EPTCS 55

Proceedings of the

15th International

Refinement Workshop

Limerick, Ireland, 20th June 2011

Edited by: John Derrick, Eerke Boiten and Steve Reeves

Published: 17th June 2011
DOI: 10.4204/EPTCS.55
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

We are proud to present the papers from the 15th Refinement Workshop, co-located with FM 2011 held in Ireland on June 20th, 2011.

Refinement is one of the cornerstones of a formal approach to software engineering: the process of developing a more detailed design or implementation from an abstract specification through a sequence of mathematically-based steps that maintain correctness with respect to the original specification.

This 15th workshop continued a 20+ year tradition in refinement workshops run under the auspices of the British Computer Society (BCS) FACS special interest group. After the first seven editions had been held in the UK, in 1998 it was combined with the Australasian Refinement Workshop to form the International Refinement Workshop, hosted at The Australian National University. Six more editions have followed in a variety of locations, all with electronic published proceedings and associated journal special issues, see the workshop homepage at `www.refinenet.org.uk` for more details.

Like the previous two editions, the 15th edition was co-located with the FM international conference, which again proved to be a very productive pairing of events. This volume contains 11 papers selected for presentation at the workshop following a peer review process. The papers cover a wide range of topics in the theory and application of refinement. Previous recent workshops have appeared in as ENTCS proceedings, this year we are publishing with EPTCS for the first time, and we would like to thank the editorial board (and in particular Rob van Glabbeek) for their help and cooperation in making this happen. This edition had a small Program Committee, whose names appear below, and we thank them for their work.

A special issue of the journal Formal Aspects of Computing is planned containing developments and extensions of the best workshop papers.

The organisers would like to thank everyone: the authors, BCS-FACS, EPTCS, and the organisers of FM 2011 for their help in organising this workshop, the participants of the workshop, and the reviewers involved in selecting the papers.

Program committee

Eerke Boiten
John Derrick
Steve Reeves

Discovery of Invariants through Automated Theory Formation *

Maria Teresa Llano† Andrew Ireland Alison Pease
Heriot-Watt University University of Edinburgh
School of Mathematical and Computer Sciences School of Informatics

Refinement is a powerful mechanism for mastering the complexities that arise when formally modelling systems. Refinement also brings with it additional proof obligations – requiring a developer to discover properties relating to their design decisions. With the goal of reducing this burden, we have investigated how a general purpose theory formation tool, HR, can be used to automate the discovery of such properties within the context of Event-B. Here we develop a heuristic approach to the automatic discovery of invariants and report upon a series of experiments that we undertook in order to evaluate our approach. The set of heuristics developed provides systematic guidance in tailoring HR for a given Event-B development. These heuristics are based upon proof-failure analysis, and have given rise to some promising results.

1 Introduction

By allowing a developer to incrementally introduce design details, refinement provides a powerful mechanism for mastering the complexities that arise when formally modelling systems. This benefit comes with proof obligations (POs) – the task of proving the correctness of each refinement step. Discharging such proof obligations typically requires a developer to supply properties – properties that relate to their design decisions. Ideally automation should be provided to support the discovery of such properties, allowing the developer to focus on design decisions rather than analysing failed proof obligations.

With this goal in mind, we have developed a heuristic approach for the automatic discovery of invariants in order to support the formal modelling of systems. Our approach, shown in Figure 1, involves three components:

- a simulation component that generates system traces,
- an Automatic Theory Formation (ATF) component that generates conjectures from the analysis of the traces and,
- a formal modelling component that supports proof and proof failure analysis.

Crucially, proof and proof failure analysis is used to tailor the theory formation component.

From a modelling perspective we have focused on Event-B [1] and the Rodin tool-set [2], in particular we have used the ProB animator plug-in [14] for the simulation component. In terms of ATF, we have used a general-purpose system called HR [4]. Generating invariants from the analysis of ProB animation traces is an approach analogous to that of the Daikon system [9]; however, while Daikon is tailored for programming languages here we focus on formal models. We come back to this in §6.

*The research reported in this paper is supported by EPSRC grants EP/F037058 and EP/F035594.
†Maria Teresa Llano is partially funded by a BAE Systems studentship.

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 1–19, doi:10.4204/EPTCS.55.1

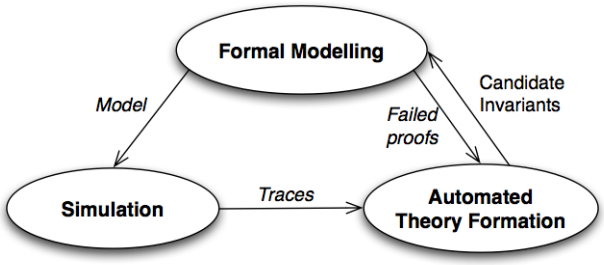


Figure 1: Approach for the automatic discovery of invariants.

Our investigation involved a series of experiments, drawing upon examples which include Abrial's "Cars on a Bridge" [1] and the Mondex case study by Butler et al. [3] . Our initial experiments highlighted the power of HR as a tool for automating the discovery of both system and gluing invariants – system invariants introduce requirements of the system while gluing invariants relate the state of the refined model with the state of the abstract model. However, our experiments also showed significant limitations: i) selecting the right configuration for HR according to the domain at hand, i.e. selection of production rules and the number of theory formation steps needed to generate the missing invariants, and ii) the overwhelming number of conjectures that are generated. This led us to consider how HR could be systematically tailored to provide practical support during an Event-B development. As a result we developed a set of heuristics which are based upon proof-failure analysis. These heuristics have given rise to some promising results and are the main focus of this paper. Although we show here the application of our technique in the context of Event-B, we believe our approach can be applied to any refinement style formal modelling framework that supports simulation and that uses proof in order to verify refinement steps.

The remainder of this paper is organised as follows. In §2 we provide background on both Event-B and HR. The application of HR within the context of Event-B is described in §3, along with the limitations highlighted above. In §4 we present our heuristics, and describe their rationale. Our experimental results are given in §5, while related and future work are discussed in §6.

2 Background

2.1 Event-B

Event-B promotes an incremental style of formal modelling, where each step of a development is underpinned by formal reasoning. An Event-B development is structured around *models* and *contexts*. A context represents the static parts of a system, i.e. *constants* and *axioms*, while the dynamic parts are represented by models. Models have a state, i.e. *variables*, which are updated via guarded actions, known as *events*, and are constrained by *invariants*.

To illustrate the basic features of a refinement consider the two events shown in Figure 2, which are part of the Mondex development [3]. The Mondex system models the transfer of money between electronic purses. The event *StartFrom* handles the initiation of a transaction on the side of the source purse. In order to initiate a transaction, the source purse must be in the *idle* state (waiting state) and after the transaction has been initiated the state of the purse must be changed to *epr* (expecting request).

As shown in Figure 2, in this step of the refinement the abstract model represents the state of purses by disjoint sets, i.e. the variables *eprP* and *idleFP*, while the concrete model handles these states through a function, i.e. the variable *statusF*, which maps a purse to an enumerated set that represents the current state, i.e. the constants *IDLEF* and *EPR*.

Abstract event

StartFrom $\widehat{=}$
  any t, p1
  where
    p1 $\in$ idleFP
    ...
  then
    eprP := eprP $\cup$ $\{p1\}$
    idleFP := idleFP $\setminus$ $\{p1\}$
    ...
end

Concrete event

StartFrom $\widehat{=}$
  refines StartFrom
  any t, p1
  where
    p1 $\mapsto$ IDLEF $\in$ statusF
    ...
  then
    statusF(p1) := EPR
    ...
end

Figure 2: Abstract and concrete views of event *StartFrom*.

Note that the keyword **refines** specifies the event being refined, while the keywords **any**, **where** and **then** delimit event *parameters*, *guards* and *actions* respectively. Note also that the concrete event on the right represents a refinement of the abstract event on the left.

In order to verify this refinement an invariant is required that relates the concrete and abstract states – these are known as *gluing invariants*. In the case of the events given above, the required gluing invariant takes the form:

$$\text{idleFP} = \text{statusF}^{-1}[\{\text{IDLEF}\}] \tag{1}$$

This invariant states that the abstract set *idleFP* can be obtained from the inverse of the function *statusF* evaluated over the enumerated set *IDLEF*. A similar gluing invariant would be required for the abstract set *eprP* and the function *statusF*. Within the Rodin toolset[1], the user is required to supply such gluing invariants. Likewise, invariants relating to state variables within a single model must also be supplied by the user – what we refer to here as *system invariants*. To illustrate, the following disjointness property represents an invariant of the abstract event above:

$$\text{eprP} \cap \text{idleFP} = \emptyset$$

From a theoretical perspective such invariants are typically not very challenging. They are however numerous and represent a significant obstacle to increasing the accessibility of formal refinement approaches such as Event-B.

2.2 Automated theory formation and HR

Lenat developed one of the earliest examples of a discovery system in mathematics; Automated Mathematician (AM) [12] and its successor Eurisko [13]. Despite subsequent methodological criticism of

[1]Rodin provides an Eclipse based platform for Event-B, with a range of modelling and reasoning plug-ins, *e.g.* UML-B [23], ProB model checker and animator [14], B4free theorem prover (`http://www.b4free.com`).

Lenat's work [22], he did show us that it is possible to formalise heuristics for discovery in mathematics. Colton has developed this intuition in his HR machine learning system[2] [4]. HR performs descriptive induction to form a theory about a set of objects of interest which are described by a set of core concepts (this is in contrast to predictive learning systems which are used to solve the particular problem of finding a definition for a target concept). Based on Colton's observation that it is possible to gain an understanding of a complex concept by decomposing it via small steps into simpler concepts, Colton defined production rules which take in concepts and make small changes to produce further concepts.

HR constructs a theory by finding examples of objects of interest, inventing new concepts, making plausible statements relating those concepts, evaluating both concepts and statements and, if working in a mathematical domain, proving or disproving the statements. Objects of interest are the entities which a theory discusses. For instance, in number theory the objects of interest are integers, in group theory they are groups, etc. Concepts are either provided by the user (core concepts) or developed by HR (non-core concepts) and have an associated data table (or table of examples). The data table is a function from an object of interest, such as the number 1, or the prime 3, to a truth value or a set of objects.

Each production rule is generic and works by performing operations on the content of one or two input data tables and a set of parameterisations in order to produce a new output data table, thus forming a new concept. The production rules and parameterisations are usually applied automatically according to a search strategy which has been entered by the user, and are applied repeatedly until HR has either exhausted the search space or has reached a user-defined number of theory formation steps to perform. Production rules include:

- The *split* rule: this extracts the list of examples of a concept for which some given parameters hold.
- The *negate* rule: this negates predicates in the new definition.
- The *compose* rule: combines predicates from two old concepts in the new concept.
- The *arithmetic* rule: performs arithmetic operations (+, -, $*$, $\div$) on specified entries of two concepts.
- The *numrelation* rule: performs arithmetic comparisons ($<$, $>$, $\leq$, $\geq$) on specified entries of two concepts.

Each time a new concept is generated, HR checks to see whether it can make conjectures with it. This could be equivalence conjectures, if the new concept has the same data table as a previous concept; implication conjectures, if the data table of the new concept either subsumes or is subsumed by that of another concept, or non-existence conjectures, if the data table for the new concept is empty.

Thus, the theories HR produces contain concepts which relate the objects of interest; conjectures which relate the concepts; and proofs which explain the conjectures. Theories are constructed via theory formation steps which attempt to construct a new concept and, if successful, formulate conjectures and evaluate the results. HR has been used for a variety of discovery projects, including mathematics and scientific domains (it has been particularly successful in number theory [6] and algebraic domains [19]) and constraint solvers [8, 21].

As an example, we show how HR produces the concept of prime numbers and the conjecture that all prime numbers are non-squares. Figure 3 shows the data tables used by HR for the formation of the concept of prime numbers.

In order to generate this concept, HR would take in the concept of divisors ($b|a$ where b is a divisor of a), represented by a data table for a subset of integers (partially shown in Figure 3 for integers from

[2] HR is named after mathematicians Godfrey Harold Hardy (1877 - 1947) and Srinivasa Aiyangar Ramanujan (1887 - 1920).

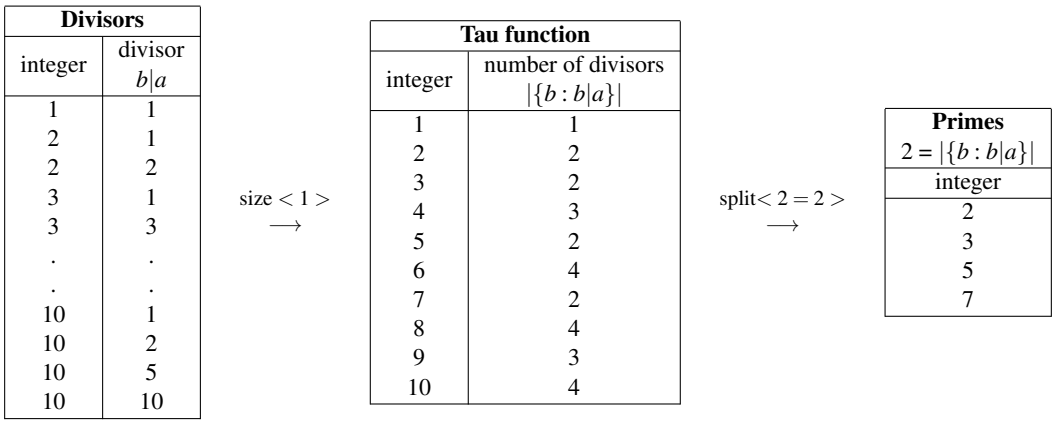


Figure 3: Steps applied by HR to produce the concept of prime numbers.

1 to 10). Then, HR would apply the size production rule with the parameterisation $< 1 >$. This means that the number of tuples for each entry in column 1 are counted, and this number is then recorded for each entry. For instance, in the data table representing the concept of divisors, 1 appears only once in the first column, 2 and 3 appear twice each, and 10 appears four times. This number is recorded next to the entries in a new data table (the table for the concept Tau function). HR then takes in this new concept and applies the split production rule with the parameterisation $< 2 = 2 >$, which means that it produces a new data table consisting of those entries in the previous data table whose value in the second column is 2. This is the concept of a prime number.

After this concept has been formed HR checks to see whether the data table is equivalent to, subsumed by, or subsumes another data table, or whether it is empty. Assuming the concept of non-square numbers has been formed previously by HR, the data tables of both the concept of prime numbers and the concept of non-square numbers, shown in Figure 4, are compared.

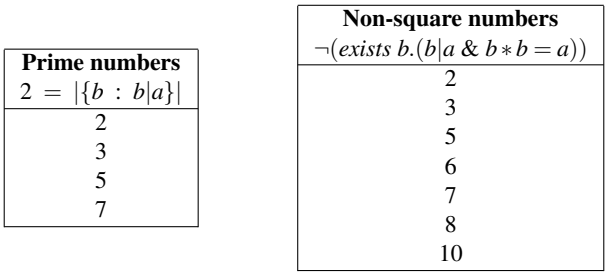


Figure 4: Data tables for the concepts of prime and non-square numbers.

HR would immediately see that all of its prime numbers are also non-squares, and so conjectures that this is true for all prime numbers. That is, it will make the following implication conjecture:

$$\underbrace{2 = |\{b : b|a\}|}_{\text{prime number}} \Rightarrow \underbrace{\neg(\mathit{exists}\ b.(b|a\ \&\ b*b = a))}_{\text{non-square number}}$$

3 Automated theory formation for Event-B models with HR

In this section we show how gluing invariant (1) introduced in the example of §2.1 can be generated through the use of theory formation and, in particular, with the HR system.

3.1 Construction of conjectures in the domain of the Mondex system

HR's input consists of a set of core concepts that describe the domain. With respect to Event-B models, these core concepts are represented by the state of the system, i.e. variables, and by the static information given in the context of the model, i.e. constants and sets. Furthermore, a concept is composed of a series of examples. Here, animation traces are used to provide HR with a list of examples for each of the concepts of an Event-B model. As mentioned before, we use ProB [14] to animate the models. For the purpose of the example, in Figure 5 we introduce some of the core concepts with their respective data tables – which were generated through the animation of the model with the ProB system.

state(A)
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
⋮
S58
S59

status(A)
IDLEF
EPR
EPA
ABORTEPR
ABORTEPA
ENDF
IDLET
EPV
ABORTEPV
ENDT

purseSet(A)
purse1
purse2
purse3
purse4
purse5

idleFP(A,B)	
S5	purse3
S6	purse3
S6	purse5
S7	purse5
S8	purse5
S19	purse4
S25	purse5
S29	purse1
S30	purse1
S31	purse1
S38	purse5
S39	purse5
S40	purse5
S44	purse5
S45	purse5
S52	purse5
S53	purse5
S54	purse5

statusF(A,B,C)		
S5	purse3	IDLEF
S6	purse3	IDLEF
S6	purse5	IDLEF
S7	purse3	EPR
S7	purse5	IDLEF
S8	purse3	EPR
S8	purse5	IDLEF
⋮	⋮	⋮
S29	purse1	IDLEF
S29	purse5	ABORTEPR
S30	purse1	IDLEF
S30	purse5	ABORTEPR
S31	purse1	IDLEF
S31	purse5	ABORTEPR
⋮	⋮	⋮
S59	purse1	ABORTEPA

Figure 5: Core concepts supplied to HR

Then, HR applied all possible combinations of concepts and production rules in order to generate new concepts and form conjectures. After the 433 step, HR formed the concept of the set of purses whose status in function *statusF* maps to *IDLEF* by applying the *split* production rule. The application of this step is illustrated in Figure 6. An intermediate output is generated with all tuples of concept *statusF* whose third column matches the parameter *IDLEF*. Since the third column is the same for all tuples of the intermediate concept, this column is removed from the final output concept.

Immediately after the generation of new concepts, HR looks for relationships with other existing concepts. As shown in Figure 7, HR found that the new concept has the same list of examples as concept *idleFP*, which gives rise to the following equivalence conjecture:

$$\forall A,B.(state(A) \wedge purseSet(B) \wedge idleFP(A,B) \Leftrightarrow status(IDLEF) \wedge statusF(A,B,IDLEF))$$

which can be represented in Event-B as (1).

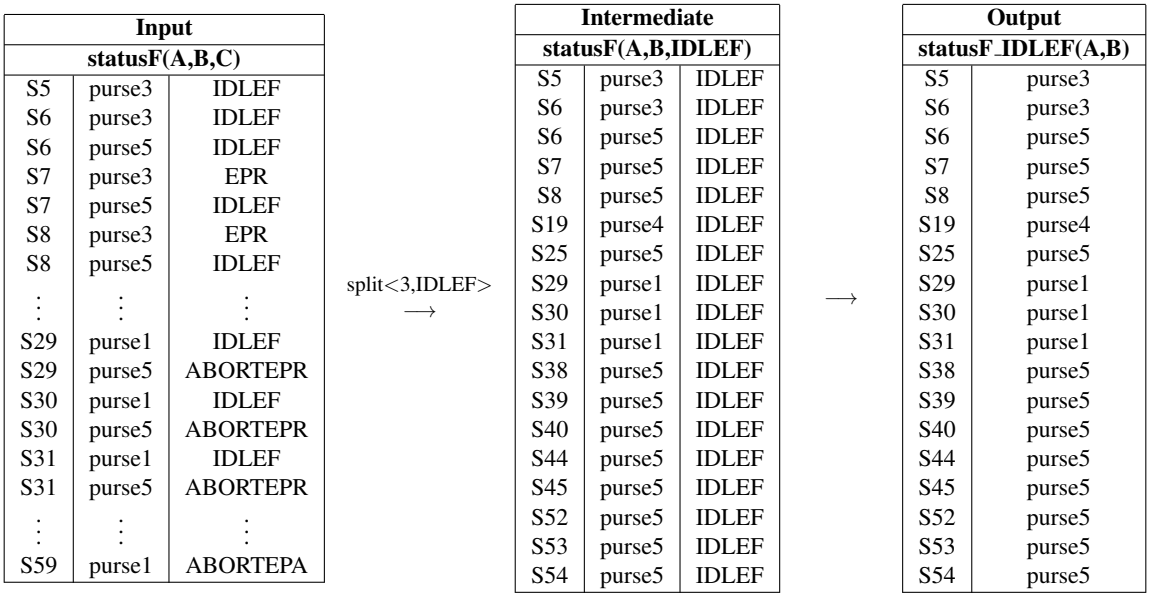


Figure 6: Split rule applied to obtain the concept of purses whose status in function *statusF* is *IDLEF*.

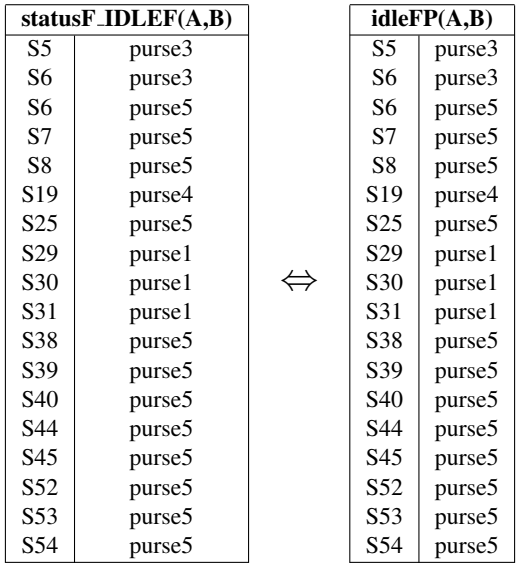


Figure 7: Formed equivalence conjecture.

3.2 Challenges in applying HR

For the domain of the Mondex system a total of 4545 conjectures were generated after 1000 formation steps. As can be observed, this is a considerable set of conjectures to analyse. In general, using HR for the discovery of invariants presented us with three main challenges:

1. The HR theory formation mechanism consists of an iterative application of production rules over

existing and new concepts. In order for HR to perform an exhaustive search, all possible combinations of production rules and concepts must be carried out. However, there is not a fixed number of theory formation steps set up for this process, since this varies depending on the domain, i.e. some domains need more theory formation steps than others. This represented a challenge for the use of HR in the discovery of invariants since it was possible that an invariant had not been formed only because not enough formation steps were run.

2. Some production rules are more effective in certain domains than others. Selecting the appropriate production rules results in the construction of a more interesting theory. For instance, if we are looking at a refinement step in an Event-B model that introduces a partition of sets we expect the new invariants to define properties over the new sets; therefore, production rules like the *arithmetic* production rule will not be of much interest in the development of the theory associated to the refinement step. Automatically selecting appropriate production rules requires knowledge about the domain; therefore, a technique was needed in order to perform this selection.
3. Finally, as highlighted in our example, HR produces a large number of conjectures – in our experiments some where in the range of 3000 to 12000 conjectures per run – from which only a very small set represented interesting invariants of the system. Thus, our main challenge was to find a way of automatically selecting the conjectures that are interesting for the domain among the conjectures obtained from HR.

In order to overcome these challenges, we have developed an approach that uses proof failure analysis to guide the search in HR. In the next section, we introduce this approach and illustrate its application, based on our running example from the Mondex case study.

4 Proof failure analysis and HR

In order to use HR, a user must first configure the system for their application domain. This involves the user in selecting production rules and conjecture making techniques, as well as deciding how many steps HR should be run. In the example introduced in §3.1, the application of the split production rule with respect to the concept *statusF*, for the value *IDLEF*, is an informed decision, based upon the user's knowledge of the model. On its own, HR does not have the capability of applying this type of reasoning. Often particular combinations of these parameters turn out to be useful for different domains. Finding the right combination is largely a process of trial and error.

Here we have developed a heuristic approach with the aim of automating this trial and error process. Our heuristics exploit the strong interplay between modelling and reasoning in Event-B. In the context of the discovery of invariants through theory formation, we use the feedback provided by failed POs to make decisions about how to configure HR in order to guide the search for invariants. Specifically, our approach consists of analysing the structure of failed POs so that we can automate:

1. the prioritisation in the development of conjectures about specific concepts,
2. the selection of appropriate production rules that increase the possibilities of producing the missing invariants and,
3. the filtering of the final set of conjectures to be analysed as possible candidate invariants.

4.1 Heuristics

Our heuristics constrain the search for invariants by focusing HR on concepts that occur within failed POs. We use two classes of heuristics – those used in configuring HR, i.e. *Pre-Heuristics (PH)*, and

those used in selecting conjectures from HR's output, i.e. *Selection Heuristics (SH)*. Below we explain each class of heuristics in turn:

4.1.1 HR configuration heuristics

We use two overall heuristics when configuring HR for a given Event-B refinement:

PH1. *Prioritise core and non-core concepts that occur within the failed POs as follows:.*

Goal concepts: concepts that appear within the goals of the failed POs.

Hypotheses concepts: concepts that appear within the hypotheses of the failed POs.

Other concepts: concepts that do not appear within the failed POs.

PH2. *Select production rules which will give rise to conjectures relating to the concepts occurring within the failed POs, i.e.*

Split rule: *is selected if members of finite sets occur.*

Arithmetic rule: *is selected if there are occurrences of arithmetic operators, e.g.* +,-,*,/.

Numrelation rule: *is selected if there are occurrences of relational operators, e.g.* $>, <, \leq, \geq$.

In addition, because of the set theoretic nature of Event-B, the compose, disjunct and negate production rules are always used in the search for invariants – where compose relates to conjunction and intersection, disjunct relates to disjunction and union and negate relates to negation and set complement.

Below we provide the rationale for these heuristics:

- As explained in §2.2, HR uses the agenda mechanism to organise the theory formation steps. The purpose of PH1 is to give higher priority to core and non-core concepts that occur within the failed POs, which means HR will generate related conjectures earlier within the theory formation process by having the prioritised concepts in the top of the agenda.

  Furthermore, we have observed that in most cases, we are able to identify the missing invariants by focusing in the first instance on the concepts that arise within the goals of the failed POs. As a result, such concepts are assigned the highest priority in the application of heuristic PH1. The concepts associated to the hypotheses follow in order of interest, while the remaining concepts are given the lesser priority.

- The missing invariants that are required in order to overcome proof failures will typically have strong syntactic similarities with the failed POs. This is the intuition behind PH2, which selects production rules that focus HR's theory formation process on such syntactic similarities.

As will be shown in §5, the empirical evidence we have gathered so far supports our rationale.

4.1.2 Conjecture selection heuristics

In order to prune the set of conjectures generated by HR, we use the following five selection heuristics:

SH1. *Select conjectures that focus purely on prioritised core and non-core concepts.*

SH2. *Select conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint.*

SH3. *Select only the most general conjectures.*

SH4. *Select conjectures that discharge the failed POs.*

SH5. *Select conjectures that minimise the number of additional proof failures that are introduced.*

The rationale for these heuristics is as follows:

- SH1 initiates the pruning of uninteresting conjectures by selecting those that describe properties about the prioritised core and non-core concepts (as identified by PH1). Furthermore, the selected conjectures should focus purely on the prioritised concepts; this means that we are interested only in equivalence and implication conjectures of the forms:

$$\alpha \Leftrightarrow \beta$$
$$\alpha \Rightarrow \beta$$
$$\beta \Rightarrow \alpha$$

  where α relates to a prioritised core or non-core concept and β to any other concept. All non-existence conjectures associated with the prioritised concepts are selected. Note that this selection criteria still gives rise to a large set of conjectures. However, as explained in the rationale of PH1 in §4.1.1, in most cases we have identified the missing invariants by focusing first on the concepts associated to the goals of the failed POs. For the selection process the same reasoning is followed and, therefore, heuristics SH1 to SH5 are focused first on conjectures associated to the concepts of the goals identified by the application of PH1. If no candidate invariants are found, or if old failures are still not addressed by the identified invariants, then the selection process starts again from SH1 to SH5 but focused on the conjectures associated with the concepts of the hypotheses.
- SH2 further prunes the set of conjectures by selecting only those that do not use the same variable(s) in both sides of the conjecture. The reason for this is that invariants in Event-B typically express relationships between different variables of the model.
- SH3 is used to eliminate redundancies amongst the set of selected conjectures by removing those that are logically implied by more general conjectures.
- SH4 is used to select candidate invariants which discharge the given failed POs.
- Potentially, overcoming one proof failure via the introduction of missing invariants may give rise to new proof fails. SH5 selects conjectures that discharge the failed POs, whilst minimising the number of new failed POs that are introduced. This iterative approach to discovering all the missing invariants is typical of Event-B developments, as described in Section 5 of [3], where invariant discovery is manual. Of course, if a development is incorrect, then this process will not terminate. We return to the issue of working with incorrect developments in §6.

Note that the selection conjectures must be applied in order from SH1 to SH5 so as to optimise the selection procedure.

4.2 Worked example

We now illustrate the application of our heuristics by returning to the refinement step described in §3.1. Recall that the gluing invariant (1) was required in order for the correctness of the refinement to be proved. When this invariant is missing from the model, an unprovable guard strengthening (*GRD*) PO[3], as shown in Figure 8, is generated. The failed PO shows that the guard $p1 \in idleFP$ of the abstract event is not implied by the guards of the concrete event.

[3] A GRD PO verifies that the guards of a refined event imply the guards of the abstract event.

Failed PO:
p1 $\mapsto$ IDLEF $\in$ statusF
t $\in$ startFromM
p1 = from(t)
Fseqno(t) = currentSeqNo(p1)
$\vdash$
p1 $\in$ idleFP

Figure 8: Failed GRD PO resulting from a missing gluing invariant

We start the process of invariant discovery with the application of heuristic PH1. We extract the list of core concepts that occur in the failed PO, giving them higher priority within the theory formation process. The extracted concepts are:

idleFP, statusF, status, startFromM, from, FSeqno and *currentSeqNo*

Except for *status*, all these concepts explicitly occur within the PO. Note that *status* is added because the constant *IDLEF* is a representative of the set *status*.

Regarding non-core concepts, the hypothesis $p1 \mapsto IDLEF \in statusF$ in the failed PO suggests that function *statusF* maps an arbitrary purse to the status *IDLEF*. This is an example of a non-core concept. This concept is obtained through the application of the split production rule over the concept *statusF* on the value *IDLEF*. No other non-core concepts are identified in the PO.

The next step is the selection of the production rules. The following production rules are used in the invariant discovery process:

compose, disjunct, negate and *split*

The compose, disjunct and negate production rules are always used in the search, as defined by heuristic PH2. The split production rule is selected because hypothesis $p1 \mapsto IDLEF \in statusF$ makes reference to a member of the finite set *status*: namely, the constant IDLEF. Thus, the split production rule is applied over the finite set *status* and the values to split are all the members of the set, i.e.: *IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV* and *ENDT*.

After the application of the PH heuristics, the initial configuration of HR is complete. By running HR for 1000 steps, 2134 conjectures were formed. This should be compared with the 4545 conjectures that are generated if our PH heuristics are not used to configure HR.

Now turning to the SH heuristics, SH1 selects conjectures that relate to the prioritised concepts that appear within the goal of the failed PO. In our example, this focuses on conjectures that involve the concept *idleFP*. After applying SH1 we obtained:

4 equivalences, 2 implications and 79 non-exists conjectures

The application of SH2 removes conjectures whose left- and right-hand sides are not disjoint with respect to the variable occurrences. The application of SH2 yields the following results:

1 equivalence, 2 implications and 79 non-exists conjectures

Through the application of SH3, less general conjectures are removed. Applying this heuristic produces:

1 equivalence, 2 implications and 46 non-exists conjectures

SH4 selects only conjectures that discharge the failed PO, the results of this step are:

1 equivalence, 0 implications and 0 non-exists conjectures

Only one conjecture discharges the failed PO. Furthermore, this conjecture does not introduce any additional failures; therefore, it represents an invariant. Within HR the invariant takes the form:

$$\forall A,B.(state(A) \wedge purseSet(B) \wedge idleFP(A,B) \Leftrightarrow status(IDLEF) \wedge statusF(A,B,IDLEF))$$

which translates into the missing gluing invariant (1). It should be noted that this conjecture was formed by HR after one theory formation step. This shows that, in this example, our heuristics guided HR to discover interesting conjectures early within the theory formation process.

5 Experimental results

The experiments we carried out were divided into two stages. The first stage involved the *development* of our heuristics, and was based upon four relatively simple Event-B models, as described below:

1. *Traffic light system:* This model represents a traffic light circuit that controls the sequencing of lights. It is composed of an abstract model and involves a single refinement. The abstract model controls the red and green lights, while the refinement introduces a third light to the sequence, i.e. an amber light.
2. Two representations of a vending machine:
   - *Set-like representation:* This model of a vending machine controls the stock of products through the use of states. It is composed of an abstract and a concrete model. The abstract model represents the states of products using state sets, while the refinement introduces a status function that maps products to their states.
   - *Arithmetic-like representation:* This model of the vending machine uses natural numbers to represent the stock and money held within the machine. While the abstract model deals with a single product, the refinement introduces a second product to the vending machine.
3. *Refinements 1 and 2 of Abrial's cars on a bridge system [1]:* Models a system that controls the flow of cars on a bridge that connects a mainland to an island. At the abstract level, cars are modelled leaving and entering the island, the first refinement introduces the requirement that the bridge only supports one way traffic, while the second refinement introduces traffic lights.

We used the second stage of our experiments to *evaluate* the heuristics developed during stage one. Here the experiments were performed on more complex Event-B models:

1. *Refinement 3 of Abrial's cars on a bridge system [1]:* The third refinement of this system models the introduction of sensors that detect the physical presence of cars.
2. *The Mondex system [3]:* Models an electronic purse that allows the transfer of money between purses. This development is composed of one abstract model and nine refinement steps. We targeted the third, fourth and eighth refinement steps. The third refinement handles dual state sets in both sides of a transaction in order to handle information locally. The fourth refinement introduces the use of messaging channels between purses and the eighth refinement introduces a status function that maps purses to their states instead of using state sets.

In the work reported in [3], it was highlighted that the manual analysis of failed POs was used to guide the construction of gluing invariants. In particular, this was illustrated in the third step of the refinement in which, through the analysis of failed POs, and after three iterations of invariant strengthening, the set of invariants needed to prove the refinement between levels three and four were added to the model. As part of our experiments we attempted the re-construction of the Mondex system in Event-B based on the development presented in [3]. In the following section we present the results obtained by the application of our approach to the refinement between levels three and four of the Mondex system, and we show that these results are similar to the ones obtained through the interactive development [3].

5.1 The Mondex system

In level three of the Mondex system a transaction is permitted to be in one of four states: *idle, pending, recover or ended*, while the refinement in level four introduces dual states to a transaction so that each side has their own local protocol state. In order to evaluate our approach, we introduced the model in level 4 with only basic typing invariants. The absence of the invariants produces the failed POs shown in Figure 9.

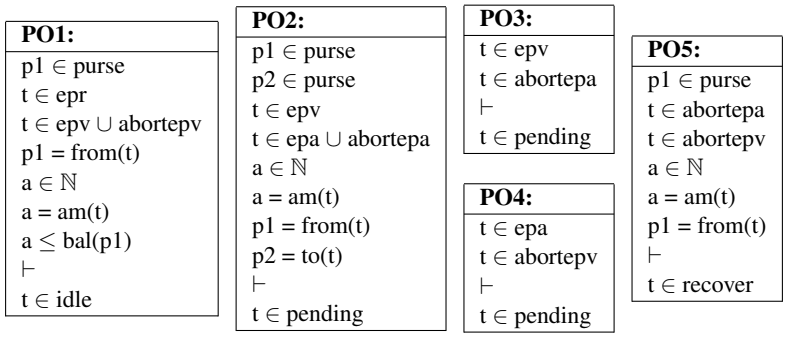


Figure 9: First set of failed POs.

We start the invariant discovery process with the application of heuristic PH1. The set of core concepts selected from the failed POs are:

idle, pending, recover, purse, epr, epv, abortepv, from, am, bal, epa, abortepa and *to*

Moreover, from the analysis of the predicates in the failed POs, we identify the following non-core concepts:

$epv \cup abortepv$ and $epa \cup abortepa$

These concepts are identified from hypotheses $t \in epv \cup abortepv$ and $t \in epa \cup abortepa$ within PO1 and PO2, respectively. Note that t does not represent a concept in the domain, it represents an arbitrary transaction passed as a parameter to the event associated with the failed POs. For this reason, only the right hand sides of the membership relations are selected as interesting non-core concepts.

The process continues with the selection of the productions rules. Based on the failed POs shown in Figure 9, the following production rules are selected for the search:

compose, disjunct, negate and *numrelation*

The compose, disjunct and negate production rules are always used in the search as stated in heuristic PH2. The numrelation production rule is selected because hypothesis $a \leq bal(p1)$ within PO1 expresses a property based on the relational operator $\leq$. After the pre-heuristics have been applied HR is run for 1000 steps, resulting in 7296 conjectures.

The selection heuristics are now applied over this set of conjectures. Heuristic SH1 suggests looking at the prioritised concepts associated to the goals of the failed POs. From the goals of the POs shown in Figure 9, we identified the concepts *idle, pending* and *recover*. Thus, we look for the conjectures associated to each of these concepts. The results from the application of this heuristic are shown in Table 1. This table also shows the results of applying heuristics SH2, SH3 and SH4 over each of the selected concepts.

Heuristic	Concept	Equivalences	Implications	Non-exists
SH1	idle	7	27	24
	pending	6	27	35
	recover	9	51	41
SH2	idle	0	27	24
	pending	0	27	35
	recover	2	51	41
SH3	idle	0	6	17
	pending	0	8	26
	recover	2	3	30
SH4	idle	0	2	0
	pending	0	2	0
	recover	1	0	0

Table 1: Results of the application of selection heuristics SH1, SH2, SH3 and SH4.

As can be observed, after applying the four initial selection heuristics we have narrowed the set of selected conjectures to a total of five conjectures: two implications involving the concept *idle*, two implications for concept *pending* and one equivalence about the concept *recover*.

The final step in the discovery process is the selection of the conjectures that produce the smaller number of new failed POs. The two implications associated with concept *idle* discharge PO1 and produce one extra failed PO. We believe that in this kind of situation it is the user who has to decide which one is the most appropriate conjecture according to his/her knowledge about the model. Thus, we present both conjectures as candidate invariants and leave the decision of which one to select to the user. Regarding the two implications associated with concept *pending*, one of them discharges PO2 and PO3 and produces two new failed POs, while the other one discharges PO4 but produces three new failed POs. As there are no other conjectures that help to overcome the failures produced by PO2, PO3 and PO4, both conjectures are suggested as candidate invariants. Finally, the equivalence conjecture associated with concept *recover* discharges PO5 and it does not produce any extra failures, so this conjecture is also suggested as a candidate invariant. The set of invariants represented by the conjectures obtained from HR in this first iteration of our approach is shown in Figure 10(a) [4].

After the new set of invariants is introduced to the model, six new failed POs are generated. We then start the analysis again by applying our approach based on the new set of failed POs. This new iteration results in the discovery of five new invariants. Again, when these invariants are added to the model, one

[4]Note that we have given the equivalent set theoretic representation of these conjectures instead of using the universally quantified format provided by HR. This is because some experiments, for instance the development of the Mondex system carried out in [3], have shown that the automatic provers do better with quantifier-free predicates.

new failed PO is generated. We discovered one new invariant after a third iteration of our approach. No further failed POs are generated when this invariant is added to the model. Figures 10(b) and Figure 10(c) shows the invariants obtained after the second and the third iteration, respectively.

$\text{epr} \subseteq \text{idle}$
$(\text{idleF} \cup \text{epr}) \subseteq \text{idle}$
$\text{epv} \cap (\text{epa} \cup \text{abortepa}) \subseteq \text{pending}$
$\text{epa} \cap (\text{epv} \cup \text{abortepv}) \subseteq \text{pending}$
$\text{abortepa} \cap \text{abortepv} = \text{recover}$
(a)

$\text{idleF} \subseteq \text{idle}$
$\text{idleT} \cap (\text{epa} \cup \text{abortepa}) = \varnothing$
$\text{idleT} \cap (\text{epv} \cup \text{abortepv}) = \varnothing$
$\text{epv} \cap \text{abortepv} = \varnothing$
$\text{epa} \cap \text{abortepa} = \varnothing$
(b)

$\text{epr} \cap \text{idleF} = \varnothing$
(c)

Figure 10: Invariants obtained through three iterations.

The invariants shown in Figure 10 are a subset of the invariants suggested in [3] for this step of the refinement. In total we obtained 11 invariants from the 17 used in [3]. However, it is important to note that we have addressed all the failures produced when proving consistency between the refinement levels. Our hypothesis, is that the extra invariants used in [3] represent new requirements of the system, which are out of the scope of our technique since we only target invariants needed to prove the refinement steps.

5.2 Summary of results

Table 2 summarises the results of the application of our approach in each of the Event-B models used during the development and the evaluation stages. Notice that all the experiments were performed over models with only basic typing invariants. This means that neither gluing nor system invariants were present in the models when using our technique. The table shows for each refinement step, the number of failed POs that arose, as well as the number of gluing and system invariants discovered through our approach. We also record the number of iterations involved in the invariant discovery process.

	Event-B model	**Step**	**Failed POs**	**Invariants**			
				Automatically discovered			
				Glue	**System**	**Total**	**Iteration**
Development set	Traffic light	Level 1-2	2	2	0	**2**	1
	Vending machine (Arith)	Level 1-2	6	3	0	**3**	1
	Vending machine (sets)	Level 1-2	6	3	0	**3**	1
	Cars on a bridge	Level 1-2	2	1	0	**1**	1
		Level 2-3	6	0	5	**5**	1
		Level 3-4	7	0	5	**5**	1
Evaluation set	Mondex	Level 3-4	5	4	0	**4**	1
			6	1	4	**5**	2
			1	0	1	**1**	3
		Level 4-5	3	0	3	**3**	1
			5	0	4	**4**	2
			4	0	2	**2**	3
		Level 8-9	14	10	0	**10**	1

Table 2: Automatically discovered invariants.

In Table 3 we compare our results with the actual invariants given in the literature for the models of the cars on a bridge [1] and the Mondex system [3]; the other developments are not compared because they are of our own authorship (note that the invariants of the refinement from levels four to five of the Mondex system are not given in the literature). All automatically discovered invariants are subsets of the

invariants given in the literature; however, it is important to highlight that the automatically discovered invariants were sufficient to prove all the refinement steps in our experimental models.

As it can be observed from Table 3, the automatic discovery of gluing invariants through the use of theory formation and the HR system has provided promising results. In most cases, the set of gluing invariants discovered through our technique was almost identical to the set of gluing invariants provided in the literature. Regarding system invariants, it can be observed that the last refinement of the cars on a bridge system shows a big gap between the invariants given in the literature and those found automatically with our approach. As mentioned previously, we believe that this difference can be explained by the introduction of new requirements, resulting in the need for extra properties in the model.

Event-B model	**Step**	**Given in Literature**			**Automatically discovered**		
		Glue	**System**	**Total**	**Glue**	**System**	**Total**
Cars on a bridge	Level 1-2	1	1	**2**	1	0	**1**
	Level 2-3	0	6	**6**	0	5	**5**
	Level 3-4	0	23	**23**	0	5	**5**
Mondex	Level 3-4	8	9	**17**	5	5	**10**
	Level 4-5	-	-	-	0	9	**9**
	Level 8-9	10	0	**10**	10	0	**10**

Table 3: Comparison between hand-crafted and automatically discovered invariants.

Figure 11, shows all the invariants that were discovered through the application of our approach. The invariants for refinement three of the Mondex system are omitted since they are shown in §5.1.

available = productStatus^{-1}[{AVAILABLE}]
limited = productStatus^{-1}[{LIMITED}]
soldOut = productStatus^{-1}[{SOLDOUT}]

(a) Vending machine (sets) invariants

stock = stockMilk + stockPlain
sold = soldMilk + soldPlain
givenCoin = EMPTY_COIN $\Leftrightarrow$ coin = NO_COIN

(b) Vending machine (arith) invariants

n = a + b + c
ml_tl = green $\Rightarrow$ c = 0
il_tl = green $\Rightarrow$ a = 0
ml_tl = red $\Rightarrow$ ml_pass = 1
il_tl = red $\Rightarrow$ il_pass = 1
il_tl = green $\Rightarrow$ ml_tl = red
ml_out_10 = TRUE $\Rightarrow$ ml_tl = green
il_out_10 = TRUE $\Rightarrow$ il_tl = green
IL_IN_SR = on $\Rightarrow$ A > 0
IL_OUT_SR = on $\Rightarrow$ B > 0
ML_IN_SR = on $\Rightarrow$ C > 0

(c) Cars on a bridge invariants

epr $\cap$ reqM $\subseteq$ epv $\cup$ abortepv
epv $\cap$ valM $\subseteq$ epa $\cup$ abortepa
endT = endF $\cup$ ackM
reqM $\cap$ idleF $\subseteq$ epv $\cup$ abortepv
valM $\cap$ idleT = $\varnothing$
epr $\cap$ valM = $\varnothing$
epr $\cap$ abortepa = $\varnothing$
valM $\cap$ idleF = $\varnothing$
abortepa $\cap$ idleF = $\varnothing$

(d) Mondex invariants (ref 4)

idleFP = statusF^{-1}[{IDLEF}]
eprP = statusF^{-1}[{EPR}]
epaP = statusF^{-1}[{EPA}]
aborteprP = statusF^{-1}[{ABORTEPR}]
abortepaP = statusF^{-1}[{ABORTEPA}]
endFP = statusF^{-1}[{ENDF}]
idleTP = statusF^{-1}[{IDLET}]
epvP = statusF^{-1}[{EPV}]
abortepvP = statusF^{-1}[{ABORTEPV}]
endTP = statusF^{-1}[{ENDT}]

(e) Mondex invariants (ref 8)

r_light = TRUE $\vee$ amber_light = TRUE $\Leftrightarrow$ red_light = TRUE
g_light = TRUE $\Leftrightarrow$ green_light = TRUE

(f) Traffic light invariants

Figure 11: Automatically discovered invariants

6 Related and future work

As far as we are aware, automated theory formation techniques have not been investigated within the context of refinement style formal modelling. The closest work we know of is Daikon [9], a system which uses templates to detect likely program invariants by analysing program execution traces. The quality of the invariants generated by both approaches depends in part upon the quality of the input data – ProB animation traces in our work and program test suites for Daikon. Like HR, Daikon is configurable. However, while HR is a general purpose theory formation tool, Daikon has been designed with program analysis in mind. It should also be stressed that Daikon is a system, whereas the work presented here is an initial investigation into developing an invariant generation tool for refinement based formal methods.

Within automated theory formation there are a number of alternative tools to HR that could be explored. For instance, IsaCoSy [11], IsaScheme [20], the CORE system [16] and MathsAid [17]. Underlying the first three of these systems is a notion of *term synthesis*, i.e. the automatic generation of candidate conjectures based upon application of domain knowledge. IsaCoSy and IsaScheme support the discovery of theorems within the context of mathematical induction, while MathsAid provides broader support for the development of mathematical theories. The CORE system has a strong software verification focus, supporting the automatic generation of frame and loop invariants for use in reasoning about pointer programs. What distinguishes these approaches from HR is that they do not rely upon animation/execution traces, instead they follow a generate-and-test approach, where the "test" component involves theorem proving. Coupled with its configurability, the trace analysis capability led us to use HR for our investigations.

As noted above, animation is key to our approach, where the quality of the invariants produced by HR strongly depends on the quality of the animation traces. The ProB animator provided good animation traces for most of our experiments; however, we found two areas where further improvements are required:

1. We believe that increasing the randomness in the production of the traces would improve our results.
2. ProB preferences only allows for the creation of sets with a few elements, as well as very limited integer ranges. This restricted some of the traces we were able to generate, and thus impacted negatively on the invariants that could be discovered. Specifically, this limitation arose during our analysis of the Mondex case study.

The process of finding a "correct" refinement will typically involve exploring many "incorrect" refinements. While the work reported here focuses on supporting the verification of correct refinements, we are currently investigating how counter-examples generated by ProB could be combined with HR in order to provide useful feedback to a developer when faced with an incorrect refinement.

Longer-term, we are looking to use theory formation within our REMO [10] formal modelling planning system. That is, when faced with a refinement failure, we aim to use theory formation, automatically tailored by refinement plans [15], to suggest modelling alternatives. Of course, such "modelling alternatives" are only suggestions, ultimately users must select which is most appropriate to their needs.

Currently, the animation traces obtained from the ProB animator are automatically converted into HR's input, i.e. a domain file with the list of examples for each of the concepts (variables, constants and sets) is created. However, the automation of the heuristics is still under development. Automating the configuration heuristics involves the prioritisation of concepts in the domain file and the creation of a macro file. The macro file records the search strategy that will be applied by HR (usually supplied by the user). Here is where the production rules selected by our proof failure analysis are specified. The

automation of the selection heuristics requires integration with a theorem prover and the Rodin toolset. HR uses the Otter theorem prover [18] to prove the conjectures. We will exploit the use of the Otter theorem prover in HR for the selection of the most general conjectures (heuristic SH3), while the Rodin toolset will be used to obtain the status of the POs after the candidate invariants have been introduced into the model (heuristics SH4 and SH5). As future work, we aim to automate this process and, as mentioned before, integrate it with our REMO tool.

7 Conclusions

We have described an investigation into how the HR theory formation tool can be used to automatically discover the kinds of invariants that developers typically have to supply in order to verify Event-B refinements. The key contribution of our work is the development of a set of heuristics. Using proof-failure analysis to prune the wealth of conjectures HR discovers, these heuristics have proven highly effective at identifying missing invariants. While more experimentation is required, we believe that our heuristics provide a firm foundation upon which to further explore techniques that support formal refinement – techniques that suggest design alternatives, whilst removing the burden of proof failure analysis from developers.

Acknowledgements: Our thanks go to Alan Bundy, Gudmund Grov and Julian Gutierrez for their feedback and encouragement with this work. Also, we want to thank Jens Bendisposto and the ProB development team for their assistance, and Simon Colton and John Charnley for their help in using the HR system.

References

[1] J-R. Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

[2] J-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta & L. Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. *STTT* 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] M. Butler & D. Yadav (2008): *An incremental development of the Mondex system in Event-B*. *Formal Aspects of Computing* 20(1), pp. 61–77, doi:10.1007/s00165-007-0061-4.

[4] S. Colton (2002): *Automated Theory Formation in Pure Mathematics*. Springer-Verlag.

[5] S. Colton, A. Bundy & T. Walsh (2000): *Automatic Identification of Mathematical Concepts*. In: *17th International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, USA, pp. 183–190.

[6] S. Colton, A. Bundy & T. Walsh (2000): *Automatic invention of integer sequences*. In: *16th IJCAI*, pp. 786–791.

[7] S. Colton, A. Bundy & T. Walsh (2000): *On the Notion of Interestingness in Automated Mathematical Discovery*. *International Journal of Human Computer Studies* 53(3), pp. 351–375, doi:10.1006/ijhc.2000.0394.

[8] S. Colton & I. Miguel (2001): *Constraint Generation via Automated Theory Formation*. In: *7th International Conference on the Principles and Practice of Constraint Programming*, doi:10.1007/3-540-45578-7_42.

[9] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz & C. Xiao (2007): *The Daikon system for dynamic detection of likely invariants*. *Science of Computer Programming* 69(1–3), pp. 35–45, doi:10.1016/j.scico.2007.01.015.

[10] A. Ireland, G. Grov, M. Llano & M. Butler (2011): *Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance*. *Science of Computer Programming* doi:10.1016/j.scico.2011.03.006.

[11] M. Johansson, L. Dixon & A. Bundy (2010): *Case-Analysis for Rippling and Inductive Proof*. In: *1st International Conference on Interactive Theorem Proving*, *LNCS* 6127, Springer, pp. 291–306, doi:10.1007/978-3-642-14052-5_21.

[12] D. Lenat (1976): *AM: An Artificial Intelligence approach to discovery in mathematics*. Ph.D. thesis, Stanford University.

[13] D. Lenat (1983): *Eurisko: A program which learns new heuristics and domain concepts.* *Artificial Intelligence* 21, doi:10.1016/S0004-3702(83)80005-8.

[14] M. Leuschel & M. Butler (2003): *ProB: A Model Checker for B*. In: *International Symposium of Formal Methods Europe*, *LNCS* 2805, Springer, pp. 855–874, doi:10.1007/978-3-540-45236-2_46.

[15] M. Llano, G. Grov & A. Ireland (2010): *Automatic Guidance for Refinement Based Formal Methods*. *5th workshop on Automated Formal Methods (AFM'10), a satellite workshop of the 22nd International Conference on Computer Aided Verification (CAV'10).* Also available via: School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0076; School of Informatics, University of Edinburgh, Report EDI-INF-RR-1371.

[16] E. Maclean, A. Ireland, L. Dixon & R. Atkey (2009): *Refinement and Term Synthesis in Loop Invariant Generation*. In: *2nd International Workshop on Invariant Generation (WING'09), a satellite workshop of ETAPS'09*.

[17] R. McCasland, A. Bundy & S. Autexier (2007): *Automated Discovery of Inductive Theorems*. In: *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, *Studies in Logic, Grammar and Rhetoric* 10(23), University of Białystok, pp. 135–149.

[18] W. McCune (2003): *OTTER 3.3 Reference Manual*. *CoRR* cs.SC/0310056.

[19] A. Meier, V. Sorge & S. Colton (2002): *Employing Theory Formation to Guide Proof Planning*. In: *AISC/Calculemus'02, LNAI 2385*, Springer, doi:10.1007/3-540-45470-5_25.

[20] O. Montano-Rivas, R. McCasland, L. Dixon & A. Bundy (2010): *Scheme-Based Synthesis of Inductive Theories*. In: *MICAI*, *LNCS* 6437, pp. 348–361, doi:10.1007/978-3-642-16761-4_31.

[21] A. Pease, A. Smaill, S. Colton, A. Ireland, M. Llano, R. Ramezani, G. Grov & M. Guhe (2010): *Applying Lakatos-style reasoning to AI problems*. In: *Thinking Machines and the philosophy of computer science: Concepts and principles*, IGI Global, PA, USA, pp. 149–174, doi:10.4018/978-1-61692-014-2.

[22] G. Ritchie & F. Hanna (1990): *AM: a case study in methodology*. In D. Partridge & Y. Wilks, editors: *The foundations of AI: a sourcebook*, CUP, Cambridge, pp. 247–265, doi:10.1017/CBO9780511663116.024.

[23] C. Snook & M. Butler (2006): *UML-B: Formal modeling and design aided by UML*. *ACM Transactions on Software Engineering and Methodology.* 15(1), pp. 92–122, doi:10.1145/1125808.1125811.

Bigraphical Refinement*

Gian Perrone Søren Debois Thomas Hildebrandt

Programming, Logic and Semantics Group
IT University of Copenhagen
Copenhagen, Denmark
{gdpe,debois,hilde}@itu.dk

We propose a mechanism for the vertical refinement of bigraphical reactive systems, based upon a mechanism for limiting observations and utilising the underlying categorical structure of bigraphs. We present a motivating example to demonstrate that the proposed notion of refinement is sensible with respect to the theory of bigraphical reactive systems; and we propose a sufficient condition for guaranteeing the existence of a safety-preserving vertical refinement. We postulate the existence of a complimentary notion of horizontal refinement for bigraphical agents, and finally we discuss the connection of this work to the general refinement of Reeves and Streader.

1 Introduction

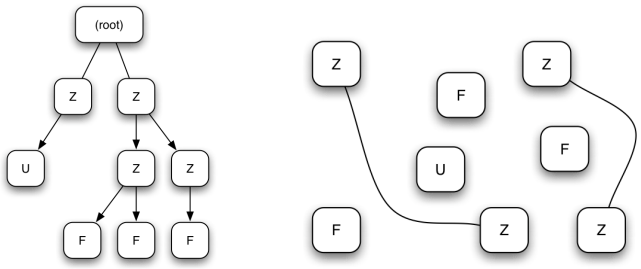
Refinement is the process of gradually developing a specification towards a suitable implementation, through a series of steps in which more concrete entities are shown to be as acceptable as the more abstract entities preceding it in the chain of refinement steps, based upon what may be observed of these entities. The utility of this method has been demonstrated through many years of application in academic and industrial settings. In this paper we attempt to bring these well-studied benefits to a new class of systems — namely, bigraphical reactive systems. We focus primarily on *vertical refinement* [3], where the aim is to relate models constructed with respect to different semantics.

A *bigraphical reactive system* [21, 19] (BRS) is a model construction paradigm proposed by Milner and colleagues that aims to enable modelling of interactive systems within a cohesive theoretical framework. While the primary long-term focus of bigraphs is on models of ubiquitous and context-aware systems [1], they have demonstrated value in other areas such as biological applications [15, 5, 6] and business processes [12, 25]. Bigraphical reactive systems also capture the syntactic and semantic structure of many formalisms associated with process modelling, providing a unifying meta-calculus within which to relate many of these well-developed theories. Already encodings into various bigraphical reactive systems have been demonstrated for amongst others the λ-calculus [20], CCS [19], the Mobile Ambients calculus [14], several variants of the π-calculus [14, 4, 8], Fusion Calculus [10] and Petri Nets [16].

Bigraphical reactive systems consist of two graphs (hence the name *bi*graph) modelling the orthogonal notions of *locality* and *connectivity* which together capture the static structure of a system, and a set of *reaction rules* that may selectively rewrite portions of the bigraph in order to capture the dynamic behaviour of that system. We will introduce bigraphs and bigraphical reactive systems (assuming no prior knowledge) in Section 2.

*This work funded in part by the Danish Research Agency (grant no.: 2106-080046) and the IT University of Copenhagen (the Jingling Genies project). The first author would like to thank Prof. Steve Reeves and Dr. David Streader for hosting him as a visiting researcher at the University of Waikato during the early stages of this work, and for helpful discussions during this time.

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 20–36, doi:10.4204/EPTCS.55.2



(a) Place Graph (b) Link Graph

Figure 1: The constituent place (1a) and link (1b) graphs that form a particular bigraph.

The usual notion of "observation" in a BRS is derived from the above notion of dynamic behaviour: a BRS gives rise to an LTS, the labels of which are simply the least context enabling reaction. The present effort towards refinement takes this connection between static structure and dynamic behaviour to heart, and attempts to short-circuit the LTS in favour of a more directly structural mechanism of refinement. This makes sense uniquely for bigraphs exactly because of the close correspondence between structure and dynamics. The primary contribution of this paper is to introduce such a mechanism as a small step towards bringing the well-established benefits of refinement to models constructed within the bigraph formalism. Additionally, we give a sufficient condition for an abstraction functor (Section 4) to give rise to a safe refinement, and show that this notion of refinement corresponds with (and indeed, in part is an instance of) the general refinement of Reeves and Streader [23, 24].

1.1 Structure of the paper

The remainder of this paper is structured as follows: We review bigraphs (assuming no prior knowledge) in Section 2. In Section 3 we introduce a running example that will be used to illustrate all of the concepts presented. In Section 4 we present our definition of vertical refinement for bigraphical reactive systems and show that the proposed refinement preserves safety properties with respect to the abstraction functor upon which it is parametrised. Additionally, we present a sufficient condition for an abstraction functor to give rise to a safe refinement. Finally, in Section 5 we discuss a candidate horizontal refinement mechanism for bigraphical agents, derived from the general refinement of Reeves and Streader [23, 24], and discuss the connection of this work to general refinement.

2 Bigraphical Reactive Systems

Bigraphical reactive systems is a graphical formalism emphasising the orthogonal notions of *locality* and *connectivity*. A BRS is a category of bigraphs and a set of reaction rules that may be applied to rewrite these bigraphs. We provide here a short, informal introduction to the anatomy of a BRS without assuming any prior knowledge. For a complete treatment of bigraphs and BRSs, readers are referred to [21, 19].

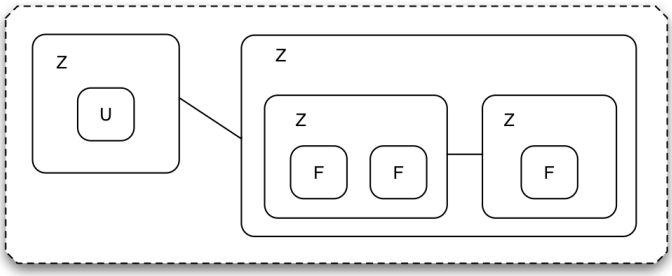


Figure 2: The bigraph resulting from the combination of the place and link graphs in Fig. 1a and Fig. 1b. This bigraph is an agent of the BRS_{notify} example BRS with signature $\Sigma = \{\mathsf{Z}, \mathsf{U}, \mathsf{F}, \mathsf{N}\}$ that we will introduce in Section 3.

2.1 Static Structure

The most basic construction within the static fragment of bigraphical reactive systems is the *node*. This follows from normal definition of a node within graph theory. To nodes we assign *controls*, which are drawn from a *signature* Σ, the set of controls. We sometimes use a convenient shorthand such that we may refer to a node as being an "X node", by which we really mean a node that has been assigned the control X. Nodes may be nested to arbitrary depth to form a tree that is known as the *place graph* (Fig. 1a). We represent this nesting by containment, as shown in Fig. 2. We distinguish between controls of two kinds: *active* and *passive* ones; we shall see later how active controls admit dynamic behaviour beneath them whereas passive controls do not. Every tree of nodes is contained by a *region* (the dotted border in Fig. 2). Bigraphs permit multiple regions (a place forest).

To controls (and therefore nodes) we assign a fixed *arity*, which defines the number of *ports* that a given node possesses. A port is a connection point on a node; it must always be connected to other such connection points by the *link graph*. The link graph (Fig. 1b) is an undirected hypergraph over the ports of the nodes of the place graph. A single (hyper) edge may connect arbitrarily many ports on different nodes.

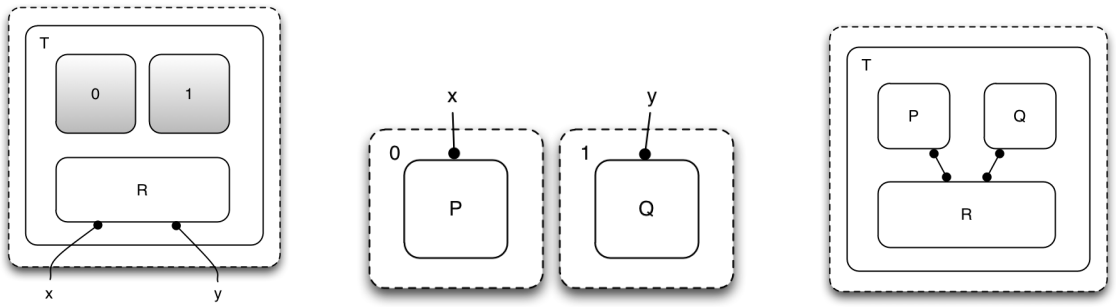
Within the place graph, in addition to regions and nodes, there may also exist *holes* (known as *sites* in some bigraphs literature), which are expressed visually as shaded grey nodes (as in Fig. 3a). A hole is a location into which a region of another bigraph may be inserted by composition. It may be helpful to think of bigraphs with holes as "contexts" and those without as "processes" or "terms".

Present also within Fig. 3 are *names* that represent (named) points at which edges of the link graph may be fused to form a single (hyper) edge. In the intuition of contexts and terms, names of bigraphs roughly correspond to unstructured names, as in the π-calculus. By convention, *outer names* are drawn upwards, and *inner names* are drawn downwards. Outer names are analogous in the link graph to regions in the place graph, while inner names are analogous to holes. Through composition of link graphs, sets of inner and outer names that agree are matched and joined.

Definition 1 (Interface)**.** *An interface is a pair $\langle j, X\rangle$ where $0 \leq j$, indicating the number of holes or regions, and X is a set of (inner or outer) names.*

Definition 2 (Bigraph)**.** *A bigraph is a 5-tuple:*

$$(V, E, ctrl, prnt, link) : \langle k, X\rangle \to \langle m, Y\rangle$$



(a) $A : \langle 2, \{x,y\}\rangle \to \langle 1, \emptyset\rangle$ (b) $B : \langle 0, \emptyset\rangle \to \langle 2, \{x,y\}\rangle$ (c) $A \circ B : \langle 0, \emptyset\rangle \to \langle 1, \emptyset\rangle$

Figure 3: The composition of two bigraphs A and B with their respective interfaces

Here V is the set of nodes, E is the set of hyperedges, $ctrl$ is the control map *that assigns controls (and therefore arities) to nodes, $prnt$ is the* parent map *that defines the tree structure in the place graph and $link$ is a link map that defines the link structure. The inner interface $\langle k, X\rangle$ indicates that the bigraph has k holes, and a set of inner names X. The outer interface $\langle m, Y\rangle$ indicates that the bigraph has m regions and a set of outer names Y.*

Definition 3 (Composition)**.** *Bigraphs are composed separately in the place and the link graphs. The interfaces of the bigraphs must be compatible in order for composition to be defined, i.e., the sets of names and the number of regions/holes must be the same. Fig. 3 illustrates the composition $A \circ B$ of bigraphs A and B. In the place graph, we insert contents of the left-most region of B into hole 0 of A, and the contents of the right-most region of B into hole 1 of A. Regions are numbered left-to-right: we insert the contents of region 0 into hole 0 etc. In the link graph, links are spliced together where there is name agreement between the inner and outer names of the bigraphs being composed. We may refer to A in this case as being a* context *into which B is inserted.*

Definition 4 (Tensor Product)**.** *There exists an additional way in which to combine bigraphs, namely the* tensor product *$A \otimes B$, where A and B are bigraphs. Where A and B do not share any inner or outer names, this just involves juxtaposing their place graphs, taking the union of their names, and increasing the indices of holes in B to make them unique with respect to A. This definition obscures some technical details. It is recommended that readers interested in following the proofs in Section 4.1 refer to [21] for a precise definition.*

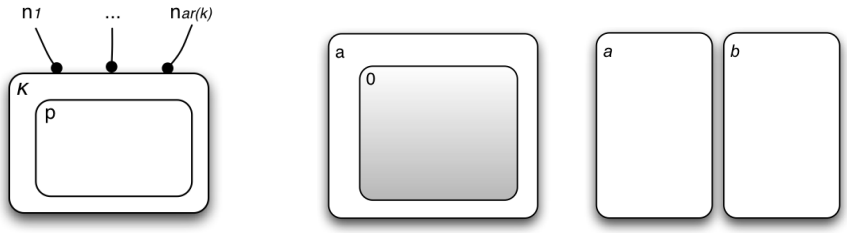
2.2 Notation

We introduce a rudimentary term language for representing bigraphs that should be familiar to most readers accustomed to the notation for process algebras. The present language is not complete, i.e., it cannot express every bigraph, but it can express the ones we will use in examples. It is a subset of a complete such language [18]. We will use this term language in conjunction with the graphical representation used in Fig. 2.

Definition 5 (Bigraph Term Language)**.**

$$p ::= \kappa(n_1, \ldots, n_{ar(\kappa)}).p \mid p \mid p \mid -_i \mid \mathsf{nil}$$

Where $\kappa \in \Sigma$.



(a) $\kappa(n_1,\ldots,n_{ar(\kappa)}).p$ (b) $a.-_0$ (c) $a.\mathsf{nil} \mid b.\mathsf{nil}$

Figure 4: Example bigraph terms with their associated graphical representation

The term language requires some explanation — $\kappa(n1,\ldots,n_{ar(\kappa)}).p$ is *prefixing* (Fig. 4a), indicating a node assigned the control κ. The arity of κ is given by $ar(\kappa)$. The sequence $n_1,\ldots,n_{ar(\kappa)}$ are the ports of the node. Finally, the suffix p is the term that is nested inside this node. $p \mid p$ is *juxtaposition* of terms (Fig. 4c), placing them as siblings within the place graph. $-_i$ is a hole (Fig. 4b), indexed by some integer $0 \leq i$. Finally, nil is the nil terminator which is simply the empty graph in the graph representation.

2.3 Dynamics

Having introduced the basic structure of *bigraphs*, the static portion of a BRS, we now introduce the *reactive* portion of a BRS that imbues a system with dynamic behaviour. This relies on *reaction rules* that define rewriting that may be applied to a bigraph. A reaction rule (R,R',η) consists of a *redex* R, a *reactum* R' and an *instantiation map* η, where the redex is a bigraph to be matched and the reactum is the bigraph with which the matched portion of the bigraph should be replaced. The instantiation map indicates how parameters matched by holes in the redex should manifest in the reactum after matching. Where the instantiation map is unambiguous (e.g., it is the identity map), we may just write $R \to R'$.

Definition 6 (Reaction)**.** *Matching of a particular reaction rule (R,R',η) against a particular bigraph G and rewriting it into some other bigraph G' proceeds by decomposition of the bigraph into a* context *C, a* match *R (the redex), and a set of* parameters *d (for portions of the bigraph that are matched by holes in the redex). This decomposition is then reassembled with the reactum R' replacing the matched portion of G, with select parts of d substituted into the holes of R', forming the resulting bigraph G'.*

$$G = C \circ R.d \to C \circ R'.\eta(d) = G'$$

We require further that the context C be active*, that is, that every control above holes of C are active (see CCS example below).*

We have suppressed details of the handling of names here by using the notation "$R.d$"; we have also suppressed details in the phrase "with select parts of d" and not explained the use of the map η. We refer the reader to [21] or [19] for details. The present paper can be read without understanding these details, as reaction in our examples always take the form of the following special case:

$$a = C \circ R \circ d \to C \circ R' \circ d\ .$$

Definition 7 (Bigraphical Reactive System)**.** *We use the notation $BG(\Sigma,\mathscr{R})$ to denote a bigraphical reactive system with a* signature Σ *(the set of constituent controls), and a set of reaction rules $\mathscr{R}$. More formally, $BG(\Sigma,\mathscr{R})$ is an* spm category *[21] in which the objects are interfaces and the arrows are bigraphs (which we refer to as* agents *of $BG(\Sigma,\mathscr{R})$), equipped with a set of reaction rules $\mathscr{R}$.*

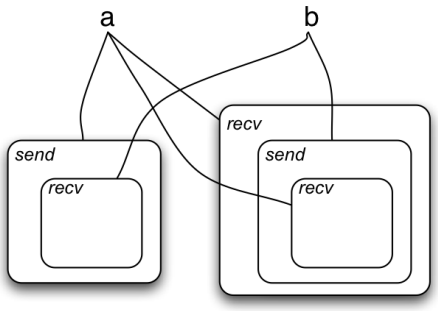


Figure 5: The process $\mathsf{send}(a).\mathsf{recv}(b).\mathsf{nil} \mid \mathsf{recv}(a).\mathsf{send}(b).\mathsf{recv}(a).\mathsf{nil}$

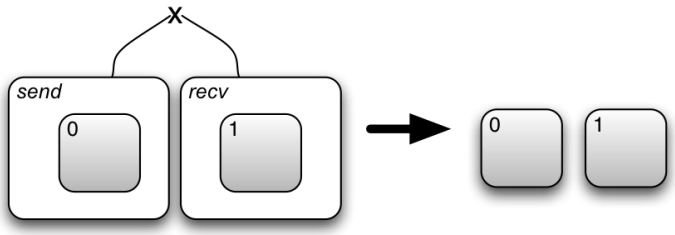


Figure 6: The R_{CCS} reaction rule

As an example, we introduce a very simple calculus in the style of the Calculus of Communicating Systems (CCS) [17], where we first give an encoding of the terms as bigraphs, and then define a reaction rule that imbues these terms with dynamic behaviour. Interested readers are referred to [21] for a real encoding of CCS.

Our calculus defines sequencing ($t.P$), parallel composition ($t \mid t$), and sending and receiving on a named channel ("x!" and "y?", respectively, where x and y are channel names). The encoding of these constructs into the bigraphical term language in Definition 5 is straightforward — these primitives are already defined in terms of the bigraphical term language, except for "send" and "receive" which we straightforwardly encode as nodes with controls send and recv, each with arity 1. Fig. 5 gives a graphical representation of the process $\mathsf{send}(a).\mathsf{recv}(b).\mathsf{nil} \mid \mathsf{recv}(a).\mathsf{send}(b).\mathsf{recv}(a).\mathsf{nil}$. According to our encoding, sequencing is represented by prefixing, parallel composition by juxtaposition, actions (such as send and recv) by *passive* controls, and channels by outer names. This is by no means the only encoding possible, but this technique is one of the most straightforward.

Having developed the encoding of our calculus within bigraphs, we can give a reaction rule R_{CCS} that will (through repeated rewriting) reduce the term as far as possible based upon agreement between parallel processes as to which action should be taken next:

$$R_{CCS} \stackrel{\text{def}}{=} \mathsf{recv}(x).-_0 \mid \mathsf{send}(x).-_1 \to -_0 \mid -_1$$

This rule is presented graphically in Fig. 6. It essentially "peels off" the outer layers of the terms where a send and a recv action are linked to the same channel name, rewriting the entire bigraph to the juxtaposition of whatever was nested inside those send and recv controls (i.e. the parts of the bigraph matched by the holes in the redex). As an example, the CCS reaction $a!.b? \mid a?.c! \to b? \mid c!$ becomes the

bigraphical reaction

$$\mathsf{send}(a).\mathsf{recv}(b).\mathsf{nil} \mid \mathsf{recv}(a).\mathsf{send}(c).\mathsf{nil} \to \mathsf{recv}(b).\mathsf{nil} \mid \mathsf{send}(c).\mathsf{nil}$$

3 Example

Aside from their role as a meta-calculus for the study of process modelling formalisms, bigraphical reactive systems are intended to provide a basis upon which to construct models of the kinds of context-aware and ubiquitous systems that are becoming increasingly popular. Consequently, we introduce an example based on modelling a context-aware social network notification system, such that a user is notified whenever a friend is in the same physical location.

We will give this example without using the link-graph part of bigraphs to keep it simple. We emphasise that the example generalises to a more interesting one in which connectivity counts — where notification is dependent not only on physical co-location but also on whether or not users and friends are virtually connected through their laptops and phones.

We will subsequently extend this to a system in which not all friends, but rather only particular designated "special friends", trigger notifications, and show that (and in what sense) the latter system is a refinement of the former.

The example system captures the dynamics of some physical environment (consisting of discrete zones within which we can detect the presence of a user by some mechanism that is outside the scope of this model) in which a user's friends move from zone to zone. When one of the user's friends is present in the same zone as the user, a notification is given, modelled by adding a "notification" node to the zone.

3.1 The abstract system: BRS_{notify}

We first define controls Z (Zone), U (User), F (Friend), N (Notification) and S (Special friend marker). Every control has arity 0 and every control is active; altogether we have a signature

$$\Sigma_N = \mathsf{Z}, \mathsf{U}, \mathsf{F}, \mathsf{N}$$

The bigraphs of our systems are thus arbitrary trees over these controls. We shall of course be interested only in those where Z are inner nodes and the remaining controls are leaves.

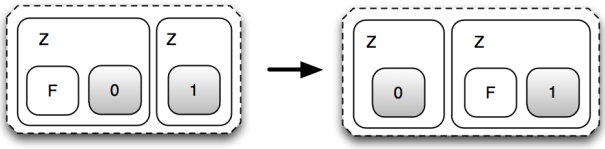
With these particular bigraphs in mind, we give reaction rules reconfiguring a bigraph by allowing nodes with control F — friends — to move between nested zones as follows. These rules are illustrated graphically in Fig. 7.

$$\begin{aligned}
M_1 &= \mathsf{Z}.(\mathsf{F} \mid -_0) \mid \mathsf{Z}.-_1 && \to && \mathsf{Z}.-_0 \mid \mathsf{Z}.(\mathsf{F} \mid -_1) \\
M_2 &= \mathsf{Z}.(\mathsf{Z}.(\mathsf{F} \mid -_0) \mid -_1) && \to && \mathsf{Z}.(\mathsf{Z}.-_0 \mid \mathsf{F} \mid -_1) \\
M_3 &= \mathsf{Z}.(\mathsf{Z}.-_0 \mid \mathsf{F} \mid -_1) && \to && \mathsf{Z}.(\mathsf{Z}.(\mathsf{F} \mid -_0) \mid -_1)
\end{aligned}$$

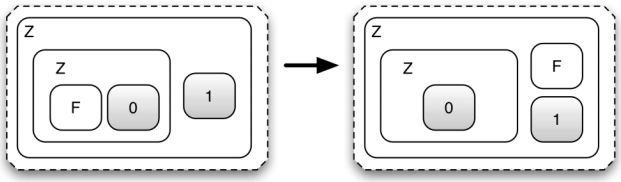
Reaction rules are here given on the form "$R \to R'$" rather than the more precise (R, R', η); recall from the above introduction to bigraphs that we use the former form whenever η is inconsequential (in this case, it is the identity map).

We extend the movement rules M with an additional rule R_1 for notifications to be issued when a U (user) and F (friend) node exist within the same zone. This reaction rule is illustrated in Fig. 8.

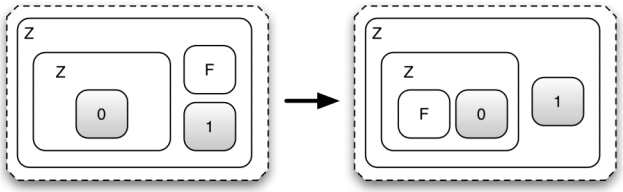
$$\begin{aligned}
&\Sigma_N = \Sigma_M \cup \{\mathsf{U}, \mathsf{N}\} \\
&R_1 = \mathsf{Z}.(\mathsf{U} \mid \mathsf{F} \mid -_0) \qquad \to \qquad \mathsf{Z}.(\mathsf{U} \mid \mathsf{F} \mid \mathsf{N} \mid -_0)
\end{aligned}$$



(a) M_1



(b) M_2



(c) M_3

Figure 7: Reaction rules M_1, M_2 and M_3 that allow $friend$ nodes to move between *zones*.

Let BRS_{notify} be the bigraphical reactive system formed by the addition of the reaction rule R_1 to the set of movement rules M:

$$BRS_{notify} = BG(\Sigma_N, M \cup \{R_1\})$$

3.2 The concrete system: $BRS_{selective}$

We now create a second bigraphical reactive system, this one refining (both intuitively and in a sense to be made precise) the system BRS_{notify} just introduced. In this new system, instead of simply notifying whenever *any* friend is present in the same zone as the user, we wish only to issue a notification in the

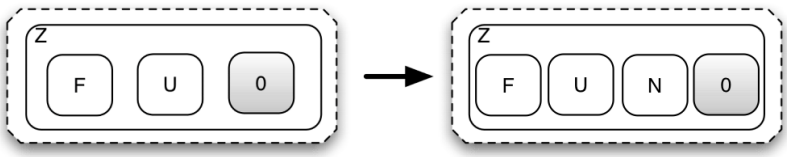


Figure 8: Reaction rule R_1

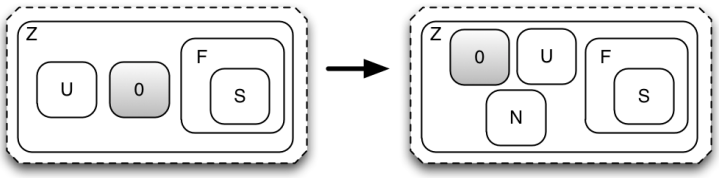


Figure 9: Reaction rule R_2

presence of a particular designated friend, distinguished by the presence of an S (special friend marker) inside the friend node in question. Consequently, we define the set of controls Σ_S for $BRS_{selective}$ to include (in addition to the controls of Σ_N) the S control. The modified reaction rule R_2 is presented graphically in Fig. 9.

$$\begin{aligned} \Sigma_S &= \Sigma_N \cup \{\mathsf{S}\} \\ R_2 &= \mathsf{Z}.(\mathsf{U} \mid \mathsf{F}.\mathsf{S} \mid -_0) \to \mathsf{Z}.(\mathsf{U} \mid \mathsf{F}.\mathsf{S} \mid \mathsf{N} \mid -_0) \\ BRS_{selective} &= BG(\Sigma_S, M \cup \{R_2\}) \end{aligned}$$

At an intuitive level, this BRS refines the one of the previous sub-section. In the following section, we shall define exactly in what sense this is the case.

4 Vertical BRS Refinement

We recall the distinction here between *horizontal* and *vertical* refinement. Vertical refinement is concerned with moving between differing levels of abstraction, or indeed completely independent modelling languages, whereas horizontal refinement instead aims to relate models specified at the same fundamental level of abstraction, and within the same modelling setting. When we refer to the refinement of a BRS, we mean a vertical refinement, indeed, this is the only meaningful interpretation, as a BRS is the category consisting of (infinitely) many actual agents of the same general shape. We will later return (briefly) to what it would mean for an *agent* to be refined, that is, to a horizontal refinement between two agents of the same BRS (each of which would be bigraphs, representing — for example — two CCS processes).

To summarise the distinction between horizontal and vertical refinement in the setting of BRSs: In the former case, we are talking about what we can observe of all such agents, whereas in the latter we are referring to what we can observe of the behaviour of a single agent. In the present section, we consider vertical refinement; we comment on horizontal refinement in the subsequent section.

4.1 Safe refinements

First, what observations can you make of bigraphical agents? While the notion of a trace is familiar within refinement literature, within bigraphical reactive systems it is unclear exactly what might correspond to an *action* within the usual definition of a trace. Consequently, we formulate a trace of a BRS such that each element of the trace is a bigraphical agent (i.e., a bigraph of that BRS). Therefore the notion of trace is not one of a system exhibiting behaviour in the form of some observable actions, rather, it is the entire state of the model as it changes over time such that every element of the trace is a bigraph, related to the next element of the trace by the application of some reaction rule. While this may seem

very crude at first glance, it is important to remember that the dynamic behaviour of a bigraph is derived from reaction rules and the structure in a perhaps more direct manner than in many other calculi. As such, it makes sense to consider the abstract specification to comprise, by itself, an entire observation — cf. the structure of agents of BRS_{notify} above.

If an observation is a complete agent of the abstract specification, what then is an observation of an agent of the concrete system? We leave that to the system constructor, merely insisting that the observations one makes of concrete implementation agents must somehow be a function of their structure. Thus, observations of concrete agents are given by a structure-preserving map from concrete agents to abstract ones. In the parlance of category theory, this is called a "functor", a functor that we shall in this instance call an *abstraction functor*.

Definition 8 (Trace, observation)**.** *For a given BRS A, a* trace *is a (possibly infinite) sequence of bigraphs (agents) $\langle a_1, a_2, \ldots \rangle$, such that for each a_i and a_{i+1} in the sequence there is a reaction $a_i \to a_{i+1}$. If $s = \langle s_1, \ldots, s_n \rangle$ and $t = \langle t_1, \ldots \rangle$ are traces and $s_n \to t_1$, we may form the* composite trace *$s;t = \langle s_1, \ldots, s_n, t_1, \ldots \rangle$. In this case we say that t is an* extension *of s. We write $Tr(A)$ for the set of all traces of a given BRS A. If $F : A \to A'$ is a functor and $\langle a_1, a_2, \ldots \rangle \in Tr(A)$ is a trace of A, we apply F pointwise to obtain a trace $F(t) = \langle F(a_1), F(a_2), \ldots \rangle$.*

Note that $Tr(x)$ is by definition prefix-closed; that is, for any trace $t \in Tr(x)$, every prefix t' of t is also in $Tr(x)$.

Of course, not just any functor will do: to have a refinement, the dynamic behaviour of the concrete implementation must be allowed by the dynamic behaviour the abstract specification allows on its agents, the observations. Altogether, our notion of refinement follows from the usual trace equality, however, because a BRS tends to permit too much observation, our bigraphical notion of refinement requires as a side condition that there exist an abstraction functor $F : C \to A$ such that for any trace $\langle c_0, c_1, \ldots \rangle$, F gives rise to a trace $\langle F(c_0), F(c_1), \ldots \rangle$. We present vertical refinement as the conjunction of two constituent definitions, separating the preservation of orthogonal safety and liveness properties through refinement.

Definition 9 (Safe Vertical Refinement)**.**

$$A \overset{\text{safe}}{\sqsubseteq}_F C \quad \overset{\text{def}}{=} \quad F(Tr(C)) \subseteq Tr(A)$$

This definition satisfies the "reduction of nondeterminism" role of refinement, in that it is always valid to simply pick one alternative and implement it in C when presented with nondeterministic choice in A.

Lemma 1. *Safe Vertical Refinement is transitive and reflexive for the identity functor.*

Proof. Reflexivity is trivial. Suppose $A \overset{\text{safe}}{\sqsubseteq}_F C$ and $C \overset{\text{safe}}{\sqsubseteq}_G D$. Then $FG(Tr(D)) \subseteq F(Tr(C)) \subseteq Tr(A)$. □

We proceed to illustrate safe refinement using the two BRSs above, then give a sufficient condition for an abstraction functor to yield a safe refinement.

Recall our claim that $BRS_{selective}$, which issues notifications upon co-location with "special friends" is a refinement of BRS_{notify}, which does so upon co-location with any friend. The latter employs an additional control S. This indicates that our abstraction functor must (at the very least) ensure that all nodes of control S must be hidden, renamed or removed so as to ensure that the codomain of F is BRS_{notify} (i.e. that F can transform any agent of $BRS_{selective}$ into a valid agent of BRS_{notify}).

By this reasoning, we arrive at an abstraction functor "pattern" that is likely applicable to many other BRSs. We call this the *hiding functor*. Its essential function is to simply hide, for a given signature Σ,

all nodes that have been assigned controls from some particular set of controls H. This includes joining any children of nodes that will be hidden to parents that will remain visible after the application of the hiding functor. For our example, the hiding set $H = \{\mathsf{S}\}$ (i.e. the designated "special" friend control).

Definition 10 (Hiding Functor)**.** *We define an abstraction functor $F_{\Sigma,H} : BG(\Sigma) \to BG(\Sigma \setminus H)$ for hiding, parametrised by Σ, the signature of the "implementation" BRS, and H, a set of controls to be hidden. On* objects*, this functor is the identity. On* arrows*, its action is $F_{\Sigma,H}((V,E,prnt,ctrl,link)) \stackrel{\text{def}}{=} (V',E,ctrl',prnt',link)$, where*

- $V' = \{v \in V | ctrl(v) \notin H\}$
- $ctrl' = ctrl \restriction V'$, *and*
- $prnt'(l) \stackrel{\text{def}}{=} \begin{cases} prnt(l) & \text{where } ctrl(prnt(l)) \notin H \\ prnt'(prnt(l)) & \text{otherwise} \end{cases}$

This "hiding functor" is an abstraction functor for our example system. Recalling the definition of a bigraphical agent (and therefore of an arrow in the category BRS_{notify} or $BRS_{selective}$) given in Definition 2, the purpose of this hiding functor is to exclude any nodes that have a control that is in the set of hidden controls H, exclude these controls from the control map $ctrl$, and recursively recreate the parent map $prnt$ such that any children of a node with a control in H is attached to its most immediate place-graph ancestor that is not marked with a control in H. We call the abstraction functor for our example notification system A_{friend}, which is defined as the hiding functor above, instantiated with $H = \{\mathsf{S}\}$.

While the hiding functor has the flavour of a forgetful functor — it dispenses with structure — it cannot reasonably be called so as it is not faithful. Many distinct configurations (e.g. special-friend controls) will map to the same bigraph. This is a technical distinction only; we use "hiding" in no special sense, except as a name for abstraction functors of this general shape.

It is easy to prove that with A_{friend} as abstraction functor, $BRS_{selective}$ is indeed a safe refinement of BRS_{notify}. However, instead of proving so directly, we shall instead provide a general theorem about abstraction functors: When they preserve reaction, and in particular, when they preserve just reaction *rules*, they give rise to safe refinement.

Theorem 1. *Let $F : C \to A$ be an abstraction functor. If F preserves reaction, that is, if $c \to c'$ implies $F(c) \to F(c')$, then $A \stackrel{\text{safe}}{\sqsubseteq}_F C$.*

Proof. Immediate from Definition 9 of safe refinement. □

From this theorem it becomes apparent that an abstraction functor may be any functor at all that obeys this property.

The terminology deceives, here: The guarantee that the concrete system has *no more* behaviour than the abstract one is in fact upheld by the abstraction functor *preserving* behaviour.

Of course, proving that a functor preserves reaction need not at all be easy. Fortunately, we can exploit the connection between static structure and dynamic behaviour of bigraphs: a functor which preserves the reaction *rules*, structurally, will also preserve (dynamic) reaction, and will thus be a safe refinement.

Theorem 2 (Safe Abstraction Functors)**.** *Let $A = BG(\Sigma,\mathscr{R})$ and $C = BG(\Sigma',\mathscr{R}')$ be BRSs. A functor $F : C \to A$ yields a safe vertical refinement $A \stackrel{\text{safe}}{\sqsubseteq}_F C$* if *it satisfies the following conditions.*

1. *It preserves and respects tensor.*
2. *It preserves active contexts.*

3. *It preserves reaction rules: For any reaction rule* $(R,R',\rho) \in \mathscr{R}'$ *(a) the* F*-image* $(F(R),F(R'),\rho)$ *is a rule in* $\mathscr{R}$*; and (b) for any parameter* d *of that rule,* $\overline{\rho}(F(d)) = F(\overline{\rho}(d))$.

Proof. Suppose $c_1,\ldots,c_n$ is a trace of C. It is sufficient to prove that for each $i < n$, there is a reaction $F(c_i) \to F(c_{i+1})$. We know that $c_i \to c_{i+1}$, so there is some reaction rule $(R,R',\rho) \in \mathscr{R}'$, context E of C, and some set of names Z s.t.

$$c_i = E \circ (R \otimes 1_Z) \circ d \qquad \to \qquad E \circ (R' \otimes 1_Z) \circ \overline{\rho}(d) = c_i'$$

Where $\overline{\rho}(d)$ is the instantiation of parameters (see [21] for details). But then, because $(F(R),F(R'),\rho)$ is a rule of $\mathscr{R}$, we compute and find $a_i = F(c_i) = F(E \circ (R \otimes 1_Z) \circ d) = F(E) \circ (F(R) \otimes 1_{F(\mathsf{Z})}) \circ F(d) \to F(E) \circ (F(R') \otimes 1_{F(\mathsf{Z})}) \circ \overline{\rho}(F(d)) = F(E) \circ (F(R') \otimes 1_{F(\mathsf{Z})}) \circ F(\overline{\rho}(d)) = F(E \circ (R' \otimes 1_Z) \circ \overline{\rho}(d)) = F(c_i') = a_i'$ □

We remark that the three conditions of this Theorem appear to be good candidates for a definition of a morphism of *parametric* reactive systems, as suggested in the forthcoming [7].

It is straightforward to verify that for our example BRSs, $BRS_{selective}$ and BRS_{notify}, the hiding functor does in fact satisfy the three conditions of this Theorem. Thus we have the following corollary:

Corollary 1. $BRS_{selective}$ *is a sound refinement of* BRS_{notify} *with respect to the abstraction functor* A_{friend}*, that is,*

$$BRS_{notify} \overset{\text{safe}}{\sqsubseteq}_{A_{friend}} BRS_{selective}$$

The $\overset{\text{safe}}{\sqsubseteq}$ relation captures safety properties of the system being refined (i.e. it does not permit a refined model any undesirable extra behaviour, provided that the abstraction functor does not hide any "undesirable" behaviour). However, it does not guarantee that the system does anything at all (i.e. an empty trace is a safe refinement of any system). To guarantee that some additional liveness properties are preserved by refinement, it is necessary to extend our definition.

4.2 Live refinements

In order to guarantee that a given concrete system actually exhibits any of the desirable behaviour of the abstract system that it refines, we must define a notion of liveness. Whereas in a process algebraic setting it might be possible to rely on the presence of a particular output (or all possible outputs) to define "desired" observable behaviour, within a bigraphical setting the lack of any primitive notions of "input" or "output" (it is up to the system designer to define what these concepts mean with respect to a particular model) means that it is necessary to explicitly choose such "desirable" behaviours.

In the absence of an intrinsic notion of desirable behaviour, we further parametrise our notion of liveness, already parametric in terms of the abstraction functor F, on the admissible traces. This parametrisation on the notion of admissibility is akin to those used in [13, 11].

Definition 11 (Live Vertical Refinement)**.** *Let* $F : C \to A$ *be an abstraction functor, let* $\mathbf{C} \subseteq Tr(C)$ *be the* admissible traces *for* C*, and let similarly* $\mathbf{A} \subseteq Tr(A)$*, the admissible traces of* A*. We then say that* $(C,\mathbf{C})$ *is a* live *refinement of* $(A,\mathbf{A})$ *iff for every trace* s *of* $Tr(C)$*, whenever* $F(s)$ *has an extension* t' *to an admissible trace* $F(s);t' \in \mathbf{A}$*, then there exists an extension* s' *of* s *to an admissible trace* $s;s' \in \mathbf{A}$ *with* $F(s') = F(t')$*. In this case we write:*

$$(A,\mathbf{A}) \overset{\text{live}}{\sqsubseteq}_F (C,\mathbf{C})\ .$$

If we wish to take the admissible traces $\mathbf{A}$ of the abstract system A as canonical, we can define $\mathbf{C}$ as those traces whose F-images are admissible.

Lemma 2. *Live Vertical Refinement is transitive.*

Proof. Suppose $(A,\mathbf{A}) \overset{\text{live}}{\sqsubseteq}_F (C,\mathbf{C})$ and $(C,\mathbf{C}) \overset{\text{live}}{\sqsubseteq}_G (D,\mathbf{D})$, and suppose $FG(t);u' \in \mathbf{A}$. Then u' has a pre-image s' with $G(t);s' \in \mathbf{C}$; but then s' has a pre-image t' with $t;t' \in \mathbf{D}$. □

Let us provide a suitable set of admissible traces for our running example, BRS_{notify}. For this BRS, the obvious notion of admissibility (think "successful") is when notification has occurred. So we define the set of admissible traces as simply those finite traces in which the user has been notified, that is, in which the final agent contains the notification control next to the user and his friend:

$$\mathbf{S}_{notified} \quad \overset{\text{def}}{=} \quad \{\langle a_1,\ldots,a_n\rangle \in Tr(BRS_{notify}) \mid \exists C.\ a_n = C \circ (\mathsf{U} \mid \mathsf{F} \mid \mathsf{N})\}$$

For $BRS_{selective}$, we transfer the notion of admissiblity:

$$\mathbf{S}_{selective} \quad \overset{\text{def}}{=} \quad \{t \in Tr(BRS_{notify}) \mid F(t) \in \mathbf{S}_{notified}\}$$

The selective system $BRS_{selective}$ under these notions of admissibility is in fact *not* a live refinement of the original one BRS_{notify}. One might think so: After all, one can extend a trace to admissibility simply by moving the special friend next to the user. Unfortunately, there need not be a special friend, and even if there were, the abstract system might extend to admissibility by moving a (non-special) friend next to the user. We will now show this in detail, thus proving of the following proposition:

Proposition 1. $(BRS_{notify},\mathbf{S}_{notify}) \overset{\text{live}}{\not\sqsubseteq}_{A_{friend}} (BRS_{selective},\mathbf{S}_{selective})$.

Proof. Consider an agent $\mathsf{Z}.(\mathsf{U} \mid \mathsf{F})$ of $BRS_{selective}$. Applying A_{friend} we find simply $A_{friend}(\mathsf{Z}.(\mathsf{U} \mid \mathsf{F})) = \mathsf{Z}.(\mathsf{U} \mid \mathsf{F})$, which succeeds after just one reaction

$$\mathsf{Z}.(\mathsf{U} \mid \mathsf{F}) \to \mathsf{Z}.(\mathsf{U} \mid \mathsf{F} \mid \mathsf{N})$$

by reaction rule R_1. Now, if we actually had a live refinement, we should be able to match this reaction in $BRS_{selective}$. A simple inspection of the rules however prove that this is not possible. □

This is, however, not a show-stopper, rather it is a welcome demonstration of the utility of such a vertical refinement mechanism. We could remedy this situation by introducing into $BRS_{selective}$ an additional reaction rule that spontaneously adds the designated friend marker S to any friend F. However, this seems to contradict the intuition of the model, so in this instance it is perhaps better to leave $BRS_{selective}$ unmodified and accept that there are (known) conditions under which this BRS cannot progress to a successful state.

Having defined our two separate (live and safe) refinement relations, we can complete the definition of safe and live vertical refinement:

Definition 12 (Safe and Live Vertical Refinement)**.**

$$(A,\mathbf{A}) \sqsubseteq_F (C,\mathbf{C}) \quad \overset{\text{def}}{=} \quad A \overset{\text{safe}}{\sqsubseteq}_F C \wedge (A,\mathbf{A}) \overset{\text{live}}{\sqsubseteq}_F (C,\mathbf{C})$$

5 Discussion & related work

Having introduced our notion of vertical BRS refinement and shown the conditions under which it is safe and live with respect to the chosen abstraction functor, we now discuss potential approaches to horizontal refinement and related work. As it happens, both topics take us to the general refinement of Reeves and Streader [23, 24].

General horizontal refinement recognises three components to refinement: *entities* E, i.e., the specifications and implementations being refined; *contexts* Ξ, which are the environment within which the entities interact; and a *user*, which defines the possible observations $O(-)$ that can be made of an entity within a particular context. Refinement is then the relation

$$A \sqsubseteq_{\Xi,O} C \stackrel{\text{def}}{=} \forall x \in \Xi . O([C]_x) \subseteq O([A]_x) \ ,$$

where Ξ is the set of contexts, O is a map assigning observations to entities in contexts, and $[-]_x$ inserts an entity into context x.

Interestingly, our proposed notion of bigraphical *vertical* refinement falls under the umbrella of general *horizontal* refinement. Entities would be BRSs (like BRS_{notify} and $BRS_{selective}$); contexts Ξ would be just the trivial context, which leaves the entity unchanged. Finally, the observation map O is in our case simply $Tr(-)$, the map that takes a BRS to the traces observable of it. We do not think this is a coincidence. It seems intuitive that horizontal refinement of an entire class of agents would correspond to vertical refinement.

What about general *vertical* refinement, then? The definition of vertical refinement within the general refinement framework [24] relies upon a notion of *layers*, representing a level of abstraction in terms of (E_L, Ξ_L, O_L), where E_L is a set of entities, Ξ_L is a set of contexts and O_L is an observation function. Vertical refinement is then defined in terms of a Galois-connection that interprets high-level entities as low-level ones and vice versa.

The analogy of this notion with our use of an abstraction functor $F : C \to A$ should be apparent: If we could find that functor F to be one of an adjoint pair, we would be in an analogous situation. Unfortunately, it remains unclear if such an adjunction would retain the intuition behind the Galois-connection of general vertical refinement: morphisms (i.e., bigraphs) do not measure approximation; they represent the agents under investigation. In particular, the hiding functors used for the example in the present paper do not appear to be part of adjoint pairs.

Leaving vertical refinement behind, what is then a good notion of horizontal refinement for bigraphs? Returning to general horizontal refinement, bigraphs actually do come with a notion of entity, context, and observation, namely agents (roughly, bigraphs with no holes/inner names), bigraph contexts (bigraphs with holes/inner names), and an LTS (given a BRS). We have in the present paper by-passed the LTS as the notion of observation, following the bigraphical connection by structure and dynamics to its extreme conclusion, using the structure of the abstract specification as the observations.

For horizontal refinement, this approach appears not sensible: We would after all be relating agents of the same BRS. Important examples (like CCS-process refinement) cannot be expressed within this particular approach, which should guide the development of other horizontal refinement strategies for bigraphical agents. One obvious choice seems now to be the LTS intrinsic to BRSs. We have yet to pursue this option; we caution that while BRS LTSs have been successful in recovering semantics of various process algebras and other models of concurrency, it has been less successful in providing useful semantics for pervasive systems, one of our key interests.

However, even leaving the question of suitable observations open, we would likely find a notion

inside general horizontal refinement by taking

$$a \sqsubseteq_O c \stackrel{\text{def}}{=} \forall x \in \Xi . O(x \circ c) \subseteq O(x \circ a) \ ,$$

where a and c are agents of some BRS B; Ξ is the set of contexts of that BRS, and O is some notion of the semantics of agents of B, perhaps traces of the LTS of B, or perhaps some other notion. Indeed, early indications are that this approach would be promising in recovering (for example) CCS process refinement, contingent upon an appropriate notion of observation.

5.1 Related Work

Restricting the set of controls admissible under a certain control, or requiring a control to be present is well-studied in bigraphs (e.g., [2, 21, 19, 22]). However, that study has invariably focused on ensuring that the bigraphical LTS theory is retained under such additional constraints, and are thus only superficially related to the present paper.

Goldsmith & Creese [9] explore an approach to refinement within bigraphs (and particularly within Spygraphs, a specialisation of bigraphs). They observe the ease with which one may derive an LTS for a BRS that is labeled exclusively by the trivial context *id* (equivalent to a τ action in a process algebraic setting). These kinds of contextual labels are not helpful for analysis, as they capture no behaviour. Similarly, the LTS semantics of bigraphs share the same intentionality inherent in the graphical presentation. While Goldsmith & Creese suggest (to good effect in a CSP setting) that it may be appropriate to perform hiding at a process-level before considering a transition into bigraphs, this would seem inappropriate for many modelling situations (e.g., those which have no convenient term or process representation). While the transformation on bigraphical reactive systems proposed by that work may give rise to a refinement that is appropriate for some situations, we aim instead in this present work to work directly within the structure of bigraphs so as to ensure generality. As bigraphs attempt to be both a modelling formalism and a general meta-calculus for existing process calculi, it seems appropropriate that the notion of refinement we introduce should be similarly general, in the hope that we may recover calculus-specific notions of refinement within this general setting.

6 Conclusion

We have presented a vertical refinement mechanism for bigraphical reactive systems that adds refinement to the toolbox of model builders working within a bigraphical setting. The addition of a sufficient condition for safe abstraction functors, and the accompanying observation that it is the *preservation* of behaviour with respect to reaction that guarantees that a refinement exhibits no undesirable behaviour, provides a firm foundation from which to explore the limits and utility of this kind of vertical refinement.

We have pointed out a clear connection to the existing work on generalising refinement across many modelling formalisms, and therefore it seems appropriate (given the application of BRSs as a *meta-calculus*) that our notion of vertical refinement is also in some sense general. We leave for future work the exploration of further mechanisms for horizontal refinement within a bigraphical setting, noting that such a notion would very likely fall within the model of general refinement, and thus likely generalise well to other modelling formalisms encoded within bigraphical reactive systems.

References

[1] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt & H. Niss (2006): *Bigraphical models of context-aware systems*. In: *Foundations of Software Science and Computation Structures*, Springer, pp. 187–201, doi:10.1007/11690634_13.

[2] L. Birkedal, S. Debois & T. Hildebrandt (2008): *On the construction of sorted reactive systems*. In: *CONCUR 2008*, Springer, pp. 218–232, doi:10.1007/978-3-540-85361-9_20.

[3] T. Bolusset & F. Oquendo (2002): *Formal refinement of software architectures based on rewriting logic*. In: *International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble*.

[4] M. Bundgaard & V. Sassone (2006): *Typed polyadic pi-calculus in bigraphs*. In: *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, ACM, pp. 1–12, doi:10.1145/1140335.1140336.

[5] T.C. Damgaard, V. Danos & J. Krivine (2008): *A Language for the Cell*. Technical Report TR-2008-116, IT University of Copenhagen.

[6] T.C. Damgaard & J. Krivine (2008): *A Generic Language for Biological Systems based on Bigraphs*. Technical Report TR-2008-115, IT University of Copenhagen.

[7] S. Debois: *Computation in the Informatic Jungle*. To appear. Draft available at `http://www.itu.dk/people/debois/pubs/computation.pdf`.

[8] E. Elsborg, T. Hildebrandt & D. Sangiorgi (2009): *Type Systems for Bigraphs*. In Christos Kaklamanis & Flemming Nielson, editors: *Proceedings of the 4th International Symposium on Trustworthy Global Computing (TGC 2008)*, *Lecture Notes in Computer Science* 5474, Springer-Verlag, pp. 126–140, doi:10.1007/978-3-642-00945-7_8.

[9] M. Goldsmith & S. Creese (2010): *Refinement-Friendly Bigraphs and Spygraphs*. In: *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, IEEE, pp. 203–207, doi:10.1109/SEFM.2010.25. Available at `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5637430`.

[10] D. Grohmann & M. Miculan (2007): *Reactive Systems over Directed Bigraphs*. In Luís Caires & Vasco Thudichum Vasconcelos, editors: *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)*, *Lecture Notes in Computer Science* 4703, Springer-Verlag, pp. 380–394, doi:10.1007/978-3-540-74407-8_26.

[11] M. Hennessy & C. Stirling (1985): *The power of the future perfect in program logics*. *Information and Control* , pp. 23–52.

[12] T. Hildebrandt, H. Niss & M. Olsen (2006): *Formalising Business Process Execution with Bigraphs and Reactive XML*. In Paolo Ciancarini & Herbert Wiklicky, editors: *Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, *Lecture Notes in Computer Science* 4038, Springer-Verlag, pp. 113–129, doi:10.1007/11767954_8.

[13] T.T. Hildebrandt (1999): *Categorical Models for Fairness and a Fully Abstract Presheaf Semantics of SCCS with Finite Delay*. In: *CTCS'99*, LNCS, doi:10.1016/S1571-0661(05)80311-1.

[14] O.H. Jensen (2006): *Mobile Processes in Bigraphs*. Available at `http://www.cl.cam.ac.uk/~rm135/Jensen-monograph.pdf`.

[15] J. Krivine, R. Milner & A. Troina (2008): *Stochastic bigraphs*. *Electronic Notes in Theoretical Computer Science* 218, pp. 73–96, doi:10.1016/j.entcs.2008.10.006.

[16] J. Leifer & R. Milner (2006): *Transition systems, link graphs and Petri nets*. *Journal of Mathematical Structures in Computer Science* 16(6), pp. 989–1047, doi:10.1017/S0960129506005664.

[17] R. Milner (1980): *A calculus of communicating systems*. Springer-Verlag.

[18] R. Milner (2005): *Axioms for Bigraphical Structure*. *Journal of Mathematical Structures in Computer Science* 15(6), pp. 1005–1032, doi:10.1017/S0960129505004809.

[19] R. Milner (2006): *Pure Bigraphs: Structure and Dynamics*. Information and Computation 204(1), pp. 60–122, doi:10.1016/j.ic.2005.07.003.

[20] R. Milner (2007): *Local Bigraphs and Confluence: Two Conjectures: (Extended Abstract)*. In Roberto Amadio & Iain Phillips, editors: *Proceedings of the 13th International Workshop on Expressiveness in Concurrency (EXPRESS 2006)*, *Electronic Notes in Theoretical Computer Science* 175, Elsevier, doi:10.1016/j.entcs.2006.07.035.

[21] R. Milner (2009): *The space and motion of communicating agents*. Cambridge University Press.

[22] S. Ó Conchúir (2009): *Kind Bigraphs*. In Anthony Seda, Menouer Boubekeur, Ted Hurley, Micheal Mac an Airchinnigh, Michel Schellekens & Glenn Strong, editors: *Proceedings of the Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2006)*, *Electronic Notes in Theoretical Computer Science* 225, Elsevier, pp. 361–377, doi:10.1016/j.entcs.2008.12.086.

[23] S. Reeves & D. Streader (2008): *General refinement, part one: interfaces, determinism and special refinement*. *Electronic Notes in Theoretical Computer Science* 214, pp. 277–307, doi:10.1016/j.entcs.2008.06.013.

[24] S. Reeves & D. Streader (2008): *General refinement, part two: flexible refinement*. *Electronic Notes in Theoretical Computer Science* 214, pp. 309–329, doi:10.1016/j.entcs.2008.06.014.

[25] M. Zhang, L. Shi, L. Zhu, Y. Wang, L. Feng & G. Pu (2008): *A Bigraphical Model of WSBPEL*. In: *Second Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'08)*, IEEE Computer Society, pp. 117–120.

Building a refinement checker for Z

John Derrick, Siobhán North and Anthony J.H. Simons
Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK.
J.Derrick@dcs.shef.ac.uk

In previous work we have described how refinements can be checked using a temporal logic based model-checker, and how we have built a model-checker for Z by providing a translation of Z into the SAL input language. In this paper we draw these two strands of work together and discuss how we have implemented refinement checking in our Z2SAL toolset.

The net effect of this work is that the SAL toolset can be used to check refinements between Z specifications supplied as input files written in the LaTeXmark-up. Two examples are used to illustrate the approach and compare it with a manual translation and refinement check.
Keywords: Z, refinement, model-checking, SAL.

1 Introduction

In this paper we discuss the development of tool support for refinement checking in Z. In doing so we draw on two strands of work - one on providing a translation of Z into the input language of the SAL tool-suite, and the other on using model checking to verify refinements in state-based languages.

The SAL [18] tool-suite is used in both strands, and is designed to support the analysis and verification of systems specified as state-transition systems. Its aim is to allow different verification tools to be combined, all working on an input language designed as a format into which programming and specification languages can be translated. The input language provides a range of features to support this aim, such as guarded commands, modules, definitions etc., and can, in fact, be used as a specification language in its own right. The tool-suite currently comprises a simulator and four model checkers [4] including those for LTL and CTL.

Our work on the first strand has resulted in a translation tool which converts Z specifications to a SAL module, which groups together a number of definitions including types, constants and modules for describing a state transition system. The declarations in a state schema in Z are translated into local variables in a SAL module, and any state predicates become appropriate invariants over the module and its transitions.

A SAL specification defines its behaviour by specifying transitions, thus it is natural to translate each Z operation into one branch of a guarded choice in the transitions of the SAL module. The predicate in the operation schema becomes a guard of the particular choice. The guard is followed by a list of assignments, one for each output and primed declaration in the operation schema. This methodology has been implemented in a tool-set, as described in [9, 8].

Our work on the second strand has derived a methodology for verifying a refinement using a model-checker by combining two specifications in a special way and verifying particular CTL properties for this combination. Specifically, [21, 22, 10] described how refinements in Z and other state-based languages could be verified by encoding downward and upward simulations as CTL theorems - the simulation conditions being the standard way to verify refinements in state-based languages such as Z, B etc.

The contribution we describe in this paper is to implement this methodology in our Z to SAL translation toolkit. This extension to the tool enables two Z specifications to be input in LaTeXformat, and for

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 37–52, doi:10.4204/EPTCS.55.3

a refinement check to be performed. Internally this is achieved by translating each specification from LaTeXto a single SAL specification upon which appropriate CTL theorems can be verified using the SAL CTL witness model-checker sal-wmc.

The purpose of this paper is to describe how this is done, using two examples as way of illustration. The structure of the paper is thus as follows. In Section 2 and Section 3 we provide background on refinement and the Z to SAL translation respectively. How specifications can be combined to enable a model checker to verify a refinement is described in Section 4, and this section also describes our implementation of this methodology. To illustrate the process we present a slightly more complicated example in Section 5 and we conclude in Section 6.

2 Refinement

Data refinement [5, 6] is a formal notion of development, based around the idea that a concrete specification can be substituted for an abstract one as long as its behaviour is consistent with that defined in the abstract specification.

Each language, method or notation has its own variants. In Z, refinement is defined so that the observable behaviour of a specification is preserved. This behaviour is in terms of the operations that are performed, and their input and output values. Values of the state variables are regarded as being internal, and thus refinement can be used to change the representation of the state of a system. Hence the term *data refinement*.

In a state-based setting such as provided by Z, data refinements are verified by defining a relation (called a *retrieve relation*) between the two specifications and verifying a set of *simulation conditions*. The retrieve relation shows how a state in one specification is represented in the other. For refinement to be complete, a relation, rather than simply a function, is required [6].

In general, there are two forms the simulation conditions take, depending on the interpretation given to an operation, specifically that given to the operation's guard or precondition [6]. The two interpretations are often called the *blocking* and *non-blocking* semantics. We consider the latter, i.e., the standard, approach in this paper.

For any interpretation, there are two simulation rules for refinement which are together complete, i.e., all possible refinements can be proved with a combination of the rules. The first rule, referred to as *downward* (or *forward*) *simulation* [6, 5], requires that

initialisation the initial states of the concrete specification are related to abstract initial states

applicability the concrete operations are enabled (at minimum) in states related to abstract states where the corresponding abstract operations are enabled, and

correctness the effect of each concrete operation is consistent with the requirements of the corresponding abstract operation.

We do not consider the alternative kind of simulation known as an *upward* simulation in this paper, although there is nothing to stop the the appropriate methodology being implemented in our tool suite.

Definition 1 *A Z specification with state schema CState, initial state schema CInit and operations* $COp_1 \ldots COp_n$ *is a downward simulation of a Z specification with state schema AState, initial state schema AInit and operations* $AOp_1 \ldots AOp_n$*, if there is a retrieve relation R such that the following hold for all* $i : 1..n$*.*

1. $\forall CState \bullet CInit \Rightarrow (\exists AState \bullet AInit \wedge R)$
2. $\forall AState;\ CState \bullet R \wedge preAOp_i \Rightarrow preCOp_i$
3. $\forall AState;\ CState;\ CState' \bullet R \wedge preAOp_i \wedge COp_i \Rightarrow (\exists AState' \bullet R' \wedge AOp_i)$

The use of a retrieve relation allows the state spaces of the abstract and concrete specifications to be different - the retrieve relation documents their relationship. The first condition ensures appropriate initial states are related, and the second that the concrete operations are defined whenever abstract ones are (modulo the retrieve relation). The third conditions ensures that the concrete operations have an effect that is consistent with the abstract, whilst also allowing non-determinism to be reduced.

As an example refinement, consider the following simple specification. It defines two operations that add and remove an input from a set s of some given type T.

$$[T] \qquad \mid max : \mathbb{N}$$

$$A = [s : \mathbb{P}\, T \mid \#s \leq max] \qquad AInit = [A' \mid s' = \varnothing]$$

AEnter

ΔA
$p? : T$

$\#s < max$
$p? \notin s$
$s' = s \cup \{p?\}$

ALeave

ΔA
$p? : T$

$p? \in s$
$s' = s \setminus \{p?\}$

A simple data refinement replaces the set s by an injective sequence l as follows (assuming the same T and max):

$$C = [l : iseq\ T \mid \#l \leq max] \qquad CInit = [C' \mid l' = \langle\rangle]$$

CEnter

ΔC
$p? : T$

$\#l < max$
$p? \notin \operatorname{ran} l$
$l' = l \frown \langle p? \rangle$

CLeave

ΔC
$p? : T$

$p? \in \operatorname{ran} l$
$l' = l \rhd (T \setminus \{p?\})$

It is easy to see that the second specification is a downward simulation of the first, using as retrieve relation the following:

$$R == [A;\ C \mid s = \operatorname{ran} l]$$

Our task is to build a tool that can automatically check this kind of refinement.

3 Z2SAL

The original idea of translating Z into SAL specifications was due to Smith and Wildman [20], however, our implementation has increasingly diverged from the original idea as optimization issues have been tackled. In [9, 8] we have described the basics of our implementation, which provides a bespoke parser and generator, written in Java, to translate from the LaTeX encoding of Z into the SAL input language.

A Z specification written in the state-plus-operations style is translated into a SAL finite state automaton, following a template-driven strategy with a number of associated heuristics. The Z-style of specification is preserved in this strategy, including postconditions that mix primed and unprimed variables arbitrarily, possibly asserting posterior states in non-constructive ways. We also preserve the Z mathematical toolkit's approach to the modelling of relations, functions and sequences as sets of tuples, permitting interchangeable views of functions, sequences and relations as sets.

A specification in the SAL input language consists of a collection of separate input files, known as *contexts*, in which all the declarations are placed. At least one *context* must contain the definition of a *module*, an automaton to be simulated or checked. In our translation strategy, we use a master *context* for the main Z specification and refer to other *context* files, which define the behaviour of data types from the mathematical toolkit. The master *context* consists of a prelude, declaring types and constants, followed by the main declaration of a SAL *module*, defining the finite state automata, which implements the behaviour of the Z state and operation schemas. The states of the SAL translation are created by aggregating the variables from the Z state schema, and the transitions are created by turning the operation schemas into *guarded commands*, triggered by preconditions on input and local (state) variables, and asserting postconditions on local and output variables.

The implementation of this basic strategy is presented in [8], here we recap on its salient points on two examples. Consider the first specification above. Upon translation the specification becomes a context, here called *a*.

The *built-in* types of Z are translated into finite subranges in SAL, according to a scheme described in [8]. For example, $\mathbb{N}$ is translated to:

```
NAT : TYPE = [0..4];
```

The *basic types* of Z are converted into finite, enumerated sets in SAL, consisting of three symbolic ground elements by default (but sometimes with an extra *bottom* element to deal with partiality of functions etc.). For example, the given type *T* is translated to:

```
T : TYPE = {T__1, T__2, T__3};
```

Where the Z specification expresses predicates involving the cardinality of sets, the translator generates a bespoke counting-context for sets containing up to the maximum number of symbolic ground elements generated for the set, as described in [8]. For this example, a `count3` context is generated; the instantiation for counting up to three elements of type `T` is named:

```
T__counter : CONTEXT = count3 {T; T__1, T__2, T__3};
```

The bounding constant *max* is an uninterpreted constant in Z, which we translate in SAL as a local variable, which can in principle take any value in the `NAT` type's range. This leads to some simulation states where the limits of the system's behaviour are reached quickly (e.g. if *max* = 0), but other states in which all three elements may be added to the set `s`.

State and initialisation schemas. The state variables from the Z state schema are translated into the *local* variables of the SAL *module*, which together constitute the aggregate states of the automaton. The state predicate is treated as follows: we define a corresponding `DEFINITION` clause to represent the

schema invariant. This is achieved by introducing an extra *local* boolean variable, called `invariant__`, and declaring a formula for this in the *definition* sub-clause.

The Z initialization schema is translated in a non-constructive style into a guarded command in the `INITIALIZATION` clause of the SAL module, with the invariant as part of the guard. Thus, for the above example, we get the following translation.

```
State : MODULE =
BEGIN
 LOCAL max : NAT
 LOCAL s : set {T;} ! Set
 INPUT p? : T
 LOCAL invariant__ : BOOLEAN
 DEFINITION
   invariant__ = (T__counter ! size?(s) <= max)
 INITIALIZATION [
     s = set {T;} ! empty AND
     invariant__
   -->
 ]
```

The challenge of the translation strategy is to deal efficiently with the large vocabulary of mathematical data types such as sets, products, relations, functions, sequences and bags. The translation tool has to represent these efficiently in SAL, whilst preserving the expressiveness and flexibility of the Z language.

The basic approach is to define one or more context files for each data type in the toolkit. For example, the set mathematical data type in Z is translated into a SAL context, which models the set as a boolean-valued membership predicate on elements (following Bryant's optimal encoding of sets for translation into BDDs, [2, 3]). All other set operations are based on this encoding:

```
set {T : TYPE; } : CONTEXT = BEGIN
Set : TYPE = [T -> BOOLEAN];
empty : Set = LAMBDA (elem : T) : FALSE;
...
contains? (set : Set, elem : T) : BOOLEAN =
 set(elem);
...
union(setA : Set, setB : Set) : Set =
 LAMBDA (elem : T) : setA(elem) OR setB(elem);
...
END
```

Similar contexts are defined for the function, relation and sequence data types. Whereas Z sets and relations are modelled as boolean maps, Z functions and sequences are modelled using SAL's total functions. We adopt a totalising strategy, introducing bottom elements for types that participate in the domain or range of functions, or range of sequences.

Translating the Z operation schemas. Each operation schema in Z contributes in two ways to the SAL translation. Firstly, an operation schema may optionally declare input, or output variables (or both), which are extracted and declared in the prelude of the *module* clause, as SAL *input* and *output* variables. Secondly, the predicate of each operation schema is converted into a *guarded command* in the *transition* sub-clause, the last sub-clause in the *module* clause.

The input and output variables are understood to exist in the local scope of each operation schema, which has consequences in the translation. The SAL translation eventually substitutes the suffix '_''_' for '!' in the output variables, since the latter is reserved.

The computation performed by each operation schema is expressed as a *guarded command* in the *transition* sub-clause. The name of the schema is used for the transition label, which aids readability. The *guarded command* has the general syntactic form: `label : guard --> assignments`.

The guards for each transition include the primed `invariant__'` as one of the conjuncts, which asserts the state predicate in the posterior state of every transition. This, combined with the assertion of the unprimed `invariant__` in the initial state, ensures that the state predicate holds universally.

Finally, a catch-all `ELSE` branch is added to the guarded commands, to ensure that the transition relation is total (for soundness of the model checking). In practice, this allows model-checking to complete, even if the simulation blocks at a given point. Admitting the `ELSE`-transition allows simulations to pass through states in which the `invariant__'` fails to hold. Normally, this does not matter, since we can also ensure that `LOCAL` state variables are not modified, whenever the `ELSE`-transition is taken.

However, a new soundness problem emerged when admitting *bottom* values, as part of a totalising strategy for partial types. Our previous practice was to assert that `INPUT` variables never took *bottom* values, as part of the invariant. However, a loophole was discovered that allowed the system to pass through states in which the invariant did not hold (due to selecting bottom values for inputs) and then recover in the following cycle, in which the invariant held once more, but undefined values had been accepted as inputs from the previous cycle. Ideally, we would have liked to rule out invalid inputs in the `ELSE`-transition, but the SAL tools do not permit this.

Instead, we now assert both the primed `invariant__'` and unprimed `invariant__` in the guard to each transition, so closing the loophole. In practice, simulations can still pass through states where the invariant fails to hold, but they are then forced to pass through `ELSE`-transitions repeatedly, until some valid input is selected. The new translation is once again sound, but simulations may have more latent cycles. Thus for the transition component of our example we have the following:

```
TRANSITION [
  AEnter :
      T__counter ! size?(s) < max AND
      NOT set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! insert(s, p?) AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE}
  []
  ALeave :
      set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! remove(s, p?) AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE}
  []
  ELSE -->  s' = s
```

]

A similar translation is produced for *C*, this time producing a SAL input file using contexts defined to model Z sequences; see Appendix A.

4 Model-checking a refinement

A series of approaches to model-checking a refinement is described in [21, 22, 10] by Smith and Derrick with varying degrees of sophistication. They all work by taking two specifications, *A* and *C* say, and building a combined system *M* which encodes the behaviour of both in such a way that it is possible to write CTL properties to check the various aspects that are needed for simulation conditions to hold. There are variations to this approach as follows.

1. Three different combinations are formed, M_{init}, M_{app}, M_{corr}, one for each of the three downward simulation conditions (and a similar methodology for upward simulations);
2. One combination is formed, *M*, encoding all three properties to be checked in one system.

These two approaches need the candidate retrieve relation to be passed to the tool, thus a final approach is

- Additionally have the model-checker search to find if such a retrieve relation exists.

For efficiency reasons (and here to aid readability) we describe our implementation of the first approach, again restricting ourselves for brevity to downward simulations. Thus in the approach we describe, which is an abbreviated discussion of [22], here three systems are formed and if all three checks are satisfied then the concrete system is indeed a downward simulation of the abstract system with the chosen retrieve relation.

To illustrate the approach, we use the example specified above, noting that although for readability we describe it as a combination of Z schemas, in our implementation the combination acts at the level of combining SAL modules. We will combine the two specifications into one system so that we can encode the simulation conditions on the combined system, thus the combined specification includes all the abstract and concrete variables. The methodology assumes the state variables of the abstract and concrete systems are disjoint (as in fact they are in our example), but if not, then renaming is applied first to achieve it.

Initialisation. The simulation condition on initial states requires that for each concrete initial state, we are able to find an abstract initial state related by the retrieve relation *R*. To encode this condition we initialise M_{init} so that the concrete part of the state is initialised. Hence in our example, the combined system's state and initialisation are as follows:

M_{init}

$s : \mathbb{P}\, T$

$l : iseq\ T$

$\#s \leq max$

$\#l \leq max$

$Init_{init}$

M'_{init}

$l' = \langle\rangle$

To check whether an abstract initial state exists that is related to any particuar concrete initial state, we use just one operation (normally called $InitA_{init}$) which changes the abstract part of the state to an initial value and leaves the concrete part unchanged. In our example this operation is then:

$$[\Delta M_{init} \mid s' = \varnothing \land l' = l]$$

For any non-trivial specification $InitA_{init}$ is total, thus we do not need the "catch-all" ELSE branch in the SAL model-checker which is needed for non-total systems as described above. Then, with a system with one operation the required initialisation condition holds if the operation can be performed such that the resulting abstract and concrete parts of the state are related by R. That is, we require that there exists a next state such that $s = \operatorname{ran} l$, i.e.:

$$\mathbf{EX}\ (s = \operatorname{ran} l)$$

Applicability. Applicability conditions in refinements check the consistency of the operations' preconditions. To encode this as a temporal formula we introduce a variable *ev* to the combined state to denote the name of the last operation that occurred, and, as in [22], we use a different font for the values of type *ev*. Since we will need an additional operation to ensure totality, the combined state for an applicability check in our example will be the following:

M_{app}

$s : \mathbb{P} T$
$l : iseq\ T$
$ev : \{\mathsf{AEnter}, \mathsf{CEnter}, \mathsf{ALeave}, \mathsf{CLeave}, \mathsf{Choose}\}$

$\#s \leq max$
$\#l \leq max$

The applicability condition requires that if abstract and concrete states are related by the retrieve relation, then the concrete operation must be applicable whenever the abstract one was. For the sake of efficiency we initialise to states which are already related by the retrieve relation, that is, here of the form[1]:

$$Init_{app} = [M'_{app} \mid s' = \operatorname{ran} l']$$

Operations are then specified, one for each abstract or concrete operation, each shadowing the behaviour of the original operation, and only specifying the values of that operation (eg $AEnter_{app}$ defines values for variables that originate from the abstract specification). In addition, we introduce a *Choose* operation.

$AEnter_{app}$

ΔM_{app}
$p? : T$

$\#s < max$
$p? \notin s$
$s' = s \cup \{p?\}$
$ev' = \mathsf{AEnter}$

$ALeave_{app}$

ΔM_{app}
$p? : T$

$p? \in s$
$s' = s \setminus \{p?\}$
$ev' = \mathsf{ALeave}$

[1] The value of *ev* can be left underspecified.

$CEnter_{app}$

ΔM_{app}
$p? : T$

$\#l < max$
$p? \notin \operatorname{ran} l$
$l' = l \frown \langle p? \rangle$
$ev' = \mathsf{CEnter}$

$CLeave_{app}$

ΔM_{app}
$p? : T$

$p? \in \operatorname{ran} l$
$l' = l \upharpoonright (T \setminus \{p?\})$
$ev' = \mathsf{CLeave}$

$Choose_{app} \mathrel{\widehat=} [\Delta M_{app} \mid ev' = \mathsf{Choose}]$

The applicability check can now be written in CTL as follows.

$$(\mathbf{EX}\ (ev = \mathsf{AEnter}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{CEnter})) \wedge (\mathbf{EX}\ (ev = \mathsf{ALeave}) \Rightarrow \mathbf{EX}\ (ev = \mathsf{CLeave}))$$

Correctness. A similar methodology is applied to check the correctness condition, and here we use the same combined state and initialisation as used for applicability, as well as the same totalisation *Choose*:

$M_{corr} \mathrel{\widehat=} M_{app}$
$Init_{corr} \mathrel{\widehat=} Init_{app}$
$Choose_{corr} \mathrel{\widehat=} Choose_{app}$

The downward simulation correctness condition requires that any after-state of a concrete operation is related by the retrieve relation to an after-state of the abstract operation. To encode this correctly one needs to ensure that each operation in the combined state does not alter variables from the portion of state it is not representing. Thus we have operations of the form:

$AOp_{corr} \mathrel{\widehat=} [AOp_{app} \mid l' = l]$
$COp_{corr} \mathrel{\widehat=} [COp_{app} \mid s' = s]$

This allows us to perform the operations COp_{corr} and AOp_{corr} in sequence so that the abstract part of the final state reached is identical to that which could have been reached by performing only AOp_{corr}, and the concrete part is identical to that which could have been reached by performing only COp_{corr}. The correctness condition is then:

$$\begin{array}{l}\mathbf{EX}\ (ev = \mathsf{AEnter}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{CEnter} \Rightarrow \mathbf{EX}\ (ev = \mathsf{AEnter} \wedge R)) \\ \quad \wedge \\ \mathbf{EX}\ (ev = \mathsf{ALeave}) \Rightarrow \mathbf{AX}\ (ev = \mathsf{CLeave} \Rightarrow \mathbf{EX}\ (ev = \mathsf{ALeave} \wedge R))\end{array}$$

Implementation in SAL. The above is described in terms of combinations of Z specifications, although, of course, it is implemented in terms of combining SAL modules in our tool-suite.

The process of combining the two LaTeX Z specifications plus retrieve relation into a single SAL specification in order to check the downward simulation conditions was achieved using an extension to our Z to SAL parser. When translating a single Z specification to SAL our compiler first parses the Z, then transforms it into an internal SAL representation and finally the SAL file is generated. In extending the tool-set to combine two specifications in the manner described above the major modification was to the

middle phase, the transformation from Z to SAL. Nevertheless the process of parsing two specifications sequentially required some modification for a number of issues.

For example, declarations in the abstract and concrete state schemas need to be checked to ensure that they contain distinct identifiers, but where types and constants occur in both specifications they have to be identical to cope with SAL's strict type checking. Neither of these problems caused much difficulty since, e.g., there already was a mechanism to ensure that a variable name used in two different Z operations did not lead to a conflict in the SAL translations (where all variables had the same scope). In our simple, single specification, translation this is achieved by prefixing the variable name by the name of its transition wherever an ambiguous name is detected and the same mechanism was used when producing a single combined specification. The only modification was that variables from axiomatic definitions were prefixed by the specification name rather than the transition name.

Treating types declared in two different specifications as the same was slightly more complicated as types from the abstract specification occurring in the concrete had to be identified. In our single translation types are canonical, for reasons explained in [8] and this had to be maintained in the combined translation without the parser rejecting a concrete specification which contains an apparently second declaration of a type which has been declared in the abstract specification. This problem also occurred with identical constants in both specifications.

Having parsed the two specifications, the retrieve relation is read in and parsed as a single Z operation with everything from both the abstract and concrete specifications in scope.

The process of transforming a single Z specification into SAL consists of fixing the finite ranges of all the types, eliminating redundant predicates, giving initial values to all the constants and identifying any named types that would have to be generated in SAL. In transforming two specifications into one SAL specification the finite ranges were fixed to the widest required by either specification but apart from that the process is essentially simple. The two sets of initial declarations were combined and the two lists of operation schemas in Z became a single list of transitions in SAL. The resulting structure is that of our internal representation of any SAL specification and a SAL text file could be generated from it in the standard way.

The result produced by our tool-kit of the two SAL modules for the correctness condition is given in Appendix B. It is then a trivial matter to check the required theorem on it.

5 A further example

A further example, which provides a comparative analysis with the manual approach to refinement checking, is given by the following (now standard) example.

The Marlowe box office allows customers to book tickets in advance using the *Book* operation – *mpool* is the set of tickets, and if a ticket is available ($mpool \neq \varnothing$) then one is allocated then and there. When the customer arrives, operation *Arrive* presents this ticket. *Ticket* is the set of all tickets, and a free type adds a possibly null ticket, and *tkt* models which tickets have been allocated.

$$[Ticket] \qquad MTicket ::= null \mid ticket\langle\langle Ticket \rangle\rangle$$

Marlowe

$mpool : \mathbb{P}\, Ticket$

$tkt : MTicket$

MInit

Marlowe

$tkt = null$

MBook

$\Delta Marlowe$

$tkt = null$
$mpool \neq \varnothing$
$tkt' \neq null$
$ticket^{-1}(tkt') \in mpool$
$mpool' = mpool \setminus \{ticket^{-1}(tkt')\}$

MArrive

$\Delta Marlowe$
$t! : Ticket$

$tkt \neq null$
$tkt' = null$
$t! = ticket^{-1}(tkt)$
$mpool' = mpool$

In an alternative description - the Kurbel - customers still book tickets in advance. However, now if there is an available ticket then this is simply recorded by the operation *Book* provided the customer has not already booked. Only when the customer actually arrives at the box office, is the ticket allocated by *Arrive*. *kpool* is the pool of tickets and *bkd* denotes whether a ticket has been booked.

$Booked ::= yes \mid no \qquad [Ticket]$

Kurbel

$kpool : \mathbb{P}\, Ticket$
$bkd : Booked$

KInit

$Kurbel$

$bkd = no$

KBook

$\Delta Kurbel$

$bkd = no$
$kpool \neq \varnothing$
$bkd' = yes$
$kpool' = kpool$

KArrive

$\Delta Kurbel$
$t! : Ticket$

$bkd = yes$
$kpool \neq \varnothing$
$bkd' = no$
$t! \in kpool$
$kpool' = kpool \setminus \{t!\}$

The Marlowe specification is a downward simulation of the Kurbel (and in fact Kurbel is an upward simulation of Marlowe). The retrieve relation linking the two that one is tempted to write down is the following:

R

$Marlowe$
$Kurbel$

$bkd = no \Rightarrow tkt = null \wedge kpool = mpool$
$bkd = yes \Rightarrow tkt \neq null \wedge kpool = (mpool \cup \{ticket^{-1}(tkt)\})$

In [22] a hand translation of these specifications into SAL was performed, followed by a merging into a single SAL specification - also performed by hand. A natural question to ask therefore is to what extent our automatic translation and combination is comparable with the manual process. The above candidate retrieve relation was used in the manual process, which revealed a failure to pass the necessary

refinement conditions - both specification and retrieve relation needing adjustment before the Marlowe was shown to be a valid downward simulation of the Kurbel.

It is interesting to note that the results of the automatic translation were broadly comparable to the manual one, and in fact due to our optimizations show slight reduction in state space size (see table below). The automatic combination essentially identical to the manual. The latter is to be expected - the combination is essentially simple once the specifications have been converted into SAL.

Step	Manual	Auto
0	1344	840
1	3360	6072
2	8544	6072
3	8544	6072
4	8544	6072

6 Conclusion

This work contributes on one hand to the strand of work on providing tool support for Z, and on the other hand to automatic refinement checking.

Recent work on providing tool support for Z includes the CZT (Community Z Tools) project [16], our own work [9], as well as related work such as ProZ [19], which adapts the ProB [15] tool for the Z notation.

Work on automatic refinement checking includes that of Bolton who has used Alloy to verify data refinements in Z [1]. There have also been a number of encoding of subsets of Z-based languages in the CSP model checker FDR [11, 17, 14], which checks that refinement holds between two specifications by comparing the failures/divergences semantics of the specifications; and simulation-based refinement can be encoded as a failures/divergences check [7, 13, 12].

Clearly there is much to be done in terms of further work here, not least some performance characterisations of when such an approach produces feasible state spaces.

Acknowledgements: This work was done as part of collaborative work with Graeme Smith and Luke Wildman of the University of Queensland. Tim Miller also gave valuable advice on the current CZT tools.

References

[1] C. Bolton (2005): *Using the Alloy Analyzer to Verify Data Refinement in Z. Electronic Notes in Theoretical Computer Science* 137(2), pp. 23–44, doi:10.1016/j.entcs.2005.04.023.

[2] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers* 35(8), pp. 677–691.

[3] Randal E. Bryant (1992): *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. ACM Comput. Surv.* 24(3), pp. 293–318.

[4] E. Clarke, O. Grumberg & D. Peled (2000): *Model Checking*. MIT Press.

[5] W.-P. de Roever & K. Engelhardt (1998): *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, doi:10.1017/CBO9780511663079.

[6] J. Derrick & E. Boiten (2001): *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, doi:10.1007/978-1-4471-0257-1.

[7] J. Derrick & E.A. Boiten (2003): *Relational Concurrent Refinement*. *Formal Aspects of Computing* 15(2-3), pp. 182–214, doi:10.1007/s00165-003-0007-4.

[8] J. Derrick, S. North & A.J. H. Simons (2008): *Z2SAL - Building a Model Checker for Z*. In E. Börger, M.J. Butler, J.P. Bowen & P. Boca, editors: *ABZ, Lecture Notes in Computer Science* 5238, Springer, pp. 280–293, doi:10.1007/978-3-540-87603-8.

[9] J. Derrick, S. North & T. Simons (2006): *Issues in Implementing a Model Checker for Z*. In Z. Liu & J. He, editors: *ICFEM, Lecture Notes in Computer Science* 4260, Springer, pp. 678–696, doi:10.1007/11901433-37.

[10] John Derrick & Graeme Smith (2008): *Using Model Checking to Automatically Find Retrieve Relations*. *Electr. Notes Theor. Comput. Sci.* 201, pp. 155–175, doi:10.1016/j.entcs.2008.02.019.

[11] C. Fischer & H. Wehrheim (1999): *Model-checking CSP-OZ specifications with FDR*. In K. Araki, A. Galloway & K. Taguchi, editors: *International Conference on Integrated Formal Methods (IFM'99)*, Springer-Verlag, pp. 315–334.

[12] J. He (1989): *Process refinement*. In J. McDermid, editor: *The Theory and Practice of Refinement*, Butterworths.

[13] M. Josephs (1988): *A state-based approach to communicating processes*. *Distributed Computing* 3, pp. 9–18.

[14] G. Kassel & G. Smith (2001): *Model Checking Object-Z Classes: Some Experiments with FDR*. In: *Asia-Pacific Software Engineering Conference (APSEC 2001)*, IEEE Computer Society Press.

[15] M. Leuschel & M. Butler (2005): *Automatic Refinement Checking for B*. In K. Lau & R. Banach, editors: *International Conference on Formal Engineering Methods, ICFEM 2005, LNCS* 3785, Springer-Verlag, pp. 345–359.

[16] Tim Miller, Leo Freitas, Petra Malik & Mark Utting (2005): *CZT Support for Z Extensions*. In Judi Romijn, Graeme Smith & Jaco Pol, editors: *Integrated Formal Methods, IFM 2005, LNCS* 3771, Springer-Verlag, pp. 227–245.

[17] A. Mota & A. Sampaio (2001): *Model-checking CSP-Z: strategy, tool support and industrial application*. *Science of Computer Programming* 40, pp. 59–96.

[18] L. de Moura, S. Owre & N. Shankar (2003): *The SAL language manual*. Technical Report SRI-CSL-01-02 (Rev.2), SRI International.

[19] Daniel Plagge & Michael Leuschel (2007): *Validating Z Specifications using the ProB Animator and Model Checker*. *LNCS* 4591, pp. 480–500.

[20] G. Smith & L. Wildman (2005): *Model checking Z specifications using SAL*. In H. Treharne, S. King, M. Henson & S. Schneider, editors: *International Conference of Z and B Users, LNCS* 3455, Springer-Verlag, pp. 87–105.

[21] Graeme Smith & John Derrick (2005): *Model Checking Downward Simulations*. *Electr. Notes Theor. Comput. Sci.* 137(2), pp. 205–224, doi:10.1016/j.entcs.2005.04.032.

[22] Graeme Smith & John Derrick (2006): *Verifying data refinements using a model checker*. *Formal Asp. Comput.* 18(3), pp. 264–287, doi:10.1007/s00165-006-0002-7.

Appendix A

Here is the SAL translation of the concrete specification from Section 2

```
c : CONTEXT = BEGIN
NAT : TYPE = [0..4];
T : TYPE = {T__1, T__2, T__3, T__B};
```

```
State : MODULE =
BEGIN
 LOCAL max : NAT
 LOCAL l : sequence {T; T__B, 3} ! Sequence
 INPUT p? : T
 LOCAL invariant__ : BOOLEAN
 DEFINITION
   invariant__ = (sequence {T; T__B, 3} ! injective?(l) AND
     sequence {T; T__B, 3} ! valid?(l) AND
     p? /= T__B AND
     sequence {T; T__B, 3} ! size?(l) <= max)
 INITIALIZATION [
     l = sequence {T; T__B, 3} ! empty AND invariant__
   -->
 ]
 TRANSITION [
   CEnter :
       sequence {T; T__B, 3} ! size?(l) < max AND
       NOT set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
       l' = sequence {T; T__B, 3} ! append(l, p?) AND
       invariant__ AND
       invariant__'
     -->
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE}
   []
   CLeave :
       set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
       l' = sequence {T; T__B, 3} ! remove(l,p?) AND
       invariant__ AND
       invariant__'
     -->
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE}
   []
   ELSE -->    l' = l
 ]
END;
END
```

Appendix B

The result of automatically combining the two SAL modules from Z specifications given in Section 2:

```
r2corr : CONTEXT = BEGIN

NAT : TYPE = [0..5];
T : TYPE = {T__1, T__2, T__3, T__B};
EVENT__ : TYPE = {AEnter, ALeave, CEnter, CLeave, Choose__};
T__counter : CONTEXT = count4 {T; T__1, T__2, T__3, T__B};

State : MODULE =
BEGIN
 LOCAL max : NAT
```

```
LOCAL max : NAT
LOCAL s : set {T;} ! Set
INPUT p? : T
LOCAL l : sequence {T; T__B, 3} ! Sequence
LOCAL ev__ : EVENT__
LOCAL invariant__ : BOOLEAN
DEFINITION
  invariant__ =
    (T__counter ! size?(s) <= max AND
    sequence {T; T__B, 3} ! injective?(l) AND
    p? /= T__B AND
    sequence {T; T__B, 3} ! valid?(l) AND
    sequence {T; T__B, 3} ! size?(l) <= max)
INITIALIZATION [
    (s = sequence {T; T__B, 3} ! range(l))
  -->
]
TRANSITION [
  AEnter :
      T__counter ! size?(s) < max AND
      NOT set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! insert(s, p?) AND
      ev__' = AEnter AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
  []
  ALeave :
      set {T;} ! contains?(s, p?) AND
      s' = set {T;} ! remove(s, p?) AND
      ev__' = ALeave AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
  []
  CEnter :
      sequence {T; T__B, 3} ! size?(l) < max AND
      NOT set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
      l' = sequence {T; T__B, 3} ! append(l, p?) AND
      ev__' = CEnter AND
      invariant__ AND
      invariant__'
    -->
      s' IN {x : set {T;} ! Set | TRUE};
      l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
      ev__' IN {x : EVENT__ | TRUE}
```

```
   []
   CLeave :
       set {T;} ! contains?(sequence {T; T__B, 3} ! range(l), p?) AND
       l' = sequence {T; T__B, 3} ! remove(l,p?) AND
       ev__' = CLeave AND
       invariant__ AND
       invariant__'
     -->
       s' IN {x : set {T;} ! Set | TRUE};
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
       ev__' IN {x : EVENT__ | TRUE}
   []
   Choose__ :
       ev__' = Choose__ AND
       invariant__ AND
       invariant__'
     -->
       s' IN {x : set {T;} ! Set | TRUE};
       l' IN {x : sequence {T; T__B, 3} ! Sequence | TRUE};
       ev__' IN {x : EVENT__ | TRUE}
 ]
END;
END
```

Refinement by interpretation in π-institutions

César J. Rodrigues
Dep. Informatics & CCTC,
Minho University, Portugal
cjr@di.uminho.pt

Manuel A. Martins
Dep. Mathematics,
Aveiro University, Portugal
martins@ua.pt

Alexandre Madeira
Dep. Informatics & CCTC,
Minho University, Portugal
Dep. Mathematics,
Aveiro University, Portugal
Critical Software S.A., Portugal
madeira@ua.pt

Luís S. Barbosa
Dep. Informatics & CCTC,
Minho University, Portugal
lsb@di.uminho.pt

The paper discusses the role of interpretations, understood as multifunctions that preserve and reflect logical consequence, as refinement witnesses in the general setting of π-institutions. This leads to a smooth generalization of the "refinement by interpretation" approach, recently introduced by the authors in more specific contexts. As a second, yet related contribution a basis is provided to build up a refinement calculus of structured specifications in and across arbitrary π-institutions.

1 Introduction

The expression *refinement by interpretation* was coined in [MMB09b] to refer to an alternative approach to refinement of equational specifications in which signature morphisms are replaced by *logical interpretations* as refinement witnesses.

Intuitively, an interpretation is a logic translation which preserves and reflects meaning. Actually, it is a central tool in the study of equivalent algebraic semantics (see, *e.g.*, [Wój88, BP89, BP01, BR03, Cze01]), a paradigmatic example being the interpretation of the *classical propositional calculus* into the *equational theory of boolean algebras* (cf. [BP01, Example 4.1.2]). Interestingly enough, and in the more operational setting of formal software development, the notion of interpretation proved effective to capture a number of transformations difficult to deal with in classical terms. Examples include data encapsulation and the decomposition of operations into atomic transactions [MMB09b].

A typical refinement pattern that is not easily captured by the classical approach concerns refinement of a subset of operations into operations defined over more specialized sorts. This kind of transformation induces the loss of the functional property on the operations' component of signature morphisms. For example, there is not a signature morphism σ to guide a refinement where a specification with operations $g : s' \to s$ and $f : s' \to s$ is transformed into one with operations $g : s' \to s_{new}$ and $f : s' \to s$, since this translation naturally induces a map $\sigma_{sort}(s) = \{s, s_{new}\}$ which violates the definition of signature morphism.

The approach seems also promising in the context of new, emerging computing paradigms which entail the need for more flexible approaches to what is taken as a valid transformation of specifications, as in, for example, [BSR04]. Later, in [MMB09a], the whole framework was generalized from the original equational setting to address deductive systems of arbitrary dimension. This made possible, for example, to refine sentential into equational specifications and the latter into modal ones. Moreover, the restriction to logics with finite consequence relations was dropped which resulted in increased flexibility

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 53–64, doi:10.4204/EPTCS.55.4

along the software development process. The interested reader is referred to both papers for a number of illustrative examples.

On the other hand, the notion of an institution [GB92], proposed by J. Goguen and R. Burstall in the late 1970s, has proven very successful in formalizing logical systems and their interrelations.

This paper aims at lifting the use of logic interpretations to witness refinement of specifications at an institutional level. This is made in the context of π-institutions [FS88] which deal directly with syntactic consequence relations rather than with semantical satisfaction, as in the original definition of an institution [GB92]. π-institutions are particularly useful in formalizing deductive systems with varying signatures, which are only indirectly handled by the methods of abstract algebraic logic, as in [BP01] on which our first generalization [MMB09a] is based. In general, π-institutions provide a more operational framework with no loss of expressiveness as any classical institution can be suitably translated.

Refinement by interpretation is proposed here at two different levels: a *macro* level relating different π-institutions, and the *micro* level of specifications inside a particular, although arbitrary, π-institution. The former discusses what is an interpretation of institutions and provides the envisaged generalization of this approach to refinement of arbitrary deductive systems. The latter, on the other hand, corresponds to a sort of *local* refinement witnessed by interpretations thought simply as multifunctions relating sentences generated by different signatures within the same institution.

As a second, although related, contribution, the paper lays the basis for a refinement-by-interpretation calculus of structured specifications in an arbitrary (and across) π-institution(s). That both levels can be addressed and related to each other comes to no surprise: a main outcome of institution theory is precisely to provide what [AN94] describes as *effective mechanisms to manipulate theories in an analogous way as our deductive calculi manipulate formulas*.

The remainder of this paper is organized as follows. π-institutions and a notion of interpretation between them are reviewed in section 2. Then, section 3 characterizes refinement by interpretation in this context, whereas the local view is discussed in section 4. The structure of a refinement calculus is discussed in section 5. Section 6 concludes and highlights some pointers to related work.

2 π-institutions and interpretations

In broad terms, an institution consists of an arbitrary category *Sign* of signatures together with two functors SEN and MOD that give, respectively, for each signature, a set of sentences and a category of models. For each signature, sentences and models are related via a satisfaction relation whose main axiom formalizes the popular aphorism *truth is invariant under change of notation* [Dia08]. Such a very generic way to capture a logical system was originally motivated by quite pragmatic concerns: to provide an abstract, language-independent framework for specificifying and reasoning about software systems, in response to the explosion of specification logics. Several current specification formalisms, notably, CafeOBJ [DF02], CASL [MHST03] and HETS [MML07] were designed to take advantage of such a general framework.

π-institutions, proposed by J. Fiadeiro and A. Sernadas in [FS88], fulfill a similar role, replacing semantical satisfaction by a syntactic consequence relation *à la* Tarski. Therefore, a π-institution introduces, for each signature, a closure operator on the set of its sentences capturing logical consequence. As remarked by G. Voutsadakis in [Vou03] π-institutions *may be viewed as the natural generalization of the notion of a deductive system on which a categorical theory of algebraizability, generalizing the theory of [BP01] may be based.* In the sequel we review the basic definition and adopt Voutsadakis's notion of interpretation to define refinement by interpretation in such a general setting.

Definition 1 *A π-institution I is a tuple $\langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ where*

- *Sign is a category of signatures and signature morphisms;*
- $\mathsf{SEN} : Sign \to Set$ *is a functor from the category of signatures to the category of small sets giving, for each $\Sigma \in |Sign|$, the set $\mathsf{SEN}(\Sigma)$ of Σ-sentences and mapping each $f : \Sigma_1 \to \Sigma_2$ to a* substitution *$\mathsf{SEN}(f) : \mathsf{SEN}(\Sigma_1) \to \mathsf{SEN}(\Sigma_2)$;*
- *for each $\Sigma \in |Sign|$, $C_\Sigma : \mathscr{P}(\mathsf{SEN}(\Sigma)) \to \mathscr{P}(\mathsf{SEN}(\Sigma))$ is a mapping, called* Σ-closure, *such that, for all $A, B \subseteq \mathsf{SEN}(\Sigma)$ and $\Sigma_1, \Sigma_2 \in Sign$;*
  - **(a)** $A \subseteq C_\Sigma(A)$
  - **(b)** $C_\Sigma(C_\Sigma(A)) = C_\Sigma(A)$
  - **(c)** $C_\Sigma(A) \subseteq C_\Sigma(B)$ *for* $A \subseteq B$
  - **(d)** $\mathsf{SEN}(f)(C_{\Sigma_1}(A)) \subseteq C_{\Sigma_2}(\mathsf{SEN}(f)(A))$

Note that the Σ-closure operator of a π-institution is not required to be finitary.

Definition 2 *A π-institution $I' = \langle Sign', \mathsf{SEN}', (C'_\Sigma)_{\Sigma\in|Sign'|}\rangle$ is a* sub-π-institution *of $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ if $Sign'$ is a sub-category of Sign and, for each $\Sigma \in |Sign'|$, $\mathsf{SEN}'(\Sigma) \subseteq \mathsf{SEN}(\Sigma)$ and the Σ-closure C'_Σ is the restriction of C_Σ.*

Roughly speaking, the notion of logical interpretation underlying [MMB09a] is that of [BP89]: a multifunction (i.e., a set-valued function) relating formulas which preserves and reflects logical consequence. Note that the expressive flexibility of interpretations comes precisely from their definition as multifunctions. A corresponding definition, to be used in the sequel, was proposed, in the context of π-institutions, in [Vou03]:

Definition 3 *Given two π-institutions $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ and $I' = \langle Sign', \mathsf{SEN}', (C'_\Sigma)_{\Sigma\in|Sign'|}\rangle$, a* translation *$\langle \mathsf{F}, \alpha\rangle : I \to I'$ consists of a functor $\mathsf{F} : Sign \to Sign'$ together with a natural transformation $\alpha : \mathsf{SEN} \to \mathscr{P}\,\mathsf{SEN}'\mathsf{F}$.*

A translation $\langle \mathsf{F}, \alpha\rangle : I \to I'$ is a semi-interpretation *if, for all $\Sigma \in |Sign|$, $\Phi \cup \{\phi\} \subseteq \mathsf{SEN}(\Sigma)$,*

$$\phi \in C_\Sigma(\Phi) \quad \Rightarrow \quad \alpha_\Sigma(\phi) \subseteq C'_{\mathsf{F}(\Sigma)}(\alpha_\Sigma(\Phi)) \tag{1}$$

It is an interpretation *if,*

$$\phi \in C_\Sigma(\Phi) \quad \Leftrightarrow \quad \alpha_\Sigma(\phi) \subseteq C'_{\mathsf{F}(\Sigma)}(\alpha_\Sigma(\Phi)) \tag{2}$$

Finally, we say that a translation $\langle \mathsf{F}, \alpha\rangle$ interprets a π-institution I, if there is a π-institution $I^0 = \langle Sign^0, \mathsf{SEN}^0, (C^0_\Sigma)_{\Sigma\in|Sign^0|}\rangle$ for which $\langle \mathsf{F}, \alpha\rangle$ is an interpretation.

Note that a translation depends only on the categories of signatures and the sentence functors involved, but not on the family of closure operators. A translation is a *self-translation* if F is the identity functor Id. On the other hand, it is said to be a *functional translation* if, for every $\Sigma \in |Sign|$, $\phi \in \mathsf{SEN}(\Sigma)$, $|\alpha_\Sigma(\phi)| = 1$. Additionally, it is an *identity translation,* if for every $\Sigma \in |Sign|$, $\phi \in \mathsf{SEN}(\Sigma)$,

$$\alpha_\Sigma(\phi) = \{\phi\} \tag{3}$$

3 Refining π-institutions by interpretation

In software development the process of *stepwise refinement* [ST88b] encompasses a chain of successive transformations of a specification

$$S_0 \rightsquigarrow S_1 \rightsquigarrow S_2 \rightsquigarrow \cdots \rightsquigarrow S_{n-1} \rightsquigarrow S_n$$

through which a complex design is produced by incrementally adding details and reducing under-specification. This is done step-by-step until the class of models becomes restricted to such an extent that a program can be easily manufactured. The discussion on what counts for a valid refinement step, represented by $S_i \rightsquigarrow S_j$, is precisely the starting point of this line of research.

The minimal requirement to be placed on a refinement relation, besides being a pre-order to allow stepwise construction, is preservation of logical consequence. In the framework of π-institutions this corresponds to the following definition:

Definition 4 (Syntactic refinement) *Let $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ and $I' = \langle Sign', \mathsf{SEN}', (C'_\Sigma)_{\Sigma\in|Sign'|}\rangle$ be two π-institutions. I' is a syntactic refinement of I if $Sign$ is a sub-category of $Sign'$ and, for each $\Sigma \in |Sign|$, $\mathsf{SEN}(\Sigma) \subseteq \mathsf{SEN}'(\Sigma)$ and $C_\Sigma(\Phi) \subseteq C'_\Sigma(\Phi)$ for $\Phi \subseteq \mathsf{SEN}'(\Sigma)$.*

Clearly, a π-institution is a syntactic refinement of any of its π-sub-institutions. Refinement by interpretation, on the other hand, goes a step further:

Definition 5 (Refinement by interpretation) *Consider two π-institutions $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ and $I' = \langle Sign', \mathsf{SEN}', (C'_\Sigma)_{\Sigma\in|Sign'|}\rangle$ and let $\langle \mathsf{F}, \alpha\rangle : I \longrightarrow I'$ be a translation. I' is a* refinement by interpretation *of I via $\langle \mathsf{F}, \alpha\rangle$, written as $I \rightsquigarrow_{\langle \mathsf{F},\alpha\rangle} I'$, if*

- *there is a π-institution $I^0 = \langle Sign', \mathsf{SEN}', (C^0_\Sigma)_{\Sigma\in Sign'}\rangle$ that interprets I under translation $\langle \mathsf{F}, \alpha\rangle$;*
- *for all $\Sigma \in |Sign|$, $\Phi \subseteq \mathsf{SEN}(\Sigma)$,*

$$\phi \in C_\Sigma(\Phi) \;\Rightarrow\; \alpha_\Sigma(\phi) \subseteq C'_{\mathsf{F}(\Sigma)}(\alpha_\Sigma(\Phi))$$

Clearly, a syntactic refinement is a refinement by interpretation for a self, identity, functional interpretation, with $\mathsf{F} = Id$. The following Lemma establishes an useful characterization of refinement via interpretation:

Lemma 1 *Let $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma\in|Sign|}\rangle$ and $I' = \langle Sign', \mathsf{SEN}', (C_\Sigma)_{\Sigma\in|Sign'|}\rangle$ be two π-institutions and $\langle \mathsf{F}, \alpha\rangle : I \longrightarrow I'$ a translation. Then, $I \rightsquigarrow_{\langle \mathsf{F},\alpha\rangle} I'$ if I' is a syntactic refinement of some interpretation of I through $\langle \mathsf{F}, \alpha\rangle$.*

Proof. Suppose I' is a syntactic refinement of an arbitrary interpretation I^0 of I along $\langle \mathsf{F}, \alpha\rangle$. Clearly the first condition in the definition of refinement by interpretation is met. For the second, let $\Sigma \in Sign$ and $\Phi \cup \{\phi\} \subseteq SEN(\Sigma)$. Assume $\phi \in C_\Sigma(\Phi)$. Then

$$\alpha_\Sigma(\phi) \subseteq C^0_{F(\Sigma)}(\alpha_\Sigma(\Phi))$$

because $\langle \mathsf{F}, \alpha\rangle$ is an interpretation. On the other hand, I' being a syntactic refinement of I^0,

$$C^0_{F(\Sigma)}(\alpha_\Sigma(\Phi)) \subseteq C'_{F(\Sigma)}(\alpha_\Sigma(\Phi))$$

Thus, $\alpha_\Sigma(\phi) \subseteq C'_{F(\Sigma)}(\alpha_\Sigma(\Phi))$.
□

Definition 5 subsumes the corresponding notion introduced in [MMB09a] for k-dimensional deductive systems, because every k-dimensional deductive system $\langle \mathscr{L}, \vdash_{\mathscr{L}}\rangle$ over a countable set of variables V, gives rise to a specific π-institution $I_{\mathscr{L}} = \langle Sign_{\mathscr{L}}, Sen_{\mathscr{L}}, (C_{\mathscr{L}_\Sigma})_{\Sigma\in|Sign_{\mathscr{L}}|}\rangle$, built in [Vou02] as follows:

(i) $Sign_{\mathscr{L}}$ is the one-object category with object V. The identity morphism is the inclusion $i_V : V \to Fm_{\mathscr{L}}(V)$, where $Fm_{\mathscr{L}}(V)$ denotes the set of formulas constructed by recursion using variables in V and connectives in $\mathscr{L}$ in the usual way. Composition $g \cdot f$ is defined by $g \cdot f = g^\star f$, where $g^\star : Fm_{\mathscr{L}}(V) \to Fm_{\mathscr{L}}(V)$ denotes the substitution uniquely extending g to $Fm_{\mathscr{L}}(V)$.

(ii) $\mathsf{SEN}_{\mathscr{L}} : Sign_{\mathscr{S}} \to Set$ maps V to $Fm^k_{\mathscr{L}}(V)$ and $f : V \to V$ to $Fm_{\mathscr{L}}(V)$ $(f^\star)^k : Fm^k_{\mathscr{L}}(V) \to Fm^k_{\mathscr{L}}(V)$. It is easy to see that $\mathsf{SEN}_{\mathscr{S}}$ is indeed a functor.

(iii) Finally, $C_{\mathscr{L}}$ is the standard closure operator $C_V : \mathscr{P}(Fm_{\mathscr{L}}(V)) \to \mathscr{P}(Fm_{\mathscr{L}}(V))$ associated with $\langle \mathscr{L}, \vdash_{\mathscr{L}} \rangle$, i.e., $C_V(\Phi) = \{\phi \in Fm^k_{\mathscr{L}}(V) : \Phi \vdash_{\mathscr{S}} \phi\}$ for all $\Phi \subseteq Fm^k_{\mathscr{L}}(V)$.

Example 1 *The π-institution of modal logic $S5^G$ forms a (syntactic) refinement of the one for classical propositional calculus (CPC). Actually, consider the modal signature $\Sigma = \{\to, \wedge, \vee, \neg, \top, \bot, \Box\}$. Modal logic K is defined as an extension of CPC by adding the axiom $\Box(p \to q) \to (\Box p \to \Box q)$ and the inference rule $\frac{p}{\Box p}$. Logic $S5^G$, on the other hand, enriches the signature of K with the symbol $\Diamond$, and K itself with the axioms $\Box p \to p$, $\Box p \to \Box\Box p$ and $\Diamond p \to \Box\Diamond p$, cf. [BP01]. Hence, since the signature of both systems contains the signature of CPC and their presentations extend that of CPC with extra axioms and inference rules, we have CPC $\rightsquigarrow$ K and CPC $\rightsquigarrow$ $S5^G$ (actually, CPC $\rightsquigarrow$ K $\rightsquigarrow$ $S5^G$). Hence, through these refinements, one may capture more complex, modally expressed requirements introduced along the refinement process.*

Given an interpretation $\tau : Fm_{\mathscr{L}}(V) \longrightarrow \mathscr{P}(Fm_{\mathscr{L}'}(V'))$ between two deductive systems $\langle \mathscr{L}, \vdash_{\mathscr{L}} \rangle$ and $\langle \mathscr{L}', \vdash_{\mathscr{L}'} \rangle$, let us define $\langle \mathsf{F}_\tau, \tau \rangle$ as the translation between π-institutions $I_{\mathscr{L}}$ and $I_{\mathscr{L}'}$, where F_τ is a functor between single object categories, mapping, at the object level, V to V'. As expected,

Lemma 2 *An l-deductive system $\langle \mathscr{L}', \vdash_{\mathscr{L}'} \rangle$ is an interpretation of a k-deductive system $\langle \mathscr{L}, \vdash_{\mathscr{L}} \rangle$ through an interpretation τ, iff $\langle F_\tau, \tau \rangle$ interprets the π-institution $I_{\mathscr{L}}$ in $I_{\mathscr{L}'}$.*

Proof. Assume $\langle \mathscr{L}, \vdash_{\mathscr{L}} \rangle$ (respectively, $\langle \mathscr{L}', \vdash_{\mathscr{L}'} \rangle$) are defined over a countable set of variables V (respectively, V'). Being an interpretation between deductive systems, τ is a multifunction $\tau : Fm_{\mathscr{L}}(V) \longrightarrow \mathscr{P}(Fm_{\mathscr{L}'}(V'))$ such that, for all $\Gamma \cup \{\phi\} \subseteq Fm_{\mathscr{L}}(V)$,

$$\Gamma \vdash_{\mathscr{L}} \phi \;\; \Leftrightarrow \;\; \tau(\Gamma) \vdash_{\mathscr{L}'} \tau(\phi) \tag{4}$$

According to the construction of $I_{\mathscr{L}}$, detailed above, this is equivalent to

$$\phi \in C_V(\Gamma) \;\; \Leftrightarrow \;\; \tau(\phi) \subseteq C_{V'}(\tau(\Gamma)) \tag{5}$$

□

Hence, it is immediate to check that

Corollary 1 *An l-deductive system $\langle \mathscr{L}', \vdash_{\mathscr{L}'} \rangle$ is a refinement of a k-deductive system $\langle \mathscr{L}, \vdash_{\mathscr{L}} \rangle$ through an interpretation τ, iff the π-institution $I_{\mathscr{L}'}$ is a refinement of $I_{\mathscr{L}}$ through $\langle F_\tau, \tau \rangle$.*

As a final remark, note that, in a very precise sense, Definition 5 also covers the case of classical institutions. Actually, a π-institution corresponding to a classical one can always be defined: for each signature Σ and set of formulas Ψ, take $C_\Sigma(\Psi)$ as the set of sentences satisfied in all models validating Ψ.

4 The local view

Having discussed refinement by interpretation of π-institutions, we address now the same sort of refinement applied to specifications inside an arbitrary π-institution. Such is the *local* view. Given an arbitrary π-institution $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma \in |Sign|} \rangle$, a basic, or *flat* specification is defined as

$$SP \;=\; \langle \Sigma, \Phi \rangle$$

where $\Sigma \in |Sign|$ and $\Phi \subseteq \mathsf{SEN}(\Sigma)$. Its meaning is the closure of Φ, i.e., $C_\Sigma(\Phi)$. D. Sannella and A. Tarlecki in [ST88a] define specification over an arbitrary institution along similar lines, but taking, as semantic domain, classes of models instead of logical consequence relations.

As expected, any morphism $\sigma : \Sigma \longrightarrow \Sigma'$ in *Sign* entails a notion of *local* refinement $\rightsquigarrow_\sigma$ in I given by

$$\langle \Sigma, \Phi \rangle \rightsquigarrow_\sigma \langle \Sigma', \Phi' \rangle \;\text{ if }\; \sigma(\Phi) \subseteq C_{\Sigma'}(\Phi') \tag{6}$$

For σ an inclusion, this may be regarded as a form of syntactic refinement.

Specifications may also be connected by interpretations which, again, correspond to multifunctions preserving and reflecting consequence. Formally,

Definition 6 *Let $\langle \Sigma, \Phi \rangle$ and $\langle \Sigma', \Phi' \rangle$ be two specifications over a π-institution $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma \in |Sign|} \rangle$ and $i : \mathsf{SEN}(\Sigma) \longrightarrow \mathscr{P}(\mathsf{SEN}(\Sigma'))$ a multifunction from $\mathsf{SEN}(\Sigma)$ to $\mathsf{SEN}(\Sigma')$. Then i is a* (local) semi-interpretation *of $\langle \Sigma, \Phi \rangle$ in $\langle \Sigma', \Phi' \rangle$ if, for all $\phi \in \mathsf{SEN}(\Sigma)$,*

$$\phi \in C_\Sigma(\Phi) \;\Rightarrow\; i(\phi) \subseteq C_{\Sigma'}(\Phi') \tag{7}$$

It is a (local) interpretation *of $\langle \Sigma, \Phi \rangle$ in $\langle \Sigma', \Phi' \rangle$ if,*

$$\phi \in C_\Sigma(\Phi) \;\Leftrightarrow\; i(\phi) \subseteq C_{\Sigma'}(\Phi') \tag{8}$$

Finally, we say that i (locally) interprets *$\langle \Sigma, \Phi \rangle$, if there is a specification $\langle \Sigma^0, \Phi^0 \rangle$ on which $\langle \Sigma, \Phi \rangle$ is interpreted by i.*

Adopting expression "*ϕ is true in* specification $\langle \Sigma, \Phi \rangle$" to abbreviate the fact that $\phi \in C_\Sigma(\Phi)$, definition (8) can be read as *ϕ is true in $\langle \Sigma, \Phi \rangle$ iff $i(\phi)$ is true in $\langle \Sigma', \Phi' \rangle$*.

Definition 7 *Let $SP = \langle \Sigma, \Phi \rangle$ be a specification and $i : \mathsf{SEN}(\Sigma) \longrightarrow \mathscr{P}(\mathsf{SEN}(\Sigma'))$ a translation which interprets SP. A specification $SP' = \langle \Sigma', \Phi' \rangle$ refines SP via local interpretation i, written as $SP \rightsquigarrow_i SP'$, if for all $\phi \in \mathsf{SEN}(\Sigma)$,*

$$\phi \in C_\Sigma(\Phi) \;\Rightarrow\; i(\phi) \subseteq C_{\Sigma'}(\Phi') \tag{9}$$

Given a $\sigma : \Sigma \to \Sigma' \in Sign$, $\mathsf{SEN}(\sigma) : \mathsf{SEN}(\Sigma) \to \mathsf{SEN}(\Sigma')$ induces a translation that maps each $\phi \in \mathsf{SEN}(\Sigma)$ into $\{\mathsf{SEN}(\sigma)(\phi)\}$. In the sequel we identify this translation simply with $\mathsf{SEN}(\sigma)$.

Definition 8 *A signature morphism $\sigma : \Sigma \to \Sigma' \in Sign$ is* conservative *if for any $\Phi \subseteq \mathsf{SEN}(\Sigma)$, $\mathsf{SEN}(\sigma)$ interprets $\langle \Sigma, \Phi \rangle$ in $SP^\sigma = \langle \Sigma', \mathsf{SEN}(\sigma)(\Phi) \rangle$.*

Observe that $\mathsf{SEN}(\sigma)$ is always a semi-interpretation from SP to SP^σ. Moreover, note that conservativeness is a stronger notion than that of interpretability.

Theorem 1 *Let $\sigma : \Sigma \to \Sigma' \in Sign$ be a conservative signature morphism, $SP = \langle \Sigma, \Phi \rangle$ a specification over I and $\Phi' \in \mathsf{SEN}(\Sigma')$. Then,*

$$\mathsf{SEN}(\sigma)(\Phi) \subseteq C_{\Sigma'}(\Phi') \text{ implies that } SP \rightsquigarrow_{\mathsf{SEN}(\sigma)} \langle \Sigma', \Phi' \rangle \tag{10}$$

In practice, new specifications are built from old through application of a number of specification constructors. As a minimum set we consider operators to join two specifications, to translate one into another, and to derive one from another going backward along a signature morphism. The following definition characterizes along these lines a notion of structured specification in an arbitrary π-institution.

Definition 9 *Structured specifications over an arbitrary π-institution $I = \langle Sign, \mathsf{SEN}, (C_\Sigma)_{\Sigma \in |Sign|} \rangle$ are defined inductively as follows, taking flat specifications as the base case.*

- *For a signature Σ, the union of specifications $SP_1 = \langle \Sigma, \Phi_1 \rangle$ and $SP_2 = \langle \Sigma, \Phi_2 \rangle$ is defined as*

$$\mathsf{union}(SP_1, SP_2) \;=\; \langle \Sigma, \Phi_1 \cup \Phi_2 \rangle$$

- *The translation of specification $SP = \langle \Sigma, \Phi \rangle$ through a morphism $\sigma : \Sigma \to \Sigma'$ in Sign is defined as*

$$\mathsf{translate}\ SP\ \mathsf{through}\ \sigma \;=\; \langle \Sigma', \mathsf{SEN}(\sigma)(\Phi) \rangle$$

- *The derivation of a Σ specification from $SP' = \langle \Sigma', \Phi' \rangle$ through a morphism $\sigma : \Sigma \to \Sigma'$ in Sign is defined as*

$$\mathsf{derive}\ SP'\ \mathsf{through}\ \sigma \;=\; \langle \Sigma, \Psi \rangle$$

where $\Psi = \{\psi |\ \mathsf{SEN}(\sigma)(\psi) \in C_{\Sigma'}(\Phi')\}$.

Of course, it is desirable that refinement be preserved by horizontal composition of specifications. In particular, refinement by interpretation should be preserved by all specification constructors in Definition 9. The result is non trivial. For union we have,

Lemma 3 *Let $i : \mathsf{SEN}(\Sigma) \longrightarrow \mathscr{P}(\mathsf{SEN}(\Sigma'))$ be a local interpretation, and $SP_1 = \langle \Sigma, \Phi_1 \rangle$, $SP_2 = \langle \Sigma, \Phi_2 \rangle$ specifications such that $SP_1 \rightsquigarrow_i SP_1'$ and $SP_2 \rightsquigarrow_i SP_2'$. If i interprets $\mathsf{union}(SP_1, SP_2)$, then $\mathsf{union}(SP_1, SP_2) \rightsquigarrow_i \mathsf{union}(SP_1', SP_2')$.*

Proof. For all $\phi \in \mathsf{SEN}(\Sigma)$, we reason

$$\begin{array}{ll}
 & SP_1 \rightsquigarrow_i SP_1' \ \wedge\ SP_2 \rightsquigarrow_i SP_2' \\
\Leftrightarrow & \quad \{\text{ definition}\} \\
 & \phi \in C_\Sigma(\Phi_1) \Rightarrow i(\phi) \subseteq C_{\Sigma'}(\Phi_1') \ \wedge\ \phi \in C_\Sigma(\Phi_2) \Rightarrow i(\phi) \subseteq C_{\Sigma'}(\Phi_2') \\
\Rightarrow & \quad \{\, C_\Sigma, C_{\Sigma'} \text{ monotonic}\} \\
 & \phi \in (C_\Sigma(\Phi_1) \cup C_\Sigma(\Phi_2)) \Rightarrow i(\phi) \subseteq (C_{\Sigma'}(\Phi_1') \cup C_{\Sigma'}(\Phi_2')) \\
\Leftrightarrow & \quad \{\text{ definition}\} \\
 & \mathsf{union}(SP_1, SP_2) \rightsquigarrow_i \mathsf{union}(SP_1', SP_2')
\end{array}$$

□

The remaining cases are not straightforward. Actually, achieving compatibility entails the need for imposing some non trivial conditions on morphisms.

5 Towards a refinement calculus

Having defined refinement by interpretation *across* π-institutions and *inside* an arbitrary π-institution, this section sketches their interconnections. Our first step is to define how a specification in an institution I translates to I' along an interpretation.

Definition 10 *Let $\rho = \langle \mathsf{F}, \alpha \rangle : I \longrightarrow I'$ be a translation between π-institutions I and I' and $SP = \langle \Sigma, \Phi \rangle$ a specification in I. The translation $\hat{\rho}(SP)$ of SP through ρ is defined by*

$$\hat{\rho}\,\langle \Sigma, \Phi \rangle \;=\; \langle \mathsf{F}(\Sigma), \alpha_\Sigma(\Phi) \rangle \tag{11}$$

Next lemma answers the following question: is refinement by interpretation over arbitrary π-institutions preserved by the specification constructors?

Lemma 4 *The definition of specification translation is structural over the specification constructors given in definition 9, i.e.*

$$\begin{aligned}
\hat{\rho}\,(\mathsf{union}(SP_1, SP_2)) &= \mathsf{union}(\hat{\rho}(SP_1), \hat{\rho}(SP_2)) \\
\hat{\rho}\,(\mathsf{translate}\ SP\ \mathsf{through}\ \sigma) &= \mathsf{translate}\ \hat{\rho}(SP)\ \mathsf{through}\ \mathsf{F}(\sigma) \\
\hat{\rho}\,(\mathsf{derive}\ SP'\ \mathsf{through}\ \sigma) &= \mathsf{derive}\ \hat{\rho}(SP')\ \mathsf{through}\ \mathsf{F}(\sigma)
\end{aligned}$$

Proof. For the first case let $SP_1 = \langle \Sigma_1, \Phi_1 \rangle$ and $SP_2 = \langle \Sigma_2, \Phi_2 \rangle$. Then,

$$\begin{aligned}
& \hat{\rho}\,(\mathsf{union}(SP_1, SP_2)) \\
= & \qquad \{ \text{ definition of } \mathsf{union} \} \\
& \hat{\rho}\,\langle \Sigma, \Phi_1 \cup \Phi_2 \rangle \\
= & \qquad \{ \text{ definition of } \hat{\rho} \} \\
& \langle \mathsf{F}(\Sigma), \alpha(\Phi_1 \cup \Phi_2) \rangle \\
= & \qquad \{ \ \alpha \text{ is a natural transformation} \} \\
& \langle \mathsf{F}(\Sigma), \alpha(\Phi_1) \cup \alpha(\Phi_2) \rangle \\
= & \qquad \{ \text{ definition of } \mathsf{union} \} \\
& \mathsf{union}(\langle \mathsf{F}(\Sigma), \alpha(\Phi_1) \rangle, \langle \mathsf{F}(\Sigma), \alpha(\Phi_2) \rangle) \\
= & \qquad \{ \text{ definition of } \hat{\rho} \} \\
& \mathsf{union}(\hat{\rho}(SP_1), \hat{\rho}(SP_2))
\end{aligned}$$

Consider now the second case (the third being similar):

$$
\begin{array}{ll}
 & \hat{\rho}\,(\mathsf{translate}\ SP\ \mathsf{through}\ \sigma) \\
= & \qquad \{\ \text{definition of translate}\} \\
 & \hat{\rho}\,\langle \Sigma', \sigma(\Phi)\rangle \\
= & \qquad \{\ \text{definition of } \hat{\rho}\} \\
 & \langle \mathsf{F}(\Sigma'), \alpha_{\Sigma'}(\sigma(\Phi))\rangle \\
= & \qquad \{\ \alpha \text{ is a natural transformation}\} \\
 & \langle \mathsf{F}(\Sigma'), \mathscr{P}(\sigma)(\alpha_{\Sigma}(\Phi))\rangle \\
= & \qquad \{\ \text{definition of translate}\} \\
 & \mathsf{translate}\ \langle \mathsf{F}(\Sigma'), \alpha_{\Sigma}(\Phi)\rangle\ \mathsf{through}\ \mathsf{F}(\sigma) \\
= & \qquad \{\ \text{definition of } \hat{\rho}\} \\
 & \mathsf{translate}\ \hat{\rho}(SP)\ \mathsf{through}\ \mathsf{F}(\sigma)
\end{array}
$$

Note a slight abuse of notation: the extension of $\hat{\rho}(SP)$ in the conclusion is actually through the powerset extension of $\mathsf{F}(\sigma)$.
□

6 Conclusions and related work

In software development, one often has to resort to a number of different logical systems to capture contrasting aspects of systems' requirements and programming paradigms. This paper uses π-institutions to formalize arbitrary logical systems and lifts to such level a recently proposed [MMB09b, MMB09a] approach to refinement based on logical interpretation.

Refinement by interpretation is formulated at both a global (*i.e.*, across π-institutions) and local (*i.e.*, between specifications inside an arbitrary π-institution) level. The paper introduces a notion of structured specification and shows that, at both levels, refinement by interpretation respects the proposed specification constructors. Actually, the institutional setting not only makes it possible to go a step further from [MMB09a] in generalizing the concept to arbitrary logics, but also provides a basis to build up a refinement calculus of "institution-independent", structured specifications.

We close the paper with a few remarks on *refinement by interpretation* in itself and some pointers to related work.

The idea of relaxing what counts as a valid refinement of an algebraic specification, by replacing *signature morphisms* by *logic interpretations* is, to the best of our knowledge, new. The piece of research initiated with [MMB09b] up to the present paper was directly inspired by the second and third author's work on algebraic logic as reported, respectively, in [Mar06] and [Mad08], where the notion of an *interpretation* plays a fundamental role (see, *e.g.*, [BP89, BP01, BR03, Cze01]) and occurs in different variants. In particular, the notion of *conservative translation* intensively studied by Feitosa and Ottaviano [FD01] is the closest to our own approach.

Refinement by interpretation should also be related to the extensive work of Maibaum, Sadler and Veloso in the 70's and the 80's, as documented, for example, in [MSV84, MVS85]. The authors resort

to interpretations between theories and conservative extensions to define a syntactic notion of refinement according to which a specification SP' refines a specification SP if there is an interpretation of SP' into a conservative extension of SP. It is shown that these refinements can be vertically composed, therefore entailing stepwise development. This notion is, however, somehow restrictive since it requires all maps to be conservative, whereas in program development it is usually enough to guarantee that requirements are preserved by the underlying translation. Moreover, in that approach the interpretation edge of a refinement diagram needs to satisfy a number of extra properties.

Related work also appears in [FM93, Vou05] where interpretations between theories are studied, as in the present paper, in the abstract framework of π-institutions. The first reference is a generalization of the work of Maibaum and his collaborators, whereas the second generalizes to π-institutions the abstract algebraic logic treatment of algebraic semantics on sentential logics. Notions of interpretation between institutions also appear in [Bor02] and [Tar95] under the designation of *institution representation*. Differently from the one used in this paper, borrowed from [Vou03], they are not defined as multifunctions. The work of José Meseguer [Mes89] on *general logics*, where a theory of interpretations between logical systems is developed, should also be mentioned.

We believe this approach to refinement through logical interpretation has a real application potential, namely to deal with specifications spanning through different specification logics. Particularly deserving to be considered, but still requiring further investigation, are observational logic [BHK03], hidden logic [Roş00, MP07] and behavioral logic [Hen97]. As remarked above, the study of refinement preservation by horizontal composition remains a challenge and a topic of current research.

Other research topics arise concerns the ways in which *global* and *local* levels interrelate. For example, we are still studying to what extent a local refinement by interpretation of a specification in a π-institution I, lifts to another local refinement of its translation induced by a global interpretation from I to another π-institution I'.

Acknowledgments

This research was partially supported by Fct (the Portuguese Foundation for Science and Technology) under contract `PTDC/EIA-CCO/108302/2008` — the MONDRIAN project, and the CIDMA research center. M. A. Martins was further supported by project *Nociones de Completud*, reference FFI2009-09345 (MICINN - Spain). Finally, A. Madeira was also supported by SFRH/BDE/ 33650/2009, a joint PhD grant by FCT and Critical Software S.A., Portugal.

References

[AN94] H. Andreka & I. Nemeti (1994): *General Algebraic Logic: A Perspective on "What is logic?"* In: *What is a Logical System? - Studies in Logic and Computation, Vol. 4*, Oxford University Press.

[BHK03] M. Bidoit, R. Hennicker & A. Kurz (2003): *Observational logic, constructor-based logic, and their duality. Theor. Comput. Sci.* 298(3), pp. 471–510, doi:10.1016/S0304-3975(02)00865-4.

[Bor02] T. Borzyszkowski (2002): *Logical Systems for Stuctured Specifications*. *Theor. Comp. Science* 286, pp. 197–245, doi:10.1016/S0304-3975(01)00317-6.

[BP89] W. Blok & D. Pigozzi (1989): *Algebraizable Logics*. *Memoirs of the American Mathmatical Society* 396. AMS - American Math. Soc., Providence.

[BP01] W. Blok & D. Pigozzi (2001): *Abstract Algebraic Logic and the Deduction Theorem*. Preprint available from `www.math.iastate.edu/dpigozzi/papers/aaldedth.pdf`.

[BR03] W. Blok & J. Rebagliato (2003): *Algebraic Semantics for Deductive Systems*. *Studia Logica* 74(1-2), pp. 153–180, doi:10.1023/A:1024626023417.

[BSR04] D. Batory, J. N. Sarvela & A. Rauschmayer (2004): *Scaling step-wise refinement*. *IEEE Trans. in Sofware Engineering* 30(6), pp. 355–371, doi:10.1109/TSE.2004.23.

[Cze01] J. Czelakowski (2001): *Protoalgebraic Logics*. Trends in logic, Studia Logica Library, Kluwer Academic Publishers.

[DF02] R. Diaconescu & K. Futatsugi (2002): *Logical foundations of CafeOBJ*. *Theor. Comput. Sci.* 285(2), pp. 289–318, doi:10.1016/S0304-3975(01)00361-9.

[Dia08] R. Diaconescu (2008): *Institution-independent Model Theory*. Birkhäuser Basel, doi:10.1007/978-3-7643-8708-2_2.

[FD01] H. A. Feitosa & I. M. L. D'Ottaviano (2001): *Conservative translations*. *Ann. Pure Appl. Logic* 108(1-3), pp. 205–227, doi:10.1016/S0168-0072(00)00046-4.

[FM93] J. Fiadeiro & T. S. E. Maibaum (1993): *Generalising Interpretations between Theories in the context of (pi-) Institutions*. In: *Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, Springer-Verlag, London, UK, pp. 126–147. Available at `http://portal.acm.org/citation.cfm?id=647322.721361`.

[FS88] J. Fiadeiro & A. Sernadas (1988): *Structuring Theories on Consequence*. In D. Sanella & A. Tarlecki, editors: *Recent Trends in Data Type Specification. Specification of Abstract Data Types (Papers from the Fifth Workshop on Specification of Abstract Datac Types, Gullane, 1987)*, *Lecture Notes in Computer Science* 332, Springer-Verlag, Berlin.

[GB92] J. Goguen & R. Burstall (1992): *Institutions: abstract model theory for specification and programming*. *J. ACM* 39(1), pp. 95–146, doi:10.1145/147508.147524.

[Hen97] R. Hennicker (1997): *Structural specifications with behavioural operators: semantics, proof methods and applications*. Habilitationsschrift.

[Mad08] Alexandre Madeira (2008): *Observational Refinement Process*. *Electr. Notes Theor. Comput. Sci.* 214, pp. 103–129, doi:10.1016/j.entcs.2008.06.006.

[Mar06] Manuel A. Martins (2006): *Behavioral Institutions and Refinements in Generalized Hidden Logics*. *J. UCS - Journ. of Universal Computer Science* 12(8), pp. 1020–1049, doi:10.3217/jucs-012-08-1020. Available at `http://www.jucs.org/jucs_12_8/behavioral_institutions_and_refinements`.

[Mes89] J. Meseguer (1989): *General Logics*. In J. Bairwise & H.J. Keisler et all, editors: *Logic Colloquium'87*, 87, Elsevier, pp. 275–330.

[MHST03] T. Mossakowski, A. Haxthausen, D. Sannella & A. Tarlecki (2003): *CASL: The Common Algebraic Specification Language: Semantics and Proof Theory*. *Computing and Informatics* 22, pp. 285–321, doi:10.1.1.10.2965.

[MMB09a] M.A. Martins, A. Madeira & L.S. Barbosa (2009): *Refinement by Interpretation in a General Setting*. In E. Boiten J. Derrick & S. Reeves, editors: *Proc. Refinement Workshop 2009*, ENTCS, Elsevier, pp. 105–121, doi:10.1016/j.entcs.2009.12.020.

[MMB09b] M.A. Martins, A. Madeira & L.S. Barbosa (2009): *Refinement via Interpretation*. In: *7th IEEE International Conf. on Software Engineeering and Formal Methods, Hanoi, Vietnam*, IEEE Computer Society Press, doi:10.1109SEFM.2009.35.

[MML07] T. Mossakowski, C. Maeder & K. Lüttich (2007): *The heterogeneous tool set, HETS*. In: *13th Int. Conf. Tools and algorithms for the construction and analysis of systems*, TACAS'07, Springer-Verlag, Berlin, Heidelberg, pp. 519–522, doi:10.1.1.67.5472. Available at `http://portal.acm.org/citation.cfm/id=1763507.1763559`.

[MP07] M. A. Martins & D. Pigozzi (2007): *Behavioural reasoning for conditional equations*. *Mathematical Structures in Computer Science* 17(5), pp. 1075–1113, doi:10.1017/S0960129507006305.

[MSV84] T. S. E. Maibaum, M. R. Sadler & Paulo A. S. Veloso (1984): *Logical Specification and Implementation*. In: *Proceedings of the Fourth Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, London, UK, pp. 13–30, doi:10.1007/3-540-13883-8-62.

[MVS85] T. S. E. Maibaum, P. A. S. Veloso & M. R. Sadler (1985): *A theory of abstract data types for program development: bridging the gap?* In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software*, Springer-Verlag, New York, NY, USA, pp. 214–230, doi:10.1007/3-540-15199-0_14. Available at `http://portal.acm.org/citation.cfm/id=22263.22277`.

[Roş00] G. Roşu (2000): *Hidden Logic*. Ph.D. thesis, University of California, San Diego.

[ST88a] D. Sannella & A. Tarlecki (1988): *Specifications in an arbitrary institution*. *Inform. and Comput.* 76, pp. 165–210, doi:10.1.1.144.2669.

[ST88b] D. Sannella & A. Tarlecki (1988): *Towards Formal Development of Programs from Algebraic Specifications: Implementations Revisited*. *Acta Informatica* (25), pp. 233–281, doi:10.1.1.17.6346.

[Tar95] A. Tarlecki (1995): *Moving Between Logical Systems*. In M. Haveraaen, O.J. Dahl & O. Owe, editors: *11th Workshop on Specification of Abstract Data Types, ADT?95*, Springer Lecture Notes in Computer Science (1130), pp. 478–502, doi:10.1.1.49.9260.

[Vou02] G. Voutsadakis (2002): *Categorical Abstract Algebraic Logic: Algebraizable Institutions*. *Applied Categorical Structures* 10, pp. 531–568, doi:10.1023/A:1020990419514.

[Vou03] G. Voutsadakis (2003): *Categorical Abstract Algebraic Logic: Equivalent Institutions*. *Studia Logica* 74, pp. 275–311, doi:10.1023/A:1024682108396.

[Vou05] G. Voutsadakis (2005): *Categorical Abstract Algebraic Logic: Models of π-Institutions*. *Notre Dame Journal of Formal Logic* 46(4), pp. 439–460, doi:10.1023/A:1020990419514.

[Wój88] R. Wójcicki (1988): *Theory of logical calculi. Basic theory of consequence operations.* Synthese Library, 199. Dordrecht etc.: Kluwer Academic Publishers.

Refinement-based verification of sequential implementations of Stateflow charts

Alvaro Miyazawa
Department of Computer Science
University of York
alvarohm@cs.york.ac.uk

Ana Cavalcanti
Department of Computer Science
University of York
ana.cavalcanti@cs.york.ac.uk

Simulink/Stateflow charts are widely used in industry for the specification of control systems, which are often safety-critical. This suggests a need for a formal treatment of such models. In previous work, we have proposed a technique for automatic generation of formal models of Stateflow blocks to support refinement-based reasoning. In this article, we present a refinement strategy that supports the verification of automatically generated sequential C implementations of Stateflow charts. In particular, we discuss how this strategy can be specialised to take advantage of architectural features in order to allow a higher level of automation.

1 Introduction

MATLAB Simulink [24] is a graphical notation widely used in the automotive and avionics industries; it supports the specification of control systems in a level of abstraction convenient for engineers. A Simulink diagram consists of blocks and wires connecting the inputs and outputs of the blocks.

Stateflow [25] is an extension of Simulink that supports the specification of state transition systems, providing a new Simulink block, namely, a Stateflow chart. It is a variant of Statecharts [12], which extends standard state-transition systems by introducing new features, such as hierarchy and parallelism.

While Simulink diagrams are typically used to specify aspects of a system that can be modelled by differential equations relating inputs and outputs, Stateflow charts usually model the control aspects. There is a wide range of tools that support Simulink and Stateflow. These include a simulation and analysis tool, a verification and validation tool, a code generator and a prototyping tool [24, 25, 23].

The extensive use of Simulink/Stateflow in the development of safety-critical systems, associated with certification standards [3, 9] that recommend the use of formal methods for the specification, design, development and verification of software, makes a formal treatment of these notations extremely useful.

We are concerned with the assessment of the correctness of implementations of Stateflow charts. A frequent approach to this problem is based on the verification of automatic code generators [4, 27, 13]. We propose an orthogonal approach based on the verification of implementations with respect to a model of the chart. An overview of our approach is given in Figure 1.

This approach consists of deriving formal models of a Stateflow chart and its implementation, and applying the refinement calculus to check the correctness of the model of the implementation with respect to the model of the chart. This is particularly suited for situations where automatically generated code is not applicable or convenient, for instance, in situations where hardware and performance requirements require changes in the generated code. Moreover, Simulink and Stateflow are frequently updated, and these updates can have a heavy impact on the cost of the verification of any code generator.

In [16], we propose an operational model of Stateflow charts, provide translation rules for deriving such models, and discuss a tool that automatically generates the model of a Stateflow chart. The model of

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 65–83, doi:10.4204/EPTCS.55.5

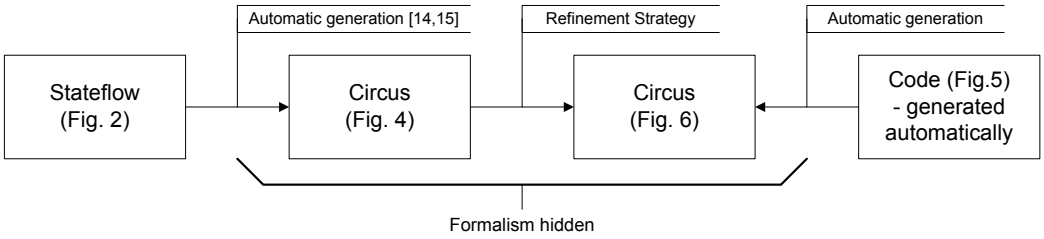


Figure 1: An approach for the verification of implementations of Stateflow charts.

a Stateflow chart is formed by the composition of two processes: the first models the general semantics of Stateflow [25], and the second models specific aspects of a chart. The model of the semantics of Stateflow chart is structured in a way that facilitates the inspection and comparison to the informal semantics found in [25], as no formal analysis can be made because the semantics of Stateflow is available in an informal way [25] or is hidden in the simulation tool.

The refinement calculus is enough for the purpose of verifying such models. However, the expertise required for such verification is often not available. Moreover, the complexity of Stateflow and the size of real charts potentially renders the manual application of the refinement calculus infeasible. We aim in our approach to hide as much of the formalism as possible, to allow it to be used in real scenarios by engineers and programmers. For that to be achieved, we must provide means for the refinement to be established at least in a semi-automatic way.

We propose a verification strategy for sequential automatically generated implementations of discrete-time Stateflow charts with respect to models of Stateflow constructed (automatically) as described in [16]. This technique is closely related to that proposed in [5] for verification of implementations of Simulink diagrams. Our work extends those results to cover a larger class of diagrams and implementations.

The implementations that we consider are those that follow the architectural pattern employed by the code generator provided by MATLAB. There are other code generators [22, 27], but as far as we know, they all cover a limited subset of the Stateflow notation. Fixing the architecture of the implementation allows us to specialise the details of the strategy to increase its level of automation.

Our models for Stateflow charts are specified in *Circus* [29], a formal notation that integrates Z [30], CSP [21], Dijkstra's language of guarded commands [8], and the refinement calculus [17]. These models are particularly adequate for refinement-based verification techniques. Our technique uses the *Circus* refinement laws to provide a tactic of refinement that can be used to prove the correctness of an implementation in a highly automated way. Soundness of the technique stems from soundness of the laws.

This article is structured as follows. Section 2 introduces the background material necessary for the presentation of our strategy. Section 3 discusses the architecture of automatically generated implementations of Stateflow charts and provides general guidelines for deriving *Circus* models of these implementations. Section 4 describes our refinement strategy for the verification of implementations of Stateflow charts. Section 5 assesses our contributions, examines related work, and discusses directions for future developments.

2 Background material

In this section, we introduce the Stateflow and *Circus* notations, and our formal models of Stateflow charts.

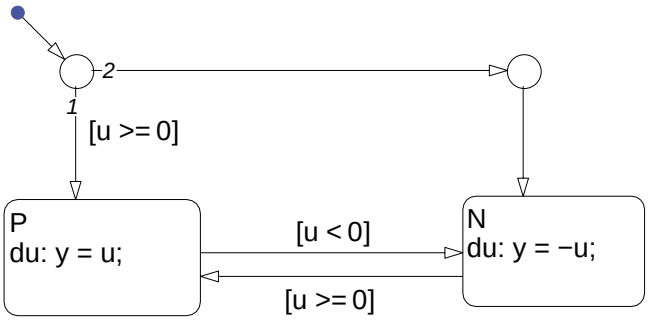


Figure 2: Absolute value chart.

2.1 Stateflow charts

Figure 2 shows our running example: a Stateflow chart adapted from an example supplied with the tool. The chart has one input variable (`u`) and one output variable (`y`); it outputs in `y` the absolute value of `u`.

A Stateflow chart is built from a series of components, such as states, transitions, junctions, data and events. States are represented by rectangles with round corners; in our example, the boxes marked with `P` and `N` are states. States, as well as charts, can have substates, which are arranged in a sequential or parallel decomposition. A state with a sequential decomposition has at most one substate active at any given time, while a state with a parallel decomposition has all of its substates active or inactive at once.

A state has a set of actions associated with it, namely, entry, during, exit, on, and binding actions. Entry, during and exit actions are executed when the state is entered, executed, and exited, respectively; on actions are executed in the same situations as during actions, with the additional requirement that a particular event is being processed; and binding actions bind a particular event or data to the state. In our example, both `P` and `N` have a during action. In `P`, `u` is assigned to `y`, and in `N`, `-u` is assigned to `y`.

Two states (within a state or chart with sequential decomposition) can be connected by one or more transitions; they are indicated by arrows and can be guarded by events and conditions. There are two types of actions associated with a transition: condition and transition actions. Condition actions are executed when the guard of the transition (event and condition) is true, and transition actions are executed when the transition leads to a state being exited.

Transitions are classified according to the relative position between its source and target states; inner transitions have the target state as a substate of the source state, and outer transitions do not. There is a special type of transition, called default transition, that has no source; it is used to indicate the default path to be taken when a state or chart is first entered. In our example, we have one default transition, and five outer transitions. Three of the transitions are guarded by a condition: `u>=0`, `u<0`, or `u>=0`.

A transition path is formed by a series of transitions linked by junctions which are represented by circles. There are two junctions in our example; they form two transition paths. A transition path is completed only when a state is reached by following all the transitions in the path. When a transition path is completed, the source of the path is exited, the transition actions of the path are executed, and the target state is entered. Additionally, there is a special type of junction, called history junction, which records the most recently activated substate of the state that contains it.

In the example in Figure 2, initially, the chart is inactive; the first time it is executed, it is activated

$$
\begin{array}{l}
\textbf{channel}\, in, in1, in2, out : \mathrm{seq}\,\mathbb{N} \\
\textbf{process}\, Merger \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, S == [y : \mathrm{seq}\,\mathbb{N}] \\
\quad InitS == [S' \mid y' = \langle\rangle] \\
\quad Merge \mathrel{\widehat=} x1, x2 : \mathrm{seq}\,\mathbb{N} \bullet \\
\qquad \left(\begin{array}{l}
\textbf{if}\, \#x1 = 0 \longrightarrow y := y \frown x2 \\
[\!]\, \#x2 = 0 \longrightarrow y := y \frown x1 \\
[\!]\, \#x1 \neq 0 \land \#x2 \neq 0 \longrightarrow \\
\qquad \left(\begin{array}{l}
\textbf{if}\, head\, x1 \leq head\, x2 \longrightarrow y := y \frown \langle head\, x1 \rangle \,;\, Merge(tail\, x1, x2) \\
[\!]\, head\, x1 > head\, x2 \longrightarrow y := y \frown \langle head\, x2 \rangle \,;\, Merge(x1, tail\, x2) \\
\textbf{fi}
\end{array} \right) \\
\textbf{fi}
\end{array} \right) \\
\quad \bullet\, InitS \,;\, in1?x1 \longrightarrow in2?x2 \longrightarrow Merge(x1, x2) \,;\, out!y \longrightarrow \textbf{Skip} \\
\textbf{end} \\
\textbf{process}\, SplitSorter \mathrel{\widehat=} \ldots \\
\textbf{process}\, ParallelSorter \mathrel{\widehat=} \left(\begin{array}{c} SplitSorter \\ [\![\{\!| in1, in2 |\!\}]\!] \\ Merger \end{array} \right) \setminus \{\!| in1, in2 |\!\}
\end{array}
$$

Figure 3: The *ParallelSort* specification.

and one of its two states is entered depending on whether `u` is greater than or equal to zero (state `P`) or not (state `N`). In the next execution, if the sign of `u` has changed, a transition takes place from one state to the other. If there is no change, the during action of the active state assigns the absolute value of `u` to `y`.

Before presenting the *Circus* model of this chart, we give, in the next section, an overview of *Circus*.

2.2 *Circus*

We present the main *Circus* features using the example in Figure 3. It models a parallel sorter that reads a sequence of natural numbers through the channel *in*, and writes on the channel *out* an ordered version of the input sequence. A detailed presentation of *Circus* can be found in [29].

A *Circus* specification is a sequence of paragraphs: Z paragraphs (axiomatic definitions, schemas, and so on), channel and channel set declarations, and process definitions. The first paragraph of our example defines four channels *in*, *in*1, *in*2, and *out*, which communicate sequences of natural numbers, that is, elements of the type $\mathrm{seq}\,\mathbb{N}$. The second paragraph is a basic process definition. It provides the name of the process (*Merger*), the definition of its state using a schema *S*, an action *InitS* defined by an operation schema, an action *Merge*, and a main action (after a $\bullet$), which defines, using the previously defined actions, the overall behaviour of the process.

In general, *Circus* actions are written using a mixture of Z and CSP constructs, and guarded commands. In our example, the main action initialises the state using *InitS*, reads a value $x1$ through the channel *in*1, reads a value $x2$ through *in*2, calls *Merge* with the values $x1$ and $x2$ as parameters, and outputs the state variable y through *out*.

The schema *S* has only one component y of type $\mathrm{seq}\,\mathbb{N}$, that is, the set of sequences of natural numbers. The schema *InitS* specifies an initialisation operation over *S* that sets y to the empty sequence ($\langle\rangle$).

Like in Z, we use y' to refer to the value of y after the operation.

The action *Merge* takes two sequences $x1$ and $x2$ of natural numbers, and appends them to the state variable y, so that if both input sequences are ordered, the final sequence in y is also ordered. The specification of *Merge* uses a conditional and assignments from the guarded commands language. If one of the sequences is empty, the non-empty sequence is appended. When both sequences are not empty, *Merge* compares the first element of each sequence ($head\,x1 \leq head\,x2$), appending the smallest of them to y, and recursively calls *Merge* on the rest of the sequence that had the smallest element ($tail\,x1$ or $tail\,x2$), and the whole of the other sequence.

Processes encapsulate their state and interact with other processes through channels. The usual CSP operators can be used to combine processes. The fourth paragraph in Figure 3 defines the process *ParallelSorter* as the parallel composition of the processes *SplitSorter* and *Merge*, communicating over the channels $in1, in2$. The process *SplitSorter* in the third paragraph is omitted; it splits a sequence of natural numbers in two, sorts each sequence in parallel, and outputs them through channels $in1$ and $in2$. In the definition of *ParallelSorter*, the channels $in1$ and $in2$ are hidden, thus yielding a process whose interface contains only the channels *in* and *out*.

In the next section, we have another example of a process; the model of the chart in Figure 2.

2.3 A formal model of Stateflow charts

In this section, we describe the *Circus* operational models of Stateflow charts that we can generate automatically. In these models, the execution of one step of the chart is initiated by reading inputs, and concluded by writing outputs, and synchronising on a channel called *end_cycle*. A more detailed description can be found in [15, 16].

Our models consist of two *Circus* processes in parallel. The first, *Simulator*, represents the simulator, and is the same for every chart. The second, the chart process, represents a particular chart. The simulator and the chart processes communicate over the channels in the set *interface* plus the channel *end_cycle*, with the channels in *interface* hidden. Figure 4 shows the structure of the automatically generated model of the chart in Figure 2.

The chart process *P_AbsoluteValue* uses a data model that defines the state, transition, and junction identifiers, as well as the states, transitions, and junctions as bindings of specific schemas. These are constants that capture information about the structure of the chart. They are represented by the first rectangle in Figure 4. These constants are collected in four other constants defined within the chart process: *identifier*, *states*, *transitions*, and *junctions*. The constant *identifier* records the identifier of the chart and *states*, *transitions*, and *junctions* are partial functions that map identifiers to the corresponding binding. These constants are declared using a schema *StateflowChart* whose definition is omitted in Figure 4. Their values are fixed in the process chart.

Next, the chart process defines a series of schemas that specify components of the state and corresponding initialisation operations. Information about which states are active and which states are recorded in the history junctions is recorded in the schema *SimulationData*, and chart variables are recorded in the schema *SimulationInstance*. These schemas are conjoined to define the schema *State* that specifies the state of the process. We adopt the convention of prefixing a $v\_$ to the name of the chart variable to clarify the nature of the name.

Next, the chart process defines a series of *Circus* actions that can be divided into four groups: actions that correspond to state and transition actions, actions that correspond to calculation of triggers and conditions, actions that read inputs and write outputs, and actions that output the structure of the chart. All these actions are grouped to define *AllActions*, which is used in the main action as shown in Figure 5.

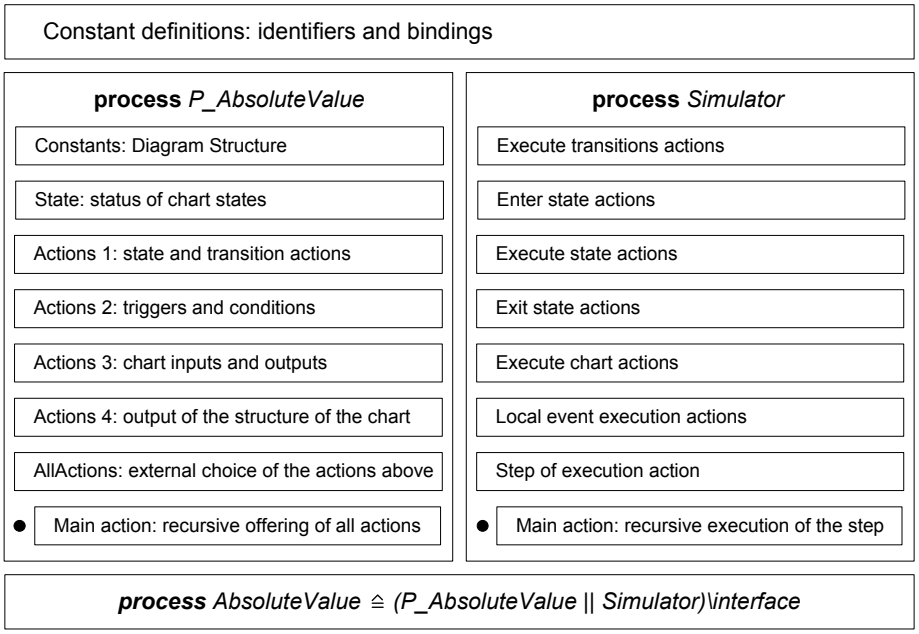


Figure 4: Structure of the model of the chart in Figure 2

$$\bullet \left(InitState\right);\ \mu X \bullet \left(\left(\left(\mu Y \bullet \left(\begin{array}{l} ((AllActions \triangle (interrupt\_chart \longrightarrow \mathbf{Skip}));\ Y) \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) ;\ X \right) \right)$$

end

Figure 5: Main action of the chart process shown in Figure 4

The main action of the chart process is shown in Figure 5; it initialises the state, and recursively offers the actions in *AllActions*, with the additional possibility that any of these actions can be interrupted at any time by a communication over the channel *interrupt_chart*. The possibility of interruption accounts for the occurrence of early return logic in the chart, that is, the interruption of the execution of the chart brought about by a state inconsistency produced by a local event broadcast. The main action has two nested recursions: the internal one corresponds to the actions that are offered to one particular step of simulation, and can be terminated by a synchronisation over the channel *end_cycle*. The external recursion corresponds to the recursive execution of simulation steps.

The process *Simulator* does not have a state; it declares a series of actions that model the execution of transitions, as well as the processes of entering, executing and exiting states. The execution of a chart is then defined in terms of the previous actions. A chart can be executed multiple times in the same time step due to the occurrence of multiple input events or the broadcast of a local event. In the first case, an action encodes the execution of the chart for each active event in the appropriate order and is used to define the step of execution. In the second case, we define an action that captures the occurrence of a broadcast and executes the chart under the appropriate setting. This action is called whenever a state or transition action is executed. All these actions are combined to define the step of execution of the

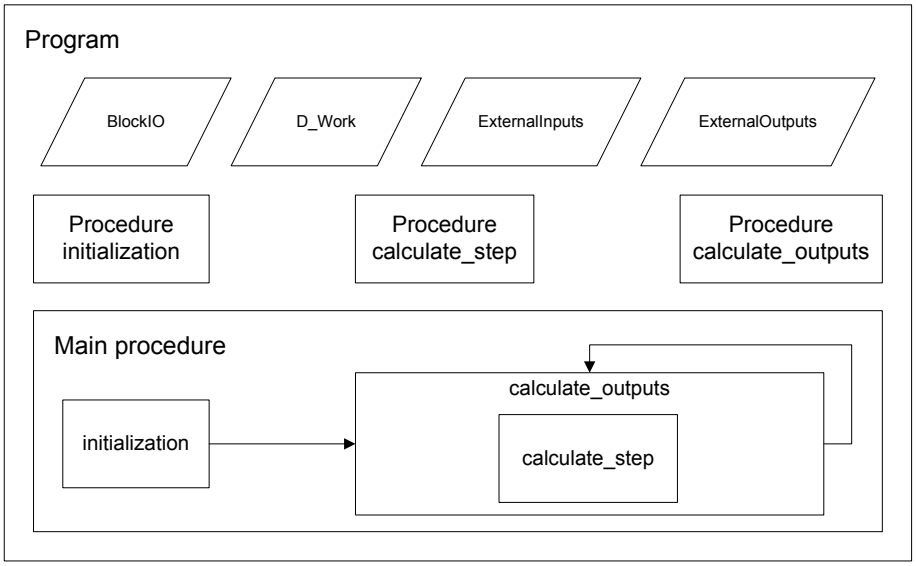


Figure 6: Architcture of the implementations of Stateflow charts

simulator, which is called recursively in the main action of the process *Simulator*.

In the next section we discuss an approach to modelling implementations of charts.

3 Implementations of Stateflow charts

Figure 6 shows the structure of implementations of Stateflow charts generated automatically by Real Time Workshop/Stateflow Coder. In general, the implementations produced consist of a series of structures that define the state of the chart (inputs, outputs, local variables, events, execution state, and so on) and a series of procedures. The procedures can be divided into those that implement the execution of the chart, which are relevant for our verification, and those that calculate the next time step. Since we capture time using synchronisation, we restrict our attention those of the first kind depicted in Figure 6 as `calculate_outputs`, `initialization` and `calculate_step`.

The procedure `calculate_outputs` implements the execution of the chart, `initialization` initialises the variables of the program, and `calculate_step` implements the control of the execution of the chart according to the number of active events. The main procedure of the implementation initializes the program and repeatedly calculate the outputs using the procedure `calculate_step`. In the case of a C implementation, the procedures shown in Figure 6 are implemented as C functions whose names reflect the name of the chart. In our example, for instance, the procedure *calculate_outputs* is implemented as the C function `AbsoluteValue_output`.

Of particular interest to us is how the implementation models information regarding the status of the states. This is done in two different ways, according to the type of decomposition of the states. In this discussion, we regard the chart as a state. If the state has a parallel decomposition, its status is modelled by a single variable in the structure *D_Work_X*, where *X* is the name of the chart. This variable is called *is_active_S*, where *S* is the name of the state; it has a numerical type (`uint8_T`), but, in fact, it is used as a boolean variable, that is, if its value is zero the state is not active, otherwise, it is active.

$$
\begin{array}{l}
D\_Work\_AbsoluteValue == [is\_active\_c1\_AbsoluteValue, is\_c1\_AbsoluteValue : \mathbb{N}] \\
\ldots \\
\textbf{process}\, AbsoluteValue \mathrel{\widehat=} \textbf{begin} \\
\textbf{state}\, AbsoluteValue\_state == [AbsoluteValue\_DWork : D\_Work\_AbsoluteValue;\ \ldots] \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue \mathrel{\widehat=} x : \mathbb{N} \bullet AbsoluteValue\_DWork := \\
\quad \langle is\_active\_c1\_AbsoluteValue == AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue, \\
\quad is\_c1\_AbsoluteValue == \_y \rangle \\
\ldots \\
AbsoluteValue\_output \mathrel{\widehat=} tid : \mathbb{Z} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{if}\, AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue = 0 \longrightarrow \\
AbsoluteValue\_DWork\_is\_active\_c1\_AbsoluteValue(1); \\
\left(\begin{array}{l}
\textbf{if}\, AbsoluteValue\_B.SineWave1 \geq 0 \longrightarrow \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue(AbsoluteValue\_IN\_P) \\
[]\neg\,(AbsoluteValue\_B.SineWave1 \geq 0) \longrightarrow \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue(AbsoluteValue\_IN\_N) \\
\textbf{fi}
\end{array} \right) \\
[]\neg\,(AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue = 0) \longrightarrow \\
\ldots \\
\textbf{fi}
\end{array} \right) \\
AbsoluteValue\_Y\_y(AbsoluteValue\_B.y)
\end{array} \right) \\
\ldots \\
\bullet\, AbsoluteValue\_initialize\,;\ \mu X \bullet Input\,;\ AbsoluteValue\_output\,;\ Output\,;\ end\_cycle \longrightarrow X
\end{array}
$$

Figure 7: *Circus* model of the implementation of the chart in Figure 2

If the state has a sequential decomposition, its status is modelled by two variables in *D_Work_X*. The variable *is_active_S* is as described above. The variable *is_S* records a number that identifies which substate is active at the time, its value is zero if there are no active substates. If a state is a child of a state with a sequential decomposition, no variable of the form *is_active_S* is created, as the information about its status is already recorded in the variable *is_P*, where *P* is the name of its parent state.

In our example, we have only states with sequential decompositions. The status of the chart is recorded by `is_active_c1_AbsoluteValue` and `is_c1_AbsoluteValue`. There is no need for variables that record the status of `P` and `N`, as this information can be obtained from `is_c1_AbsoluteValue`.

We model the implementation as a series of schemas and a single process. Figure 7 gives a partial view of the model of the implementation of our example. Schemas model the records in the implementation. For instance, `D_Work_AbsoluteValue` is modelled by the schema *D_Work_AbsoluteValue*.

For each of the relevant C function in the implementation, we define a *Circus* action that models it. In Figure 7, we present the *Circus* action that models the function `AbsoluteValue_output`. The main action of the process is fixed and consists of calling the initialisation action (*AbsoluteValue_initialize* for our example), and recursively reading the inputs (with the action *Input*), producing the outputs (with the action *AbsoluteValue_output*), offering the outputs (with the action *Output*), and signalling the end of the "time-step" by synchronising on *end_cycle*. The actions *Input* and *Output* abstract as communications the shared variables used to implement inputs and outputs.

In the modelling of the functions, we map C constructs to similar *Circus* constructs. Loops are

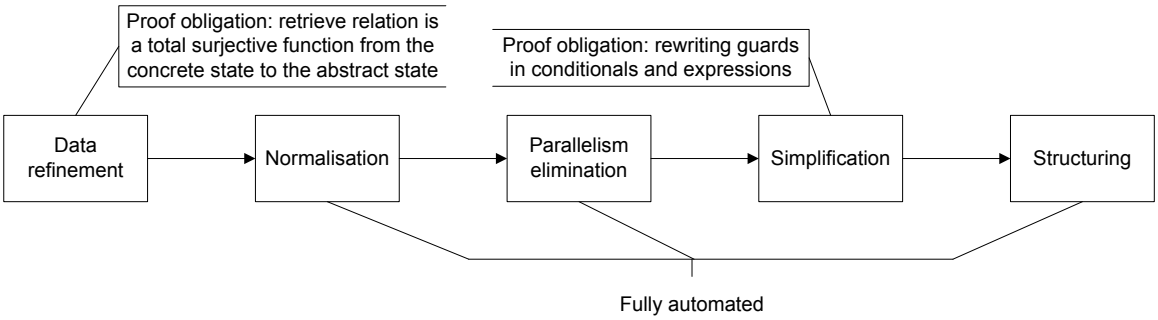


Figure 8: Overview of the refinement strategy

modelled using recursion. In general, the translation of implementation constructs is direct, except for the assignment to a variable of a structure. Since we cannot write $b.f := v$ (as a translation of `b.f = v`), for a variable b of type binding, we define *Circus* actions that specify the assignment of a binding to the variable, as the action *AbsoluteValue_DWork_is_c*1*_AbsoluteValue* in Figure 7. This action takes one parameter x of type $\mathbb{N}$, and assigns a binding of the schema *D_Work_AbsoluteValue* to the state component *AbsoluteValue_DWork*. The binding is formed by associating each component of the schema to a value, the component *is_active_c*1*_AbsoluteValue* is associated to the value of the same component, and *is_c*1*_AbsoluteValue* is associated to the value of the parameter.

In the next section, we discuss a refinement strategy that supports the verification of the models of implementations just described with respect to the models discussed in Section 2.3.

4 Refinement Strategy

Our refinement strategy consists of five phases: data refinement, normalisation, parallelism elimination, simplification, and structuring. Figure 8 illustrates the strategy; it identifies the fully automated phases and the proof obligations that stem from the other phases.

In the data refinement phase, we modify the state of the chart process in order to conform to the state of the implementation model. The normalisation phase transforms the parallel composition of the chart and simulation processes into a single process whose main action initialises the state and recursively offers an action that encodes a step of execution of the chart. The parallelism elimination phase collapses the parallel actions that occur in the resulting process. This is necessary because the parallelism in the diagram model reflects the operational semantics of Stateflow, not a parallel design for a program. In the simplification phase, we simplify expressions and predicates, and move assumptions through the model to eliminate unreachable branches of alternations. Finally, in the structuring phase, we rewrite the main action to match the functions of the implementation, and therefore the actions of its model.

The strategy proposed in this section is generic enough to be applied to a large class of charts and implementations. It takes advantage of restrictions on the architecture of the implementation to support a high degree of automation. We consider here only sequential implementations of Stateflow diagrams. The steps of the strategy are, however, useful in the refinement to parallel implementations as well.

In the sequel, we describe the details of each phase. They define procedures to apply existing and novel *Circus* refinement laws whose soundness guarantees the soundness of our verification strategy.

ConcreteState

$$AbsoluteValue\_state$$

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \in \{0,1,2\}$$

Figure 9: Restricted concrete state

4.1 Data refinement

In this phase, we construct a retrieve relation between the abstract state of the chart process and the concrete state of the implementation model. With that, we use the *Circus* calculus to construct a refinement of the chart process, and so preserve its structure, and transform the assignments, operation schemas, and communications. Precisely, we follow the procedure below for constructing retrieve relations that are total surjective functions from concrete to abstract states. They allow us to proceed with the data refinement in a calculational fashion.

The state components of the implementation model belong to one of four groups: execution specific data, like `AbsoluteValue_DWork` in our example, local variables (like `AbsoluteValue_B`), input variables (like `AbsoluteValue_U`), and output variables (`AbsoluteValue_Y`). Additionally, we can restrict the components of the concrete state to take values only over the appropriate sets. For example, Figure 9 shows the state of our implementation model with one additional invariant; it requires that *is_c*1*_AbsoluteValue* takes values from $\{0,1,2\}$.

The retrieve relation maps the execution specific data to the components of *SimulationData*, and the local, input and output variables to components of *SimulationInstance*. The correspondence between the input and output variables is trivial. It is obtained by equating the concrete variables to the abstract variable whose name is the same except for a prefix *v_*. The specification of the relation between the execution specific data of the concrete state and the function *state_status* in *SimulationData* is obtained by using a set comprehension where, for each state identifier *s*, we define a boolean *active* that determines its status from the concrete state. These conditions can be calculated as follows. For a chart name *C*, and each component of *D_Work_C* (in our example, the schema *D_Work_AbsoluteValue*) named *is_active_name* (*is_active_c*1*_AbsoluteValue*, for example), where *name* is the name of state (or chart), we have the following condition.

$$s = name \land active = (\textbf{if } C\_DWork.is\_active\_name > 0 \textbf{ then True else False})$$

For instance, the condition for *is_active_c*1*_AbsoluteValue* equates *s* to *c_AbsoluteValue*, and *active* to **True** or **False** depending on whether the value of *is_active_c*1*_AbsoluteValue* is greater than zero or not.

For each component of the schema *D_Work_C* of the form *is_name*, where *name* is the name of a state (or chart), we formulate a condition in the following way. For each substate *X* of *name*, we define the condition below.

$$s = s\_X \land active = (\textbf{if } C\_DWork.is\_name = C\_IN\_X \textbf{ then True else False})$$

In our example, the condition for the state `P` equates *s* to *s_P*, and *active* to **True** or **False** depending on whether the value of *is_c*1*_AbsoluteValue* is *AbsoluteValue_IN_P* or not. All these conditions are composed in a disjunction as shown in the definition of the retrieve relation for our example in Figure 10.

RetrieveFunction

$$
\begin{array}{l}
P\_AbsoluteValue\_S \\
ConcreteState \\
\hline
state\_status = \{s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet \\
\quad s = s\_N \land active = \textbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = \\
\qquad AbsoluteValue\_IN\_N\ \textbf{then True else False} \lor \\
\quad s = s\_P \land active = \textbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = \\
\qquad AbsoluteValue\_IN\_P\ \textbf{then True else False} \lor \\
\quad s = c\_AbsoluteValue \land active = \\
\qquad \textbf{if}\ AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue > 0\ \textbf{then True else False}\} \\
state\_history = \{\} \\
v\_u = AbsoluteValue\_U.u \\
v\_y = AbsoluteValue\_B.y
\end{array}
$$

Figure 10: Total surjective functional retrieve relation

Since our example does not contain history junctions, the implementation has no state components that model the state component *state_history* of the chart process. The model of the chart establishes that this state component is the empty partial function, therefore we equate *state_history* to the empty set.

The retrieve relation in Figure 10 is functional because each abstract state component is defined by a function of a component of the concrete state. Since no restriction is imposed on the concrete state for the applicability of the function, the relation is also total. Moreover, for every abstract state A, it is possible find a concrete state that is related to A by the retrieve relation because the functions are invertible.

Using this retrieve relation, we apply the laws of simulation [6, 19] to obtain a *Circus* process *C_P_AbsoluteValue* by data refinement of *P_AbsoluteValue*, and to refine the process *AbsoluteValue* to a process *CAbsoluteValue*, as shown in Figure 11. We define the constant *ss* to increase the readability of *C_P_AbsoluteValue*. This function is defined as the characterisation of *state_status* in Figure 10.

The main action does not change, but components of the actions that it uses are transformed. For example, *Activate* and *InitState* are data refined to operate over the concrete state. Figure 11 shows part of the definition of *CInitState* (the data refinement of the schema *InitState*); it shows the part of the predicate that defines the operation. The actions *condition_P_N*, *Inputs*, and *Outputs* are also data refined; the first has the component *v_u* rewritten to *AbsoluteValue_U.u*, the second has the assignment transformed into an action that assigns a value to a component of a schema binding (as mentioned in Section 3), and the third has the component *v_y* substituted by *AbsoluteValue_B.y*, in accordance with the retrieve relation.

In the next section, we describe how to collapse the parallel composition in *CAbsoluteValue*.

4.2 Normalisation

In this phase, we first collapse the parallelism between the chart and simulator processes in the process *CAbsoluteValue*, and rewrite the main action of the resulting new process to a normal form: an initialisation action, followed by a recursive action. This allows us to focus on the body of the recursion that characterises one step of execution.

process $C\_P\_AbsoluteValue \mathrel{\widehat=}$ **begin**

$StateflowChart$

$identifier = c\_AbsoluteValue$
$states = \{(s\_P, S\_P), (s\_N, S\_N), (c\_AbsoluteValue, C\_AbsoluteValue)\}$
$transitions = \{(t\_P\_N, T\_P\_N), \ldots, (t\_6\_N, T\_6\_N)\}$
$junctions = \{(j5, J5), (j6, J6)\}$

$ss : SID \pfun \mathbb{B}$

$ss = \{s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet$
$\quad s = s\_N \land active =$ **if** $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N$
$\qquad$ **then True else False** $\lor$
$\quad s = s\_P \land active =$ **if** $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P$
$\qquad$ **then True else False** $\lor$
$\quad s = c\_AbsoluteValue \land active = AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue\}$

state $ConcreteState$

$CActivate == [\Delta ConcreteState;\ x? : SID \mid \ldots]$
$CInitState == [ConcreteState' \mid AbsoluteValue\_DWork'.is\_active\_c1\_AbsoluteValue = \mathbf{False} \land \ldots]$

$\ldots$
$condition\_P\_N \mathrel{\widehat=} \mathbf{if}((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow \ldots\ldots$
$Inputs \mathrel{\widehat=} (read\_inputs \longrightarrow (i\_u?x \longrightarrow AbsoluteValue\_U\_u(x)))$
$Outputs \mathrel{\widehat=} (write\_outputs \longrightarrow (o\_y!(AbsoluteValue\_B.y) \longrightarrow \mathbf{Skip}))$

$$\bullet \left(CInitState\right);\ \mu X \bullet \left(\left(\left(\mu Y \bullet \left(\begin{array}{l} ((AllActions \triangle (interrupt\_chart \longrightarrow \mathbf{Skip}));\ Y) \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) \right);\ X \right)$$

end

process $CAbsoluteValue \mathrel{\widehat=} Simulator \llbracket interface \cup \{\!| end\_cycle |\!\} \rrbracket C\_P\_AbsoluteValue$

Figure 11: Data refinement of the processes shown in Figure 4

$$\bullet \left(\begin{array}{l} \left(\begin{array}{l} \big(CInitState\big); \\ \mu X \bullet \left(\mu Y \bullet \left(\begin{array}{l} \left(\begin{array}{l} AllActions \\ \triangle \\ (interrupt\_chart \longrightarrow \mathbf{Skip}) \end{array} \right) ; Y \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) ; X \end{array} \right) \\ [\![\{AbsoluteValue\_B, AbsoluteValue\_DWork\} \mid interface \cup \{\!| end\_cycle |\!\} \mid \{\}]\!] \\ (\mu X \bullet Step ; X) \end{array} \right) \setminus interface$$

Figure 12: Combined main action after merging the two processes.

$$\bullet \left(\begin{array}{l} \big(CInitState\big); \\ \mu X \bullet \left(\mu Y \bullet \left(\begin{array}{l} \left(\begin{array}{l} AllActions \\ \triangle \\ (interrupt\_chart \longrightarrow \mathbf{Skip}) \end{array} \right) ; Y \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) \setminus interface ; X \\ [\![\ldots]\!] \\ Step \end{array} \right)$$

Figure 13: Main action after the normalisation phase (We abbreviate the parallelism).

We construct the new process by taking the state of the chart process (the simulator process is stateless), and combining the main actions of the chart and simulator processes in the same way the processes were combined, as shown in Figure 12. This is a direct application of the definition of the semantics of process parallelism in *Circus*.

Next, we move the schema action *CInitState* out of the parallel composition, distribute the hiding over the sequential composition, and eliminate the hiding over *CInitState*. The external recursion in the first action of the parallel composition, and the recursion in the second action are then merged. This is possible because *Step* necessarily terminates in a synchronisation over the channel *end_cycle*, since this channel is in the synchronisation set, and this synchronisation stops the inner recursion, and starts a new cycle of the external recursion. We are left with the action in Figure 13, which calls *CInitState*, and recursively executes the parallel composition of a recursion (the recursion over *Y* in Figure 12), and the action *Step*, with the channels in *interface* hidden.

4.3 Parallelism elimination

In this phase, we eliminate the parallelism still embedded in the main action. The parallel action inside the recursion of Figure 12 reads an input event, reads the input variables, executes the chart, writes the outputs, and ends the cycle. In general, the step of execution involves a series of communications over channels that may or may not be in the synchronisation set. Communications over channels not in the synchronisation set are moved outside the parallel composition, and the others are used to select an action from *AllActions* (or trigger an interruption). We proceed as follows to evaluate the communications and remove the parallelism.

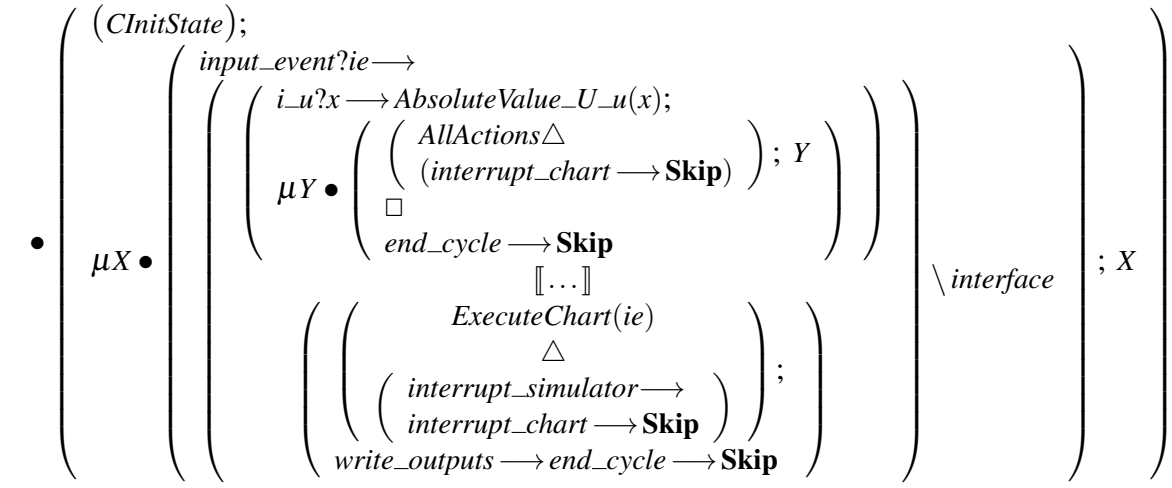


Figure 14: Main action after the communications over *input_event* and *read_inputs* are treated.

By expanding the definition of *Step* in Figure 13, we have two communications one after the other. The first is a communication over the channel *input_event* that is not in the synchronisation set, and the second is a synchronisation over *read_inputs*, which is in the synchronisation set. We move the communication outside the parallel composition, unfold the recursion over Y, and resolve the communication on the channel *read_inputs*. This produces a prefixing action identical to the body of the action *Inputs* in Figure 11. Because the communication is hidden, we eliminate it and obtain the action in Figure 14.

The actions that can be selected can be an atomic information request, as exemplified by *read_inputs*, a non-atomic information request, or a Stateflow action request. In the first case, as already shown, the parallelism can be removed by resolving the communication. In the case of a non-atomic request (for instance, a trigger action of Figure 4), the action is composed of two communications. To eliminate the parallelism we resolve them; this potentially involves resolving a conditional expression that selects the appropriate value to communicate. A Stateflow action request consists of a communication that identifies the appropriate action, a series of *Circus* actions that encode the Stateflow action, and a synchronisation that indicates completion. In this case, the initial communication and the final synchronisation are treated as usual. The encoding of the Stateflow action contains assignments and local event broadcasts. Assignments are moved out of the parallel composition, and broadcasts produce a recursive execution of the chart, which can be treated using the same strategy.

The strategy for eliminating parallelism can be seen as a two level strategy. The first level is guided by the structure of the simulator process, which potentially leads to the execution of an action of the chart process, and the second level is guided by a chart process action that has been executed. Since the simulator process is the same for all charts, we explore its structure to define the refinement strategy.The same is not true for the chart process, but due to the simple structure of the actions in the chart process, we can explore the limited patterns that occur.

At the end, we obtain a main action whose structure is as shown in Figure 16. There, we have already simplified expressions, which is the objective of the next phase.

$$
\left(\begin{array}{l}
\left(\mu Y \bullet \left(\begin{array}{l} AllActions\,;\ Y \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array}\right)\right) \\
\qquad [\![\ldots]\!] \\
\left(\begin{array}{l} status!(states(c\_AbsoluteValue).identifier)?active \longrightarrow \\ \left(\begin{array}{l} \mathbf{if}\ active = \mathbf{True} \longrightarrow \\ \qquad \left(\begin{array}{l} ExecuteActiveChart(states(c\_AbsoluteValue), ie); \\ write\_outputs \longrightarrow end\_cycle \longrightarrow \mathbf{Skip} \end{array}\right) \\ [\!]\ active = \mathbf{False} \longrightarrow \\ \qquad \left(\begin{array}{l} ExecuteInactiveChart(states(c\_AbsoluteValue), ie); \\ write\_outputs \longrightarrow end\_cycle \longrightarrow \mathbf{Skip} \end{array}\right) \\ \mathbf{fi} \end{array}\right) \end{array}\right)
\end{array}\right) \setminus interface
$$

Figure 15: Parallel action of the main action in Figure 14 after further refinement steps.

4.4 Simplification

In the simplification phase, we transform expressions and eliminate unreachable branches of conditional statements. The simplification of expressions takes advantage of the constants that model the structure of the chart, as well as state invariants. For instance, Figure 15 contains an expression that appears frequently in communication resolutions: $states(c\_AbsoluteValue).identifier$. It evaluates to the identifier of the state whose identifier is *c_AbsoluteValue*, but this is exactly *c_AbsoluteValue*, thus we simplify it.

After these simplifications are carried, we obtain a main action as in Figure 16. The last branch of the second conditional has the guard: $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_N$. Since *AbsoluteValue_IN_N*=2, the invariant of the concrete state implies that

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 1 \vee AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 0$$

Therefore, since *AbsoluteValue_IN_P* is a constant defined as 1, we replace the above guard with

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P \vee$$
$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 0$$

This guard is then broken in two, and a new branch is added to the conditional statement. We proceed in this way for every guard defined by a disjunction that checks the status of a state. This simplification is applied whenever the status of a state in a sequential decomposition is checked because our model always includes a branch whose guard is an inequality. It is necessary because, while our model contains only binary conditionals, the implementations may have conditionals with more than two branches.

We traverse the resulting action and for each conditional statement found, we attempt to simplify the guard. If we can simplify a guard to false, we eliminate the branch. If the guard is true, we reduce the whole conditional to the action it guards; this is correct, because all our conditionals are of the form $\mathbf{if}\, b \longrightarrow \ldots [\!] \neg b \longrightarrow \ldots \mathbf{fi}$. If a conditional statement cannot be simplified, it should match a conditional statement in the model of the implementation. For example, the first conditional statement in Figure 16 corresponds to the first conditional statement of the action *AbsoluteValue_output* in Figure 7.

$$
\begin{array}{l}
\bullet\ CInitState;\ \mu X \bullet input\_event?ie \longrightarrow i\_u?x \longrightarrow AbsoluteValue\_U\_u(x);\\
\left(\begin{array}{l}
\mathbf{if}\ ss(c\_AbsoluteValue) = \mathbf{False} \longrightarrow \ldots\\
[\!]\ ss(c\_AbsoluteValue) = \mathbf{True} \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}((AbsoluteValue\_U.u \geq_{\mathscr{A}} 0) \neq 0) \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P \longrightarrow\\
Deactivate\_P;\ Deactivate\_N;\ Activate\_P;\\
\quad \left(\begin{array}{l}
\mathbf{if}(AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow \ldots\\
[\!]\ \neg\ (((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0)) \longrightarrow \ldots\\
\mathbf{fi}
\end{array}\right)\\
[\!]\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_P \longrightarrow\\
Deactivate\_N;\ Activate\_P;\\
\quad \left(\begin{array}{l}
\mathbf{if}((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow \ldots\\
[\!]\ \neg\ (((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0)) \longrightarrow \ldots\\
\mathbf{fi}
\end{array}\right)\\
\mathbf{fi}
\end{array}\right)\\
[\!]\ \neg\ (((AbsoluteValue\_U.u \geq_{\mathscr{A}} 0) \neq 0)) \longrightarrow AbsoluteValue\_B\_y(-\ AbsoluteValue\_U.u);\\
\quad \left(\begin{array}{l}
\mathbf{if}((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N \longrightarrow\\
\qquad \ldots\\
[\!]\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_N \longrightarrow\\
\qquad \ldots\\
\mathbf{fi}
\end{array}\right)\\
[\!]\ \neg\ (((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0)) \longrightarrow \ldots\\
\mathbf{fi}
\end{array}\right)\\
\mathbf{fi}
\end{array}\right)\\
[\!]\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_N \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow\\
\quad \left(\begin{array}{l}
\mathbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N \longrightarrow \ldots\\
[\!]\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_N \longrightarrow \ldots\\
\mathbf{fi}
\end{array}\right)\\
[\!]\ \neg\ (((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0)) \longrightarrow \ldots\\
\mathbf{fi}
\end{array}\right)\\
\mathbf{fi}
\end{array}\right)\\
\mathbf{fi}
\end{array}\right);\ X
\end{array}
$$

Figure 16: Partially simplified main action of process *AbsoluteValue*.

The fourth conditional statement in Figure 16 is an example of a statement that can be simplified. The guard of the first branch is $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P$, but this is inside a branch whose guard is $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N$, and since $is\_c1\_AbsoluteValue$ cannot have both values (and no statement modifies this component between the two branches), the first guard is false, and that branch can be eliminated. In this way, we put the model of the chart in the same shape as the model of the implementation, and once this is achieved, the structuring phase takes place. Formalisation of this strategy using refinement requires a law to introduce assumptions based on the guards of the conditional, laws to distribute and use assumptions, and finally a law to remove an assumption when it is no longer needed. These are standard laws that are valid in *Circus* as shown in [19].

4.5 Structuring

The structuring phase identifies each component of the main action that corresponds to an auxiliary action in the model of the implementation. It introduces this extra action in the process *CAbsoluteValue*, and uses the copy rule to replace the component of its main action by a call to it. The rationale behind this phase is to match the main action to that of the model of the implementation.

For instance, we compare the action *AbsoluteValue_output* in the model of the implementation to the subactions of the main action obtained from the previous phase. We identify a subaction that is a match, introduce its definition, and substitute the name *AbsoluteValue_output* in the main action. The result of all this should be exactly the model of the implementation (as shown in Figure 7 for our example). If this is not the case, the verification has failed: either the program is wrong, or it does not conform to the architectural pattern that we can handle.

The detailed application of this strategy to our example can be found in [14].

5 Conclusion

We have proposed a refinement-based verification strategy for implementations of Stateflow charts. This strategy is guided by the structure of the models of Stateflow charts described in [16]. We have also discussed how such a strategy can take advantage of the architecture imposed on generated code.

We have provided a procedure for obtaining retrieve relations that support the data refinement of the specification in a calculational style, thus rendering the data refinement phase also suitable for automation. In the case of the normalisation and parallelism elimination phases, the possibility of automation stems from the fixed structure of our models. The simplification phase can be semi-automated because the main action consists of a number of nested if-statements, and assumptions generated by the guards of the conditional statements (among others) can be moved into the associated action, potentially falsifying some of the conditions in an internal conditional statement. Finally, the structuring phase can be guided by matching actions from the model of the implementation to subactions of the action being refined.

The refinement strategy for Simulink presented in [5] consists of four steps that systematically collapse the massive parallelism of the diagram specification to match the processes of the implementation model, prove that each of the procedures in the implementation refine the action that specifies it in the corresponding process, and finally, prove that the parallel programs refine the process that specifies the system. The main actions of component processes are put in a normal form where they are defined as the iterative execution of a step that consists of reading the inputs in interleaving, calculating the outputs and updating the state, writing the outputs in interleaving, and synchronising on the channel *end_cycle*.

Our strategy has a similar nature, however, it is worth mentioning some important differences. Our models of Stateflow charts owe their parallelism to the separation between the structure of the model and the operational semantics of the simulator, not to any implicit or explicit parallelism in the chart. Therefore, while in [5] collapsing the parallelism is guided by the implementation model, in our strategy it is performed until there are no parallel actions left. The beginning of our parallelism elimination phase is similar to the process of putting the main action in a normal form in the strategy for Simulink diagrams. The equivalent step in our model is, however, not as linear as in [5]. The decision to end the cycle comes from the action inherited from the simulator process. Nevertheless, by the end of the parallelism elimination phase, we have a process whose main action is in the normal form of [5]. This suggests that not only our models can be integrated to the models of Simulink diagrams, but also that our strategy can be used to put a Stateflow block in the normal form prescribed in [5]. This will support the integrated use of these verification techniques for Simulink diagrams involving Stateflow blocks.

There are several approaches to the formal analysis of Stateflow diagrams. These works aim at the analysis of diagrams, not of their implementations. Operational and denotational semantics are proposed in [11] and [10]; these support static analysis, interpretation, and compilation of Stateflow charts. Translations of Stateflow into notations that support model checking are presented in [1], [26], [22], and [7]. Verification in these approaches is based on temporal logics and bisimulation, rather than refinement, thus verification of implementations is not the objective. An approach based on Z to verify that the chart satisfies a set of requirements of the system being modelled is presented in [28]. However, it places strong restrictions on the Stateflow notation.

Olderog [18] integrates three views of a system (trace specification, process algebra and Petri nets) by formalising a relation between them. While this approach ends in a graphical notation, namely Petri nets, we take the opposite direction: from a graphical notation to a program. In [2], the refinement calculus is adapted to Simulink diagrams, but they do not cover Stateflow charts, and their goal is not the verification of implementations, but the development of diagrams from contracts. In [20], a semantics for μ-Charts is constructed in Z, and a notion of refinement of μ-Charts is derived from the existing Z refinement calculus. This approach is similar to that presented in [16], where we define a semantics of Stateflow charts in *Circus*, thus allowing the *Circus* refinement calculus to be applied, but it differs in the sense that we focus in a industrial non-formal notation, while the μ-Charts notation is a simplification of Statecharts mainly used in academia. Moreover, our approach goes beyond the application of the refinement calculus to Stateflow charts; it addresses the problem of automation of the refinement process.

As far as we know, this is the first work to address the issue of verification of implementations of Stateflow charts. Moreover, as explained in detail [15], our models of Stateflow charts used as the base for the verification eliminate many of the restrictions imposed in other formalisations.

Given the generality of our refinement strategy, we believe it scales well to modified implementations. In particular, an implementation that does not modify the use of the variable *AbsoluteValue_DWork* should still be amenable to the specialisation of the refinement strategy discussed in Section 4.1. Moreover, our strategy can be used as a preliminary phase in the verification of parallel implementations, as all the parallelism eliminated by the strategy derives from the structure of the model.

As future work, we will address the issue of verification of parallel implementations of Stateflow charts. Parallel implementations are not common, and as far as we know there are no code generators that produce parallel implementations. We will extend the current strategy to allow the verification to be carried out after the introduction of parallelism in the implementation using fixed design patterns.

References

[1] C. Banphawatthanarak & B. H. Krogh (2000): *Verification of stateflow diagrams using smv: sf2smv 2.0*. Technical Report CMU-ECE-2000-020, Carnegie Mellon University.

[2] P. Boström et al. (2007): *Stepwise Development of Simulink Models Using the Refinement Calculus Framework*. In: *ICTAC 2007*, *LNCS* 4711, pp. 79–93, doi:10.1007/978-3-540-75292-9_6.

[3] BS EN 61508-3:2002 (2002): *Functional safety of electrical/electronic/programmable electronic safety related systems – Part 1: General requirements*.

[4] P. Caspi et al. (2003): *From Simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications*. *ACM SIGPLAN Notices* 38(7), pp. 153–162,doi:10.1145/780753.780754.

[5] A. L. C. Cavalcanti et al. (2011): *From control law diagrams to Ada via Circus*. *FAC* , pp. 1–48, doi:10.1007/s00165-010-0170-3.

[6] A. L. C. Cavalcanti et al. (2003): *A Refinement Strategy for Circus*. *FAC* 15(2 - 3), pp. 146–181, doi:10.1007/s00165-003-0006-5.

[7] C. Chen (2010): *Formal Analysis for Stateflow Diagrams*. In: *SSIRI-C '10*, pp. 102–109, doi:10.1109/SSIRI-C.2010.29.

[8] E. W. Dijkstra (1975): *Guarded commands, nondeterminacy and formal derivation of programs*. *Commun. ACM* 18(8), pp. 453–457, doi:10.1145/360933.360975.

[9] DO-178b (1992): *Software Considerations in Airborne Systems and Equipment Certification*.

[10] G. Hamon (2005): *A denotational semantics for Stateflow*. In: *EMSOFT*, ACM, pp. 164–172, doi:10.1145/1086228.1086260.

[11] G. Hamon & J. Rushby (2004): *An Operational Semantics for Stateflow*. In: *FASE*, *LCNS* 2984, Springer-Verlag, Barcelona, Spain, pp. 229–243, doi:10.1007/s10009-007-0049-7.

[12] D. Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *SCP* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[13] R. Lublinerman et al. (2009): *Modular code generation from synchronous block diagrams: modularity vs. code size*. In: *POPL' 09*, ACM, pp. 78–89, doi:10.1145/1480881.1480893.

[14] A. Miyazawa & A. L. C. Cavalcanti: *Refinement of the AbsoluteValue chart*. Available at `www.cs.york.ac.uk/~alvarohm/refinement`.

[15] A. Miyazawa & A. L. C. Cavalcanti (2011): *A formal semantics of Stateflow charts*. Technical Report YCS-2011-461, University of York.

[16] A. Miyazawa & A. L. C. Cavalcanti (2011): *Refinement-oriented models of Stateflow charts*. *SCP* (to appear).

[17] C. C. Morgan (1994): *Programming from Specifications*. Prentice Hall International Series in Computer Science.

[18] E.-R. Olderog (1991): *Nets, terms and formulas: three views of concurrent processes and their relationship*. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511526589.

[19] M. V. M. Oliveira (2006): *Formal Derivation of State-Rich Reactive Programs using Circus*. Ph.D. thesis, Department of Computer Science - University of York, UK. YCST-2006-02.

[20] G. Reeve & S. Reeves (2006): *Logic and refinement for charts*. ACSC '06, Australia, pp. 13–23.

[21] A. W. Roscoe (1998): *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science, Prentice-Hall, New York. Oxford.

[22] N. Scaife et al. (2004): *Defining and translating a "safe" subset of simulink/stateflow into lustre*. In: *EMSOFT*, ACM, pp. 259–268, doi:10.1145/1017753.1017795.

[23] The MathWorks,Inc.: *Real-Time Workshop*. `www.mathworks.co.uk/products`.

[24] The MathWorks,Inc.: *Simulink*. `www.mathworks.co.uk/products`.

[25] The MathWorks,Inc.: *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.co.uk/products`.

[26] A. Tiwari (2002): *Formal semantics and analysis methods for Simulink Stateflow models*. Technical Report, SRI International. `www.csl.sri.com/~tiwari/stateflow.html`.

[27] A. Toom et al. (2008): *Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos*. In: *ERTS'08*.

[28] I. Toyn & A. Galloway (2005): *Proving properties of Stateflow models using ISO Standard Z and CADiZ*. In: *ZB-2005, vol. 3455 of LNCS*, pp. 104–123,doi:10.1007/11415787_7.

[29] J. C. P. Woodcock & A. L. C. Cavalcanti (2002): *The Semantics of Circus*. In: *ZB 2002: Formal Specification and Development in Z and B*, *LCNS* 2272, Springer-Verlag, pp. 184–203,doi:10.1007/3-540-45648-1_10.

[30] J. C. P. Woodcock & J. Davies (1996): *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc.

Refinement for Probabilistic Systems with Nondeterminism

Steve Reeves
Department of Computer Science
University of Waikato
Hamilton
New Zealand
stever@cs.waikato.ac.nz

David Streader
Department of Computer Science
University of Waikato
Hamilton
New Zealand
dstr@cs.waikato.ac.nz

Before we combine actions and probabilities two very obvious questions should be asked. Firstly, what does "the probability of an action" mean? Secondly, how does probability interact with nondeterminism? Neither question has a single universally agreed upon answer but by considering these questions at the outset we build a novel and hopefully intuitive probabilistic event-based formalism.

In previous work we have characterised refinement via the notion of testing. Basically, if one system passes all the tests that another system passes (and maybe more) we say the first system is a refinement of the second. This is, in our view, an important way of characterising refinement, via the question "what sort of refinement should I be using?"

We use testing in this paper as the basis for our refinement. We develop tests for probabilistic systems by analogy with the tests developed for non-probabilistic systems. We make sure that our probabilistic tests, when performed on non-probabilistic automata, give us refinement relations which agree with for those non-probabilistic automata. We formalise this property as a vertical refinement.

1 Introduction

Event-based models are frequently based on finite automata (FA, also called labelled transition systems) and probabilistic event-based systems are frequently based on FA where the transitions are also labelled by a probability as well as by an action. Before we combine events and probabilities two very obvious questions then arise. Firstly, what does "the probability of an event" mean, or what does it mean for an event to "behave in a probabilistic fashion"? Secondly, how does probability interact with nondeterminism? Neither question has a single universally agreed upon answer but by considering these questions at the outset we build a novel and hopefully intuitive probabilistic event-based formalism.

Throughout we will be motivated by a wish to, in the end, develop a notion of refinement for probabilistic systems. In fact, refinement will be the starting point of our story here as well as the desired end point.

In previous work we have characterised refinement via the notion of testing. Basically, if one system passes all the tests that another system passes (and maybe more) we say the first system is a refinement of the second. This is, in our view, an important way of characterising refinement since the question "what sort of refinement should I be using?" can be answered by saying "you should be using the sort of refinement that is characterised by the sort of tests which characterise the contexts within which your system will find itself, i.e. choose your refinement by looking at what contexts your systems will be used in."

Because this seems such a natural and useful answer, we use testing again in this paper as the basis for our refinement. We develop tests for probabilistic systems by analogy with the tests developed for non-probabilistic systems, all the while hoping to make sure that our probabilistic tests, when performed on non-probabilistic automata (and just noting whether a probability distribution is empty or not), give

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 84–100, doi:10.4204/EPTCS.55.6

us refinement relations which agree with for those non-probabilistic automata: this gives us confidence that our new notions make sense. We formalise this property in Section 7.

The real test (!) in all this comes when we consider probabilistic automata which also contain nondeterminism. Again, we are guided by the wish that our probabilistic tests, when used on nondeterministic, non-probabilistic automata, give us a refinement ordering which agrees with that originally given for those automata when probability was not considered. We also find that the algebraic properties that characterise the non-probabilistic case carry over into our new domain.

We formalise a notion of refinement based upon probabilistic tests and then try to (re-)capture what nondeterminism means in this probabilistic setting.

We will first introduce transition systems as a semantic foundation for non-probabilistic automata and recap previous work on using testing to define refinement for such systems.

It will turn out that part of the key to doing this for probabilistic systems is to be clear about two different philosophical bases for probability, so we next review those. Another part of the key to this work will be a consideration of how nondeterminism is characterised, so we will go on to discuss that subsequently. This will finally suggest how we might adapt transition systems to allow consideration of probability, and we finally show how this adaptation can be used to also allow a treatment of nondeterministic probabilistic systems, all the while retaining our testing-based notion of refinement.

We also show (via a selection) that expected properties hold for our refinement.

2 Transition systems

Definition 1 *Finite Automata (FA). Let Act be a set of actions and let Act^τ be the same set along with the special action τ, which represents actions interacting to form events. Let N_A be a finite set of nodes.*

The finite automaton A is given by the triple $(N_\mathsf{A}, S_\mathsf{A}, T_\mathsf{A})$ where

1. *$S_\mathsf{A} \subseteq N_\mathsf{A}$ is a set of start nodes*
2. *$T_\mathsf{A} \subseteq \{(n,\mathsf{a},m) | n,m \in N_\mathsf{A} \wedge \mathsf{a} \in Act^\tau\}$ shows the effect of each action.*

We write $x \xrightarrow{\mathsf{a}}_\mathsf{A} y$ for $(x,\mathsf{a},y) \in T_\mathsf{A}$ and $x \xrightarrow{\mathsf{a}} y$ where A is obvious from context. We write $n \xrightarrow{\mathsf{a}}$ for $\exists m.(n,\mathsf{a},m) \in T_\mathsf{A}$, and $m \xrightarrow{\rho} n$ for

$$\exists m_1 \ldots m_i . m \xrightarrow{\rho_1} m_1, m_1 \xrightarrow{\rho_2} m_2, \ldots m_i \xrightarrow{\rho_i} n$$

and $m \xrightarrow{\rho}$ for

$$\exists m_1 \ldots m_i, n . m \xrightarrow{\rho_1} m_1, m_1 \xrightarrow{\rho_2} m_2, \ldots m_i \xrightarrow{\rho_i} n$$

when $\rho = (\rho_1, ..., \rho_i)$, a finite sequence of actions.

We write $n \Longrightarrow m$ for $n \xrightarrow{\tau^*} m$, $n \overset{\mathsf{a}}{\Longrightarrow} m$ for $\exists j,k . n \Longrightarrow j \wedge j \xrightarrow{\mathsf{a}} k \wedge k \Longrightarrow m$ and $n \overset{\mathsf{a}}{\Longrightarrow}$ for $\exists j,k,m . n \Longrightarrow j \wedge j \xrightarrow{\mathsf{a}} k \wedge k \Longrightarrow m$.

$m \overset{\rho}{\Longrightarrow}$ and $m \overset{\rho}{\Longrightarrow} n$ are defined similarly to the cases for $\longrightarrow$.

Where ρ is a sequence of actions over Act^τ we write ρ_0 for ρ with the τs removed.

The traces are $Tr(\mathsf{A}) \overset{\text{def}}{=} \{\rho \mid s \in S_\mathsf{A} \wedge s \overset{\rho}{\Longrightarrow}\}$.

The complete traces[1] are $Tr^c(\mathsf{A}) \overset{\text{def}}{=} \{\rho \mid (s \in S_\mathsf{A} \wedge s \overset{\rho}{\Longrightarrow} n \wedge \pi(n) = \emptyset)$ where $\pi(n) \overset{\text{def}}{=} \{m \mid n \xrightarrow{\times}_\mathsf{A} m\}$.

[1] We deal with only acyclic automata and so we do not need to deal with infinite traces, though all the work of this paper can be extended to infinite traces and cyclic automata in the standard way [1].

We wish to model, using our automata, components that, like CSP processes, can immediately be nondeterministic. But, unlike CSP, we wish hiding (abstraction) to distribute through choice (so τs are used only for unobservable actions or for events, and not pressed into service to encode nondeterministic choice between starting states). There is a subtle difference between how external choice in CSP and choice in CCS behave with processes containing initial τ actions. This has been explained either by regarding the choice operators as being different, see [2] "The unique choice operator of CCS, denoted by +, is a mixture between external and internal choices" or by viewing CSP's use of τ actions to model a nondetermined start state as different to CCS's use of τ actions [3]. By allowing automata to have a set of start states we both avoid having to distinguish external choice and CCS choice and allow hiding to distribute through choice [3].

Also, choice can be defined ([4, 5]) between FAs with one start state each by gluing the two start states together to make a new single start state. Here, due to our generalisation, we glue together two sets of start states.

Let $S = \{s_1, s_2, \ldots, s_n\}$ and $S' = \{s'_1, s'_2, \ldots, s'_m\}$ be two sets of starting states and then define $\{S/S \times S'\}$ to be the n substitutions $\{s_i \in S | s_i / \{(s_i, s'_1), \ldots, (s_i, s'_m)\}\}$ and define $\{S'/S \times S'\}$ to be the m substitutions $\{s'_j \in S' | s'_j / \{(s_1, s'_j), \ldots, (s_n, s'_j)\}\}$.

We define $\{SS'/S \times S'\}$ to be the $n+m$ simultaneous substitutions $\{S/S \times S'\} \cup \{S'/S \times S'\}$. The first n substitutions replace each element of $\{s_1, s_2, \ldots, s_n\}$ with a set of m nodes and the last m substitutions simultaneously replace each element of $\{s'_1, s'_2, \ldots, s'_m\}$ with a set of n nodes. Consequently $\{S_\mathsf{A} S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\}$ will identify the two sets of nodes S_A and S_B as $S_\mathsf{A}\{S_\mathsf{A} S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\}$ and $S_\mathsf{B}\{S_\mathsf{A} S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\}$ are both the $n \times m$ set of nodes $S_\mathsf{A} \times S_\mathsf{B}$.

Since single states may now become sets of states under the substitution, we also have to define what it means to have sets of nodes in a transition:

$$T \xrightarrow{\mathsf{x}} T' \stackrel{\text{def}}{=} \{t \xrightarrow{\mathsf{x}} t' | t \in T, t' \in T'\}$$

Definition 2 *Process operators. Let* A *be* $(N_\mathsf{A}, S_\mathsf{A}, T_\mathsf{A})$ *and let* B *be* $(N_\mathsf{B}, S_\mathsf{B}, T_\mathsf{B})$.
Action Prefixing $\mathsf{a}.\mathsf{B} \stackrel{\text{def}}{=} (\{s\} \cup N_\mathsf{B}, \{s\}, \{s \xrightarrow{\mathsf{a}} x | x \in S_\mathsf{B}\} \cup T_\mathsf{B})$ *where s is a new state.*
Internal choice $\mathsf{A} \sqcap \mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B}, S_\mathsf{A} \cup S_\mathsf{B}, T_\mathsf{A} \cup T_\mathsf{B})$
External choice is, informally, internal choice where start states are combined according to the substitutions above. Let $S_{\mathsf{A}\Box\mathsf{B}}$ *be* $\bigcup((S_\mathsf{A} \cup S_\mathsf{B})\{S_\mathsf{A} S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\})$*, i.e. we combine start states as above. Then,*
External choice $\mathsf{A}\Box\mathsf{B} \stackrel{\text{def}}{=} ((N_\mathsf{A} \cup N_\mathsf{B}) \setminus (S_\mathsf{A} \cup S_\mathsf{B}) \cup S_{\mathsf{A}\Box\mathsf{B}}, S_{\mathsf{A}\Box\mathsf{B}}, (T_\mathsf{A} \cup T_\mathsf{B})\{S_\mathsf{A} S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\})$
Parallel composition: $\mathsf{A} \|_P \mathsf{B} \stackrel{\text{def}}{=} (N_{\mathsf{A}\|_P\mathsf{B}}, S_{\mathsf{A}\|_P\mathsf{B}}, T_{\mathsf{A}\|_P\mathsf{B}})$ *where* $P \subseteq N_\mathsf{A} \cap N_\mathsf{B}$, $N_{\mathsf{A}\|_P\mathsf{B}} = N_\mathsf{A} \times N_\mathsf{B}$, $S_{\mathsf{A}\|_P\mathsf{B}} = S_\mathsf{A} \times S_\mathsf{B}$ *and* $T_{\mathsf{A}\|_P\mathsf{B}}$ *is defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} l, m \xrightarrow{\mathsf{x}}_\mathsf{B} k, \mathsf{x} \in P}{(n,m) \xrightarrow{\tau}_{\mathsf{A}\|_P\mathsf{B}} (l,k)}$$

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} l, (\mathsf{x} \notin P \wedge m \in N_\mathsf{B})}{(n,m) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_P\mathsf{B}} (l,m)} \qquad \frac{n \xrightarrow{\mathsf{x}}_\mathsf{B} l, (\mathsf{x} \notin P \wedge m \in N_\mathsf{A})}{(m,n) \xrightarrow{\mathsf{x}}_{\mathsf{A}\|_P\mathsf{B}} (m,l)}$$

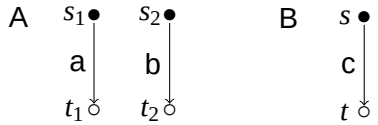
Example 1 *Let* A *be*

$$(\{s_1, s_2, t_1, t_2\}, \{s_1, s_2\}, \{s_1 \xrightarrow{\mathsf{a}}_\mathsf{A} t_1, s_2 \xrightarrow{\mathsf{b}}_\mathsf{A} t_2\})$$

and let B *be*

$$(\{s, s_2, t\}, \{s\}, \{s \xrightarrow{\mathsf{c}}_\mathsf{B} t\})$$

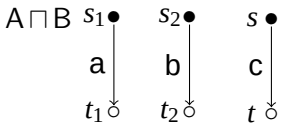
or, in diagram form,



Then $\mathsf{A} \sqcap \mathsf{B}$ *is*

$$(\{s_1, s_2, s, t_1, t_2, t\}, \{s_1, s_2, s\}, \{s_1 \xrightarrow{\mathsf{a}}_{\mathsf{A}\sqcap\mathsf{B}} t_1, s_2 \xrightarrow{\mathsf{b}}_{\mathsf{A}\sqcap\mathsf{B}} t_2, s \xrightarrow{\mathsf{c}}_{\mathsf{A}\sqcap\mathsf{B}} t\})$$

or, as a diagram,



Given that $S_{\mathsf{A}\Box\mathsf{B}}$ *is*

$$\bigcup \{s_1, s_2, s\}\{s_1/\{(s_1, s)\}, s_2/\{(s_2, s)\}, s/\{(s_1, s), (s_2, s)\}\} = \{(s_1, s), (s_2, s)\}$$

then $\mathsf{A}\Box\mathsf{B}$ *is*

$$(\{t_2, t_3, (s_1, s), (s_2, s)\}, \{(s_1, s), (s_2, s)\},$$

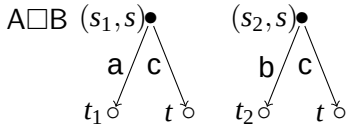
$$\{(s_1, s) \xrightarrow{\mathsf{a}}_{\mathsf{A}\Box\mathsf{B}} t_1, (s_2, s) \xrightarrow{\mathsf{b}}_{\mathsf{A}\Box\mathsf{B}} t_2, \{(s_1, s), (s_2, s)\} \xrightarrow{\mathsf{c}}_{\mathsf{A}\Box\mathsf{B}} t\})$$

which is

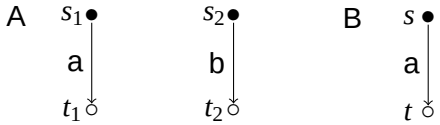
$$(\{t_2, t, (s_1, s), (s_2, s)\}, \{(s_1, s), (s_2, s)\},$$

$$\{(s_1, s) \xrightarrow{\mathsf{a}}_{\mathsf{A}\Box\mathsf{B}} t_1, (s_2, s) \xrightarrow{\mathsf{b}}_{\mathsf{A}\Box\mathsf{B}} t_2, (s_1, s) \xrightarrow{\mathsf{c}}_{\mathsf{A}\Box\mathsf{B}} t\}, (s_2, s) \xrightarrow{\mathsf{c}}_{\mathsf{A}\Box\mathsf{B}} t\})$$

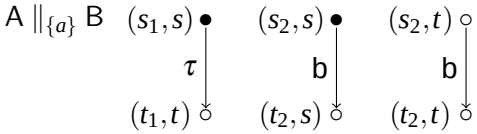
and as a diagram



Finally, $\mathsf{A} \parallel_{\{a\}} \mathsf{B}$ *with (note that* B*'s action is now* a*)*



is



□

3 Testing semantics

The definitions in this section are taken from [6] where they have been applied to both state-based and event-based models.

One of our tests, of a process E, taken from a set of processes $\mathbb{E}$, consists of placing E in some context X taken from a set of possible contexts Ξ. E in context X is written $[\mathsf{E}]_{\mathsf{X}}$. We then observe the resulting system. Each observation made is taken from a set of possible observations $\mathbb{O}$.

We turn first to our general definition of testing semantics for nondeterministic processes and contexts. In this setting a test may return (nondeterministically) one observation from a set of possible observations.

A specification is interpreted as a *contract* consisting of the *assumption* that the process will be placed only in one of the specified contexts Ξ and a *guarantee* that the observation of its behaviour will be one of the observations defined by the mapping $O : \mathbb{E} \to \Xi \to \wp\mathbb{O}$. The mapping O defines what can be observed for all processes in any of the assumed contexts. Hence for any fixed Ξ and O we have a definition of the semantics and the refinement of processes.

Definition 3 *Let Ξ be a set of contexts each of which the processes $\mathsf{A}, \mathsf{C} \in \mathbb{E}$ can communicate privately with, and let $O : \mathbb{E} \to \Xi \to \wp\mathbb{O}$ be a function which returns a set of observations, i.e. a subset of $\mathbb{O}$. Then, the relational semantics of a process A is a subset of $\Xi \times \mathbb{O}$.*

$$[\![\mathsf{A}]\!]_{\Xi,O} \stackrel{\text{def}}{=} \{(x,o) | x \in \Xi \wedge o \in O([\mathsf{A}]_x)\}$$

and refinement is given by

$$\mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{C} \stackrel{\text{def}}{=} [\![\mathsf{C}]\!]_{\Xi,O} \subseteq [\![\mathsf{A}]\!]_{\Xi,O}$$

and equality is

$$\mathsf{A} =_{\Xi,O} \mathsf{C} \stackrel{\text{def}}{=} [\![\mathsf{C}]\!]_{\Xi,O} = [\![\mathsf{A}]\!]_{\Xi,O}$$

□

Given a rich enough class of tests the use of nondeterministic tests is redundant, as what can be observed using a nondeterministic test will be the union of what can be observed using a set of deterministic tests. Hence nondeterministic tests add no further information and will be ignored.

For all the processes considered in this paper, placing a process A in a context X, i.e. $[\mathsf{A}]_{\mathsf{X}}$, will mean executing process A in parallel with X, i.e. $\mathsf{A} \parallel_N \mathsf{X}$ (where N is some set of actions over which the context and process communicate, i.e. synchronize) and the observation function O is either the trace function Tr (if only safety properties are of interest) or (if liveness properties are of interest) the complete trace function Tr^c.

Definition 4 *Let Ξ_{FA} be FA and let $\sqsubseteq_{FA}$ be $\sqsubseteq_{\Xi_{FA},Tr^c}$.* □

Theorem 1 *Refinement distributes through parallel composition: Let $\mathsf{X}, \mathsf{Y}, \mathsf{P}, \mathsf{Q} \in FA$*

$$\frac{\mathsf{X} \sqsubseteq_{FA} \mathsf{Y}, \mathsf{P} \sqsubseteq_{FA} \mathsf{Q}}{\mathsf{X} \parallel_N \mathsf{P} \sqsubseteq_{FA} \mathsf{Y} \parallel_N \mathsf{Q}}$$

□

4 Probabilities—Two Interpretations

There are two (main) interpretations of probability, the *frequentist* and the *Bayesian*.

The frequentists' definition sees probability as the long-run expected frequency of occurrence. The probability of event A happening, where n is the number of times event A occurs in N opportunities, is $P(A) = n/N$.

The Bayesians' view of probability is related to degree of belief or state of knowledge. It is a measure of the plausibility of an event given incomplete knowledge. The Bayesian probabilist specifies some given or assumed prior probabilities, which are then used in the computation of other probabilities. That is to say, anything that is nondeterministic or unknown must either be assigned some probability or have its probability computed from other, more primitive, known probabilities. Bayesian statisticians have developed several "objective" methods for specifying prior probabilities.

The frequentists' view is based upon repeatedly performing the same test many times and, where the behaviour of the item under test is nondeterministic, aggregating the results of all the tests. Extending an event-based testing semantics to record not just the set of possible observations but the probability with which they occur is a simple uniform way to extend event-based testing semantics to event-based probabilistic testing semantics. This can be further generalised by representing both the process under test and the test process itself with probabilistic automata.

The Bayesian view fits well with Hoare's comment on nondeterminism [7, p81]:

> "*There is nothing mysterious about this kind of nondeterminism: it arises from a deliberate decision to ignore the factors which influence the selection*"

So, nondeterminism in a process is merely a case of not having analysed it enough to quantify it, i.e. attach to it some probabilities. Nondeterministic choice is probabilistic choice with unknown probabilities. Surprisingly, this is not how testing semantics have been defined in the literature.

As probabilities quantify (i.e. attach a number to, or make quantitative) nondeterministic behaviour, it is clearly crucial when modelling some real process to distinguish between the behaviour of the process being deterministic and the behaviour being nondeterministic. Similarly when the process is observed interacting in some context it is crucial to distinguish the nondeterminism of the process from the nondeterminism of the context.

Give a coin to a frequentist statistician and they experiment by flipping the coin a large number of times noting down the number of times they observe heads being uppermost and the number of times they observe tails. From this experiment they can compute the probability.

An important point to note is that, to the frequentist, probabilities define how likely it is that an action is executed, or equivalently how likely it is that the execution ends in a particular state. The probability of an event occurring when the event cannot be executed must be zero.

The Bayesian statistician, given a coin, knows that the only observations are heads and tails, and has no further information. The skill of the Bayesian statistician is to assign a prior probability based on understanding the world that agrees with the frequentist. It becomes very important when we try to add probabilities to event-based processes that we either follow the frequentist and perform experiments (tests) or follow the Bayesian statistician and think clearly about the behaviour in the world of what we are modelling.

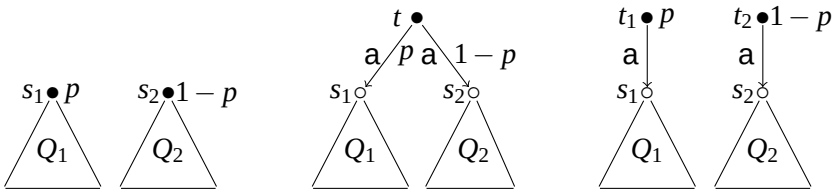


Figure 1: Probabilities on starting states

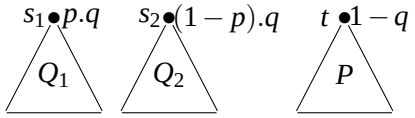


Figure 2: More general probabilistic combination

5 Probabilistic Finite Automata

5.1 Probability

We introduce probabilities on choice by attaching probabilities to the start states of a process. There are two things to notice here: as in the non-probabilistic case with FAs, we represent nondeterminism on the initial state of a process by allowing the process to start in one of a *set* of states; and we generalise this idea to represent the *probability* of starting in some state of a process by attaching probabilities to each of its start states so that we can see what the probability of each possible start state being actually chosen for some particular execution of the process.

The first of these points is inherited from work [8] which seeks to remove the need to use unobservable actions to also "encode" or represent nondeterminism in a process by assuming the process makes an unobserved transition to its "real" starting state (which may be one of many) from some single "dummy" formal starting state. (And, of course, this is just a case of using the usual "set of states" model uniformly for start states as well as all other states, which is something we are all familiar with from the "classic" algorithm that constructs a deterministic finite-state automaton from a nondeterministic one.) Such unobserved actions can then be used exclusively to denote (synchronisation between) events. This idea is, in the second point above, carried over into the probabilistic realm so that initial probabilistic choice is replaced by a probability distribution over the possible starting states.

So, if P is the process that starts with a choice between Q_1 and Q_2, which have (single, for this illustration) starting states s_1 and s_2 respectively, with probabilities p of starting in state s_1 and $1-p$ of starting in state s_2 then we might picture P as in the left of Figure 1. We might represent the picture by saying $S(P) = \{s_1 \mapsto p, s_2 \mapsto 1-p\}$, where S is a probability distribution function over start states of P.

Further, if we now form the process $a.P$ (i.e. the event a happens then the process P happens) then we might picture this as in the middle of Figure 1, and here notice how the probabilities have migrated to the occurrences of event a. This picture suggests that transitions now represent the effect of an action on an initial state moving the system, according to some probability distribution, to the next state, when it synchronizes with the same action in some other process, i.e. when the two actions combine to form an event which takes place with the indicated probability.

So in $a.P$, the action a has the potential to move us from state t to state s_1 with probability p and to s_2 with probability $1-p$ when synchronized to form an event which actually does take place

with the indicated probabilities. We formalise all this by saying that the transitions of $a.P$ include $\{t \xrightarrow{a} d \mid d(s_1) = p \wedge d(s_2) = 1-p\}$. An alternative picture might be as shown in the right of Figure 1, and here notice how the probabilities on the new start states for the new process $a.P$ have migrated from the old start states of P and we have $S(a.P) = \{t_1 \mapsto p, t_2 \mapsto 1-p\}$. This picture might be considered a useful, though perhaps more unusual, alternative way of thinking of our system in the previous picture.

Note that the original form of transitions as in FAs can be recovered by using the domain of the probability distribution function to tell us what the relevant post-states are.

As processes are combined together, the probabilities for the various component start states are combined to form the probabilities for the start states of the combination. As an example, see Figure 2, which shows what the resultant start-state probabilities are for $(Q_1 +_p Q_2) +_q P$, where s_1, s_2 and t are the start states for Q_1, Q_2 and P respectively.

5.2 Probability and nondeterminism

From statistics, the *law of large numbers* tells us that nondeterministic behaviour is the same as probabilistic behaviour where the probabilistic behaviour is unknown but can be found by repeating the right experiment a large number of times.

In process algebras τ actions indicate hidden, unobservable, uncontrollable actions or events (a special case being when two processes synchronize on some actions, which we consider to be private and uncontrollable). Remember Hoare's comment that we cited in Section 4. We have said above that we view this as agreeing with the Bayesian idea that probability indicates a lack of information.

As probabilities refer to frequencies of executable behaviour, i.e. the probability of an event occurring, they naturally occur on τ actions. The intuitive relationship between nondeterminism and probability is widely held. For example,

> *"nondeterminism represents possible choices that can be resolved in a wholly unpredictable way. With probabilistic constructs the resolution becomes predictable up to a point, in that it is quantified"* [9]

We can view this as saying that probabilistic processes contain more information than nondeterministic processes but less than deterministic processes. Consequently what can be observed in any single observation of a probabilistic process is the same as what can be observed of the underlying non-probabilistic process. But by aggregating the observations of a large number of executions we can compute a probability distribution or verify a previously computed probability distribution.

As τ events are built by composing two actions that are observable (via parallel composition, i.e. synchronization) it would be useful to find some way to compute the probability of the executable τ event from the prior "probabilities" of their observable parts. This we do below in Definition 8.

The addition of probabilities to *observable* actions where there is *no* nondeterminism has proven both hard to interpret and hard to formalise, especially when we want to ensure that the models have desirable properties. One reason, in our opinion, that this has turned out to be so hard to do is that the probabilities on the observable actions need, obviously, to define the behaviour of the processes not just in one context but in *all* contexts.[2]

[2]We go no further with this point in this paper, but note that, in the non-probabilistic setting, we have considered this previously in [10].

5.3 Nondeterminism

We represent nondeterminism not by a separate set of operators but by allowing probabilities to be denoted not just by real numbers in the range 0 to 1 but also by real-valued terms (in that range) that contain variables or parameters. This introduces the idea of a starting-state distribution which is not completely determined or which has undetermined aspects, and hence allows us to represent nondeterminism with the same machinery that we introduce for probabilities.

This idea is motivated by the Bayesian view that the more we know about a mechanism, the more certain we can be about the probabilities attached to its behaviour: to talk of nondeterministic behaviour is merely to admit having more or less incomplete information about how something behaves, and this incomplete information can be represented by having parameters in the terms which denote probabilities. This also accords with Hoare's view that nondeterminism arises from ignoring or hiding (or, we would go further and say, being ignorant of) some aspects of a process. Further analysis of the mechanism would uncover ("unhide") more of the mechanism. This view dissolves nondeterminism; there is no such thing really, since it is just arises from not knowing (for whatever reason) enough about the actual distribution of probabilities amongst actions that might be taken when a choice is presented or confronted.

5.4 Probabilistic testing semantics

For probabilistic tests all we need change is that the user records not just a set of observations but a probability distribution over a set of observations, hence $\mathbb{O} \stackrel{\text{def}}{=} Act^* \to \mathbb{R}$.

The relational semantics of process A when probability distributions are observed is a subset of $\Xi \times (Act^* \to \mathbb{R})$. If a process is experimented upon (frequentist perspective) and the results noted then what is observed will be a function $\Xi \to (Act^* \to \mathbb{R})$ and hence there is no nondeterminism and no possibility of refinement.

But approaching automata from the Bayesian perspective, if we can define the processes and tests as prior "probabilistic" automata then we might be able to use probabilistic parallel composition to compute the probabilistic relational semantics of the processes. From the Bayesian point of view, the probabilities on actions are prior probabilities that, until the action takes part in an event by being synchronized with another process along the same action, do not play any role. Obviously the probability of an unexecuted action is prior to the probability of an execution—in particular, not until we factor in the probability of the synchronizing action do we know (via their product) what the probability of the executed event (denoted by τ) will be. So, it is the Bayesian ideas that allow us to make sense of attaching probabilities to something that has not yet happened, and which will only be a part of what happens.

6 Formalising probabilistic automata

In this section we will formalise the discussion in Section 5.2 and see that automata that contain both probabilistic and nondeterministic choice are called *partially probabilistic* introduced as parameterised probabilistic finite automata (PPFA). Here we take what we see as the standard statistical approach and model nondeterministic choice as probabilistic choice with unknown probability. So our probabilities are no longer only real numbers but may also be real-valued terms (parameterised terms, hence the name) that may contain variables, the unknown probabilities. Automata where nondeterminism has been completely replaced by probabilistic choice are *deterministic* probabilistic finite automata (DPFA).

Definition 5 *Parameterised Probabilistic Finite Automata (PPFA). Let N_{A} be a finite set of nodes. The parameterised probabilistic finite automaton A is given by the triple $(N_{\mathsf{A}}, S_{\mathsf{A}}, T_{\mathsf{A}})$ where*

1. *S_{A} is a "starting distribution", i.e. a parameterised probability distribution such that $dom(S_{\mathsf{A}}) \subseteq N_{\mathsf{A}}$, where $dom(S_A)$ are the starting states of* A
2. *$T_{\mathsf{A}} \subseteq \{(n, \mathsf{a}, d) | n \in N_{\mathsf{A}} \wedge \mathsf{a} \in Act^{\tau} \wedge d \in D_{\mathsf{A}}\}$, such that for each $n \in N_{\mathsf{A}}$ and $a \in Act^{\tau}$ there exists no more than one element of T_{A} with first component n and second component* a*, and recall that nondeterminism is modelled by a parameter in the range of the probability distribution d. Finally, D_A is a set of probability distributions over states.*

Deterministic Probabilistic Finite Automata (DPFA) are PPFA with the restrictions that:

1. *The ranges of all probability distributions are sets of real values, not sets of possibly parameterised terms, i.e. the elements of the ranges contain no variables;*
2. *$(n, \mathsf{a}, d) \in T_{\mathsf{A}}$ implies $\mathsf{a} \in Act$.*

□

Let the variables $\mathbb{X}, \mathbb{Y}$ be taken from some set *Var* and $\overline{\mathbb{X}}$ be a list of variables and $\psi_{\overline{\mathbb{X}}}$ be an instantiation of the variables in the list taken from the set of all such instantiations $\Psi_{\overline{\mathbb{X}}}$. We will write $\mathsf{A}(\overline{\mathbb{X}})$ for a PPFA containing variables $\overline{\mathbb{X}}$, but where not needed the list of variables will be dropped and we will write A. We interpret the variables in $\mathsf{A}(\overline{\mathbb{X}})$ as being *globally bound* and take the usual α-congruence of terms and identify PPFA that differ only by the names of variables used. Similarly we assume α-renaming to prevent confusion and variable capture when composing PPFAs.

We write $x \xrightarrow{\mathsf{a}}_{\mathsf{A},p} y$ for $(x, \mathsf{a}, d) \in T_{\mathsf{A}} \wedge d(y) = p$ and $x \xrightarrow{\mathsf{a}}_{p} y$ where A is obvious from context. In addition when we want to talk about a "complete" transition, i.e. one that has its associated final state distribution, we write $x \xrightarrow{\mathsf{a}}_{\mathsf{A}} d$ for $(x, \mathsf{a}, d) \in T_{\mathsf{A}}$.

Definition 6 *The probability of the computation following a path, a sequence of transitions starting from a start state s, is the product of the probability of its component transitions and the probability of starting in the start state $S_{\mathsf{A}}(s)$. Let p be the path $s \xrightarrow{\rho_1}_{p_1} m_1, m_1 \xrightarrow{\rho_2}_{p_2} m_2, \ldots m_{n-1} \xrightarrow{\rho_n}_{p_n} m_n$. Then the probability that p is executed is*

$$d(p) \stackrel{\text{def}}{=} S_{\mathsf{A}}(s) \times p_1 \times p_2 \times \ldots p_n$$

and we say that the path p can be observed as trace $\rho = \rho_1, \rho_2, \ldots, \rho_n$.

The probability of observing a trace ρ is the sum of the all probabilities of the computation following any path that can be observed as trace ρ:

$$d(\rho) = \sum_{tr(p_i)=\rho} d(p_i)$$

where $tr(p) \stackrel{\text{def}}{=} \{\rho | p = s \xrightarrow{\rho_1}_{p_1} m_1, m_1 \xrightarrow{\rho_2}_{p_2} m_2, \ldots m_{n-1} \xrightarrow{\rho_n}_{p_n} m_n\}$.

Writing $S_{\mathsf{A}} \xrightarrow{\rho}_{p}$ informs us that p is the probability of seeing the trace ρ when starting in any of the start states in $dom(S_{\mathsf{A}})$ and following some appropriate path, i.e. $d(\rho) = p$. $S_{\mathsf{A}} \xrightarrow{\rho}_{p} n$ means that p is the probability of seeing the trace ρ when starting in any of the start states in $dom(S_{\mathsf{A}})$ and ending in state n.

Definition 7 *The probability distribution over complete traces is*

$$D^c(\mathsf{A}) \stackrel{\text{def}}{=} \{\rho \mapsto \sum_{q \in P} q \mid P = \{q \mid n \in N_{\mathsf{A}} \wedge \pi(n) = \emptyset \wedge S_{\mathsf{A}} \xrightarrow{\rho}_{q} n\}\}$$

Definition 8 *Process operators*
Action Prefixing $\mathsf{a}.\mathsf{B} \stackrel{\text{def}}{=} (\{s_a\} \cup N_\mathsf{B}, \{s_a \mapsto 1\}, \{s_a \xrightarrow{\mathsf{a}} S_\mathsf{B}\} \cup T_\mathsf{B})$ *where* s_a *is a new state*
Internal choice $\mathsf{A} \sqcap \mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B}, S_\mathsf{A} \sqcap S_\mathsf{B}, T_\mathsf{A} \cup T_\mathsf{B})$ *where*
$(S_\mathsf{A} \sqcap S_\mathsf{B})(n) = \mathbb{X} \times S_\mathsf{A}(n)$ *if* $n \in dom(S_\mathsf{A})$ *else* $(1-\mathbb{X}) \times S_\mathsf{B}(n)$ *if* $n \in dom(S_\mathsf{B})$*, where* $\mathbb{X}$ *is a fresh parameter, and note that now* $dom(S_\mathsf{A} \sqcap S_\mathsf{B}) = dom(S_\mathsf{A}) \cup dom(S_\mathsf{B})$.
Probabilistic choice $\mathsf{A} \oplus_p \mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B}, S_\mathsf{A} \oplus_p S_\mathsf{B}, T_\mathsf{A} \cup T_\mathsf{B})$ *where* $(S_\mathsf{A} \oplus_p S_\mathsf{B})(n) = p \times S_\mathsf{A}(n)$ *if* $n \in dom(S_\mathsf{A})$ *else* $(1-p) \times S_\mathsf{B}(n)$ *if* $n \in dom(S_\mathsf{B})$*, and note that now* $dom(S_\mathsf{A} \oplus_p S_\mathsf{B}) = dom(S_\mathsf{A}) \cup dom(S_\mathsf{B})$. *We note immediately from this that internal choice is probabilistic choice with unknown probability between the two choices.*
External choice $\mathsf{A} \Box \mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B} \setminus (dom(S_\mathsf{A}) \cup dom(S_\mathsf{B})) \cup dom(S_{\mathsf{A}\Box\mathsf{B}}), S_{\mathsf{A}\Box\mathsf{B}}, T_\mathsf{A} \cup T_\mathsf{B}\{\{S_\mathsf{A}S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\}\})$ *where* $S_{\mathsf{A}\Box\mathsf{B}}(n_\mathsf{A}, n_\mathsf{B}) = S_\mathsf{A}(n_\mathsf{A}) \times S_\mathsf{B}(n_\mathsf{B})$ *and* $\{\{S_\mathsf{A}S_\mathsf{B}/S_\mathsf{A} \times S_\mathsf{B}\}\}$ *now, of course, uses the* domains *of the start state distributions in order to build the substitutions over start states.*
Parallel composition:

$$\mathsf{A} \parallel_P \mathsf{B} \stackrel{\text{def}}{=} (N_{\mathsf{A}\parallel_P\mathsf{B}}, S_{\mathsf{A}\parallel_P\mathsf{B}}, T_{\mathsf{A}\parallel_P\mathsf{B}})$$

$$N_{\mathsf{A}\parallel_P\mathsf{B}} = N_\mathsf{A} \times N_\mathsf{B}$$

$$S_{\mathsf{A}\parallel_P\mathsf{B}}(n_\mathsf{A}, n_\mathsf{B}) = S_\mathsf{A}(n_\mathsf{A}) \times S_\mathsf{B}(n_\mathsf{B}) \; if \; n_\mathsf{A} \in dom(S_\mathsf{A}) \wedge n_\mathsf{B} \in dom(S_\mathsf{B})$$

and $T_{\mathsf{A}\parallel_P\mathsf{B}}$ *is defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} d_\mathsf{A}, m \xrightarrow{\mathsf{x}}_\mathsf{B} d_\mathsf{B}, \mathsf{x} \in P}{(n,m) \xrightarrow{\tau}_{(\mathsf{A}\parallel_P\mathsf{B})} d_\mathsf{A} \times d_\mathsf{B}}$$

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} d_\mathsf{A}, (\mathsf{x} \notin P \wedge m \in N_\mathsf{B})}{(n,m) \xrightarrow{\mathsf{x}}_{(\mathsf{A}\parallel_P\mathsf{B})} d_\mathsf{A} \times m} \qquad \frac{n \xrightarrow{\mathsf{x}}_\mathsf{B} d_\mathsf{B}, (\mathsf{x} \notin P \wedge m \in N_\mathsf{A})}{(m,n) \xrightarrow{\mathsf{x}}_{(\mathsf{A}\parallel_P\mathsf{B})} m \times d_\mathsf{B}}$$

where

$$d_\mathsf{A} \times d_\mathsf{B} \stackrel{\text{def}}{=} \{(x,y) \mapsto d_\mathsf{A}(x).d_\mathsf{B}(y) | n \xrightarrow{\mathsf{x}}_A x \wedge m \xrightarrow{\mathsf{x}}_B y\}$$

and

$$d_\mathsf{A} \times m \stackrel{\text{def}}{=} \{(x,m) \mapsto d_\mathsf{A}(x) | n \xrightarrow{\mathsf{x}}_\mathsf{A} x\}$$

and

$$m \times d_\mathsf{B} \stackrel{\text{def}}{=} \{(m,y) \mapsto d_\mathsf{B}(y) | n \xrightarrow{\mathsf{x}}_\mathsf{B} y\}$$

Example 2 *Consider the PPFAs given by the expressions* $a.(Q_1 +_p Q_2)$ *and* $a.Q_1 +_p a.Q_2$*. Then, assuming the start states, states and transitions of* Q_1 *and* Q_2 *are given by* s_1*,* s_2*,* N_1*,* N_2*,* T_1 *and* T_2 *respectively, we have*

$$a.Q_1 +_p a.Q_2 = (\{t_1, t_2\} \cup N_1 \cup N_2, \{t_1 \mapsto p, t_2 \mapsto 1-p\},$$
$$\{t_1 \xrightarrow{\mathsf{a}} d_1, t_2 \xrightarrow{\mathsf{a}} d_2 | d_1(s_1) = d_2(s_2) = 1\} \cup T_1 \cup T_2)$$

$$a.(Q_1 +_p Q_2) = (\{t\} \cup N_1 \cup N_2, \{t \mapsto 1\},$$
$$\{t \xrightarrow{\mathsf{a}} d | d(s_1) = p, d(s_2) = 1-p\} \cup T_1 \cup T_2)$$

In fact, these PPFAs are indistinguishable by testing, so they are equal (they "refine both ways") as far as our testing semantics goes. This result can be generalised so that probability distributions on transitions can always be "migrated" to the starting state distribution.

6.1 Testing of probabilistic processes

Recall from Section 3 that we said in the definition of our testing semantics for FA that we will use $[\mathsf{A}]_{\mathsf{X}} = \mathsf{A} \parallel_N \mathsf{X}$ and $O_{FA} = Tr^c$. For probabilistic FAs we need to use parallel composition from Definition 8 (as defined for DPFA and PPFA). The observation of a single execution of a DPFA is still a trace but what can be "observed" over many executions is no longer simply a set of traces but, if we also record the frequency of occurrence of the traces, a probability distribution over the set of traces hence $O_{DPFA} = D^c$. We treat PPFA similarly and let $\Xi_{PPFA} = PPFA$ and $O_{PPFA} = D^c$ except that now the observed probability distributions may be parameterised.

Definition 9 *The relational semantics of an entity* $\mathsf{A}(\overline{\mathbb{X}})$ *is (where* $\Psi_{\overline{\mathbb{X}}}$ *is the set of instantiations for the parameters in* $\overline{\mathbb{X}}$*)*

$$[\![\mathsf{A}(\overline{\mathbb{X}})]\!]_{\Xi_{PPFA},D^c} \stackrel{\text{def}}{=} \{(x,o).x \in \Xi_{PPFA} \wedge o \in \psi_{\overline{\mathbb{X}}}(D^c(([\mathsf{A}(\overline{\mathbb{X}})]_x))) \wedge \psi_{\overline{\mathbb{X}}} \in \Psi_{\overline{\mathbb{X}}}\}$$

$$\mathsf{A}(\overline{\mathbb{X}}) \sqsubseteq_{\Xi_{PPFA},D^c} \mathsf{C}(\overline{\mathbb{Y}}) \stackrel{\text{def}}{=} [\![\mathsf{C}(\overline{\mathbb{Y}})]\!]_{\Xi_{PPFA},D^c} \subseteq [\![\mathsf{A}(\overline{\mathbb{X}})]\!]_{\Xi_{PPFA},D^c}$$

$$\mathsf{A}(\overline{\mathbb{X}}) =_{PPFA} \mathsf{C}(\overline{\mathbb{Y}}) \stackrel{\text{def}}{=} [\![\mathsf{C}(\overline{\mathbb{Y}})]\!]_{\Xi_{PPFA},D^c} = [\![\mathsf{A}(\overline{\mathbb{X}})]\!]_{\Xi_{PPFA},D^c}$$

Note here that we have given the meaning of PPFAs as a relation from contexts (PPFAs) to probability distributions:

$$[\![\mathsf{A}(\overline{\mathbb{X}})]\!]_{\Xi_{PPFA},D^c} \subseteq \Xi_{PPFA} \times (Act^* \to Real)$$

by instantiating all the open distributions that might be observed to get plain probability distributions "with no unknowns".

Let $\sqsubseteq_{PPFA} \stackrel{\text{def}}{=} \sqsubseteq_{\Xi_{PPFA},D^c}$. That is, we write $\sqsubseteq_{PPFA}$ for this general definition of refinement. When $\sqsubseteq_{PPFA}$ relates two DPFA processes it is of little interest, i.e. there are no opportunities for refinement as there is no nondeterminism (though there are, perhaps, probabilities).

In Section 7 we will show refinement of PPFA is strongly related to refinement of an underlying FA.

6.2 Simple results from the definitions

Theorem 2 *Refinement distributes through parallel composition. Let* X, Y, P *and* Q *be arbitrary PPFAs and let* $N \subseteq Act$*. Then*

$$\frac{\mathsf{X} \sqsubseteq_{PPFA} \mathsf{Y}, \mathsf{P} \sqsubseteq_{PPFA} \mathsf{Q}}{\mathsf{X} \parallel_N \mathsf{P} \sqsubseteq_{PPFA} \mathsf{Y} \parallel_N \mathsf{Q}}$$

For an arbitrary PPFA $\mathsf{P}(\overline{\mathbb{Y}})$ we have the following theorems.

Theorem 3 $\sqcap$ *is idempotent.* $\mathsf{P}(\overline{\mathbb{Y}}) =_{PPFA} \mathsf{P}(\overline{\mathbb{Y}}) \sqcap \mathsf{P}(\overline{\mathbb{Y}})$

Proof: From Definition 8 it can be seen that the graph of $\mathsf{P}(\overline{\mathbb{Y}}) \sqcap \mathsf{P}(\overline{\mathbb{Y}})$ consists of two copies of the graph of $\mathsf{P}(\overline{\mathbb{Y}})$ which ever copy is selected the behaviour is exactly that of $\mathsf{P}(\overline{\mathbb{Y}})$. Hence he equality.

Theorem 4 $\oplus_p$ *is idempotent* $\mathsf{P}(\overline{\mathbb{Y}}) =_{PPFA} \mathsf{P}(\overline{\mathbb{Y}}) \oplus_p \mathsf{P}(\overline{\mathbb{Y}})$

Proof: Similar to Theorem 3.

7 Relating finite automata to parameterised probabilistic finite automata

We construct $[\![\_]\!]^{FA}_{PPFA}$, an embedding of FA into PPFA and a forgetful mapping from PPFA to FA, and then show that these mappings form a Galois connection between the refinement relations $\sqsubseteq_{PPFA}$ and $\sqsubseteq_{FA}$.

The embedding $[\![\_]\!]^{FA}_{PPFA}$ of FA in PPFA will map all nondeterministic choices in FA processes into probabilistic choice with unknown probabilities in the PPFA processes.

Definition 10 *Semantic mappings $[\![\_]\!]^{FA}_{PPFA}$ and vA^{FA}_{PPFA} between finite automata A and parameterised probabilistic finite automata Ap are defined so that:*

$$[\![(N_{\mathsf{A}}, S_{\mathsf{A}}, T_{\mathsf{A}})]\!]^{FA}_{PPFA} \stackrel{\text{def}}{=} (N_{\mathsf{Ap}}, S_{\mathsf{Ap}}, T_{\mathsf{Ap}})$$

where

$$N_{\mathsf{Ap}} \stackrel{\text{def}}{=} N_{\mathsf{A}}$$

and

$$S_{\mathsf{Ap}} \stackrel{\text{def}}{=} \{(s, \mathbb{X}) \mid s \in S_{\mathsf{A}} \wedge \mathbb{X} \; is \; fresh \wedge (\Sigma_{n \in dom(S_{\mathsf{Ap}})} S_{\mathsf{Ap}}(n)) = 1\}$$

$$T_{\mathsf{Ap}} = \{(n, \mathsf{a}, d) \mid d = \{m \mapsto v \mid n \xrightarrow{\mathsf{a}} m \wedge v \; is \; fresh\} \wedge (\Sigma_{m \in dom(d)} d(m)) = 1\}$$

The mapping vA^{FA}_{PPFA} from PPFA in to FA forgets all probability distributions:

$$vA^{FA}_{PPFA}(N_{\mathsf{Ap}}, s_{\mathsf{Ap}}, T_{\mathsf{Ap}}) = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}})$$

where

$$N_{\mathsf{A}} \stackrel{\text{def}}{=} N_{\mathsf{Ap}}$$

and

$$S_{\mathsf{A}} \stackrel{\text{def}}{=} dom(S_{\mathsf{Sp}})$$

and

$$T_{\mathsf{A}} = \{(n, \mathsf{a}, m) | n \xrightarrow{\mathsf{a}}_{\mathsf{Ap}} d \wedge m \in dom(d)\}$$

□

The pair of mappings $([\![\_]\!]^{FA}_{PPFA}, vA^{FA}_{PPFA})$ define a vertical refinement $\sqsubseteq^{FA}_{PPFA}$ as they are a Galois connection [10]. This is the content of Theorem 7, but first some preliminary results.

Lemma 1 *For any FAs X and Y*
$Tr^c(\mathsf{X}) \subseteq Tr^c(\mathsf{Y}) \Rightarrow D^c([\![\mathsf{X}]\!]^{FA}_{PPFA}) \subseteq D^c([\![\mathsf{Y}]\!]^{FA}_{PPFA})$

Proof (Sketch) The application of $[\![\_]\!]^{FA}_{PPFA}$ to a FA simply adds parameterised probabilities spanning any nondeterministic choice. The set of all possible observation traces is $Tr^c(\mathsf{X})$. This is also the set of all possible observation traces of $[\![\mathsf{X}]\!]^{FA}_{PPFA}$ but now what is "observed" is not one trace but any probability distribution over any subset of $O(\mathsf{X})$ (we need to use subset as when the probability of observing a trace is 0 it is no longer in the domain of the distribution).

Hence $d \in D^c([\![\mathsf{X}]\!]^{FA}_{PPFA}) \Leftrightarrow dom(d) \subseteq Tr^c(\mathsf{X})$. Consequently if $d \in D^c([\![\mathsf{X}]\!]^{FA}_{PPFA})$ then $dom(d) \subseteq Tr^c(\mathsf{X})$ and since $Tr^c(\mathsf{X}) \subseteq Tr^c(\mathsf{Y})$, from the assumption of the lemma, we further have $dom(d) \subseteq Tr^c(\mathsf{Y})$. Then $d \in D^c([\![\mathsf{Y}]\!]^{FA}_{PPFA})$ follows from the argument above with Y in place of X. •

Theorem 5 *Let* X *and* Y *be FAs, and let* $N \subseteq Act$. *Then,*

$$[\![\mathsf{X} \parallel_N \mathsf{Y}]\!]^{FA}_{PPFA} = [\![\mathsf{X}]\!]^{FA}_{PPFA} \parallel_N [\![\mathsf{Y}]\!]^{FA}_{PPFA}$$

Theorem 6 *Let* X *and* Y *be PPFAs, and let* $N \subseteq Act$. *Then,*

$$vA^{FA}_{PPFA}(\mathsf{X} \parallel_N \mathsf{Y}) = vA^{FA}_{PPFA}(\mathsf{X}) \parallel_N vA^{FA}_{PPFA}(\mathsf{Y})$$

Definition 11 *Deterministic automata.*

$Det_{FA} \stackrel{\text{def}}{=} \{\mathsf{P} | (n \xrightarrow{\mathsf{a}} k \wedge n \xrightarrow{\mathsf{a}} l \Rightarrow k = l) \wedge |S_{\mathsf{A}}| = 1\}$

$Det_{PPFA} \stackrel{\text{def}}{=} \{\mathsf{P} | (n \xrightarrow{\mathsf{a}}_p k \wedge n \xrightarrow{\mathsf{a}}_q l \Rightarrow k = l \wedge p = q = 1) \wedge |S_{\mathsf{A}}| = 1\}$

Lemma 2 *Results involving deterministic automata.*

1. *(a)* $\{\mathsf{X} \in Det_{FA} \mid [\![\mathsf{X}]\!]^{FA}_{PPFA}\} = Det_{PPFA}$ *and*
   (b) $\{\mathsf{Y} \in Det_{PPFA} \mid vA^{FA}_{PPFA}(\mathsf{Y})\} = Det_{FA}$
2. *Let* A *and* C *be FAs. Then* $\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Leftrightarrow \forall_{x \in Det_{FA}}. Tr^c([\mathsf{A}]_x) \supseteq Tr^c([\mathsf{C}]_x)$
3. *Let* A *and* C *be PPFAs. Then* $\mathsf{A} \sqsubseteq_{PPFA} \mathsf{C} \Leftrightarrow \forall_{x \in Det_{PPFA}}. D^c([\mathsf{A}]_x) \supseteq D^c([\mathsf{C}]_x)$

Proof (Sketch).

$1(a)$ and $1(b)$ follow from definitions.

Re 2: With non-probabilistic processes and tests, what can be observed when applying a nondeterministic test is the union of what can be observed when applying each element of the set of deterministic alternatives (where here we picture, as usual, a nondeterministic computation as a set of deterministic ones which covers all the possible choices) and hence:

$$\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Leftrightarrow \forall_{x \in Det_{FA}}. Tr^c([\mathsf{A}]_x) \supseteq Tr^c([\mathsf{C}]_x)$$

Re 3: With probabilistic processes and tests, what can be observed when applying a probabilistic test is the distribution, inferred from the test, of what can be observed when applying the deterministic components that the probabilistic choice spans. Hence a set of test processes for PPFA that is sufficient to establish refinement is the image after applying $[\![\_]\!]^{FA}_{PPFA}$ to a sufficient set of FA processes, i.e. since Det_{FA} is sufficient for FA then Det_{PPFA} is sufficient for PPFA, hence:

$$\mathsf{A} \sqsubseteq_{PPFA} \mathsf{C} \Leftrightarrow \forall_{x \in Det_{PPFA}}. D^c([\mathsf{A}]_x) \supseteq D^c([\mathsf{C}]_x)$$

Theorem 7

$$\forall \mathsf{X} \in FA, \mathsf{Y} \in PPFA. [\![\mathsf{X}]\!]^{FA}_{PPFA} \sqsubseteq_{PPFA} \mathsf{Y} \Leftrightarrow \mathsf{X} \sqsubseteq_{FA} vA^{FA}_{PPFA}(\mathsf{Y})$$

Proof: (Sketch)

It is a well-known result (e.g. [11]) that to prove a Galois connection it is sufficient to prove for arbitrary X

$$vA^{FA}_{PPFA}([\![\mathsf{X}]\!]^{FA}_{PPFA}) \sqsubseteq_{FA} id_{FA}\mathsf{X}$$

and for arbitrary Y

$$[\![vA^{FA}_{PPFA}(\mathsf{Y})]\!]^{FA}_{PPFA} \sqsubseteq_{PPFA} id_{PPFA}\mathsf{Y}$$

and in addition to prove both relations $[\![\_]\!]^{FA}_{PPFA}$ and vA^{FA}_{PPFA} are monotone.

We can see directly from the definitions that $[\![\_]\!]^{FA}_{PPFA}$ adds parameterised probabilities to any nondeterministic choice and vA^{FA}_{PPFA} forgets this addition hence, for arbitrary X :

$$vA^{FA}_{PPFA}([\![\mathsf{X}]\!]^{FA}_{PPFA}) =_{FA} id_{FA}\mathsf{X}$$

which gives our first inequality.

The effect of $[\![vA^{FA}_{PPFA}\mathsf{Y}]\!]^{FA}_{PPFA}$ is to first replace probabilistic choice with nondeterministic choice (by ignoring probabilities) and then reintroducing probabilities-with-parameters due to the nondeterminism and this can be refined, along with other possibilities, back into its original value, which gives our second inequality.

Re: show $[\![\_]\!]^{FA}_{PPFA}$ is monotone: $\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Rightarrow [\![\mathsf{A}]\!]^{FA}_{PPFA} \sqsubseteq_{PPFA} [\![\mathsf{C}]\!]^{FA}_{PPFA}$

From Definition 3 we have $\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Leftrightarrow \forall_{x\in\Xi_{FA}}.Tr^c([\mathsf{A}]_x) \supseteq Tr^c([\mathsf{C}]_x)$ and as $Det_{FA} \subseteq \Xi_{FA}$ we also have

$\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Leftrightarrow \forall_{x\in Det_{FA}}.Tr^c([\mathsf{A}]_x) \supseteq Tr^c([\mathsf{C}]_x)$ (1)

From Lemma 1 we then have

$\mathsf{A} \sqsubseteq_{FA} \mathsf{C} \Rightarrow \forall_{x\in Det_{FA}}.D^c([\![[\mathsf{A}]_x]\!]^{FA}_{PPFA}) \supseteq D^c([\![[\mathsf{C}]_x]\!]^{FA}_{PPFA})$.

Then,

$\forall_{x\in Det_{FA}}.D^c([\![[\mathsf{A}]_x]\!]^{FA}_{PPFA}) \supseteq D^c([\![[\mathsf{C}]_x]\!]^{FA}_{PPFA})$

$\forall_{x\in Det_{FA}}.D^c([[\![\mathsf{A}]\!]^{FA}_{PPFA}]_{[\![x]\!]^{FA}_{PPFA}}) \supseteq D^c([[\![\mathsf{C}]\!]^{FA}_{PPFA}]_{[\![x]\!]^{FA}_{PPFA}})$ from Theorem 5

$\forall_{x\in Det_{PPFA}}.D^c([[\![\mathsf{A}]\!]^{FA}_{PPFA}]_x) \supseteq D^c([[\![\mathsf{C}]\!]^{FA}_{PPFA}]_x)$ Lemma 2 part 1(a)

$[\![\mathsf{A}]\!]^{FA}_{PPFA} \sqsubseteq_{PPFA} [\![\mathsf{C}]\!]^{FA}_{PPFA}$ from Definition 9

4. Re: show vA^{FA}_{PPFA} is monotone: $\mathsf{A} \sqsubseteq_{PPFA} \mathsf{C} \Rightarrow vA^{FA}_{PPFA}\mathsf{A} \sqsubseteq_{FA} vA^{FA}_{PPFA}\mathsf{C}$

From $\mathsf{A} \sqsubseteq_{PPFA} \mathsf{C}$ and definitions we have: $\forall_{x\in\Xi_{PPFA}}.D^c([\mathsf{A}]_x) \supseteq D^c([\mathsf{C}]_x)$

as $Det_{PPFA} \subset \Xi_{PPFA}$ we have

$\forall_{x\in Det_{PPFA}}.D^c([\mathsf{A}]_x) \supseteq D^c([\mathsf{C}]_x)$ (2)

For all o in $Tr^c(vA^{FA}_{PPFA}([\mathsf{C}]_x))$ there must exist a d in $D^c([\mathsf{C}]_x)$ such that $o \in dom(d)$ and from (2) we know that d is in $D^c([\mathsf{A}]_x)$ and with $o \in dom(d)$ we can conclude that o in $Tr^c(vA^{FA}_{PPFA}([\mathsf{A}]_x))$ so:

$\forall_{x\in Det_{PPFA}}.Tr^c(vA^{FA}_{PPFA}([\mathsf{A}]_x)) \supseteq Tr^c(vA^{FA}_{PPFA}([\mathsf{C}]_x))$

$\forall_{x\in Det_{PPFA}}.Tr^c([vA^{FA}_{PPFA}\mathsf{A}]_{vA^{FA}_{PPFA}x}) \supseteq Tr^c([vA^{FA}_{PPFA}\mathsf{C}]_{vA^{FA}_{PPFA}x})$ Theorem 5

$\forall_{x\in Det_{FA}}.Tr^c([vA^{FA}_{PPFA}\mathsf{A}]_x) \supseteq Tr^c([vA^{FA}_{PPFA}\mathsf{C}]_x)$ from Lemma 2 part 1(b)

$\forall_{x\in\Xi_{FA}}.Tr^c([vA^{FA}_{PPFA}\mathsf{A}]_x) \supseteq Tr^c([vA^{FA}_{PPFA}\mathsf{C}]_x)$ from Lemma 2 part 3

$vA^{FA}_{PPFA}\mathsf{A} \sqsubseteq_{FA} vA^{FA}_{PPFA}\mathsf{C}$ Definition 3

•

The embedding $[\![\_]\!]^{FA}_{PPFA}$ can be used to add probability to a non-probabilistic finite automata during the stepwise development, i.e. refinement, of a model or specification. This use of Galois connections is nothing new but to the best of our knowledge it is the first time it has been used to allow the introduction of probability part of the way through the development of a process.

8 Conclusions

Others have used the same testing framework to treat probabilistic processes, but in one notable case [9] it was found that many of the expected algebraic results were false according to the testing used. This meant the abandonment of testing as a basis for refinement and a notion of simulation was introduced. We believe that the reason that many of the "sanity checks" turned out to be false for the testing-based refinement in that paper was that the original formalisation of nondeterminism found in non-probabilistic

systems was kept and that this led to problems when probabilistic tests on nondeterministic probabilistic systems were considered.

Instead of abandoning refinement based on testing, we handle nondeterminism in a way that is compatible with probability, rather than using the original formalisations of testing nondeterminism found in non-probabilistic systems.

We also note that, having shown we can (as a (vertical) refinement) move from non-probabilistic models to probabilistic ones (and back again, if we wish), the introduction of probabilities can happen as a design step *during* development of a system via refinement steps. So, we are free to take a very general non-probabilistic specification and, if it turns out to be necessary to do so to deal with some aspects of the specification, introduce probabilities as we make progress towards a more concrete form of the system. We have not yet explored this possibility, but it does introduce another freedom to the developer which might turn out to be useful.

The framework we have introduced in this paper is really only a first step towards a sensible language for specifying systems containing probability. What still needs to be done is to recognise that some sorts of probabilistic choice do not "make sense", i.e. that there are right and wrong places to use such choice. For example, if we have a vending machine with two buttons on, one for tea and one for coffee, it clearly does not make sense to specify the choice here as a probabilistic one—the vending machine would be a very odd one if it allowed me to choose tea only 75% of the time!

On the other hand, it does make sense (though perhaps inventing plausible uses for such a thing might be hard!) to specify a robot which can make choices from a vending machine that offers tea or coffee, where the robot prefers tea over coffee, so it chooses tea 75% of the time.

The difference between these two cases is one of *causality*. The robot's actions cause the vending machine's, and not *vice versa*. So, our specification language would need to allow us to make this distinction and, most helpfully, only allow probabilistic choice to be specified in situations where it makes sense, as in the case of specifying the robot. We have done previous work on adding causality (back) into process algebras, and the work presented here forms the basis for a probabilistic causal process algebra (CPA) [12], or for a probabilistic language for interactive branching processes (IBPs) [10] which we have also talked about before, which forms the subject of another paper yet to be published.

A final interesting point to note is that, because we can always migrate probabilities on actions right up the probabilities on start states, we have a normal form for our automata. In this form, the only place that probabilities appear is on the start states (so the only non-trivial probabilistic distribution over states is the start-state distribution). This makes it very clear that one needs only one roll (of dice with enough faces) in order to conduct a probabilistic computation.

References

[1] Hennessy, M.: Algebraic Theory of Processes. The MIT Press (1988)

[2] López, N., Núñez, M.: An overview of probabilistic process algebras and their equivalences. In Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P., Siegle, M., eds.: Validation of Stochastic Systems. Volume 2925 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 283–298 doi:10.1007/978-3-540-24611-4_3

[3] Reeves, S., Streader, D.: Unifying state and process determinism. Technical report, University of Waikato, http://hdl.handle.net/10289/1001 (2004)

[4] Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science 18 (1990)

[5] Winskel, G., Nielsen, M.: Models for concurrency. Technical Report DAIMI PB 429, Computer Science Dept. Aarhus Universty (1992)

[6] Reeves, S., Streader, D.: Guarded operations, refinement and simulation. In: Proc Fourteenth BAC-FACS Refinement Workshop (REFINE 2009). Volume 259 of Electronic Notes in Theoretical Computer Science., Eindhoven, The Netherlands, Elsevier (2009) 177–191 doi:10.1016/j.entcs.2009.12.024

[7] Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)

[8] Reeves, S., Streader, D.: Atomic Components. In Liu, Z., Araki, K., eds.: Theoretical Aspects of Computing - ICTAC 2004: First International Colloquium. Volume 3407 of Lecture Notes in Computer Science., Springer-Verlag (2004) 128–139

[9] Deng, Y., van Glabbeek, R., Hennessy, M., Morgan, C., Zhang, C.: Remarks on testing probabilistic processes. Electron. Notes Theor. Comput. Sci. **172** (2007) 359–397 doi:10.1016/j.entcs.2007.02.013

[10] Reeves, S., Streader, D.: Contexts, refinement and determinism. Science of Computer Programming, DOI: 10.1016/j.scico.2010.11.011 (2010) doi:10.1016/j.scico.2010.11.011

[11] Taylor, P.: Practical Foundations of Mathematics. Cambridge University Press (1999) Cambridge studies in advanced mathematics 59.

[12] Reeves, S., Streader, D.: Causal process algebra: always getting the right drink from a coffee machine. In: Proc EPSRC RefineNet Refinement Workshop, Seventh International Conference on Formal Engineering Methods (ICFEM 2005). (2005) 1–14

Model exploration and analysis for quantitative safety refinement in probabilistic B

Ukachukwu Ndukwu* and Annabelle McIver †
Department of Computing, Macquarie University, NSW 2109 Australia.
{ukachukwu.ndukwu,annabelle.mciver}@mq.edu.au

The role played by counterexamples in standard system analysis is well known; but less common is a notion of counterexample in probabilistic systems refinement. In this paper we extend previous work using counterexamples to inductive invariant properties of probabilistic systems, demonstrating how they can be used to extend the technique of bounded model checking-style analysis for the refinement of quantitative safety specifications in the probabilistic B language. In particular, we show how the method can be adapted to cope with refinements incorporating probabilistic loops. Finally, we demonstrate the technique on pB models summarising a one-step refinement of a randomised algorithm for finding the minimum cut of undirected graphs, and that for the dependability analysis of a controller design.

Keywords Probabilistic B, quantitative safety specification, refinement, counterexamples.

1 Introduction

The B method [1] and more recently its successor Event-B [2] comprises a method and its automation for modelling complex software systems. It is based on the top-down refinement where specifications can be elaborated with detail and additional features, whilst the automated prover checks consistency between the refinements. Hoang's probabilistic B or pB [15] extension of standard B gave designers the ability to refer to probability and access to the specification of quantitative safety properties.

In probabilistic systems, the generalisation of traditional safety properties allows the specification of random variables whose expected value must always remain above some given threshold. Elsewhere [23, 25] we have provided automation to check this requirement by analysing pB models using an automatic translation of their quantitative safety specifications as PRISM reward structures [14]. Our technique allows pB modellers to explore the quantitative safety properties encoded within their models to obtain diagnostic feedback in the form of counterexample traces in the case that their model does not satisfy the quantitative specification. Counterexamples become sets of execution traces each with some probability of occurring and jointly implying that the specified threshold is not maintained. Moreover pB's consistency checking enforces inductive invariance of the quantitative safety property, thus the counterexample traces also demonstrate specific points in the models execution where the inductive property fails.

The paradigm of abstraction and refinement supports stepwise development of probabilistic systems aimed at improving probabilistic results. Unfortunately, for quantitative safety specifications (our focus here), a human verifier has no way of inspecting that this requirement is met even though the automated

*This author acknowledges support from the Australian Commonwealth Endeavor International Postgraduate Research Scholarship (E-IPRS) Fund.
†This author acknowledges support from the Australian Research Council (ARC) Grant Number DP0879529.

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 101–120, doi:10.4204/EPTCS.55.7

prover readily establishes consistency between the refinements. One way to resolve this uncertainty is to explore algorithmic approaches similar to probabilistic model checking techniques which can provide exact diagnostics summarising the failure (if indeed it exists) of the refinement goal.

In this paper we extend some practical uses of counterexamples to probabilistic systems refinement with respect to quantitative safety specifications particular to the pB language. We show how to use them to generalise bounded model checking-style analysis for probabilistic programs so that an iteration can be verified by exhaustive search provided that quantitative invariants are inductive for all reachable states. We also show how the use of probabilistic counterexamples in quantitative dependability analysis can be used to determine "failure modes" and "critical sets" which thus enables their extension to estimating components severity.

We illustrate the techniques on two case studies: one based on a probabilistic algorithm [20] to find the minimum cut set in a graph, and the other a probabilistic design for a controller mechanism [11].

The outline of the paper is as follows. In Sec.2 we summarise the underlying theory of pB; in Sec.3 we discuss the probabilistic counterexamples we can derive from the models and a bounded model checking approach to probabilistic iteration. In Sec.5 we illustrate the technique on the specification of a randomised "min-cut". We discuss probabilistic diagnostics of dependability in Sec.6 and demonstrate with a case study in Sec.7. We discuss related work and then conclude.

1.0.1 Notation

Function application is represented by a dot, as in $f.x$ (rather than $f(x)$). We use an abstract finite state space S. Given predicate *pred* we write lift*pred* for the *characteristic* function mapping states satisfying *pred* to 1 and to 0 otherwise, punning 1 and 0 with "True" and "False" respectively. We write $\mathscr{E}S$ as the set of real-valued functions from S, *i.e.* the set of expectations; and whenever $e, e' \in \mathscr{E}S$ we write $e \Rrightarrow e'$ to mean that $(\forall s \in S.\ e.s \leq e'.s)$. We let $\mathbb{D}S$ be the set of all discrete probability distributions over S; and write $Exp.\delta.e = \sum_{s \in S} (\delta.s) \times e.s$ for the expected value of e over S where $\delta \in \mathbb{D}S$ and $e \in \mathscr{E}S$. Finally we write S^* for the finite sequences of states in S.

2 Probabilistic annotations

When probabilistic programs execute they make random updates; in the semantics that behaviour is modelled by discrete probability distributions over possible final values of the program variables. Given a program *Prog* operating over S we write $[\![Prog]\!] : S \to (S \to [0,1])$ for the semantic function taking initial states to distributions over final states. For example, the program fragment

$$pInc \ \triangleq\ s{:=}s{+}1 \ {}_p\oplus\ s{:=}s{-}1 \tag{1}$$

increments state variable s with probability p, or decrements it with probability $1-p$. The semantics $[\![pInc]\!]$ for each initial state s is a probability distribution returning p or $(1-p)$ for (final) states $s' = (s+1)$ or $s' = (s-1)$ respectively. Rather than working with this semantics directly, we shall focus on the dual logical view generalisation of Hoare logic [16].

Probabilistic Hoare logic [22] takes account of the probabilistic judgements that can be made about probabilistic programs, in particular it can express when predicates can be established only *with some probability*. However, as we shall see, it is even more general than that, capable of expressing general expected properties of random variables over the program state. We use *Real*-valued annotations of the

Name	*Prog*	*Wp.Prog.Expt*
identity	**skip**	$Expt$
assignment	$x := f$	$Expt[x := f]$
composition	$Prog; Prog'$	$Wp.Prog.(wp \cdot Prog' \cdot Expt)$
choice	$Prog \lhd G \rhd Prog'$	$Wp.Prog.Expt \lhd G \rhd Wp.Prog'.Expt$
probability	$Prog \ {}_p\!\oplus \ Prog'$	$Wp.Prog.Expt \ {}_p\!\oplus \ Wp.Prog'.Expt$
nondeterminism	$Prog \sqcap Prog'$	$Wp.Prog.Expt$ min $Wp.Prog'.Expt$
weak iteration	it $Prog$ ti	$\nu X \bullet (Wp.Prog.X$ min $Expt)$

Given a program command *Prog* and expectation *Expt* of type $\mathscr{E}S$, *Wp.Prog* is of type $\mathscr{E}S \to \mathscr{E}S$. Note also that we write $Exp.([\![Prog]\!].s).Expt$ to mean $Wp.Prog.Expt.s$.

Figure 1: Structural definition of the expectation transformer-style semantics.

program variables interpreted as expectations; a program annotation is said to be valid exactly when the expected value over the post-annotation is at least the value given by the pre-annotation. In detail

$$\{pre\} \ \ Prog \ \ \{post\} \ , \tag{2}$$

is valid exactly when $Exp.[\![Prog]\!].post.s \geq pre.s$ for all states $s \in S$, where *post* is interpreted as a random variable over final states and *pre* as a real-valued function.

With our notational convention, a correct annotation for *pInc* (at (1)) is given by the triple

$$\{p \times \mathsf{lift}(s = -1) + (1-p) \times \mathsf{lift}(s = 1)\} \quad pInc \quad \{\mathsf{lift}(s = 0)\} \ , \tag{3}$$

which expresses the probability of establishing the state $s = 0$ finally, depending on the initial state from which *pInc* executes. Thus if the initial state is $s = -1$ then that probability is p, but it is $(1-p)$ if the initial state is $s = 1$.

Rather than use the distribution-centered semantics outlined above, we shall use a generalisation of Dijkstra's weakest precondition or *Wp* semantics defined on the program syntax of the probabilistic Guarded Command Language or *pGCL* [22]. The semantics of the language is set out in Fig. 1. As for standard *Wp* this formulation allows annotations to be checked mechanically [15, 17]; moreover we see that annotation (2) is valid exactly when $pre \Rrightarrow Wp.Prog.post$.

In this paper we shall concentrate on certifying probabilistic safety expressible using probabilistic annotations. Informally, a probabilistic safety property is a random variable whose expected value cannot be decreased on execution of the program. (This idea generalises standard safety, where the *truth* of a safety predicate cannot be violated on execution of the program.) Safety properties are characterised by *inductive invariants*: for example the valid annotation $\{Expt \times \mathsf{lift} pred\}$ *Prog* $\{Expt\}$ says that *Expt* is an inductive invariant for *Prog* provided it is executed in an initial state satisfying *pred*. To illustrate, the annotation

$$\{s\} \quad pInc \quad \{s\} \ , \tag{4}$$

means that the expected value of s is never decreased (and it is therefore only valid if $p \geq 1/2$).

Inductive invariants will be a significant component of the refinement of quantitative safety specifications in our pB machines, to which we now turn.

MACHINE	Faulty
SEES	Int_TYPE, Real_TYPE
CONSTANTS	p
PROPERTIES	$p \in REAL \wedge p \geq real(0) \wedge p \leq real(1)$
VARIABLES	cc
INVARIANT	$cc \in \mathbb{N}$
INITIALISATION	$cc := 0$
OPERATIONS	
	OpX $\triangleq$ **BEGIN**
	PCHOICE p **OF** $cc := cc+1$
	OR $cc := cc-1$ **END;**
	OpY $\triangleq$ $cc := 0$
EXPECTATIONS	$real(0) \Rrightarrow cc$
END	

Bold texts on the left column capture the fields (or clauses) used to describe the machine. The **PCHOICE** keyword introduces a probabilistic binary operator; the **EXPECTATIONS** clause expresses the notion of probabilistic quantitative safety.

Figure 2: A simple pB machine.

2.1 Probabilistic safety and refinement in pB

Probabilistic B or pB [15], is an extension of standard B [1] to support the specification and refinement of probabilistic systems. Systems are specified by a collection of *pB machines* which consist of operations describing possible program executions, together with variable declarations and invariants prescribing correct behaviour.

The machine set out in Fig. 2 illustrates some key features of the language. There are two operations –*OpX* and *OpY*– which can update a variable *cc*. *OpX* can either increment *cc* by 1 or decrement it by the same value with probability p or $(1-p)$ respectively, while *OpY* just resets the current value of *cc* to 0. In general, operations can execute only if their preconditions hold. But in the absence of preconditions as in this case, the choice of which operation to execute is made nondeterministically.

The remaining clauses ascribe more information to the variables, constants and behaviour of the operations. Declarations are made in the CONSTANTS and VARIABLES clauses; PROPERTIES and SEES clauses state assumed properties and context of the constants and variables. The INVARIANT clause sets out invariant properties. The expression in the INITIALISATION clause must establish the invariant and the operations *OpX* and *OpY* must maintain it afterwards.

We shall concentrate on the EXPECTATIONS clause[1], which was introduced by Hoang [15] to express quantitative invariant or safety properties. The form of an EXPECTATIONS clause is given by

$$E \Rrightarrow Expt\,, \tag{5}$$

where both E and *Expt* are expectations. It specifies that the expected value of *Expt* should always be *at least* E, where the expected value is determined by the distribution over the state space after any valid execution of the machine's operations, following its initialisation. Hoang showed that this is guaranteed by the following valid annotations:

[1]However, Hoang [15] showed that another way to check that a real-value Ω is indeed an expectation is to evaluate the language-specific boolean function $expectation(\Omega)$. Therefore we shall interchangeably use both forms to denote expectations-based expressions with no loss of generality.

$$\{E\} \quad init \quad \{Expt\} \qquad and \qquad \{\mathsf{lift}pred \times Expt\} \quad Op \quad \{Expt\}\,, \tag{6}$$

where *Op* is any operation with precondition *pred* and *init* is the machine's initialisation. In what follows we shall refer to (6) as the *proof obligations* for the associated expectations clause (5).

Checking the validity of program annotation, and in particular inductive invariants for loop-free program fragments can be done mechanically based on the semantics set out in Fig. 1. In some cases the proof obligation cannot be discharged, and there are two possible reasons for this. The first possibility is that *Expt* is too weak to be an inductive invariant for the machine's operations, and must be strengthened by finding $Expt' \Rrightarrow Expt$ so that the original safety property can be validated. The second possibility is that the machine's operations actually violate the probabilistic safety property.

The same reasoning can be extended to refinement of abstract pB machines. We note that quantitative safety specifications in pB can also be refined in the usual way with respect to expectation pairs. Thus another way of expressing (5) is to say that any program command *P* satisfies the bounded expectation pair $[E, Expt]$ if execution from its initial state guarantees that

$$E \Rrightarrow Wp.P.Expt. \tag{7}$$

Refinement is then implied by the ordering of program commands so that more refined programs improve probabilistic results. More specifically, we write

$$P \sqsubseteq Q \quad iff \quad (\forall E \in \mathscr{E}S \cdot Wp.P.E \Rrightarrow Wp.Q.E), \tag{8}$$

to mean that the program command *Q* is a refinement of the program command *P*. In addition we note that the preservation of an expression like (5) is implied by the *monotone* property of *Wp*.

The refinement of abstract pB machines embedding quantitative safety statements is dealt with in the language framework by introducing the IMPLEMENTATION and REFINES clauses. The former clause specifies the refinement of an abstract machine specified in the latter clause. The refinement process is then aimed at preserving the bounds of expectations in the original specification statement (the machine to be refined) so that the validity of an expression like (6) can be checked mechanically.

Our aim in the next section is to use probabilistic counterexamples adopted in model checking techniques to interpret failure of proofs of refinement of probabilistic machines in the pB language. We will find that a counterexample is a trace (or a set of traces) from the initialisation to a state where the inductive invariant fails to hold after inspecting the EXPECTATIONS clause over the refinement.

3 Probabilistic safety in Markov Decision Processes

In abstract terms *pGCL* programs and pB machines may be modelled as a Markov Decision Process (*MDP*). Recall that an *MDP* combines the notion of probabilistic updates together with some arbitrary choice between those updates [27]: that combination of probabilistic choices together with nondeterministic choices is present in *pGCL* and captures both features.

In this section we summarise pB models[2] and their quantitative safety specifications in terms of *MDP*s, and show how to apply model checking's search techniques for counterexamples to prove quantitative safety as a first step towards generalising standard bounded model checking verification. Inductive invariance is then crucial to the application of exhaustive state exploration for the intended goal.

[2]We note that an abstract pB model begins with the MACHINE keyword while a refinement is a pB model that begins with the IMPLEMENTATION keyword.

Here we consider an *MDP* expressed as a nondeterministic selection $P \triangleq P_0 \sqcap \ldots \sqcap P_n$ of deterministic *pGCL* programs, where the nondeterminism corresponds to the arbitrary choice, and each P_i corresponds to the probabilistic update for a choice i. When P is iterated for some arbitrarily-many steps, we identify a *computation path* as a finite sequence of states $\langle s_0, s_1, s_2, \ldots, s_n \rangle$ where each (s_i, s_{i+1}) is a probabilistic transition of P, *i.e.* s_{i+1} can occur with non-zero probability by executing P from s_i. Note that the choice (between $0 \ldots n$) can depend on the previous computation path since for example guards for the individual operations P_i must hold for their selection to be enabled.

Standard safety properties identify a set of "safe" states — the safety property then holds provided that all states reachable from the initial state under specified state transitions are amongst the selected safe states. A generalisation of this for probabilistic systems specifies thresholds on the probability for which the reachable states are always amongst the safe states. The quantitative safety properties encapsulated by the EXPECTATIONS clause are even more general than that, allowing the possibility to specify thresholds on arbitrary expected properties. The next definition sets out the mathematical model for interpreting general quantitative safety properties.

Since *MDP*s contain both nondeterministic and probabilistic choice, taking expected values only makes sense over well-defined probability distributions — we need to resolve the nondeterministic choice in all possible ways to yield a set of probability distributions. The next definition sets out a mechanism for doing just that.

Definition 1 *Given a program P, an* execution schedule *is a map $\aleph : S^* \to \mathbb{D}S$ so that $\aleph.\alpha \in [\![P]\!].s$ picks a particular resolution of the nondeterminism in P to execute after the trace α, where s is the last item of α. (A more uniform formalisation would give the distribution of initial states as $\aleph.\langle\rangle$; but we prefer to give initial states explicitly.)*

Once a particular schedule has been selected, the resulting behaviour generates a probability distribution over computation path. We call such a distribution a *probabilistic computation tree*; such distributions are well-defined with respect to Borel algebras based on the traces.

Definition 2 *Given a program P, initial state s_0 and execution schedule $\aleph$, we define the corresponding trace distribution $\langle\!|P_\aleph|\!\rangle.s_0$ of type $S^* \to [0,1]$ to be*

$$\begin{array}{rcl} \langle\!|P_\aleph|\!\rangle.s_0.(s') & \triangleq & 1 \textit{ if } s' = s_0 \textit{ else } 0 \\ \textit{and} \quad \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s s') & \triangleq & \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s) \times \aleph.(\alpha s).s' \end{array}$$

Computation trees of finite depth generate a *distribution over endpoints* as follows. If we take K steps from some initial s_0 according to the schedule $\aleph$, then the probability of ending in state s' is given by

$$[\![P_\aleph^K]\!].s_0.s' \triangleq \sum_{|\alpha|=K} \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s') \ .$$

General quantitative safety properties are intuitively specified via a numeric threshold e and a random variable *Expt* over the state space S: the expected value of *Expt* with respect to any distribution over endpoints should never fall below the threshold e.

Definition 3 *Given threshold e and an expectation Expt the* general quantitative safety *property is satisfied by the program P if for all schedules $\aleph$ and $K \geq 0$, we have that $Exp.[\![P_\aleph^K]\!].Expt.s_0 \geq e$.*

The probabilistic Computation Tree Logic or *pCTL* [13] safety property, which places a threshold on the probability that the reachable states always satisfy the identified "safe" states is expressible using

Def. 3 via characteristic expectation $\mathsf{lift} sa f e$. However many more general properties are also expressible, including expected time complexity [14].

We shall be interested in identifying situations where the inequality in Def. 3 does not hold. Evidence for the failure is a (finite) computation tree whose distribution over endpoints illustrates the failure to meet the threshold.

Definition 4 *Given a probabilistic safety property, a failure tree is defined by a scheduler* $\aleph$ *and an integer* $K \geq 0$ *such that* $Exp.[\![P^K_\aleph]\!].Expt.s_0 < e$.

Elsewhere [24] we showed that if *Expt* is an inductive invariant, then the safety property based on *Expt* is implied, provided that $e \leq Expt.s_0$. In fact, given a failure tree, there must be some finite trace α such that $\langle\!|P_\aleph|\!\rangle.s_0.(\alpha s) > 0$ and $Wp.(P \sqcap \mathbf{skip}).Expt.s < Expt.s$ [24]. Thus, as for standard model checking, we are able to locate specific traces which lead to the failure of the invariant property. We define a counterexample to *inductive invariance* as follows.

Definition 5 *Given a scheduler* $\aleph$*, an expectation Expt and a program P, a counterexample to inductive invariance safety property is a trace* (αs) *which can occur with non-zero probability, and such that* $Wp.P.Expt.s < Expt.s$*. A state such as s is a witness to failure.*

But note that in practice there will be a number of counterexamples. Our technique is able to identify them all given any depth K of computation. Next we discuss how the strategy can be extended to probabilistic loops reasoning.

3.1 Analysis of loops

We assume a loop of the form $loop \triangleq$ **while** G **do** *body* **od** where G is a predicate over the program state representing the loop guard; *body* is a probabilistic program consisting of a finite nondeterministic choice over probabilistic updates. Our aim in this section is to generalise the technique of bounded model checking to prove the safety assertion of the form

$$\{e\} \;\; loop \;\; \{inv\} \,. \tag{9}$$

In the case that (9) does not hold there must be a failure tree (Def. 4) to witness that fact, together with a set of failures to inductive invariance of *inv*. We shall be interested in the complementary problem, in the case that the property does hold. For standard programs this can be established by exhaustively searching the reachable states; any revisiting of a state terminates the search at that point, so that the method is complete for finite state programs: either a counterexample is discovered or all reachable states are visited, and each one checked for satisfaction of the (qualitative) safety property.

The situation is not quite so straightforward for probabilistic programs, and that is because the technique of exhaustive search does not generalise immediately to quantitative safety properties. However *via* inductive invariants it does. Consider the program which repeatedly sets a variable x uniformly in the set $\{0,1,2\}$ after the initialisation $x := 1$, and terminates whenever x is set to 2. In this case we might like to verify the safety property that $x \in \{1,2\}$ with probability at least $1/2$. Expressed as an assertion, it becomes

$$\{1/2\} \qquad x := 1; \mathbf{while}\ (x = 1)\ \mathbf{do}\ \ x{:=}0 \,{}_{1/3}\!\oplus (x{:=}1 \,{}_{1/2}\!\oplus x{:=}2)\ \mathbf{od} \qquad \{post\}\,, \tag{10}$$

where $post \triangleq \{\mathsf{lift}(x \in \{1,2\})\}$. A *quantitative inductive invariant* establishing that fact is given by $x/2$, expressing the probability that the safety property is always satisfied at that state. (When x is 2 that

probability is 1, when x is 1, it is $1/2$ and when x is 0 it is 0.) In fact the property (10) is equivalently formulated by setting $post \triangleq x/2$, which can be seen as a strengthening of $\{\mathsf{lift}(x \in \{1,2\})\}$.

Since the triple (10) does indeed hold, no failure trees exist; more generally, in standard model checking and for finite state spaces such a failure to establish the presence of a failure tree can be converted to a proof that the property holds (provided all reachable states are examined). For probabilistic systems however, it is not clear when to terminate a state exploration, since $Exp.[\![body^K_{\aleph}]\!].x/2$ steadily approaches $1/2$ from above (where here *body* is taken to be the guarded loop body of (10)). However we can recover the termination property even for probabilistic systems by looking at inductive invariants, as the next lemma shows.

Lemma 1 *Let P be a probabilistic program operating over a finite state space S; let s_0 be the initial state. If for all states s, reachable from s_0 under executions via P, the inductive invariance property $Wp.P.inv.s \geq inv.s$ holds, then $Exp.[\![P^K_{\aleph}]\!].inv \geq inv.s_0$ for all K and schedules $\aleph$.*

Proof 1 *(Sketch) We use proof by induction on K.*

When $K = 1$ we note that $Exp.[\![P^1_{\aleph}]\!].inv \geq inv.s_0$ is a consequence of the assumption since $Exp.[\![P^1_{\aleph}]\!].inv \geq Wp.P.inv.s_0$.

For the general step, we observe similarly that $Exp.[\![P^{K+1}_{\aleph}]\!].inv \geq Exp.[\![P^K_{\aleph}]\!].(Wp.P.inv)$. The result follows through monotonicity of the expectation operator.

Lem. 1 implies that we can use exhaustive search to verify quantitative safety properties using inductive invariants and exhaustive state exploration. The search terminates once all reachable states have been verified as satisfying the inductive property. In the case of (10), using $x/2$ for the invariant, each of the three states satisfies the inductive property. Next we summarise a prototype tool framework for locating and presenting counterexamples.

4 Automating counterexamples generation

YAGA [25] is a prototype suite of programs for inspecting safety specifications of abstract pB machines and their refinements. Importantly, it allows a pB machine designer to explore experimentally the details of system construction in order to ascertain the cause(s) of failure of a pB safety encoding as in (5).

YAGA inputs a pB machine or its refinement violating a specific safety property expressed in its EXPECTATIONS clause, and generates its equivalent MDP representation in the PRISM language [14]. PRISM is a probabilistic model checker that permits pB models as MDPs in the tool framework and thus can investigate critical expected values of random variables as "reward structures" — a part of PRISM's specification language. PRISM can then be used to explore the computation of $Exp.[\![P^K_{\aleph}]\!].Expt.s_0$ for values of $K \geq 0$, and thus (modulo computing resources) can determine values of K for which the expectations clause fails. If such a K is discovered, YAGA is able to extract the resultant failure tree as an "extremal scheduler" that fails the inductivity test. The extremal scheduler is a transition probability matrix which gives a description of the best (or worst-case) deterministic scheduler of the PRISM representation of an abstract 'faulty' pB machine — *i.e.* one whose probability (or reward) of reaching a state where our intended safety specification is violated is maximal (or minimal).

Finally, YAGA analyses the resultant extremal scheduler using algorithmic techniques set out in [24] and generates 'the most useful' diagnostic information composed of finite execution traces as sequences of operations and their state valuations leading from the initial state of the pB machine to a state where the property is violated. Details of the underlying theory of YAGA, its algorithms and implementation can be found elsewhere [25, 24]. In the next section we discuss practical details on how to use exhaustive search of pB machines to verify compliance of inductivity for finite probabilistic models.

```
IMPLEMENTATION      contractionImp
REFINES             contraction
SEES                Bool_Type, Int_TYPE, Real_TYPE
OPERATIONS
  ans ⟵ contraction (NN) ≙   VAR nn IN
                             nn := NN; ans := TRUE;

                             WHILE (nn > 2) DO
                             ans ⟵ merge(nn, ans);
                             nn := nn − 1
                             VARIANT    nn
                             INVARIANT nn ∈ ℕ ∧ nn ≤ NN ∧ 2 ≤ nn ∧ ans ∈ BOOL ∧
                             expectation(frac(2, nn × (nn − 1)) × lift ans)
END;
END
```

Figure 3: A pB refinement of the contraction specification of the Mincut algorithm.

5 Case study one: min-cut

We discuss one of Hoang's pB models [15]: a randomised solution to finding the "minimum cut" in an undirected graph. The probabilistic algorithm is originally due to Karger [20]. We also report experimental results after running our diagnostic tool.

Let an undirected graph be given by (N,E) where N is a set of nodes and E is a set of edges. The graph is said to be *disconnected* if N is a disjoint union of two nonempty sets N_0, N_1 such that any edge in E connects nodes in N_0 or N_1; a graph is *connected* if it is not disconnected. A *cut* in a connected graph is a subset $E' \subseteq E$ such that $(N, E \backslash E')$ is disconnected; a cut is minimal if there is no cut with strictly smaller size. Cuts are useful in optimisation problems but are difficult to find. Karger's algorithm uses a randomisation technique which is not guaranteed to find the minimal cut, but only with some probability. The idea of the algorithm is to use a "contraction" step, where first an edge e connecting two nodes (n_1, n_2) is selected at random and then a new graph created from the old by "merging" n_1 and n_2 into a single node n_{12}; edges in the merged graph are the same as in the original graph except for edges that connected either n_1 or n_2. In that case if (n_1, a), say was an edge in the original graph then (n_{12}, a) is an edge in the merged graph. We keep merging while the number of nodes is greater than 2. The specification of the merge function for an initial number of nodes NN is such that

$$ans \longleftarrow \mathbf{merge}(nn, aa) \;\triangleq\; nn \in NN \wedge aa \in BOOL \mid ans := (\mathbf{false} \;{}_{\leq 2/nn}\!\oplus\; aa).$$

It expresses that with a probability of at most $2/nn$, the minimum cut will be destroyed by the contraction step. Otherwise the minimum cut is guaranteed to be found. Contraction satisfies an interesting combinatorial property which is that if the edge is chosen uniformly at random from the set of edges then the merged graph has the same minimum cut as does the unmerged graph with probability at least $2/(NN(NN-1))$. Although this probability can be small, it can be amplified by repeating the algorithm to give a probability of assurance to within any specified threshold.

The pB implementation in Fig. 3 sets out part of the refinement step for the min-cut algorithm. The refinement describes an iteration where the **merge** function is called to perform the contraction described above. The result of a call to merge is that the number of nodes in the graph (given by the variable *nn*) is diminished by 1 and either the original minimum cut is preserved (with probability mentioned above), or it is not; the Boolean *ans* is used to indicate which of these possibilities has been selected.

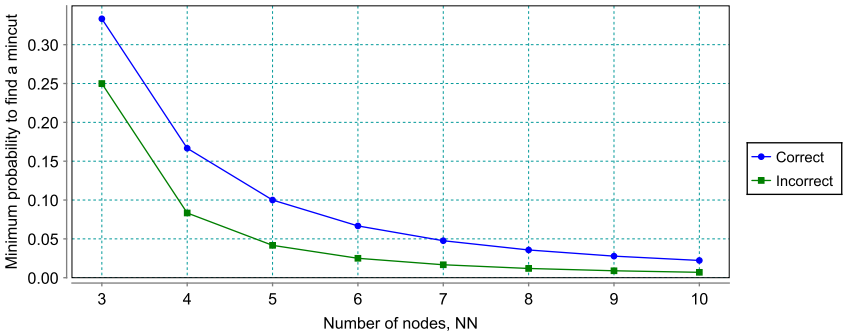


Figure 4: Graph comparing the probabilities to find a min-cut for the correct and incorrect implementations of the contraction specification of the mincut algorithm. The incorrect implementation is where we have introduced a high probability in the left branch of the **merge** operation thus forcing the variable *ans* to become **false** often.

```
******* Starting Error Reporting  for Failure Traces located on step 2 *********
          Sequence of operations leading to bad state ::>>>
                      [{INIT} (3,true), {Skip} (3,true)
              Probability mass of failure trace is:>>>> 1
************ Finished Error Reporting***************
```

Figure 5: Diagnostics detailing a failure of the inductive invariance at the implementation step (for $NN = 3$) involving the **merge** operation. Note that this is a counterexample since the execution of the merge operation will result in an endpoint distribution which yields a decreased expectation (see Def.5). That is, there is a witness s ($nn = 3$, $ans = true$) such that $Wp.\mathbf{merge}.2/(nn(nn-1)).s = 1/12 < 2/(nn(nn-1)).s = 1/3$. Note that every trace component of the counterexample is marked with a pair which denotes the state valuations of the program variables occurring in the EXPECTATIONS clause, in this case (*nn*, *ans*).

Here we use the $expectation(.)$ function to check that the expression $\mathrm{lift}ans \times 2/(nn(nn-1))$ simplifies to an inductive property; that is, that the probability of preserving the minimum cut should always be at least $2/(nn(nn-1))$ while *ans* remains **true**, but is 0 if *ans* ever becomes **false**. Note that if this property holds then we are able to deduce exactly that the overall probability that the original minimum cut is preserved when the graph is merged to one of 2 nodes is the theoretically predicted $2/(NN(NN-1))$.

Next we describe bounded model checking style experiments to analyse the refinement.

5.1 Experiments for min cut

5.1.1 Counterexample diagnostics

In our first experiment we introduce an error [3] in the design of the **merge** function. The graph depicted in Fig. 4 shows a failure to preserve the expected probability threshold of the mincut algorithm. Specifically the graph shows that the probability falls below $2/(NN(NN-1))$. An examination of the resultant failure tree produces the counterexample depicted in Fig. 5. It clearly reveals a problem ultimately leading to a witness after executing the **merge** operation.

[3] We set the probability of choosing the left branch in the merge specification to be "at most" 3/4 so that the new specification becomes $ans := (\mathbf{false} \leq_{3/4} \oplus\ aa)$

PRISM model checking results for mincut algorithm for varying node sizes			
NN	**States, transitions**	**Probability to find a mincut**	**Duration (secs)**
10	72517, 128078	2.2222 E-1	18.046
50	412797, 732718	8.1633 E-4	131.363
100	797647, 1416518	2.0202 E-4	277.605

Table 1: Performance result of inductive invariance checking for mincut

5.1.2 Proof of correctness for small models

In the next experiment we fix the error in the **merge** function and attempt a verification of mincut for specific (small) model sizes. In particular, we use YAGA to check that the EXPECTATIONS clause satisfies the inductive property for all reachable states. The result is shown in Table 1. It depicts the various sizes of the PRISM model relative to the number of nodes NN of interest of the original graph.

6 Probabilistic diagnostics of dependability

In this section we investigate how the use of probabilistic counterexamples can play a role in the analysis of dependability, especially in compiling quantitative diagnostics related to specific "failure modes".

We assume a probabilistic model of a critical system, and we shall use the notation and conventions set up in Sec.3. In addition, we shall reserve the symbol F for a special designated state corresponding to "complete failure"; in the case that a system completely fails (i.e. enters the F state) we shall posit that no more actions are possible. In the design of dependable systems, one of the goals is to understand what behaviours lead to complete failure, and how the design is able to cope overall with the situation where partial failures occur. For example, the design of the system should be able to prevent complete failure even if one or more components fail. Regrettably, some combinations of component failures will eventually lead to complete failure — those combinations are usually referred to as *failure modes*. In such cases, dependability analysis would seek to confirm that the relevant failure modes were very unlikely to occur and also, to produce some estimate of the time to complete failure once the failure mode arose.

We first set out definitions of failure modes and related concepts relative to an MDP model. In the definitions below we refer to P as an MDP, with F a designated state to indicate "complete failure", such that the annotation $\{F\}\ P\ \{F\}$ holds. Let ϕ be a predicate over the state space and α a sequence of states indicating an execution trace of P. We define the the path formula $\diamond\phi$ to be $(\diamond\phi).\alpha = \textbf{true}$ if and only if there is some $n \geq 0$ such that $\alpha.n$ satisfies ϕ, corresponding to the usual definition of "eventuality" [13].

Our next definition identifies a failure mode: it is a predicate which, if ever satisfied, leads to failure with probability 1. We formalise this as the *conditional probability i.e.* that F occurs given that the failure mode occurs. We use the standard formulation for conditional probability: if μ is a distribution over an event space, we write $\mu.A$ for the probability that event A occurs and $\mu.(A \mid B)$ for the probability that event A occurs given that event B occurs. It is defined by the quotient $\mu.(A \wedge B)/\mu.B$.

Standard approaches for dependability analysis largely rely on the failure mode and effects analysis or (FMEA) [18] for identifying a "critical set" — the minimal set of components whose simultaneous failure constitutes a failure mode. Next we shall show how probabilistic model checking can be used to generalize this procedure.

Definition 6 *Let P be an MDP and let $\aleph$ be a scheduler; we say that a predicate ϕ over the state space*

is a failure mode for $\aleph$ *if the probability that* F *occurs given that* ϕ *ever holds is* 1*:*

$$[\![P^K_\aleph]\!].s_0.(\diamond F \mid \diamond \phi) \quad = \quad 1 \,,$$

where we write $Exp.[\![P^K_\aleph]\!].s_0.(\diamond F \mid \diamond \phi)$ *as the conditional probability over traces such that* F *is reachable from the initial state* s_0 *given that* ϕ *previously occurred. We say that* ϕ *defines a critical set if* ϕ *is a weakest predicate which is also a failure mode.*

Given the assumption that once the system enters the state F, it can never leave it, Def. 6 consequently identify states of the system which certainly lead to failure.

Once a critical set has been identified, we can use probabilistic analysis to give detailed quantitative profiles, including the probability that it occurs, and estimates of the time to complete failure once it has been entered. The probability that a critical set ϕ occurs for a scheduler $\aleph$ is given by $Exp.\langle\!|P_\aleph|\!\rangle.(\diamond\phi)$. The next definition sets out the basic definition for measuring the time to failure — it is based on the conditional probability measured at various depths of the execution tree.

Definition 7 *Let* P *be an MDP,* $\aleph$ *a scheduler and let* K *refer to the depth of the associated execution tree. Furthermore let* ϕ *be a critical set. The probability that complete failure has occurred at depth* K *given that* ϕ *has occurred is given by:*

$$[\![P^K_\aleph]\!].s_0.(\diamond F \mid \diamond \phi) \,.$$

Thus even though a failure mode has been entered, the analysis can determine the approximate depth of computation $k \leq K$ before complete failure occurs.

6.1 Instrumenting model checking with failure mode analysis

In this section we describe how the definitions above can be realised within a probabilistic model checking environment in order to identify and analyse particular combinations of actions that lead to failure.[4]

6.1.1 Identification of failure modes

The first task is to interpret Def. 6 as a model checking problem: this relies on the calculation of *conditional probabilities* which is not usually possible using standard techniques. However, adopting the more general expectations approach — instrumented as reward structures of MDPs — we are able to compute lower bounds on conditional probabilities after all.

Lemma 2 *Let* P *be a pGCL program and* $\aleph$ *a scheduler,* X, C *are predicates over* S*, and* λ *is a real value at least* 0*. Starting from an initial state* s_0*, the following relationship holds.*[5]

$$Exp.[\![P_\aleph]\!].s_0.(\mathit{lift}(C \wedge X) - \lambda \times \mathit{lift}C) \geq 0 \quad \mathit{iff} \quad Exp.[\![P_\aleph]\!].s_0.(X \mid C) \geq \lambda \,.$$

Proof 2 *Follows from linearity of the expectation operator and the definition of conditional probability as* $Exp.[\![P_\aleph]\!].s_0.\mathit{lift}(C \wedge X)/Exp.[\![P_\aleph]\!].s_0.\mathit{lift}C$ *provided that* C *has a non-zero probability of occurring.*

[4] Note that YAGA computes probabilities over endpoints rather than over traces, thus we assume that failure modes can be identified by entering a state which persists according to Def. 6. These will be deadlock states of the MDP being analysed.

[5] This expression may be generalised to allow for non-determinism: $Exp.[\![P]\!].s_0.(\mathsf{lift}(C \wedge X) - \lambda \times \mathsf{lift}C) \geq 0$ *iff* $[\![P_\aleph]\!].s_0.(X \mid C) \geq \lambda$, for any scheduler $\aleph$. Note also that if C does not hold with a non-zero probability then this definition assumes that the conditional probability is still defined and is maximal.

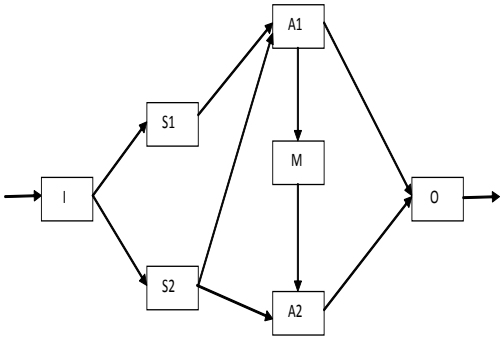


Figure 6: An embedded control system.

From Lem. 2 we can see that (putting $\lambda = 1$) if $Exp.[\![P_{\aleph}]\!].s_0.(\mathsf{lift}(C \wedge X) - \mathsf{lift}C) \geq 0$ then the conditional probability $Exp.[\![P_{\aleph}]\!].s_0.(X \mid C) = 1$. On the other hand, we can verify the expression $Exp.[\![P_{\aleph}]\!].s_0.(\mathsf{lift}(C \wedge X) - \mathsf{lift}C) \geq 0$ directly using YAGA's output. Thus the following steps summarise our proposed method for failure mode analysis.

(a) Use YAGA to identify a failure tree consisting of traces which terminate in F.

(b) From the failure tree identify candidate combinations of events C which correspond to traces terminating in F.

(c) Using YAGA's output, verify that the candidate combinations C are indeed failure modes by evaluating the constraint $Exp.[\![P_{\aleph}]\!].s_0.(\mathsf{lift}(C \wedge X) - \mathsf{lift}C) \geq 0$ *i.e.* after setting $\lambda = 1$.

(d) Compute expected times to failure for the identified failure modes.

In the next section we shall illustrate this technique on a case study of an embedded controller design.

7 Case study two: controller design

Here we show how YAGA can be used to provide important diagnostics feedback to a pB developer summarising the failure the EXPECTATIONS clause in a pB machine refinement. We incorporate the key dimensions of systems dependability — *availability* — the probability that a system resource(s) can be assessed; *reliability* — the probability that a system meets its stated requirement; *safety* — expresses that nothing bad happens.

The design in Fig. 6 is originally based on the work by Güdemann and Ortmeier [11]. It consists of two redundant input sensors (S1 and S2) measuring some input signal (I). This signal is then processed in an arithmetic unit to generate the required output signal (O). Two arithmetic units exist, a primary unit (A1) and its backup unit (A2). A1 gets an input signal from both S1 and S2, and A2 only from one of the two sensors. The sensors deliver a signal in finite intervals (but this requirement is not a key design issue since we assume that signals will always be propagated). If A1 produces no output signal, then a monitoring unit (M) switches to A2 for the generation of the output signal. A2 should only produce outputs when it has been triggered by M.

An abstract description of the behaviour of the controller is captured in the specification of Fig. 7. The reliability of the system is given by the real value rr; we encode this in the safety specification within the $expectation(.)$ function. State labels $sg = 2$ and $sg = 3$ denote signal success and failure respectively. Otherwise state labels $sg = 0$ and $sg = 1$ respectively denote idle state and signal in transit.

MACHINE SignalTracker ($maxtime, s1p, s2p, a1p, a2p, mp$)
SEES Int_TYPE, Real_Type
CONSTRAINTS $maxtime \in \mathbb{N} \wedge s1p, s2p, a1p, a2p, mp \in REAL \wedge s1p, s2p, a1p, a2p, mp :\in real(0)..real(1)$
CONSTANTS rr
PROPERTIES $rr \in REAL \wedge rr \geq real(0) \wedge rr \leq real(1)$
OPERATIONS
$sgout \longleftarrow sendsignal \triangleq$
PRE $expectation(real(rr))$ **THEN**
ANY sg **WHERE**
$sg \geq 0 \wedge sg \leq 3 \wedge expectation(\mathsf{lift}(sg = 0 \vee sg = 1) \times real(rr) + \mathsf{lift}(sg = 2))$
THEN
$sgout := sg$
END;
END;
END

Figure 7: Again we use the $expectation(.)$ function to specify that states where $sg = 0 (or 1)$ are worth the system reliability rr; states where $sg = 2$ are worth 1 and states where $sg = 3$ are worth 0. This encoding is a safety property for the *sendsignal* operation and must be preserved by any refinement of the abstract machine.

7.1 Refining the controller specification

Here we provide an implementation of the controller by refining the abstract specification in Fig. 7. We also show how to adapt the standard *B*-style modelling of timing constraints [7, 6] to pB models. We use the EXPECTATIONS clause of the form $q \Rrightarrow p \times \mathsf{lift}(s \neq F) \sqcup \mathsf{lift}success$, which captures the idea that the probability of reaching the "success" state should exceed the given threshold q. Here p is a parameter which could vary over the state, but which should initially be at least the value of q. Observe that F denotes a state where signal is lost.

But before we do this, we assign individual availability to components of the controller and include the information in the CONSTANTS clause of their abstract machine descriptions. The implementation of the controller as well as the abstract descriptions of its components are in the Appendix. In the next section, we show how to perform dependability analysis on the controller after setting all the components availability to 95% $(s1p = s2p = a1p = a2p = mp = 0.95)$. To do this, we use YAGA to provide an equivalent MDP interpretation of the refinement in the PRISM language. This then permits experimental analysis of the refinement and hence generation of system diagnostics to summarise the process.

7.2 Experiment 1: identification of critical sets

Step 1:

We set the parameters $q, p := 1$ in the expression $q \Rrightarrow p \times \mathsf{lift}(s \neq F) \sqcup \mathsf{lift}success$ to identify all failure traces for chosen values of the components availability. Fig. 8 lists three of the failure traces (out of a total of 5) relevant to our discussion, resulting in a maximum probability of failure of 0.0025 after the 6th execution time stamp *i.e.* $maxtime = 6$.

Step 2: From inspection of the above traces we notice that the failure of $A1$ and M enables us to identify them as potential candidates for the construction of our critical set.

Step 3: We verify that their failure will indeed result in overall failure by examining the value of the expectation $\mathsf{lift}(F \wedge A1 \wedge M) - \mathsf{lift}(A1 \wedge M)$.

For candidates such as A1 and M, we use the diagnostic traces to calculate the conditional probabilities as in Def. 6. To do this we extract all the traces which result in F and then examine the variations of the component failures in the traces to identify those which corresponded to a failure configuration.

```
***** Starting Error Reporting  for Failure Traces located on step 6 *****

            Sequence of operations leading to bad state ::>>>
            [{INIT} (1,0,0,0,0,0), {Sensor2Action} (1,0,1,0,0,0),
        {PrimaryAction} (1,0,1,2,0,0), {MonitorAction} (1,0,1,2,0,2),
{Skip} (1,0,1,2,0,2), {Sensor1Action} (1,2,1,2,0,2), {SendSignal} (3,2,1,2,0,2)]
                Probability mass of failure trace is:>>>> 0.00012

            Sequence of operations leading to bad state ::>>>
            [{INIT} (1,0,0,0,0,0), {Sensor2Action} (1,0,2,0,0,0),
        {Sensor1Action} (1,1,2,0,0,0), {PrimaryAction} (1,1,2,2,0,0),
{MonitorAction} (1,1,2,2,0,2), {Skip} (1,1,2,2,0,2), {SendSignal} (3,1,2,2,0,2)]
                Probability mass of failure trace is:>>>> 0.00012

            Sequence of operations leading to bad state ::>>>
             [{INIT} (1,0,0,0,0,0), {Sensor2Action} (1,0,1,0,0,0),
        {PrimaryAction} (1,0,1,2,0,0), {MonitorAction} (1,0,1,2,0,2),
{Skip} (1,0,1,2,0,2), {Sensor1Action} (1,1,1,2,0,2), {SendSignal} (3,1,1,2,0,2)]
                Probability mass of failure trace is:>>>> 0.00226

        ************ Finished Error Reporting  ... ***************
```

Figure 8: Diagnostic feedback revealing single traces at endpoint probability distributions (after setting parameter $maxtime = 6$) corresponding to the failure of the controller to deliver an output signal. Note that the state tuple in this case is given by (sg, $s1$, $s2$, $a1$, $a2$,m).

The results were unsurprising and included for example, identifying that a simultaneous failure of the primary unit $A1$ and the backup monitor M. On the other hand, once the *pB* modelling was completed, the generation of the failure traces was automatic improving the confidence of full coverage. To illustrate this point, a programming mistake was uncovered using this analysis where $A1$ was mistakenly programmed to extract a correct reading only if it received signals from both sensors, rather than from at least 1.

7.3 Experiment 2: investigating time to failure

This experiment investigates the time to first occurrence of failure given a particular critical set. In fact, the results show that members of the set of interest are indeed critical after verifying their overall conditional probabilities of failure. In summary, for example, a failure tree corresponding to depth $K = 6$ yields distributions over endpoints traces whose components time to failure is shown in Table 2.

8 Related work

Traditional approaches for safety analysis via model exploration rely on qualitative assessment — exploring the causal relationship between system subcomponents to determine if some types of failure or accident scenarios are feasible. This is the method largely employed in techniques like the Deductive Cause Consequence Analysis (DCCA) [26], which provides a generalisation of the Fault Tree Analysis (FTA) [19]. Other Industrial methods that support this kind of analysis also include the Failure Modes and Effects Analysis (FMEA) [18] and the Hazard Operability Studies (HAZOP) [8]. But the efficiency

Identifying critical components time to first failure		
Critical Components	Time step to first failure	Maximum probability of failure
S1, S2	2 steps	2.5000 E-3
A1, M	3 steps	2.4938 E-3
A1, A2	4 steps	2.4938 E-3
A1, S2	3 steps	2.4938 E-3

Table 2: Maximum probabilities of failure are computed with respect to endpoint distributions of failure traces (Fig. 8) and conditional probabilities are given by Def. 6.

of these techniques is largely dependent on the experience of their practitioners. Moreover, with probabilistic systems, where an interplay of random probabilistic updates and nondeterminism characterise system behaviours, such methods are not likely to scale especially with the dependability analysis of industrial sized systems.

The use of probabilistic model-based analysis to explore dependability features in systems construction has recently become a topical issue [21, 10, 11, 3]. One way to achieve this is to use probabilistic counterexamples [12, 4, 5] which can guarantee profiles refuting the desired property *i.e.* after visiting the reachable states of the supposedly 'finite' probabilistic model.

What we have done here is to show how a similar investigation can be achieved for the refinement of proof-based models by taking advantage of the state exploration facility offered by probabilistic model checking. Our method is very precise since it can guarantee the goal of refinement — improving probabilistic results. However, if this does not hold then we are able to provide exact diagnostics summarising the failure provided that computation resources are not scarce.

9 Conclusion and future work

This paper has summarised an approach based on model exploration for the refinement of proof-based probabilistic systems with respect to quantitative safety specifications in the pB language. Our method can provide a pB designer with information necessary to make judgements relating to dependability features of distributed probabilistic systems. We have shown how this can be done for probabilistic loops hence generalising standard models.

Even though most of the failure analysis conjectured herein have been based on intuition, it should be mentioned that a more interesting investigation would be to explore the use of constraint programming techniques to support full coverage of probabilistic system models. This will enable us target larger refinement frameworks as in [9] where probability is not currently being supported.

Acknowledgement: The authors are grateful to Thai Son Hoang for assistance with the pB models of the embedded controller. We also appreciate the anonymous reviewers for their very helpful comments.

References

[1] J. R. Abrial (1996): *The B-Book: Assigning programs to meaning.* Cambridge University Press.

[2] J. R. Abrial (2009): *Modeling in Event-B: system and software engineering. To appear*. Cambridge University Press. Available at `http://www.event-b.org`.

[3] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner & S. Leue (2009): *Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples*. In proceedings of QEST'09, pp. 299–308, doi:10.1109/QEST.2009.8.

[4] H. Aljazzar & S. Leue (2009): *Generation of counterexamples for model checking of Markov Decision Processes*. In proceedings of QEST'09, pp. 197–206, doi:10.1109/QEST.2009.10.

[5] M. E. Andrés, P. D' Argenio & P. v Rossum (2009): *Significant diagnostic counterexamples in probabilistic model checking*. In proceedings of HVC'08. Lecture Notes in Computer Science 5394, pp. 129–148, doi:10.1007/978-3-642-01702-5_15.

[6] M. Butler (2009): *Using Event-B refinement to verify a control strategy*. Technical Report, University of Southampton, United Kingdom.

[7] D. Cansell, D. Mèry & J. Rehm (2006): *Time constraint patterns for Event-B development*. In proceedings of B'07. Lecture Notes in Computer Science 4355. Springer, pp. 140–154, doi:10.1007/11955757_13.

[8] Chemical Industries Association Limited, London (1987): *CIA.: A guide to hazard and operability studies.*

[9] : *Deploy*. Available at http://www.deploy-project.eu/.

[10] L. Grunske, R. Colvin & K. Winter (2007): *Probabilistic model checking support for FMEA*. In proceedings of QEST'07, doi:10.1109/QEST.2007.18.

[11] M. Gudemann & F. Ortmeier (2010): *Probabilistic model-based safety analysis*. In proceedings of QAPL'10. EPTCS 28, pp. 114–128, doi:10.4204/EPTCS.28.8.

[12] T. Han, J.-P Katoen & B. Damman (2009): *Counterexamples generation in probabilistic model checking*. *IEEE Transaction on software engineering* 32(2), pp. 241–257, doi:10.1007/978-3-540-71209-1_8.

[13] H. Hansson & B. Jonsson (1994): *A logic for reasoning about time and reliability*. *Formal Aspects of Computing* 6(5), pp. 512–535, doi:10.1007/BF01211866.

[14] A. Hinton, M. Kwiatkowska, G. Norman & D. Parker (2006): *PRISM: A tool for automatic verification of probabilistic systems*. In proceedings of TACAS'06. Lecture Notes in Computer Science 3920. Springer, pp. 441–444, doi:10.1007/11691372_29.

[15] T. S. Hoang (2005): *Developing a probabilistic B-Method and a supporting toolkit*. Ph.D. thesis, University of New South Wales, Australia.

[16] C. A. R. Hoare (1969): *An axiomatic basis for computer programming*. *Communications of the ACM* 12(10), pp. 576–580, doi:10.1145/357980.358001.

[17] J. Hurd (2002): *Formal verification of probabilistic algorithms*. Ph.D. thesis, University of Cambridge, United Kingdom.

[18] Internatinal Electrotechnical Commission, Geneva (1985): *IEC International Standard 812: "Analysis techniques for system reliability: procedures for failure mode and effect analysis*.

[19] Internatinal Electrotechnical Commission, Geneva (1990): *International Standard IEC 1025: Fault Tree Analysis (FTA)*.

[20] D.R. Karger (1993): *Global min-cuts in RNC, and other ramifications of a simple min-out algorithm*. In proceedings of fourth annual ACM-SIAM symposium on discrete algorithms. pp 21-30, Austin, Texas, United States.

[21] M. Kwiatkowska, G. Norman & D. Parker (2007): *Controller dependability analysis by probabilistic model checking*. *Control Engineering Practice* 15(11), pp. 1427–1434, doi:10.1016/j.conengprac.2006.07.003.

[22] A.K. McIver & C.C. Morgan (2004): *Abstraction, refinement and proof for probabilistic systems*. Monographs in Computer Science. Springer Verlag.

[23] U. Ndukwu (2009): *Quantitative safety: linking proof-based verification with model checking for probabilistic systems*. In proceedings of QFM'09. EPTCS 13, pp. 27–39, doi:10.4204/EPTCS.13.3.

[24] U. Ndukwu (2010): *Generating counterexamples for quantitative safety specifications in probabilistic B.* Accepted for inclusion in the journal of logic and algebraic programming.

[25] U. Ndukwu & A.K. McIver (2010): *YAGA: Automated analysis of quantitative safety specifications in probabilistic B.* In proceedings of ATVA'10. Lecture Notes in Computer Science 6252. Springer, pp. 378–386, doi:10.1007/978-3-642-15643-4_31.

[26] F. Ortmeier, W. Reif & G. Schellhorn (2006): *Deductive cause-consequence analysis (DCCA).* In proceedings of IFAC World Congress, Elsevier.

[27] M.L. Puterman (1994): *Markov Decision Processes.* Wiley.

Appendix

MACHINE	Clock (*maxtime*)
CONSTRAINTS	$maxtime \in \mathbb{N}$
VARIABLES	$time, action$
INVARIANT	$time \in \mathbb{N} \wedge action \in \mathbb{N} \wedge time \geq 0 \wedge time \leq maxtime$
INITIALISATION	$time, action := 0, 0$
OPERATIONS	
$timeout \longleftarrow initClock \triangleq$	**BEGIN**
	$action := 0 \mid\mid timeout := 0$.
	END;
$timeout \longleftarrow clockAction(label) \triangleq$	
	PRE $label \in \mathbb{N} \wedge time < maxtime$ **THEN**
	BEGIN
	$action := label \mid\mid time := time + 1$
	END;
	END;
	$timeout := time;$
END	

Figure 9: The specification of the discrete Clock is such that whenever an action due to the components or even a **Skip** action fires, time is incremented while also marking the specific action. We use the action variable as a marker to abstract the identification of the operations constituting the the diagnostic traces (See Fig. 8).

MACHINE	Cmp (cp)
SEES	Real_TYPE
CONSTRAINTS	$cp \in REAL \ \wedge \ cp \geq real(0) \wedge cp \leq real(1)$
OPERATIONS	
$cout \longleftarrow componentaction \triangleq$	**PCHOICE** cp **OF** .
	$cout := 1$
	OR
	$cout := 2$
	END;
END	

Figure 10: Here we model an abstract stateless machine for components with similar behaviours. Later on, we shall use pB's IMPORT clause to clone Sensor1, Sensor2, PrimaryUnit, Monitor and Backup Units via variable renaming. The specification of the abstract Cmp machine is such that it can probabilistically either respond to a signal request $(cout = 1[active])$ or it fails to do so $(cout = 2[dead])$. The probability cp is a paremeter of the machine and specifies the availability of the component.

MACHINE SignalProcess($s1p, s2p, a1p, a2p, mp$)
CONSTRAINTS $s1p, s2p, a1p, a2p, mp \in REAL \wedge s1p, s2p, a1p, a2p, mp :\in real(0)..real(1)$
INCLUDES Sensor1.Cmp(s1p), Sensor2.Cmp(s2p), PrimaryUnit.Cmp(a1p), BackupUnit.Cmp(a2p), Monitor.Cmp(mp)
VARIABLES $s1, s2, a1, a2, m$
INVARIANT $s1, s2, a1, a2, m \in \mathbb{N} \wedge s1, s2, a1, a2, m :: [0,2]$
INITIALISATION $s1, s2, a1, a2, m := 0$
OPERATIONS

$label \longleftarrow action \triangleq$

SELECT $s1 = 0$ **THEN**
$s1 \longleftarrow Sensor1.componentaction \;||\; label := 1$
WHEN $s2 = 0$ **THEN**
$s2 \longleftarrow Sensor2.componentaction \;||\; label := 2$
WHEN $a1 = 0 \wedge s1 = 1$ **THEN**
$a1 \longleftarrow PrimaryUnit.componentaction \;||\; label := 3$
WHEN $a1 = 0 \wedge s2 = 1$ **THEN**
$a1 \longleftarrow PrimaryUnit.componentaction \;||\; label := 3$
WHEN $a1 = 2$ **THEN**
$m \longleftarrow Monitor.componentaction \;||\; label := 4$
WHEN $m = 1$ **THEN**
$a2 \longleftarrow BackupUnit.componentaction \;||\; label := 5$
ELSE $label := 6$

$s1out, s2out, a1out, a2out, mout \longleftarrow getState \triangleq$ **BEGIN** s1out,s2out,a1out,a2out,mout := s1,s2,a1,a2,m **END**;
END

Figure 11: The nondeterministic behaviour of the components is specified in this machine. An individual component can probabilistically respond to a signal request by setting its state value to 1 or 2 denoting 'active' and 'dead' respectively, after leaving the initial state with value 0 ('idle').

IMPLEMENTATION SignalTrackerI($maxtime, s1p, s2p, a1p, a2p, mp$)
REFINES SignalTracker
SEES Real_TYPE, Int_TYPE
IMPORTS SignalProcess($s1p, s2p, a1p, a2p, mp$), Clock($maxtime$)
OPERATIONS

$sgout \longleftarrow sendsignal \triangleq$ **VAR** sg, s1, s2, a1, a2, m, t **IN**
$t \leftarrow initClock$;

WHILE ($t \leq maxtime$) **DO**
$act \longleftarrow action$; $t \leftarrow clockAction(act)$;
$s1, s2, a1, a2, m \longleftarrow getState$;

IF $(a2 = 1) \wedge (s2 = 1)$ **THEN**
$sg := 2$;
ELSIF $(a1 = 1) \wedge (s1 = 1)$ **THEN**
$sg := 2$;
ELSIF $(a1 = 1) \wedge (s2 = 1)$ **THEN**
$sg := 2$;
ELSE
$sg := 3$;
END;
$sgout := sg$;
INVARIANT $s1, s2, a1, a2, m, t \in \mathbb{N} \wedge s1, s2, a1, a2, m :: [1,2] \wedge sg :: [0,3] \wedge t \leq maxtime$
EXPECTATIONS $real(rr) \Rrightarrow (\mathsf{lift}(sg = 0 \vee sg = 1) \times real(rr) + \mathsf{lift}(sg = 2)) \times \mathsf{lift}(t = maxtime)$
END;
END

Figure 12: SignalTrackerI uses a **WHILE-DO** loop structure to model the passage of discrete time. The **PCHOICE** operation provides implementation constructs of the abstract probabilistic branching statements with respect to the availability of the controller components.

Formalising the Continuous/Discrete Modeling Step

Richard Banach*
School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Huibiao Zhu† Wen Su
Software Engineering Institute, East China Normal University,
3663 Zhongshan Road North, Shanghai 200062, P.R. China.
{hbzhu,wensu}@sei.ecnu.edu.cn

Runlei Huang
Alcatel-Lucent Shanghai Bell, 388 Ningqiao Road,
Pudong Jinqiao, Shanghai 201206, P.R. China.
runleihuang@alcatel-sbell.com.cn

Formally capturing the transition from a continuous model to a discrete model is investigated using model based refinement techniques. A very simple model for stopping (eg. of a train) is developed in both the continuous and discrete domains. The difference between the two is quantified using generic results from ODE theory, and these estimates can be compared with the exact solutions. Such results do not fit well into a conventional model based refinement framework; however they can be accommodated into a model based retrenchment. The retrenchment is described, and the way it can interface to refinement development on both the continuous and discrete sides is outlined. The approach is compared to what can be achieved using hybrid systems techniques.

1 Introduction

Conventional model based formal refinement technologies (see for example [37, 19, 38, 1, 34, 43, 2]) are based on purely discrete mathematical and logical concepts. These turn out to be ill suited to modeling and formally developing applications whose usual models are best expressed using continuous mathematics. Nevertheless, many such applications, control systems in particular, are these days implemented using digital techniques. So there is a mismatch between continuous modeling and discrete development techniques.

In this paper we tackle this mismatch head on. Although traditional model based refinement is too exacting to straddle the continuous to discrete demarcation line, a judicious weakening of it, retrenchment, proves to be adaptable enough to do the job, which we show. Importantly, retrenchment techniques interface well with refinement, so that a development starting from continuous and ending at discrete can be captured in an integrated way.

In this paper we tackle the continuous to discrete issue by taking a simple running example, one that can be solved fully by analytic means in both the continuous and discrete domains, and tracing it through the critical formal development step. We start with a continuous control problem: bringing an object (eg. a train) to a halt. This is formulated as a continuous control problem, and given the (deliberately chosen) simplicity of the problem, an exact solution is presented. In reality, continuous control is implemented these days via digital controllers. These periodically read inputs and recompute

*The majority of the work reported in this paper was done while the first author was a visiting researcher at the Software Engineering Institute at East China Normal University. The support of ECNU is gratefully acknowledged.

†Huibiao Zhu is supported by the National Basic Research Program of China (No. 2011CB302904), the National Natural Science Foundation of China (No. 61061130541), China HGJ Significant Project (No. 2009ZX01038-001-07), and Doctoral Program Foundation of Institutions of Higher Education of China (No. 200802690018).

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 121–138, doi:10.4204/EPTCS.55.8

outputs at multiples of a sampling interval during the dynamics. In this sense, the control becomes discretized, although the discretized control is obviously still played out in the continuous real world. We thus remodel the continuous problem as a discrete control problem, and derive a formal description of the discretization step via a suitable retrenchment, drawing on rigorous results from the theory of ordinary differential equations (ODEs) to supply the justification. Given the limited size of this paper, our technical focus is on this critical step, and the remainder of the development (comprising the associated refinements either side of it) is sketched rather than treated in detail. The latter is a task for which a fuller treatment will be given in the extended version of the paper.

The rest of the paper is as follows. We start in Section 2 by describing relevant existing work in the hybrid systems domain and how it contrasts with our own approach, after which we get down to details. Section 3 then formulates our train stopping problem as a conventional open loop continuous control problem. Section 4 then describes the discretization of the control problem using a simple zero order hold strategy. In Section 5 we review what we need of ASM refinement and retrenchment in a form suitable for our problem. Section 6 then shows how our earlier discretization process can be captured using a suitable retrenchment, citing the needed ODE results. Section 7 sketches how all this can fit into a wider formal development strategy, in which the greater flexibility of retrenchment can be combined with the stronger guarantees offered by refinement via the Tower Pattern [8, 28]. Section 8 concludes.

2 Related Work

The relationship between continuous and discrete transition systems has long been a topic for investigation in the hybrid systems field. Earlier work includes [4, 26, 5, 25]; also, the International Conference on Hybrid Systems: Computation and Control, has been the venue for a large amount of research in this area. A more recent reference is [42].

Hybrid systems are dynamical systems that mix smooth, continuous transitions with discrete, discontinuous ones. The major focus in this field has been the automatic verification of properties of such systems. Obviously, such verification demands the representation of the systems in question in discrete and finite terms, whether by means of an explicitly constructed finite state space (which is manipulated directly), or a state space whose states arise via the symbolic representation of the less tractable state space of a previously constructed underlying system (which is manipulated symbolically).

The main tool for bringing an intractable state space within the scope of computable techniques is the equivalence relation. Regions of the state space are gathered into equivalence classes, and a representation of these equivalence classes (whether as individual elements in a simple approach, or as symbolic expressions that denote the equivalence class in question) constitutes the state space of the abstraction. Transitions between these states are introduced to mirror the behaviour of the underlying system. The properties of interest can then be checked against the abstract system. For instance, properties that can be expressed as reachability properties fall within the scope of model checking approaches that are applied to the abstraction.

Of course what has been constructed thereby is a (bi)simulation, and a major strand of hybrid systems research is the investigation of such (bi)simulations. The same remarks apply when there is an external control applied to the systems.

One disadvantage of the above approach is the frequent reliance on brittle properties of the studied systems. Put most simply, a number of techniques rely on the parameters of the problem falling within a subset of measure zero of the parameter space. Real systems can never hit such small targets reliably. Equally, the simulation relations studied can also be just as brittle. To alleviate this, and to address

other issues of interest, the notion of *approximate (bi)simulation* has been studied in recent years ([42] gives a good introduction). Here, instead of defining the simulation relation $R(u,v)$ between an abstract state u and a concrete state v as a simple predicate on states, it is defined via a distance function $\mathbf{d}$ as $R_\varepsilon(u,v) \equiv \mathbf{d}(f(u),v) \leq \varepsilon$, where f is a precise relationship between the two state spaces which is in some sense "semantically natural" (we don't have space to elaborate on this aspect here). For bisimulation you need a symmetrical arrangement of course.

(Bi)simulation depends on assuming the appropriate relation between the two before-states and re-establishing it in the after-states of suitable pairs of transitions. To preserve a relationship based on distance, the dynamics needs to be inherently *stable*. The obvious centre of attention thus becomes stable control systems, normally *linear* stable control systems, because of their calculational tractability. These are discussed in very many places, eg. [32, 20, 22, 18, 3, 40, 11, 6].

In a stable system all trajectories converge to a single point, so the distance between two trajectories decreases monotonically; hence a simulation relation based on distance between trajectories is maintained. But although most systems are designed to be stable in this sense, some are not, and there can be parts of a system phase space in which trajectories diverge rather than converge, without rendering the system useless. Below, we treat in detail a very simple example which happens to be unstable in the sense just discussed. We know it is not stable because we solve it exactly.

Also, in the usual hybrid systems literature, it is normal that the discrete approximation to a given system is manufatured from it (eg. by constructing equivalence classes, as indicated above). In our approach, by contrast, we take a more "off the shelf" attitude to discretization, analysing a straightforward "zero order hold" version of the continuous system (in which the new output values to be sent to the actuators are recalculated at regular intervals, and the new values are "held" for the duration of the next inteval[1]) rather than something extracted from an analysis of the original system. In this sense our approach is closer to conventional engineering practice, since it is directed at the typical practical approach. Of course these two ways of doing things are not mutually exclusive: the parameters of the zero order hold may fall within the parameters of a discrete approximation extracted by analysis of the original system, and *vice versa*. Finally, our approach is via retrenchment, one consequence of which is that our analysis is not confined to the purely stable case. In effect, the greater expressiveness of retrenchment permits (the analogue of) the simulation relation mentioned above, to increase its permitted margin of error, as well as to decrease it, although this emerges indirectly.

3 Train Stopping: a Continuous Control System

Our target application domain is control problems in the railway sphere. In this paper we have train stopping as a specific case study. Of course, in reality, train position control is a complex problem [41, 27], relying on the co-operation of many mechanisms to achieve a reliable outcome, and we do not have the space to deal with all these aspects and their subtle interactions. Instead we focus on a single technical issue —the relationship between a continuous control problem and its discrete counterpart— in a very simple way, commenting on the extreme simplicity below.

Suppose a train, of mass M, is traveling at its cruise velocity V, when it needs to stop. We assume that a linearly increasing deceleration rate a is appropriate. (It has to be said here that our notion of appropriateness is not quite the usual one. Rather than usability or any similar consideration governing our choice, simplicity is the priority. A constant deceleration would have been even simpler — unfortunately

[1] "Zero order" refers here to "holding" the output value constant throughout the interval, in contrast to a higher order hold which would use a suitably designed higher order polynomial.

the zero order hold approximation to constant deceleration is identical to it, trivialising our problem.) To bring the train to a standstill using linearly increasing deceleration, a force $F = -Mat$ (where t is time) has to be applied, by Newton's Law. We will assume that M is known, so that we can focus on just the kinematic aspects.

A cursory knowledge of kinematics is enough to reveal that under linear deceleration, the deceleration, distance and stopping time are linked. We suppose that there is a single stopping episode, which starts at time 0 and at x position 0, and which ends at time T_{Stop}, with the train having traveled to position $x = D$. Representing time derivatives with a dot, if v is the velocity, then we know that

$$\dot{v} \;=\; -at \qquad\qquad v(0) \;=\; V \qquad\qquad v(T_{Stop}) \;=\; 0 \tag{1}$$

Regarding the distance traveled x, we know that

$$\dot{x} \;=\; v \qquad\qquad x(0) \;=\; 0 \qquad\qquad x(T_{Stop}) \;=\; D \tag{2}$$

Integrating these, rapidly brings us to

$$V \;=\; \frac{1}{2}aT_{Stop}^2 \qquad\qquad D \;=\; VT_{Stop} - \frac{1}{3!}aT_{Stop}^3 \;=\; \frac{2}{3}VT_{Stop} \tag{3}$$

We now recast the above as a control theory problem. At the introductory level, control theory is usually developed in the frequency domain [32, 20, 22, 18], because of the relative simplicity and perspicuity of the design techniques in that domain. However, for results sufficiently rigorous to interface to formal techniques, we need to go to the state space formulation favoured by more mathematically precise treatments [3, 40, 15, 14, 11, 6]. In the state space picture, the system consists of a number of state variables, and their evolution is governed by a corresponding number of first order differential equations. State variables and differential equations mirror the states and transition systems of model based refinement formalisms sufficiently closely that we can hope to make a connection between them.

To use the first order framework in our example, the state has to consist of both the position $x(t)$ and the velocity $v(t)$. So we get the state vector

$$\boldsymbol{x}(t) \;=\; \begin{bmatrix} x(t) \\ v(t) \end{bmatrix} \tag{4}$$

The dynamics of the system is captured in the equation[2]

$$\dot{\boldsymbol{x}}(t) \;=\; \begin{bmatrix} \dot{x}(t) \\ \dot{v}(t) \end{bmatrix} \;=\; \boldsymbol{f}(\dot{x}(t), u(t)) \;=\; \begin{bmatrix} v(t) \\ u(t) \end{bmatrix} \tag{5}$$

where

$$u(t) \;=\; -at \tag{6}$$

is the external control control signal. We also have the initial condition

$$\boldsymbol{x}(0) \;=\; \begin{bmatrix} 0 \\ V \end{bmatrix} \tag{7}$$

[2]It is clear that when (5) is expressed as a linear control law (with external control signal), the linear part has only zero eigenvalues. Thus it is not stable in the usual (Liapunov) sense.

4 From Continuous Control to Discrete Control

To truly implement a continuous control model, such as our case study, requires analogue apparatus. In the highly digitized world of today, hardly any such systems are built. Instead, continuous control designs are discretized, and it is the corresponding digital control systems that are implemented.

The digital approach to control has many parallels with the continuous case — in the frequency domain the main difference is the use of the z-transform rather than the Laplace transform. The state based picture too boasts many parallels, with first order difference equations replacing first order differential equations [23, 24, 33, 29].

In this section we examine a discrete counterpart of the previous continuous control problem, in preparation for a formal reappraisal in the next section. One advantage of the extreme simplicity of our example, is that it admits an analytic solution in both continuous and discrete domains, enabling an incisive evaluation to be made later, of the reappraisal in Section 6.3.

The starting point for our problem remains as before: the train, traveling at velocity V, needs to stop after time T_{Stop}, having gone a distance D_D.[3] Instead of doing so continuously though, it will do it in a number of discrete episodes. For this purpose, let us assume that T_{Stop} is divided into N short periods, each of length T, so that

$$T_{Stop} \;=\; NT \tag{8}$$

Our discretization scheme will be based on a zero order hold, in which the same control input value is maintained throughout an individual time period. The counterpart of the linear deceleration rate a of the continuous treatment, will be a piecewise constant deceleration, with the constant rate decreasing by an additional multiple of a constant a_D after each time interval of length T.

Calling the discretized velocity variable v_D, we have for the acceleration

$$\dot{v}_D(t) \;=\; -ka_DT \tag{9}$$

where

$$k \;=\; \left\lceil \frac{t}{T} \right\rceil \tag{10}$$

and k ranges over the values $1 \ldots N$. If we set, for a general t,

$$\delta t_k \;=\; t-(k-1)T \;=\; t-\left\lfloor \frac{t}{T} \right\rfloor T \tag{11}$$

then recalling that the initial velocity is V, provided $(k-1)T < t < kT$, the velocity during the k'th period is

$$v_D(t) \;=\; V - a_DT^2 - 2a_DT^2 - \ldots - (k-1)a_DT^2 - ka_DT\,\delta t_k \tag{12}$$

Since the final velocity is zero, we derive

$$V \;=\; a_DT^2 + 2a_DT^2 + \ldots + Na_DT^2 \;=\; \frac{1}{2}a_DT^2N(N+1) \tag{13}$$

[3] We will use a subscript 'D' to indicate quantities in the discretized model that differ from their continuous counterparts.

Knowing the velocity, we can integrate again, and work out the distance traveled. Calling the displacement in the discretized world x_D, the contribution to x_D during the period $(k-1)T < t < kT$ comes out as

$$(V - a_D T^2 - 2a_D T^2 - \ldots - (k-1)a_D T^2)\delta t_k - \frac{1}{2}k a_D T \delta t_k^2 \tag{14}$$

Thus for the total distance we find

$$\begin{aligned} D_D &= NVT - a_D T^3 \sum_{k=1}^{N-1}(N-k)k - \frac{1}{2}a_D T^3 \sum_{k=1}^{N} k \\ &= VT_{Stop} - \frac{1}{12}a_D T^3(2N^3 + 3N^2 + N) \end{aligned} \tag{15}$$

Both (13) and (15) feature a_D. Substituting the a_D value from (13) into (15) gives

$$D_D = VT_{Stop}\left[1 - \frac{2N^2+3N+1}{6N^2+6N}\right] = \frac{2}{3}VT_{Stop}\left[1 - \frac{1}{4N} + O(N^{-2})\right] \tag{16}$$

We see that (16) for D_D contains an $O(1/N)$ correction compared with (3) for D (assuming we keep V and T_{Stop} the same). This is because we have an extra constraint generated by the requirement that T_{Stop} is an integral multiple of T, making the problem overconstrained if we wished D and D_D to be the same.

Recasting the preceding as an initial value first order system along the lines of (4)-(7) is not hard. The state vector is

$$\boldsymbol{x}_D(t) = \begin{bmatrix} x_D(t) \\ v_D(t) \end{bmatrix} \tag{17}$$

and the dynamics of the system is captured in the equation

$$\dot{\boldsymbol{x}}_D(t) = \begin{bmatrix} \dot{x}_D(t) \\ \dot{v}_D(t) \end{bmatrix} = \boldsymbol{f}(\dot{x}_D(t), u_D(t)) = \begin{bmatrix} v_D(t) \\ u_D(t) \end{bmatrix} \tag{18}$$

where

$$u_D(t) = \dot{v}_D(t) = -ka_D T \tag{19}$$

as given by (9), is the external control. We also have the initial condition

$$\boldsymbol{x}_D(0) = \begin{bmatrix} 0 \\ V \end{bmatrix} \tag{20}$$

It is hard not to notice how much more complicated the above is compared with (1)-(7). It is always so with discrete systems — hence the strong desire to model systems in the continuous domain. The very rapid ramp-up in complexity when we consider the discrete version of a continuous problem is our justification for restricting to a particularly simple example. The ability to keep the complexity still low enough to permit an exact solution, is extremely useful in an investigation such as this one, allowing a comparison between exact and approximate approaches to be made with confidence.

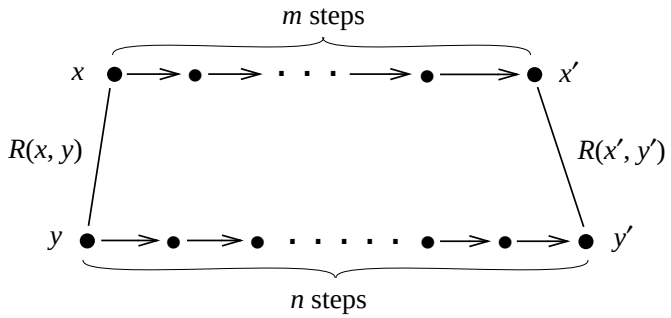


Figure 1: An ASM (m,n) diagram, showing how m abstract steps, going from state x to state x' simulate n concrete steps, going from y to y'. The simulation is embodied in the retrieve relation R, which holds for the before-states of the series of steps $R(x,y)$, and is re-established for the after-states of the series $R(x',y')$.

5 ASM Refinement and Retrenchment

In this section we review what we need of ASM refinement and retrenchment, which will be the vehicles for formalization in this paper. The standard reference for the ASM method is [13], building on the earlier [12]. In general, to prove an ASM refinement, one verifies so-called (m,n) diagrams, in which m abstract steps simulate n concrete ones. The situation is illustrated in Fig. 1, in whch we suppress input and output for clarity. For this paper, it will be sufficient to focus on the refinement proof obligations (POs) which are the embodiment of this policy. The first is the initialization PO:

$$\forall y' \bullet CInit(y') \Rightarrow (\exists x' \bullet AInit(x') \wedge R(x',y')) \tag{21}$$

In (21), it is demanded that for each concrete initial state y', there is an abstract initial state x' such that the retrieve or abstraction relation $R(x',y')$ holds.

The second PO is correctness, and is concerned with the verification of the (m,n) diagrams. For this, we have to have some way of deciding which (m,n) diagrams are sufficient for the application. Let us assume that we have done this. Let $CFrags$ be the set of fragments of concrete execution sequences that we have previously determined will permit a covering of all the concrete execution sequences of interest for the application. We write $y :: ys :: y' \in CFrags$ to denote an element of $CFrags$ starting with concrete state y, ending with concrete state y', and with intervening concrete state sequence ys. Likewise $x :: xs :: x' \in AFrags$ for abstract fragments. Also, let is, js, os, ps denote the sequences of abstract inputs, concrete inputs, abstract outputs, concrete outputs, respectively, belonging to $x :: xs :: x'$ and $y :: ys :: y'$, and let $In(is, js)$ and $Out(os, ps)$ denote suitable input and output relations. Then the correctness PO reads:

$$\begin{aligned}&\forall x, is, y, ys, y', js, ps \bullet y :: ys :: y' \in CFrags \wedge R(x,y) \wedge In_{AOps,COps}(is, js) \wedge \\ &\quad COps(y :: ys :: y', js, ps) \\ &\qquad \Rightarrow (\exists xs, x', os \bullet AOps(x :: xs :: x', is, os) \wedge R(x',y') \wedge Out_{AOps,COps}(os, ps))\end{aligned} \tag{22}$$

In (22), it is demanded that when there is a concrete execution fragment of the form $COps(y :: ys :: y', js, ps)$, carried out by a sequence of concrete operations $COps$, with state sequence $y :: ys :: y'$, input sequence js and output sequence ps, such that the retrieve and input relations $R(x,y) \wedge In(is, js)$ hold

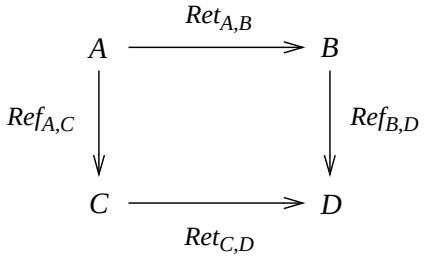


Figure 2: The *Tower Pattern* basic square, with refinements vertical, retrenchments horizontal.

between concrete and abstract before-states and inputs, then an abstract execution fragment $AOps(x :: xs :: x', is, os)$ can be found to re-establish the retrieve and output relations $R(x', y') \wedge Out(os, ps)$.

The ASM refinement policy also demands that non-termination be preserved from concrete to abstract, but we will not need that in this paper. We now turn to retrenchment.

For retrenchment, [10, 9] give definitive accounts; latest developments are found in [36]. See also [7] for formulations of retrenchment adapted to several specific model based refinement formalisms including ASM. Like refinement, retrenchment is also characterized by POs: an initialization PO identical to (21), and a "correctness" PO which weakens (22) by inserting *within*, *output* and *concedes* relations, W_{Op}, O_{Op}, C_{Op} respectively into (22), to give extra flexibility and expressivity. In particular, the concession C_{Op} weakens the conclusions of (22) disjunctively, giving room for many kinds of "exceptional" behaviour. The result is:

$$\begin{array}{l} \forall x, is, y, ys, y', js, ps \bullet y :: ys :: y' \in CFrags \wedge R(x,y) \wedge W_{AOps,COps}(is, js, x, y) \wedge \\ \quad COps(y :: ys :: y', js, ps) \\ \quad\quad \Rightarrow (\exists xs, x', os \bullet AOps(x :: xs :: x', is, os) \wedge \\ \quad\quad\quad\quad ((R(x', y') \wedge O_{AOps,COps}(x :: xs :: x', is, os, y :: ys :: y', js, ps)) \vee \\ \quad\quad\quad\quad\quad C_{AOps,COps}(x :: xs :: x', is, os, y :: ys :: y', js, ps))) \end{array} \tag{23}$$

To ensure that retrenchment only deals with well defined transitions, and to ensure smooth retrenchment/refinement interworking, we also insist that $R \wedge W_{Op}$ always falls in the domain of the requisite operations, though this is another thing not needed here.

The smooth interworking between refinements and retrenchments is guaranteed by the *Tower Pattern*. The basic construction for this is shown in Fig. 2. There, refinements are vertical arrows and retrenchments are horizontal, and the two paths round the square from A to D (given by composing $Ref_{A,C}$ with $Ret_{C,D}$ on the one hand, and on the other, by composing $Ret_{A,B}$ with $Ref_{B,D}$) are compatible, in the sense that they each define a portion of a (potentially larger) retrenchment from A to D.

At this point one might legitimately ask what all the above has to do with our case study, in which the dynamics that we considered is entirely in the continuous domain (albeit taking into account discontinuous control inputs when necessary). The answer lies in the focus on the use of paths through the system at both abstract and concrete levels in the POs of ASM. With this focus, it is unproblematic to reconfigure the (m,n) rules (22) and (23) to deal with continuous paths rather than discrete ones. Thus $CFrags$ and $AFrags$ can now refer to fragements of continuous system trajectories, rather than sequences of state-to-state hops. Likewise the is and js in $W_{AOps,COps}(is, js, x, y)$ now refer to the continuous input

signals along the trajectories, and so on for the other terms in (22) and (23). We see this exemplified in detail in the retrenchment of Section 6.2.

6 Formalizing the Continuous to Discrete Modeling Change

In the control literature, one finds many ways of discretizing continuous designs (see loc. cit.), and the evaluation of the relationship between continuous and discrete is often based on *ad hoc* engineering rules of thumb. While these typically yield perfectly good results in practice, the criteria used fall far short of the kind of precision needed for a good fit with model based formal development techniques. As a consequence, when model based formal development techniques are used to support the digital implementation of the discrete counterpart of some continuous design, the formal modeling inevitably starts already in the discrete domain. Obviously this yields a weaker formal support for the process than if the formal modeling had started earlier, at the continuous design stage, and was integrated into all the subsequent design steps, including the change from continuous to discrete.

Our objective in this paper is to illustrate how to make a judgement about the discretization of a control problem, that has enough precision to integrate well with model based formal technologies. To achieve this we have recourse to the rigorous theory of ODEs. It can be shown[4] that two instances of a control problem which differ solely in the input control satisfy an inequality:

$$||\boldsymbol{x}^{u} - \boldsymbol{x}_D^{u_D}|| \;\leq\; K2||u - u_D||_2 \tag{24}$$

In (24), $||\boldsymbol{x}^{u} - \boldsymbol{x}_D^{u_D}||$ is the $\mathscr{L}^{\infty}$ norm of $\boldsymbol{x}^{u} - \boldsymbol{x}_D^{u_D}$, or, in plain English, the maximum value over the interval $[0 \ldots T_{Stop}]$ attained by the difference between continuous and discrete values of any state component. Likewise, $||u - u_D||_2$ is the $\mathscr{L}^2$ norm of $u - u_D$, or, in plain English, the root integrated square difference between u and u_D, calculated over the interval $[0 \ldots T_{Stop}]$. Finally, $K2$ is a constant.

We note that the continuous and discrete versions of our case study, with initial states (7) and (20), over the time interval from 0 to T_{Stop}, characterize just such a scenario, since (5) and (7) differ from (18) and (20) only in the use of u_D rather than u among the independent variables.

6.1 Rigorous Bounds on Continuous and Discrete Systems

We now flesh out what (24) means for our little case study. We consider the values of the quantities on the right hand side of (24) in order to obtain a bound for the value of the left hand side. Referring to (24), theory furnishes an explicit value for the constant $K2$, namely

$$K2 \;=\; e^{K_f}||k_u||_2 \tag{25}$$

In (25) K_f is $k_f T_{Stop}$, where k_f is the $\mathscr{L}^{\infty}$ norm of $\boldsymbol{f}_{\boldsymbol{x}}$, or, the absolute maximum value (over the interval $[0 \ldots T_{Stop}]$) of the Lipschitz constant governing the variation of the control law $\boldsymbol{f}$ with respect to the state. In our application, the form of the control law is

$$\boldsymbol{f}(v(t), u(t)) \;=\; \begin{bmatrix} v(t) \\ u(t) \end{bmatrix} \tag{26}$$

[4] In the extended version of this paper it *is* shown.

and it is clear that there is only one component of $\boldsymbol{f}$ with a non-zero partial derivative with respect to either x or v, namely the first

$$\frac{\partial \boldsymbol{f}_1}{\partial v} \;=\; 1 \tag{27}$$

With this, the first factor of (25) is just $e^{T_{Stop}}$.

Regarding the second factor, $||k_u||_2$ is the root integrated square value of the Lipschitz constant governing the variation of the control law with respect to the input control signal. Again there is only one component of $\boldsymbol{f}$ with a non-zero partial derivative with respect to u, namely the second

$$\frac{\partial \boldsymbol{f}_2}{\partial u} \;=\; 1 \tag{28}$$

so the root integrated square reduces to $\sqrt{T_{Stop}}$. So we get

$$K2 \;=\; e^{T_{Stop}}\sqrt{T_{Stop}} \tag{29}$$

Turning to the second factor on the right hand side of (24), $||u-u_D||_2$, we recall that we know explicitly what u and u_D are from our earlier calculations. From (6) and (19) we know that

$$u(t) \;=\; -at \qquad\qquad u_D(t) \;=\; -ka_DT \tag{30}$$

where, from (3) and (13)

$$a \;=\; \frac{2V}{T_{Stop}^2} \qquad\qquad a_D \;=\; \frac{2V}{T_{Stop}^2(1+1/N)} \tag{31}$$

Now (30) shows that $u(t)$ decreases linearly, and that $u_D(t)$ is a staircase function, decreasing in equal sized steps near $u(t)$. It is clear from (30) that in the limit $t \to 0+$, we have $u(0+) = 0$ and $u_D(0+) = -a_DT$, so that $u(0+) - u_D(0+) = a_DT$. It is also clear from (30) that in the limit $t \to T_{Stop}-$, we have $u(T_{Stop}-) = -aT_{Stop}$ and $u_D(T_{Stop}-) = -Na_DT = -a_DT_{Stop}$, so that $u(T_{Stop}-) - u_D(T_{Stop}-) = (a_D - a)T_{Stop} = a_DT_{Stop}[1-(1+1/N)] = -a_DT_{Stop}/N = -a_DT$. Since the staircase has equal sized steps, it evidently the case that the staircase $u_D(t)$ ranges around $u(t)$ within a bound a_DT.

$$|u(t) - u_D(t)| \;\leq\; a_DT \tag{32}$$

This furnishes a suitable overestimate for the root integrated square difference between $u(t)$ and $u_D(t)$ as follows

$$||u-u_D||_2 \;\leq\; \sqrt{\int_{t=0}^{T_{Stop}} [a_DT]^2 dt} \;=\; a_DT\sqrt{T_{Stop}} \tag{33}$$

Substituting all the values we have obtained into (24), we get

$$||\boldsymbol{x}^u - \boldsymbol{x}_D^{u_D}|| \;\leq\; e^{T_{Stop}}\sqrt{T_{Stop}} \times a_DT\sqrt{T_{Stop}} \;=\; e^{T_{Stop}}a_DTT_{Stop} \tag{34}$$

We see that despite the potential for the deviation between $u(t)$ and $u_D(T)$ to grow exponentially with the size of the time interval, a possibility severely exacerbated by our rather crude bound (33), it is always possible to reduce it by an arbitrary amount by making the discretization, measured by N, fine enough.

6.2 Turning Rigorous Bounds into Retrenchment Data

Now that we have a precise relationship between the continuous and discrete control systems, we can look to incorporate this into our model based formal description.

In general, the exigencies of model based formal refinement are too exacting to be able to accommodate the kind of relationships just derived. Retrenchment though, has been purposely designed to be more forgiving in this regard, so that is what we will use.

Regardless though, of which model based formal description technique is adopted, is the issue that all such techniques are designed for discrete state transitions, and presume a well defined notion of "next state", to which an equally clear notion of "current state" can be related.

In continuous dynamics there is no sensible notion of next state that we can immediately use. However, as we noted above, the (m,n) diagram approach of ASM refinement makes clear that it is *paths* at abstract and concrete levels that are being related. Thus, although we avoid technical details in this paper, we extend the ASM approach to incorporate *continuous paths* as well as discrete ones. The incentive to do this was one strong reason for choosing ASMs in this work. (Note that this perspective on refinememt between paths is equally applicable to both the continuous and discretized versions of our control problem. In the continuous problem there is a single continuous path. In the discretized problem there are N consecutive shorter continuous "zero order held" paths, interleaved, at the instants at kT, by the discrete recalculations of the output signal, thus constituting a path comprising both continuous and discrete components.)

Since the rigorous results we use concern the same starting state for the two systems, our formal statement is constrained to be an end-to-end one. It will express an end-to-end relationship between the smooth dynamics at the continuous level, and the discretized level's dynamics (which is continuous too, though punctuated at every multiple of T by a discontinuous change in the acceleration).

As we saw before, a retrenchment between two specific operation sequences consists of four things: a retrieve relation between the state spaces, a within relation for the before-states and inputs, an output relation for the after-states and outputs (and before-states and inputs too if necessary), and a concedes relation for the after-states and outputs (and before-states and inputs too if needed). In the relations below, we use some *ad hoc* notations whose meaning should be obvious from the preceding material.

Regarding the retrieve relation R, there is a very natural one that we might expect to use, namely the identity between state values in the continuous and discretized worlds. However, even though in our specific case study the two models start out in the same state thus making such a putative R true in the hypothesis of the PO (23), in most cases, that assumption will not hold, and so we prefer to follow a more generic approach, which will be applicable in a wider set of scenarios. A second proposal for R would see it express a margin of tolerance between the state values in the continuous and discretized worlds, as discussed in Section 2. This proposal would also work after a fashion, but such a proposal works best when the relationship between the two system states is stable throughout the dynamics — we have then a kind of refinement. In our case study, this assumption does not hold since the discrepancy between the two system states grows steadily through the dynamics.

To accomodate inconvenient situations such as these, retrenchment makes provisions for expressing the relationship (or just aspects of the relationship) between the states at the before- point of the transition being discussed in the within relation W instead of (or in addition to) in R. Since the facts expressed in W do not need to be re-established in the conclusion of the PO (23), this provides the most flexible way of incorporating appropriate facts about the systems' before-states in the PO. With this strategy, a global

retrieve relation is not appropriate, and we set R to true

$$R(\langle x(t), v(t)\rangle, \langle x_D(t), v_D(t)\rangle) \equiv \mathsf{true} \tag{35}$$

The job of expressing that the before-states are suitably matched in the PO, taking into account the input control signals throughout the interval of interest, is thus taken on by the within relation W

$$\begin{aligned} &W(u(t \in [0 \ldots T_{Stop}]), u_D([t \in 0 \ldots T_{Stop}]), \langle x(0), v(0)\rangle, \langle x_D(0), v_D(0)\rangle) \equiv \\ &\quad x(0) = x_D(0) \wedge v(0) = v_D(0) \quad \wedge \quad ||u - u_D||_2 \leq a_D T \sqrt{T_{Stop}} \end{aligned} \tag{36}$$

Note that while W relates just the continuous and discrete before-states, it also relates the whole of the continuous and discrete control inputs.

The output relation O says what happens at the end of the period of interest. In our case, on the basis of the rather heavy calculations that came earlier, we can use O to say that the after-states diverge by no more than the bound derived in (34)

$$\begin{aligned} &O(\langle x(T_{Stop}), v(T_{Stop})\rangle, \langle x_D(T_{Stop}), v_D(T_{Stop})\rangle) \equiv \\ &\quad |x(T_{Stop}) - x_D(T_{Stop})| \leq e^{T_{Stop}} a_D T T_{Stop} \quad \wedge \quad |v(T_{Stop}) - v_D(T_{Stop})| \leq e^{T_{Stop}} a_D T T_{Stop} \end{aligned} \tag{37}$$

Note that although O itself speaks *explicitly* only about the after-states that are attained by the two systems, the fact that we derived the properties of the after-states in question using an $\mathscr{L}^\infty$ analysis, means that the same bound holds *throughout* the interval of interest. The advantage of this formulation is that we automatically get a discreteness of the description in terms of before- and after- states, which will integrate neatly with discrete system reasoners (in the event that such modeling is eventually incorporated into mechanised tools), while yet providing guarantees that hold throughout the interval of interest.

Since our system is so simple, O already captures all that we need to say, and the kind of exceptional behaviour that may need to be taken into account in more realistic engineering situations is not present. This is also connected wsiyth the fact that we have trivialised the retrieve relation. Accordingly we can set the concedes relation C to false

$$C(\langle x(T_{Stop}), v(T_{Stop})\rangle, \langle x_D(T_{Stop}), v_D(T_{Stop})\rangle) \equiv \mathsf{false} \tag{38}$$

With these data, the proof obligation (23) becomes provable on the basis of the results cited earlier, which establishes the formal connection between the continuous and discrete domains in a way that can be integrated with formal refinements on both the continuous and discrete sides.

Particularly noteworthy is the fact that the discrepancy between the states grows linearly with time; and that this is a property of the exact solutions and not just an artifact of some approximation scheme. If we tried to handle this in a pure refinement framework, using a retrieve relation R to capture the relationship between states in the two models (regardless of whether R was an exact, pointwise relationship, or an approximate one, analogous to the approximate simulation relations discussed in Section 2), then assuming such an R for the before-states would not enable us to re-establish it for the after-states, and the correctness PO could not be proved. The greater flexibiity of retrenchment permits us to handle the before-states in the within relation and the after-states in the output relation, overcoming the problem.

6.3 Corroboration

In our case study, exact solvability of the control models in both continuous and discrete domains gives us additional and independent confirmation of the approach we are advocating in this paper.

Both continuous and discrete models "run" for the same amount of time, T_{Stop}, and the output relation (37) gives an estimate for the discrepancy between the continuous and discrete states reached in the two models after that time. The states themselves consist of two components, the displacements and the velocities.

Regarding the velocities, both models come to a standstill after exactly T_{Stop}. Consequently both $v(T_{Stop})$ and $v_D(T_{Stop})$ are zero, so that $|v(T_{Stop}) - v_D(T_{Stop})| = 0$, and any positive upper bound is bound to be sound. So (37), which gives the overestimate $e^{T_{Stop}} a_D T T_{Stop}$ for $|v(T_{Stop}) - v_D(T_{Stop})|$ is correct regarding the velocities, but in an unsurprising way.

Regarding the displacements, the quantization of T_{Stop} in the discrete case, leads to the continuous and discrete dynamics stopping at slightly different places, D and D_D respectively, which we calculated earlier. On that basis, we can calculate the exact difference (disregarding $O(N^{-2})$ and beyond):

$$\begin{aligned} |x(T_{Stop}) - x_D(T_{Stop})| &= \frac{2}{3}\frac{V T_{Stop}}{4N} = \frac{1}{2} a_D T_{Stop}^2 \left(1 + \frac{1}{N}\right) \frac{T_{Stop}}{6N} \\ &= \frac{1}{12} a_D T T_{Stop}^2 \left(1 + \frac{1}{N}\right) \end{aligned} \tag{39}$$

On the other hand, the output relation (37) gives the estimate $e^{T_{Stop}} a_D T T_{Stop}$ for this quantity. Thus the exact value falls within the bounds of the estimate, as it should, if and only if (after cancelling the common factor $a_D T T_{Stop}$):

$$\frac{T_{Stop}}{12}\left(1 + \frac{1}{N}\right) \leq e^{T_{Stop}} \tag{40}$$

Since a linear function of T_{Stop} of slope less than 1 can never catch an exponential function of T_{Stop} with coefficient 1, (40) is obviously true, and we have our corroboration.

7 Continuous to Discrete Modeling in a Wider Design Process

The previous sections focused in detail on how the rigorous theory of ODEs was capable of yielding results that could be integrated with existing model based refinement centred development methodologies, all in the context of a very simple example. The essence of the process is to identify useful results from the mathematical theory, and then to drill down into the details of the proof to identify explicit values for the constants etc. that figure in them. The latter process is often required, since it is frequently the case that the goal of a proof of interest is satisfied by merely asserting the existence of the requisite constant, without a specific value being calculated, since that is usually enough to enable the existence of some limit to be proved. By contrast, for us, the existence of the limit is insufficient, since no engineering process can completely traverse the infinite road required to reach it. Rather, we need the explicit value of everything, so that we can judge how far down the road we have to go before we can be sure that we have gone "far enough" to achieve the engineering quality we require.

In this section, we outline how a retrenchment obtained in this way could be placed in the context of a development methodology of wider scope. For lack of space we touch on a number of technical issues that are only dealt with properly in the extended version of this paper. The key idea for the integration is the Tower Pattern, mentioned already in Section 5. This allows the extreme flexibility of retrenchment with its ability to accomodate a very wide variety of system properties, to be shored up with the much stricter guarantees that model based refinement offers, the latter coming at the price of

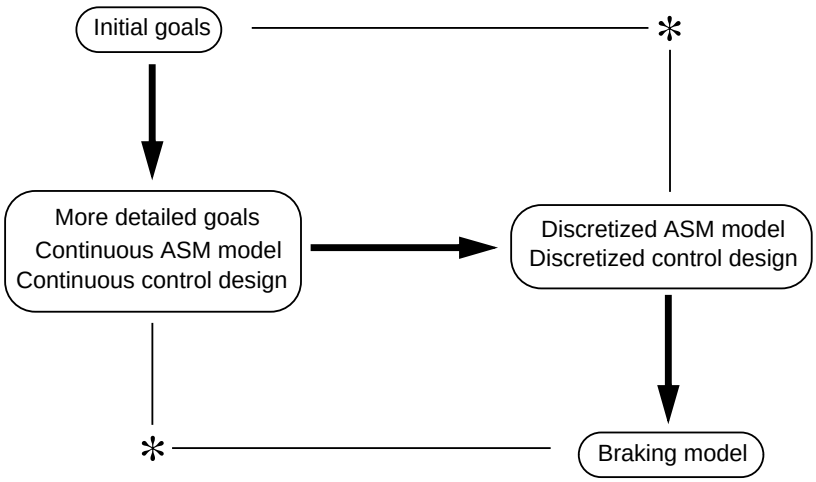


Figure 3: An overview of a complete development, starting with abstract goals, proceeding through explicit continuous and discrete deceleration models, and continuing with further low level models. Vertical arrows are (perhaps successive) model based refinements. Horizontal arrows are retrenchments, suited to relating models too different to be connected by refinement.

much more restricted expressivity as regards system properties. Although we do not have the space to discuss the point at length, we claim that a judicious combination of the two techniques can give better coverage of the route from high level domain centred requirements goals to low level implementation, than either technique alone. Thus on the one hand, use of refinement alone, forces the consideration of and commitment to, low level restrictions such as finiteness limits on arithmetic, far too early in the process, in order that all later models can (in effect) be conservative extensions of their predecessors. On the other hand, use of retrenchment alone makes it much harder to track how system properties evolve as the development proceeds, since successive models can be connected to their predecessors in a very loose manner, requiring much tighter focus on *post hoc* validation.

In our case, it is appropriate to use retrenchment to capture the properties of the discretization step, since that is something that has eluded model based refinement techniques.[5] However, either side of the discretization step, we are free to use refinement, since on each side individually, the models display much more consistency regarding the kind of properties that can be handled with sufficient eloquence using refinement alone.

The complete process that we have in mind may be summarized in Fig. 3. The thick arrows trace a path through a family of models that a development route could plausibly take. The left hand side of the diagram concerns continuous models. At the start, we have high level requirements goals, expressed in a notation with formal underpinnings. We have in mind a formalism like KAOS [30, 31] (or more precisely, an adaptation of it to deal more honestly with continuous processes). These requirements goals can then be formally refined till they can be *operationalized*, i.e. transformed into the operations of a methodology such as ASM (again, adapted to deal with continuous evolution). Then comes our discretization step, necessitating the use of retrenchment. Once we have crossed the continuous/discrete boundary, we are free to revert to traditional model based refinement techniques for discrete state transition systems — no

[5] It has to be noted that the introduction of approximate simulations has improved the situation recently with regard to stable systems, but in a more general context the observation remains true.

worries about continuous phenomena any more. In Fig. 3 we indicate how the discrete kinematics that we investigated earlier might be refined to a model of train braking, in which concern with the dynamics is replaced by a focus on the actuators that would implement the deceleration increments in practice.

Fig. 3 also features other models, indicated by asterisks. These are models whose existence is guaranteed by the Tower theorems [8, 28], making the squares of Fig. 3 commuting in an appropriate sense. However, we argue that these models are less useful than the others. Thus the lower left model would be a continuous version of the braking model, an unrealistic overidealisation so close to implementation. The upper right model would be a discretized version of the highest level requirements goals for train stopping. Again this would be inappropriate at such a high level, since it clutters what ought to be the most perspicuous expression of the system goals with a lot of material concerning low level details of the discretization scheme. This bears out what we said above about a combination of refinement and retrenchment techniques providing the best coverage of the route from high level requirements to low level implementation.

Above, we mentioned adaptations of KAOS and ASM to deal with continuous behaviour. We discuss these briefly now. Regarding ASM, a major part of what we need is already available in the literature, eg. [16, 39] which deal with *(Real) Timed ASM*. The essential observation is that in the context of continuous time, system states should be modeled as persisting over half-open half-closed time intervals, eg. $(t_0, t_1]$. This allows the typical discontinuous state transition in a typical discrete transition system, say of a state variable v, to be represented as the move from $v(t_0)$ (the value of v at t_0, which lies outside $(t_0, t_1]$ and is the right hand endpoint of the preceding interval), to $\lim_{\varepsilon \to 0+} v(t_0 + \varepsilon)$ (the left hand limit at t_0 from the right, of values of v within the interval $(t_0, t_1]$). Likewise, a period of continuous evolution can be understood as persisting over such a half-closed interval, governed by a suitably well posed ODE initial value problem, and with the truth of the initial conditions for the initial value problem at the end of the preceding interval being the trigger for the system's subsequently following a trajectory specified by the ODE problem. With these conventions, a version of ASM in which discrete steps alternate with continuous flows can be developed, reflecting many of the characteristics of hybrid automata.

A similar approach can be adopted for KAOS. Although KAOS depends on a notion of time from the outset, in the normal KAOS formalism, time is discrete, typically indexed by the integers, with requirements goals expressed as temporal logic formulae over time. For a version over continuous time, while some temporal operators, eg. *always*, *until*, offer no conceptual difficulties, the *next* operator needs to be rethought. Again half-open half-closed intervals, with successor states being defined via the limit from the right at the left hand end of a half-closed interval, can be used. To avoid problems arising due to an accumulation of *next* operators, syntactic restrictions have to be imposed on the permitted temporal formulae. However, the kinds of restrictions that need to be imposed are satisfied by the patterns that KAOS requirements are normally built out of.

8 Conclusion

In this paper we introduced a small continuous control problem in state space format, and then treated a discretized counterpart of it, utilising a zero order hold. Then came the main novel contribution of the paper, a rigorous treatment of the continuous to discrete modeling transformation, based on cited results from ODE theory. That done, we were able to integrate the results into a retrenchment which related from continuous and discrete models. As noted earlier, model based formal development normally starts already in the discrete domain, so the ability to connect this with the continuous world in a reasoned way, is a significant extension of the potential of model based formal techniques to underpin developments

of such systems. Equally importantly, in making essential use of retrenchment to forge the connection between continuous modeling and discrete modeling, this work gives a fresh confirmation of the utility of the concept as a worthwhile adjunct to refinement in tackling the wider issues connected with real world formal developments.

Of course, this paper is by no means the last word in developments of this kind. As well as tackling a control problem that was almost trivial technically, the rigorous result from mathematical control theory that we utilized was relatively limited, insisting, as it did, that the two behaviours that were compared, started from the same state, using a rather crude $\mathscr{L}^2$ estimate of the difference in the control inputs to derive its conclusion, and being based on rather generic properties of the ODEs that govern the dynamics of the control problem. (These simple contraints also meant that relatively little of the expressive power of retrenchment was used in this case study.) In more realistic cases, the problem will be less amenable to analytic solution, and feedback mechanisms will help alleviate the inherent uncertainty that arises. Moreover, while a crude $\mathscr{L}^2$ estimate of the difference in the control inputs allows the two control inputs to get as far away from each other as the bounds on the control space allow, in practice, feedback mechanisms will tend to push them together, and this could be exploited to derive more stringent estimates of the difference between continuous and discrete control. All of this remains to be discussed in future work, as does the extension of the KAOS and ASM formalisms (or any alternatives that might be contemplated to act in their place), that can encompass the continuous behaviours that we have described.

Our work is to be contrasted with the possibilities offerd by the hybrid systems approach [42]. There, the insistence on (approximate) bisimulation between a continuous system and a discrete counterpart restricts attention to control systems which are stable in the Liapunov sense. In any event, the intense focus on considerations of algorithmic decidability in that field, with automata homomorphism as such a prominent relationship between system models, can inhibit design expressivity for the purposes that concern us. For instance, techniques that rely on stability, are, strictly speaking, not applicable to our simple case study.

Once a suitable collection of widely applicable and useful results of the kind discussed here have been established, the way is open for the incorporation of these into appropriate formal development tools. These would be of a different flavour to those typically developed for the hybrid systems field, since they would have more emphasis on interactive proving than is typically the case there. One snag that would have to be overcome is that most proving based tools cope rather badly with the kind of applied mathematics and rigorous analysis techniques that are required for this work. A notable exception is the PVS suite [17, 35], for which substantial library support exists to underpin both applied mathematics and its more rigorous counterparts, eg. [21]. This would be the obvious jumping off point for the development of tools that aligned well with our approach.

References

[1] J-R. Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] J-R. Abrial (2010): *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[3] N. Ahmed (2006): *Dynamic Systems and Control With Applications*. World Scientific.

[4] R. Alur, C. Courcoubetis, T. Henzinger & P-H. Ho (1993): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems*. In: *Proc. Workshop on Theory of Hybrid Systems*, *LNCS* 736, Springer, pp. 209–229.

[5] R. Alur & D. Dill (1994): *A Theory of Timed Automata*. *Theor. Comp. Sci.* 126, pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[6] P. Antsaklis & A. Michel (2006): *Linear Systems*. Birkhauser.

[7] R. Banach: *Model Based Refinement and the Design of Retrenchments.* Available from [36].

[8] R. Banach & C. Jeske: *Retrenchment and Refinement Interworking: the Tower Theorems.* Submitted.

[9] R. Banach, C. Jeske & M. Poppleton (2008): *Composition Mechanisms for Retrenchment*. *J. Log. Alg. Prog.* 75, pp. 209–229, doi:10.1016/j.jlap.2007.11.001.

[10] R. Banach, M. Poppleton, C. Jeske & S. Stepney (2007): *Engineering and Theoretical Underpinnings of Retrenchment*. *Sci. Comp. Prog.* 67, pp. 301–329, doi:10.1016/j.scico.2007.04.002.

[11] S. Barnett (1975): *Introduction to Mathematical Control Theory*. Oxford University Press.

[12] E. Börger (2003): *The ASM Refinement Method*. *F.A.C.J.* 15, pp. 237–257.

[13] E. Börger & R.F. Stärk (2003): *Abstract State Machines. A Method for High Level System Design and Analysis*. Springer.

[14] F. Clarke (1987): *Optimization and Nonsmooth Analysis*. Society for Industrial Mathematics.

[15] F. Clarke, Y. Ledyaev, R. Stern & P. Wolenski (1997): *Nonsmooth Analysis and Control Theory*. Springer.

[16] J. Cohen & A. Slissenko (2008): *Implementation of Timed Abstract State Machines with Instantaneous Actions by Machines with Delays*. Technical Report TR-LACL-2008-2, LACL, University of Paris-12.

[17] J. Crow, S. Owre, J. Rushby, N. Shankar & M. Srivas (1995): *A Tutorial Introduction to PVS*. In R. France, S. Gerhart & M. Larrondo-Petrie, editors: *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, IEEE Computer Society Press.

[18] J. D'Azzo & C. Houpis (1995): *Linear Control System Analysis and Design: Conventional and Modern*. McGraw Hill.

[19] J Derrick & E Boiten (2001): *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer-Verlag UK, doi:10.1007/978-1-4471-0257-1.

[20] R. Dorf & R. Bishop (2010): *Modern Control Systems*. Pearson.

[21] B. Dutertre (1996): *Elements of Mathematical Analysis in PVS*. In: *TPHOLS 1996*, *LNCS* 1125, Springer.

[22] K. Dutton, S. Thompson & B. Barraclough (1997): *The Art of Control Engineering*. Addison Wesley.

[23] M. Fadali & A. Visioli (2009): *Digital Control Engineering: Analysis and Design*. Academic Press.

[24] G. Franklin, J. Powell & M. Workman (1996): *Digital Control Systems*. Prentice Hall.

[25] J. He (1994): *From CSP to hybrid systems*. In A.W. Roscoe, editor: *A Classical Mind, Essays in Honour of C.A.R. Hoare*, Prentice-Hall International, pp. 171–189.

[26] T. A. Henzinger (1996): *The Theory of Hybrid Automata*. In: *Proc. IEEE LICS-96*, IEEE, pp. 278–292. See also `http://mtc.epfl.ch/~tah/Publications/the_theory_of_hybrid_automata.pdf`.

[27] IEEE Standard 1474: IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements: IEEE Std 1474.1-2004; IEEE Standard for User Interface Requirements in Communications-Based Train Control (CBTC) Systems: IEEE Std 1474.2-2003; IEEE Recommended Practice for Communications-Based Train Control (CBTC) System Design and Functional Allocations: IEEE Std 1474.3-2008.

[28] C. Jeske (2005): *Algebraic Integration of Retrenchment and Refinement*. Ph.D. thesis, University of Manchester.

[29] B. Kuo (1992): *Digital Control Systems*. Oxford University Press.

[30] A. van Lamsweerde (2009): *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[31] Letier, E. (2001): *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Ph.D. thesis, Dépt. Ingénierie Informatique, Université Catholique de Louvain.

[32] K. Ogata (2008): *Modern Control Engineering*. Pearson.

[33] P. Paraskevopoulos (1996): *Digital Control Systems*. Prentice Hall.

[34] B Potter, J Sinclair & D Till (1996): *An Introduction to Formal Specification and Z*, 2nd. edition. Prentice Hall.

[35] PVS Homepage: `http://pvs.csl.sri.com`.

[36] Retrenchment Homepage: `http://www.cs.man.ac.uk/retrenchment`.

[37] W P de Roever & K Engelhardt (1998): *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press.

[38] E Sekerinski & K Sere (1998): *Program Development by Refinement: Case Studies Using the B-Method*. Springer.

[39] A. Slissenko & P. Vasilyev (2008): *Simulation of Timed Abstract State Machines with Predicate Logic model Checking*. *J.U.C.S.* 14, pp. 1984–2006.

[40] E. Sontag (1998): *Mathematical Control Theory*. Springer.

[41] W. Su, F. Yang, X. Wu, J. Gou & H. Zhu (2011): *Formal Approaches to Mode Conversion and Positioning for Vehicle Systems*. In: *Proc. 3rd IEEE International Workshop on Security Aspects of Process and Services Engineering*. To appear.

[42] P. Tabuada (2009): *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer.

[43] J Woodcock & J Davies (1996): *Using Z, Specification, Refinement and Proof*. Prentice Hall.

A CSP account of Event-B refinement

Steve Schneider
Department of Computing, University of Surrey
S.Schneider@surrey.ac.uk

Helen Treharne
Department of Computing, University of Surrey
H.Treharne@surrey.ac.uk

Heike Wehrheim
Department of Computer Science, University of Paderborn
wehrheim@uni-paderborn.de

Event-B provides a flexible framework for stepwise system development via refinement. The framework supports steps for (a) refining events (one-by-one), (b) splitting events (one-by-many), and (c) introducing new events. In each of the steps events can moreover possibly be anticipated or convergent. All such steps are accompanied with precise proof obligations. Still, it remains unclear what the exact relationship - in terms of a *behaviour-oriented semantics* - between an Event-B machine and its refinement is. In this paper, we give a CSP account of Event-B refinement, with a treatment for the first time of splitting events and of anticipated events. To this end, we define a CSP semantics for Event-B and show how the different forms of Event-B refinement can be captured as CSP refinement.

1 Introduction

Event-B [1] provides a framework for system development through stepwise refinement. Individual refinement steps are verified with respect to their proof obligations, and the transitivity of refinement ensures that the final system description is a refinement of the initial one. The refinement process allows new events to be introduced through the refinement process, in order to provide the more concrete implementation details necessary as refinement proceeds.

The framework allows for a great deal of flexibility as to cover a broad range of system developments. The recent book [1] comprising case studies from rather diverse areas shows that this goal is actually met. The flexibility is a result of the different ways of dealing with events during refinement. At each step existing events of an Event-B machine need to be refined. This can be achieved by (a) simply keeping the event as is, (b) refining it into another event, possibly because of a change of the state variables, or (c) splitting it into several events[1]. Furthermore, every refinement step allows for the introduction of new events. To help reasoning about divergence, events are in addition classified as ordinary, *anticipated* or *convergent*. Anticipated and convergent events both introduce new details into the machine specification. Convergent events must not be executed forever, while for anticipated events this condition is deferred to later refinement steps. All of these steps come with precise proof obligations; appropriate tool support helps in discharging these [3, 2]. Event-B is essentially a *state-based* specification technique, and proof obligations therefore reason about predicates on states.

Like Event-B, CSP comes with a notion of refinement. In order to understand their relationship, these two refinement concepts need to be set in a single framework. Both formalisms moreover support a variety of different forms of refinement: Event-B by means of several proof obligations related to refinement, out of which the system designer chooses an appropriate set; CSP by means of its different

[1] A fourth option is merging of events which we do not consider here.

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 139–154, doi:10.4204/EPTCS.55.9

semantic domains of traces, failures and divergences. The aim of this paper is to give a precise account of Event-B refinement in terms of CSP's behaviour-oriented process refinement. This will also provide the underlying results that support refinement in the combined formalism Event-B‖CSP. Our work is thus in line with previous studies relating state-based with behaviour-oriented refinement (see e.g. [5, 9, 4]). It turns out that CSP supports an approach to refinement consistent with that of Event-B. It faithfully reflects all of Event-B's possibilities for refinement, including splitting events and new events. It moreover also deals with the Event-B approach of anticipated events as a means to defer consideration of divergence-freedom. Our results involves support for individual refinement steps as well as for the resulting refinement chain.

The paper is structured as follows. The next section introduces the necessary background on Event-B and CSP. Section 3 gives the CSP semantics for Event-B based on weakest preconditions. In Section 4 we precisely fix the notion of refinement used in this paper, both for CSP and for Event-B, and Section 5 will then set these definitions in relation. It turns out that the appropriate refinement concept of CSP in this combination with Event-B is infinite-traces-divergences refinement. The last section concludes.

2 Background

We start with a short introduction to CSP and Event-B. For more detailed information see [17] and [1] respectively.

2.1 CSP

CSP, Communicating Sequential Processes, introduced by Hoare [11] is a formal specification language aiming at the description of communicating processes. A process is characterised by the events it can engage in and their ordering. Events will in the following be denoted by $a_1, a_2, \ldots$ or $evt0, evt1, \ldots$. Process expressions are built out of events using a number of composition operators. In this paper, we will make use of just three of them: *interleaving* ($P_1 \,|||\, P_2$), executing two processes in parallel without any synchronisation; *hiding* ($P \setminus N$), making a set N of events internal; and *renaming* ($f(P)$ and $f^{-1}(P)$), changing the names of events according to a renaming function f. If f is a non-injective function, $f^{-1}(P)$ will offer a choice of events b such that $f(b) = a$ whenever P offers event a.

Every CSP process P has an alphabet αP. Its semantics is given using the Failures/Divergences/Infinite Traces semantic model for CSP. This is presented as $\mathscr{U}$ in [16] or FDI in [17]. The semantics of a process can be understood in terms of four sets, T, F, D, I, which are respectively the traces, failures, divergences, and infinite traces of P. These are understood as observations of possible executions of the process P, in terms of the events from αP that it can engage in.

Traces are finite sequences of events from P's alphabet: $tr \in \alpha P^*$. The set $traces(P)$ represents the possible finite sequences of events that P can perform. Failures will not be considered in this paper and are therefore not explained here.

Divergences are finite sequences of events on which the process might diverge: perform an infinite sequence of internal events (such as an infinite loop) at some point during or at the end of the sequence. The set $divergences(P)$ is the set of all possible divergences for P. Infinite traces $u \in \alpha P^\omega$ are infinite sequences of events. The set $infinites(P)$ is the set of infinite traces that P can exhibit. For technical reasons it also contains those infinite traces which have some prefix which is a divergence.

Definition 2.1 *A process P is* divergence-free *if $divergences(P) = \{\}$.*

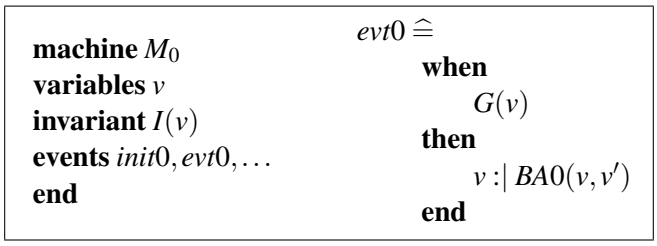


Figure 1: Template of an Event-B machine and an event.

We use tr to refer to finite traces. These can also be written explicitly as $\langle a_1, a_2, \ldots, a_n \rangle$. The empty trace is $\langle\rangle$, concatenation of traces is written as $tr_1 \frown tr_2$. We use u to refer to infinite traces. Given a set of events A, the *projections* $tr \upharpoonright A$ and $u \upharpoonright A$ are the traces restricted to only those events in A. Note that $u \upharpoonright A$ might be finite, if only finitely many A events appear in u. Conversely, $tr \setminus A$ and $u \setminus A$ are those traces with the events in A removed. The length operator $\#tr$ and $\#u$ gives the length of the trace it is applied to. As a first observation, we get the following.

Lemma 2.2 *If P is divergence-free, and for any infinite trace u of P we have $\#(u \setminus A) = \infty$, then $P \setminus A$ is divergence-free.*

Proof 2.3 *Follows immediately from the semantics of the hiding operator.*

Later, we furthermore use *specifications* on traces or, more generally, on CSP processes. Specifications are given in terms of predicates. If S is a predicate on a particular semantic element, then we write P **sat** S to denote that all relevant elements in the semantics of P meet the predicate S. For example, if $S(u)$ is a predicate on infinite traces, then P **sat** $S(u)$ is equivalent to $\forall u \in infinites(P) \,.\, S(u)$.

2.2 Event-B

Event-B [1, 13] is a state-based specification formalism based on set theory. Here we describe the basic parts of an Event-B machine required for this paper; a full description of the formalism can be found in [1].

A machine specification usually defines a list of variables, given as v. Event-B also in general allows sets s and constants c. However, for our purposes the treatment of elements such as sets and constants are independent of the results of this paper, and so we will not include them here. However, they can be directly incorporated without affecting our results.

There are many clauses that may appear in Event-B machines, and we concentrate on those clauses concerned with the state. We will therefore describe a machine M_0 with a list of state variables v, a state invariant $I(v)$, and a set of events $evt0, \ldots$ to update the state (see left of Fig.1). Initialisation is a special event $init0$.

A machine M_0 will have various proof obligations on it. These include consistency obligations, that events preserve the invariant. They can also include (optional) deadlock-freeness obligations: that at least one event guard is always true.

Central to an Event-B description is the definition of the events, each consisting of a *guard* $G(v)$ over the variables, and a *body*, usually written as an assignment S on the variables. The body defines a *before-after predicate* $BA(v, v')$ describing changes of variables upon event execution, in terms of the relationship between the variable values before (v) and after (v'). The body can also be written as $v :|$

$BA(v,v')$, whose execution assigns to v any value v' which makes the predicate $BA(v,v')$ true (see right of Fig. 1).

3 CSP semantics for Event-B machine

Event-B machines are particular instances of action systems, so Morgan's CSP semantics for action systems [14] allows traces, failures, and divergences to be defined for Event-B machines, in terms of the sequences of events that they can and cannot engage in. Butler's extension to handle unbounded nondeterminism [6] defines the infinite traces for action systems. These together give a way of considering Event-B machines as CSP processes, and treating them within the CSP semantic framework. In this paper we use the infinite traces model in order to give a proper treatment of divergence under hiding. This is required to establish our main result concerning divergence-freedom under hiding of new events. Consideration of finite traces alone is not sufficient for this result.

Note that the notion of *traces* for machines is different to that presented in [1], where traces are considered as sequences of *states* rather than our treatment of traces as sequences of *events*.

The CSP semantics is based on the weakest precondition semantics of events. Let S be a statement (of an event). Then $[S]R$ denotes the weakest precondition for statement S to establish postcondition R. Weakest preconditions for events of the form "**when** $G(v)$ **then** $S(v)$ **end**" are given by considering them as guarded commands:

$$[\,\textbf{when}\ G(v)\ \textbf{then}\ S(v)\ \textbf{end}]P \quad = \quad G(v) \Rightarrow [S(v)]P$$

Events in the general form "**when** $G(v)$ **then** $v :| BA(v,v')$ **end**" have a weakest precondition semantics as follows:

$$[\,\textbf{when}\ G(v)\ \textbf{then}\ v :| BA(v,v')\ \textbf{end}]P \quad = \quad G(v) \Rightarrow \forall x.(BA(v,x) \Rightarrow P[x/v])$$

Observe that for the case $P = true$ we have

$$[\,\textbf{when}\ G(v)\ \textbf{then}\ v :| BA(v,v')\ \textbf{end}]true \quad = \quad true$$

Based on the weakest precondition, we can define the traces, divergences and infinite traces of an Event-B machine[2].

Traces The traces of a machine M are those sequences of events $tr = \langle a_1, \ldots, a_n \rangle$ which are possible for M (after initialisation *init*): those that do not establish *false*:

$$traces(M) \quad = \quad \{tr \mid \neg[init;tr]false\}$$

Here, the weakest precondition on a sequence of events is the weakest precondition of the sequential composition of those events: $[\langle a_1, \ldots, a_n \rangle]P$ is given as $[a_1;\ \ldots\ ;\ a_n]P = [a_1](\ldots([a_n]P)\ldots)$.

Divergences A sequence of events tr is a divergence if the sequence of events is not guaranteed to terminate, i.e. $\neg[init;\ tr]true$. Thus

$$divergences(M) \quad = \quad \{tr \mid \neg[init;tr]true\}$$

Note that any Event-B machine M with events of the form *evt* given above is divergence-free. This is because $[evt]true = true$ for such events (and for *init*), and so $[init;\ tr]true = true$. Thus no potential divergence tr meets the condition $\neg[init;\ tr]true$.

[2] Failures can be defined as well but are omitted since they are not needed for our approach.

Infinite Traces The technical definition of infinite traces is given in [6], in terms of least fixed points of predicate transformers on infinite vectors of predicates. Informally, an infinite sequence of events $u = \langle u_0, u_1, \ldots \rangle$ is an infinite trace of M if there is an infinite sequence of predicates P_i such that $\neg[init](\neg P_0)$ (i.e. some execution of *init* reaches a state where P_0 holds), and $P_i \Rightarrow \neg[u_i](\neg P_{i+1})$ for each i (i.e. if P_i holds then some execution of u_i can reach a state where P_{i+1} holds).

$$\begin{array}{lcl} infinites(M) & = & \{u \mid \text{there is a sequence} \langle P_i \rangle_{i \in \mathbb{N}} \,.\ \neg[init](\neg P_0) \wedge \\ & & \qquad \text{for all } i \,.\ P_i \Rightarrow \neg[u_i](\neg P_{i+1}) \ \} \end{array}$$

These definitions give the CSP Traces/Divergences/Infinite Traces semantics of Event-B machines in terms of the weakest precondition semantics of events.

4 Refinement

In this paper, we intend to give a CSP account of Event-B refinement. The previous section provides us with a technique for relating Event-B machines to the semantic domain of CSP processes. Next, we will briefly rephrase the refinement concepts in CSP and Event-B before explaining Event-B refinement in terms of CSP refinement.

4.1 CSP refinement

Based on the semantic domains of traces, failures, divergences and infinite traces, different forms of refinement can be given for CSP. The basic idea underlying these concepts is - however - always the same: the refining process should not exhibit a behaviour which was not possible in the refined process. The different semantic domains then supply us with different forms of "behaviour". In this paper we will use the following refinement relation, based on traces and divergences:

$$\begin{array}{lcl} P \sqsubseteq_{TDI} Q & \widehat{=} & traces(Q) \subseteq traces(P) \\ & & \wedge\ divergences(Q) \subseteq divergences(P) \\ & & \wedge\ infinites(Q) \subseteq infinites(P) \end{array}$$

Refinement in Event-B also allows for the possibility of introducing new events. To capture this aspect in CSP, we need a way of incorporating this into process refinement. As a first idea, we could *hide* the new events in the refining process. This potentially introduces divergences, namely, when there is an infinite sequence of new events in the infinite traces. In order to separate out consideration of divergence from reasoning about traces, we will use $P \,|||\, RUN_N$ as a lazy abstraction operator instead. RUN_N defines a divergence free process capable of executing any order of events from the set N. This will enable us to characterise Event-B refinement introducing new events in CSP terms. The following lemma gives the relationship between refinement involving interleaving, and refinement involving hiding.

Lemma 4.1 *If* $P_0 \,|||\, RUN_N \sqsubseteq_{TDI} P_1$ *and* $N \cap \alpha P_0 = \{\}$ *and* $P_1 \setminus N$ *is divergence-free, then* $P_0 \sqsubseteq_{TDI} P_1 \setminus N$.

Proof: Assume that (1) $P_0 \,|||\, RUN_N \sqsubseteq_{TDI} P_1$, (2) $N \cap \alpha P_0 = \{\}$ and (3) $P_1 \setminus N$ is divergence-free. We need to show that the (finite and infinite) traces as well as divergences of $P_1 \setminus N$ are contained in those of P_0.

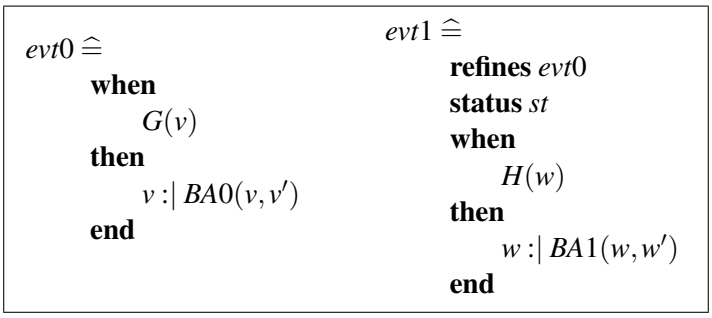


Figure 2: An event and its refinement

Traces Let $tr \in traces(P_1 \setminus N)$. By semantics of hiding there is some $tr' \in traces(P_1)$ s.t. $tr' \setminus N = tr$. By (1) $tr' \in traces(P_0 \,|||\, RUN_N)$. By (2) and the semantics of $|||$ we get $tr' \setminus N \in traces(P_0)$ and thus $tr \in traces(P_0)$.

Divergences By (3) $divergences(P_1 \setminus N) = \{\}$, thus nothing to be proven here.

Infinites Let $u \in infinites(P_1 \setminus N)$. By the semantics of hiding there is some $u' \in infinites(P_1)$ such that $u' \setminus N = u$ and $\#(u' \setminus N) = \infty$. By (1) $u' \in infinites(P_0 \,|||\, RUN_N)$ and by (2) and semantics of interleave we get $u' \setminus N = u \in infinites(P_0)$.

□

4.2 Event-B refinement

In Event-B, the (intended) refinement relationship between machines is directly written into the machine definitions. As a consequence of writing a refining machine, a number of proof obligations come up. Here, we assume a machine and its refinement to take the following form:

machine M_0
variables v
invariant $I(v)$
events $init0, evt0, \ldots$
end

machine M_1
refines M_0
variables w
invariant $J(v,w)$
events $init1, evt1, \ldots$
variant $V(w)$
end

The machine M_0 is actually refined by machine M_1, written $M_0 \preccurlyeq M_1$, if the given *linking invariant* J on the variables of the two machines is established by their initialisations, and preserved by all events, in the sense that any event of M_1 can be matched by an event of M_0 (or *skip* for newly introduced events) to maintain J. This is the standard notion of downwards simulation data refinement [8]. We next look at this in more detail, and in particular give the proof obligations associated to these conditions.

First of all, we need to look at events again. Figure 2 gives the shape of an event and its refinement. We see that an event in the refinement now also gets a *status*. The status can be ordinary (also called *remaining*), or *anticipated* or *convergent*. Convergent events are those which must not be executed forever, and anticipated events are those that will be made convergent at some later refinement step. New events must either have status anticipated or convergent. Both of these introduce further proof obligations: to prevent execution "forever" the refining machine has to give a variant V (see above in

M_1), and V has to be decreased by every convergent event and must not be increased by anticipated events.

We now describe each of the proof obligations in turn. We have simplified them from their form in [13] by removing explicit references to sets and constants. Alternative forms of these proof obligations are given in [1, Section 5.2: Proof Obligation Rules].

FIS_REF: Feasibility Feasibility of an event is the property that, if the event is enabled (i.e. the guard is true), then there is some after-state. In other words, the body of the event will not block when the event is enabled.

The rule for feasibility of a concrete event is:

$$\frac{I(v) \wedge J(v,w) \wedge H(w) \vdash \exists w'.BA1(w,w')}{\textbf{FIS\_REF}}$$

GRD_REF: Guard Strengthening This requires that when a concrete event is enabled, then so is the abstract one. The rule is:

$$\frac{I(v) \wedge J(v,w) \wedge H(w) \vdash G(v)}{\textbf{GRD\_REF}}$$

INV_REF: Simulation This ensures that the occurrence of events in the concrete machine can be matched in the abstract one (including the initialization event). New events are treated as refinements of *skip*. The rule is:

$$\frac{I(v) \wedge J(v,w) \wedge H(w) \wedge BA1(w,w') \vdash \exists v'.(BA0(v,v') \wedge J(v',w'))}{\textbf{INV\_REF}}$$

Event-B also allows a variety of further proof obligations for refinement, depending on what is appropriate for the application. The two parts of the variant rule WFD_REF below must hold respectively for all convergent and anticipated events, including all newly-introduced events.

WFD_REF: Variant This rule ensures that the proposed variant V satisfies the appropriate properties: that it is a natural number, that it decreases on occurrence of any convergent event, and that it does not increase on occurrence of any anticipated event:

$$\frac{I(v) \wedge J(v,w) \wedge H(w) \wedge BA1(w,w') \vdash V(w) \in \mathbb{N} \wedge V(w') < V(w)}{\textbf{WFD\_REF (convergent event)}}$$

$$\frac{I(v) \wedge J(v,w) \wedge H(w) \wedge BA1(w,w') \vdash V(w) \in \mathbb{N} \wedge V(w') \leqslant V(w)}{\textbf{WFD\_REF (anticipated event)}}$$

We will use the refinement relation $M_0 \preccurlyeq M_1$ to mean that the four proof obligations *FIS_REF*, *GRD_REF*, *INV_REF*, and *WFD_REF* hold between abstract machine M_0 and concrete machine M_1.

5 Event-B refinement as CSP refinement

With these definitions in place, we can now look at our main issue, the characterisation of Event-B refinement via CSP refinement. Here, we in particular need to look at the different forms of events in Event-B during refinement. Events can have status convergent or anticipated, or might have no status. This partitions the set of events of M into three sets: anticipated A, convergent C, and remaining events R (neither anticipated nor convergent). The alphabet of M, the set of all possible events, is thus given by $\alpha M = A \cup C \cup R$. In the CSP refinement, these will take different roles.

Now consider an Event-B Machine M_0 and its refinement M_1: $M_0 \preccurlyeq M_1$. The machine M_0 has anticipated events A_0, convergent events C_0, and remaining events R_0, and M_1 similarly has event sets A_1, C_1, and R_1. Each event ev_1 in M_1 either refines a single event ev_0 in M_0 (indicated by the clause 'refines ev_0' in the description of ev_1) or does not refine any event of M_0. The set of new events N_1 is those events which are not refinements of events in M_0.

$M_0 \preccurlyeq M_1$ thus induces a partial surjective function $f_1 : \alpha M_1 \twoheadrightarrow \alpha M_0$ where $f_1(ev_1) = ev_0 \Leftrightarrow ev_1$ refines ev_0. Observe that αM_1 is partitioned by $f_1^{-1}(\alpha M_0)$ and N_1. The rules for refinement between events in Event-B impose restrictions on these sets:

1. each event of M_0 is refined by at least one event of M_1;
2. each new event in M_1 is either anticipated or convergent;
3. each event in M_1 which refines an anticipated event of M_0 is itself either convergent or anticipated;
4. refinements of convergent or remaining events of M_0 are remaining in M_1, i.e. they are not given a status.

The conditions imposed by the rules are formalised as follows:

1. $ran(f_1) = A_0 \cup C_0 \cup R_0$;
2. $N_1 \subseteq A_1 \cup C_1$;
3. $f_1^{-1}(A_0) \subseteq A_1 \cup C_1$;
4. $f_1^{-1}(C_0 \cup R_0) = f_1^{-1}(C_0) \cup f_1^{-1}(R_0) = R_1$.

These relationships between the classes of events are illustrated in Figure 3.

5.1 New events

For the new events arising in the refinement, we can use the lazy abstraction operator via the *RUN* process to get our desired result, disregarding the issue of divergence for a moment. The following lemma gives our first result on the relationship between Event-B refinement and CSP refinement.

Lemma 5.1 *If $M_0 \preccurlyeq M_1$ and the refinement introduces new events N_1 and uses the mapping f_1, then $f_1^{-1}(M_0) \;|||\; RUN_{N_1} \sqsubseteq_{TDI} M_1$.*

Proof: We assume state variables of M_0 and M_1 named as given above, i.e. state variables of M_0 are v and of M_1 are w. Let $tr = \langle a_1, \ldots, a_n \rangle \in traces(M_1)$. We need to show that $tr \in traces(f_1^{-1}(M_0) \;|||\; RUN_{N_1})$. First of all note that the interleaving operator merges the traces of two processes together, i.e., the traces of $f_1^{-1}(M_0) \;|||\; RUN_{N_1}$ are simply those of $f_1^{-1}(M_0)$ with new events arbitrarily inserted. The proof proceeds by induction on the length of the trace.

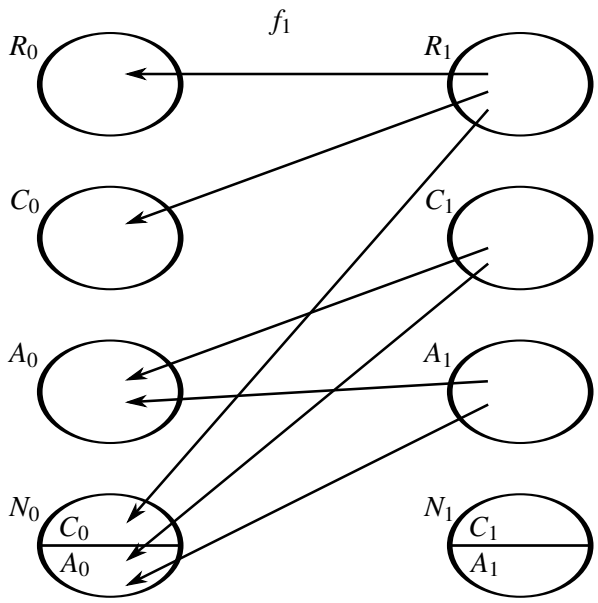


Figure 3: Relationship between events in a refinement step: f_1 maps events in M_1 to events in M_0 that they refine.

Induction base Assume $n = 0$, i.e., $tr = \langle\rangle$. By definition this means that the initialisation event *init*1 has been executed bringing the machine M_1 into a state w_1. By INV_REF (using init as event), we find a state v_1 such that $J(v_1, w_1)$ and furthermore $\langle\rangle \in traces(M_0)$ and hence also in $traces(f_1^{-1}(M_0) \,|||\, RUN_{N_1})$.

Induction step Assume that for a trace $tr = \langle a_1, \ldots, a_{j-1}\rangle \in traces(M_1)$ we have already shown that $tr \in traces(f_1^{-1}(M_0) \,|||\, RUN_{N_1})$ and this has led us to a pair of states v_{j-1}, w_{j-1} such that $J(v_{j-1}, w_{j-1})$. Now two cases need to be considered:

1. $a_j \notin N_1$: Assume a_j in M_1 to be of the form

   $$\textbf{when } H(w) \textbf{ then } w :| BA1(w, w') \textbf{ end}$$

   and $f_1(a_j)$ in M_0 of the form

   $$\textbf{when } G(v) \textbf{ then } v :| BA(v, v') \textbf{ end}$$

   Since a_j is executed in w_{j-1} we have $H(w_{j-1})$. By GRD_REF we thus get $G(v_{j-1})$. Furthermore, for w_j with $BA1(w_{j-1}, w_j)$ we find – by INV_REF – a state v_j such that $J(v_j, w_j)$ and $BA(v_{j-1}, v_j)$. Hence $tr^\frown\langle a_j\rangle \in traces(f_1^{-1}(M_0) \,|||\, RUN_{N_1})$.
2. $a_j \in N_1$: Similar to the previous case. Here, a_j refines skip and thus $v_j = v_{j-1}$ and the event a_j is coming from RUN_{N_1}.

In the same way we can carry out a proof for infinite traces. For divergences it is even simpler as $divergences(M_1) = \{\}$. □

This lemma can be generalised to a chain of refinement steps. For this, we assume that we are given a sequence of Event-B machines M_i with their associated processes P_i, and every refinement step introduces some set of new events N_i.

Theorem 5.2 *If a sequence of processes P_i, mappings f_i, and sets N_i are such that*

$$f_{i+1}^{-1}(P_i) \,|||\, RUN_{N_{i+1}} \quad \sqsubseteq_{TDI} \quad P_{i+1} \tag{1}$$

for each i, then

$$f_n^{-1}(\ldots(f_1^{-1}(P_0))\ldots) \,|||\, RUN_{f_n^{-1}(..f_2^{-1}(N_1)...)\cup\ldots\cup f_n^{-1}(N_{n-1})\cup N_n} \quad \sqsubseteq_{TDI} \quad P_n$$

Proof: Two successive refinement steps combine to provide a relationship between P_0 and P_2 of the same form as Line 1 above, as follows:

$$\begin{array}{rlll}
f_2^{-1}(P_1) \,|||\, RUN_{N_2} & \sqsubseteq_{TDI} & P_2 & \text{(given)} \\
f_2^{-1}(f_1^{-1}(P_0) \,|||\, RUN_{N_1}) \,|||\, RUN_{N_2} & \sqsubseteq_{TDI} & P_2 & \text{(line (1), transitivity of } \sqsubseteq) \\
f_2^{-1}(f_1^{-1}(P_0)) \,|||\, RUN_{f_2^{-1}(N_1)} \,|||\, RUN_{N_2} & \sqsubseteq_{TDI} & P_2 & (\text{Law: } f^{-1}(P \,|||\, Q) = f^{-1}(P) \,|||\, f^{-1}(Q)) \\
f_2^{-1}(f_1^{-1}(P_0)) \,|||\, RUN_{f_2^{-1}(N_1)\cup N_2} & \sqsubseteq_{TDI} & P_2 & (\text{Law: } RUN_A \,|||\, RUN_B = RUN_{A\cup B})
\end{array}$$

Hence the whole chain of refinement steps can be collected together, yielding the result. □

5.2 Convergent and anticipated events

The previous result lets us relate the first and last Event-B machine in a chain of refinements. Due to the lazy abstraction operator (and the resulting possibility of defining refinement without hiding new events), we considered divergence free processes there: all processes P_i representing Event-B machines, are divergence free by definition. However, Event-B refinement is concerned with a particular form of divergence and its avoidance. A sort of divergence would arise when new events (or more specifically, convergent events) could be executed forever, and this is what the proof rules for variants rule out.

We would like to capture the impact of convergence and anticipated sets of events in the CSP semantics as well. To do so, we first of all define the specification predicate

$$CA(C,R)(u) \quad \widehat{=} \quad (\#(u \upharpoonright C) = \infty \Rightarrow \#(u \upharpoonright R) = \infty)$$

Intuitively, this states that all infinite traces having infinitely many convergent (C) events also have infinitely many (R) remaining events (and thus cannot execute convergent events alone forever). In this case we say that the Event-B machine *does not diverge on C events*.

Definition 5.3 *Let M be an Event-B machine with its alphabet αM containing event sets C and R with $C \cap R = \{\}$. M* does not diverge on C events *if M* **sat** $CA(C,R)$.

Convergent events in Event-B machines only come into play during refinement. Thus a plain, single Event-B machine has no convergent events ($C = \{\}$) and thus trivially satisfies the specification predicate.

Lemma 5.4 *If $M_0 \preccurlyeq M_1$, and M_1 has convergent, anticipated, and remaining events C_1, A_1, and R_1 respectively, then M_1* **sat** $CA(C_1,R_1)$

Proof: We prove this by contradiction. Assume $\neg M_1$ **sat** $CA(C_1,R_1)$. Then there is some $u \in infinites(M_1)$ such that $\#(u \upharpoonright C_1) = \infty$ and $\#(u \upharpoonright R_1) < \infty$. Then there must be some tr_0, u' such that $u = tr_0 \frown u'$ with $u' \in (C_1 \cup A_1)^\omega$ (i.e. tr_0 is a prefix of u containing all the R_1 events). Moreover, $\#u' \upharpoonright C_1 = \infty$.

Now since $M_0 \preccurlyeq M_1$ we have by GRD_REF and INV_REF that there is some pair of states (v,w) (abstract and concrete state) reached after executing tr_0 for which $J(v,w)$ and $I(v)$ is true. Furthermore,

$V(w)$ is a natural number. Also by $M_0 \preccurlyeq M_1$ we have an infinite sequence of pairs of states (v_i, w_i) (for the remaining infinite trace u') such that $J(v_i, w_i)$. Since each event in u' is in A_1 or C_1 we have from WFD_REF that $V(w_{i+1}) \leqslant V(w_i)$ for each i. Further, for infinitely many i's (i.e. those events in C_1) we have $V(w_{i+1}) < V(w_i)$. Thus we have a sequence of values $V(w_i)$ decreasing infinitely often without ever increasing. This contradicts the fact that the $V(w_i) \in \mathbb{N}$. □

A number of further interesting properties can be deduced for the specification predicate *CA*.

Lemma 5.5 *Let P be a CSP process and $C, C', R \subseteq \alpha P$ nonempty finite sets of events.*

1. *If P **sat** $CA(C,R)$ then $f^{-1}(P)$ **sat** $CA(f^{-1}(C), f^{-1}(R))$.*
2. *If P **sat** $CA(C,R)$ and $N \cap C = \{\}$ then $P \,|||\, RUN_N$ **sat** $CA(C,R)$.*
3. *If P **sat** $CA(C,R)$ and P **sat** $CA(C', C \cup R)$ then P **sat** $CA(C \cup C', R)$.*
4. *If P **sat** $CA(C,R)$ and $C \cap R = \{\}$ then $P \setminus C$ is divergence-free.*

Proof:

1. Assume that $u \in \mathit{infinites}(f^{-1}(P))$ and $\#(u \upharpoonright f^{-1}(C)) = \infty$. From the first we get $f(u) \in \mathit{infinites}(P)$. From the latter it follows that $\#(f(u) \upharpoonright C) = \infty$. With P **sat** $CA(C,R)$ we have $\#(f(u) \upharpoonright R) = \infty$ and hence $\#(u \upharpoonright f^{-1}(R)) = \infty$.
2. Let $u \in \mathit{infinites}(P \,|||\, RUN_N)$ and $\#(u \upharpoonright C) = \infty$. With $N \cap C = \{\}$ we get $\#((u \setminus N) \upharpoonright C) = \infty$. By definition of $|||$ we have $u \setminus N \in \mathit{infinites}(P)$ ($u \setminus N$ is infinite since $\#((u \setminus N) \upharpoonright C) = \infty$). By P **sat** $CA(C,R)$ we get $\#((u \setminus N) \upharpoonright R) = \infty$, hence $\#(u \upharpoonright R) = \infty$.
3. Let $u \in \mathit{infinites}(P)$ such that $\#(u \upharpoonright (C \cup C')) = \infty$. Both C and C' are finite sets hence either $\#(u \upharpoonright C) = \mathit{infty}$ or $\#(u \upharpoonright C') = \infty$ (or both). In the first case we get $\#(u \upharpoonright R) = \infty$ by P **sat** $CA(C,R)$. In the second case it follows that $\#(u \upharpoonright (C \cup R)) = \infty$ and hence again $\#(u \upharpoonright C) = \infty$ or directly $\#(u \upharpoonright R) = \infty$.
4. First of all note that if P **sat** $CA(C,R)$ then P is divergence free. Now assume that there is a trace $tr \in \mathit{divergences}(P \setminus C)$. Then there exists a trace $u \in \mathit{infinites}(P)$ such that $tr = u \setminus C$, and so $\#(u \setminus C) < \infty$. Hence $\#(u \upharpoonright C) = \infty$. However, — as $C \cap R = \{\}$ — $\#(u \upharpoonright R) \neq \infty$ which contradicts P **sat** $CA(C,R)$.

□

The most interesting of these properties is probably the last one: it relates the specification predicate to the definition of divergence freedom in CSP. In CSP, a process does not diverge on a set of events C if $P \setminus C$ is divergence-free.

This gives us some results about the specification predicate for single Event-B machines and CSP processes. Next, we would like to apply this to refinements. First, we again consider just two machines.

Lemma 5.6 *Let $M_0 \preccurlyeq M_1$ with an associated refinement function f_1 and let M_0 **sat** $CA(C_0, R_0)$. Then M_1 **sat** $CA(f_1^{-1}(C_0) \cup C_1\ ,\ f_1^{-1}(R_0))$.*

Proof: Assume $u \in \mathit{infinites}(M_1)$ and $\#(u \upharpoonright (f_1^{-1}(C_0) \cup C_1) = \infty$. We aim to establish that $\#(u \upharpoonright f_1^{-1}(R_0)) = \infty$. We have $\#(u \upharpoonright f_1^{-1}(C_0)) = \infty$ or $\#(u \upharpoonright C_1) = \infty$.

In the former case, Lemma 5.1 yields that $f_1(u \upharpoonright f^{-1}(\alpha M_0)) \in \mathit{infinites}(M_0)$. Then

$$
\begin{array}{rl}
\#(u \upharpoonright f_1^{-1}(C_0)) = \infty & \text{(given)} \\
\#(f_1(u \upharpoonright f^{-1}(C_0)) \upharpoonright C_0) = \infty & \text{(since renaming preserves length)} \\
\#(f_1(u \upharpoonright f^{-1}(\alpha M_0)) \upharpoonright C_0) = \infty & \text{(since } C_0 \subseteq \alpha M_0\text{)} \\
\#(f_1(u \upharpoonright f^{-1}(\alpha M_0)) \upharpoonright R_0) = \infty & \text{(by } M_0 \textbf{ sat } CA(C_0, R_0)\text{)} \\
\#(u \upharpoonright f^{-1}(\alpha M_0)) \upharpoonright f^{-1}(R_0) = \infty & \text{(since renaming preserves length)} \\
\#(u \upharpoonright f_1^{-1}(R_0)) = \infty & \text{(since } R_0 \subseteq \alpha M_0\text{)}
\end{array}
$$

In the latter case Lemma 5.4 yields that $\#(u \upharpoonright R_1) = \infty$. Then

$$\begin{array}{rl} \#(u \upharpoonright R_1) = \infty & \\ \#(u \upharpoonright f_1^{-1}(R_0 \cup C_0)) = \infty & (\text{since } R_1 = f_1^{-1}(C_0 \cup R_0)) \\ \#(u \upharpoonright f_1^{-1}(R_0)) = \infty \vee \#(u \upharpoonright f_1^{-1}(C_0)) = \infty & \end{array}$$

The first disjunct is the desired result, the second is the one already treated above.

□

Note that by Lemma 5.5 (4) the above result implies that the machine M_1 does not diverge on $f_1^{-1}(C_0) \cup C_1$, in particular $M_0 \setminus (f_1^{-1}(C_0) \cup C_1)$ is divergence-free.

Similar to the previous case, we can lift this to chains of refinement steps. Consider the last result with respect to two refinement steps $M_0 \preccurlyeq M_1 \preccurlyeq M_2$:

$$\begin{array}{rlll} M_0 & \textbf{sat} & CA(C_0, R_0) & (\text{given}) \\ f^{-1}(M_0) & \textbf{sat} & CA(f^{-1}(C_0), f^{-1}(R_0)) & (\text{lemma 5.5 (1)}) \\ f^{-1}(M_0) \,|||\, RUN_{N_1} & \textbf{sat} & CA(f^{-1}(C_0), f^{-1}(R_0)) & (\text{lemma 5.5 (2), since } f_1^{-1}(C_0) \cap N_1 = \{\}) \\ M_1 & \textbf{sat} & CA(f^{-1}(C_0), f^{-1}(R_0)) & (\text{lemma 5.1}) \\ f_2^{-1}(M_1) & \textbf{sat} & CA(f_2^{-1}(f^{-1}(C_0)), f_2^{-1}(f^{-1}(R_0))) & (\text{lemma 5.5 (1)}) \\ f_2^{-1}(M_1) \,|||\, RUN_{N_2} & \textbf{sat} & CA(f_2^{-1}(f^{-1}(C_0)), f_2^{-1}(f^{-1}(R_0))) & (\text{lemma 5.5 (2)}) \\ M_2 & \textbf{sat} & CA(f_2^{-1}(f^{-1}(C_0)), f_2^{-1}(f^{-1}(R_0))) & (\text{lemma 5.1}) \\ M_2 & \textbf{sat} & CA(C_2 \cup f_2^{-1}(C_1)\,,\, f_2^{-1}(R_1)) & (\text{lemma 5.6}) \end{array}$$

Then by applying Lemma 5.5(3) to the final two lines, with $R = f_2^{-1}(f_1^{-1}(R_0))$, $C = f_2^{-1}(f_1^{-1}(C_0))$, and $C' = C_2 \cup f_2^{-1}(C_1)$, we obtain

$$M_2 \quad \textbf{sat} \quad CA(C_2 \cup f_2^{-1}(C_1) \cup f_2^{-1}(f_1^{-1}(C_0))\,,\, f_2^{-1}(f_1^{-1}(R_0))$$

Thus if

$$M_0 \preccurlyeq M_1 \preccurlyeq \ldots \preccurlyeq M_n$$

then collecting together all the steps yields that

$$M_n \quad \textbf{sat} \quad CA((f_n^{-1}(\ldots f_1^{-1}(C_0)\ldots) \cup \ldots f_n^{-1}(C_{n-1}) \cup C_n)\,, f_n^{-1}(\ldots f_1^{-1}(R_0)\ldots)) \tag{2}$$

Finally, we would like to put together these results into one result relating the initial machine M_0 to the final machine M_n in the refinement chain. This result should use hiding for the treatment of new events, and – by stating the relationship between M_0 and $M_n \setminus \{\textit{new events}\}$ via infinite-traces-divergences refinement – show that Event-B refinement actually does not introduce divergences on new events. For such chains of refinement steps we always assume that $A_0 = C_0 = \{\}$ (initially we have neither anticipated nor convergent events), and $A_n = \{\}$ (at the end all anticipated events have become convergent).

For this, we first of all need to find out what the "new events" are in the final machine. Define $g_{i,j}$ as the functional composition of the event mappings from f_j to f_i:

$$g_{i,j} \quad = \quad f_i; f_{i+1}; \ldots; f_j$$

Then noting the disjointness of the union, by repeated application of

$$C_j \uplus A_j \uplus R_j \quad = \quad f_j^{-1}(C_{j-1} \uplus A_{j-1} \uplus R_{j-1}) \uplus N_j$$

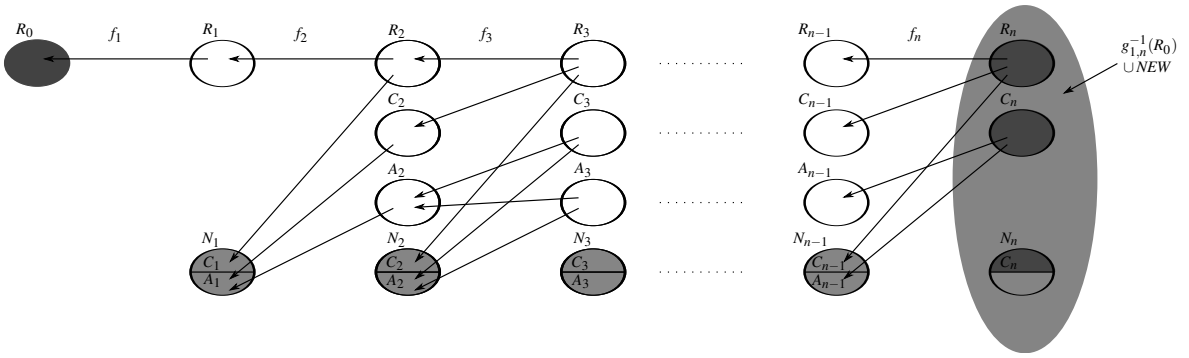


Figure 4: Constructing *NEW*

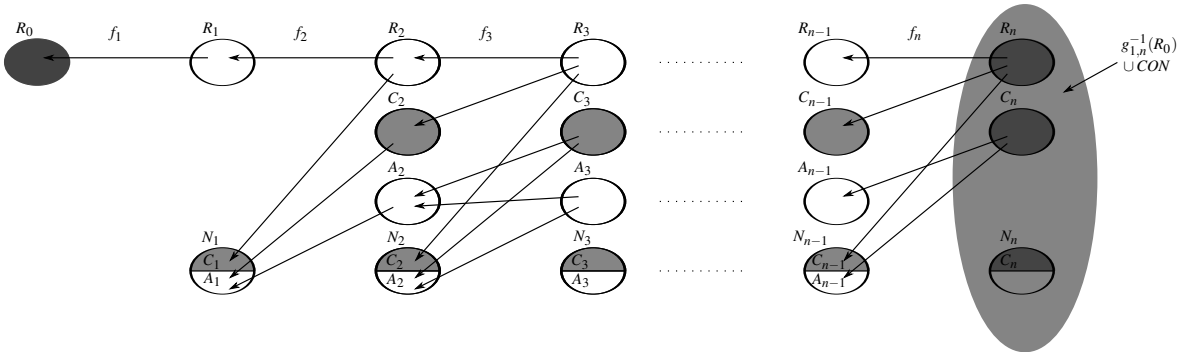


Figure 5: Constructing *CON*

we obtain

$$C_j \uplus A_j \uplus R_j \quad = \quad g_{1,j}^{-1}(C_0 \uplus A_0 \uplus R_0) \uplus g_{2,j}^{-1}(N_1) \uplus \ldots \uplus g_{j,j}^{-1}(N_{j-1}) \uplus N_j \tag{3}$$

Observe that this is a partition of $C_j \uplus A_j \uplus R_j$. Also, by repeated application of

$$R_j \quad = \quad f_j^{-1}(R_{j-1}) \uplus f_j^{-1}(C_{j-1})$$

we obtain

$$R_j \uplus C_j \quad = \quad g_{1,j}^{-1}(R_0) \uplus g_{1,j}^{-1}(C_0) \uplus g_{2,j}^{-1}(C_1) \uplus \ldots \uplus g_{j,j}^{-1}(C_{j-1}) \uplus C_j \tag{4}$$

Observe that this is a partition of $C_j \uplus R_j$.

In a full refinement chain $M_0 \preccurlyeq \ldots \preccurlyeq M_n$ we have that $A_0 = \{\}$,$C_0 = \{\}$, and $A_n = \{\}$. Define:

$$NEW \quad = \quad g_{2,n}^{-1}(N_1) \uplus \ldots \uplus g_{n,n}^{-1}(N_{j-1}) \uplus N_n$$

$$CON \quad = \quad g_{1,n}^{-1}(C_0) \uplus \ldots \uplus g_{n,n}^{-1}(C_{j-1}) \uplus C_n$$

These constructions are illustrated in Figures 4 and 5.

Then from Equation 3 above with $j = n$, and using $A_0 = C_0 = A_n = \{\}$ we obtain

$$C_n \uplus R_n \quad = \quad g_{1,n}^{-1}(R_0) \uplus NEW$$

From Equation 4 above with $j = n$ we obtain

$$C_n \uplus R_n \quad = \quad g_{1,n}^{-1}(R_0) \uplus CON$$

Hence $NEW = CON$. From Theorem 5.2 and Line (2) above respectively we obtain that

$$f_n^{-1}(\ldots(f_1^{-1}(M_0))\ldots) \;|||\; RUN_{NEW} \sqsubseteq_{TDI} M_n$$

$$\text{and} \qquad M_n \textbf{ sat } CA(CON\,, f_n^{-1}(\ldots f_1^{-1}(R_0)\ldots)\,)$$

Lemma 5.5(4) yields that $M_n \setminus CON$ is divergence-free, i.e., $M_n \setminus NEW$ is divergence-free. Hence by Lemma 4.1 we obtain that

$$f_n^{-1}(\ldots(f_1^{-1}(M_0))\ldots) \quad \sqsubseteq_{TDI} \quad M_n \setminus NEW \tag{5}$$

or, equivalently, that the following theorem holds true.

Theorem 5.7 *Let $M_0 \preccurlyeq M_1 \preccurlyeq \ldots \preccurlyeq M_n$ be a chain of refinement steps such that $A_0 = C_0 = \{\}$ and $A_n = \{\}$, refining events according to functions f_i, and let NEW be the set of events as calculated above. Then*

$$M_0 \sqsubseteq_{TDI} f_1(f_2(\ldots f_n(M_n \setminus NEW)\ldots))$$

Proof: This follows from the result in Line 5 above, using the CSP law $f(f^{-1}(P)) = P$. □

This result guarantees that Event-B refinement (a) does neither introduce "new traces on old events" nor (b) does it introduce divergences on new events. This gives us the precise account of Event-B refinement in terms of CSP which we were aiming at.

6 Conclusion

In this paper, we have given a CSP account of Event-B refinement. The approach builds on Butler's semantics for action systems [6]. Butler's refinement rules allow new convergent events to be introduced into action systems, so that refinement steps satisfy $M_i \sqsubseteq_{TDI} (M_{i+1} \setminus N_{i+1})$, and hiding new events does not introduce divergence. Abrial's approach to Event-B refinement generalises this approach, allowing new events to be *anticipated* as well as *convergent*, and also allowing splitting of events. Our approach to refinement using CSP semantics reflects this generalisation and thus extends Butler's, in order to encompass these different forms of event treatment in Event-B refinement. We do not yet handle merging events, and this is the subject of current research.

Recently, an Event-B‖CSP approach has been introduced [19]. It aims to combine Event-B machine descriptions with CSP [17] control processes, in order to support a more explicit view of control. In this, it follows previous works on integration of formal methods [7, 22, 15, 18, 12], which aim at complementing a state-based specification formalism with a process algebra.

The account of refinement presented here provides the basis for a flexible refinement framework in Event-B‖CSP, and this is presented in [21]. The semantics justifies the introduction of a new status of *devolved*, for refinement events which are anticipated in the Event-B machine but convergent in the CSP controller. This approach has been applied to an initial Event-B‖CSP case study of a Bounded Retransmission Protocol [20]. We aim to develop investigate further case studies. We are in particular

interested in finding out whether the work of showing divergence-freedom (and also deadlock-freedom) can be divided onto the Event-B and CSP part such that for some events convergence is guaranteed by showing the corresponding proof obligations in Event-B while for others we just look at divergence-freedom of the CSP process. The latter part could then be supported by model checking tools for CSP, like FDR [10].

References

[1] J-R. Abrial (2010): *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[2] J-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta & L. Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. *STTT* 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] J-R. Abrial, M. J. Butler, S. Hallerstede & L. Voisin (2008): *A Roadmap for the Rodin Toolset*. In E. Börger, M. J. Butler, J. P. Bowen & P. Boca, editors: *ABZ*, *Lecture Notes in Computer Science* 5238, Springer, p. 347, doi:10.1007/978-3-540-87603-8.

[4] E. A. Boiten & J. Derrick (2009): *Modelling Divergence in Relational Concurrent Refinement*. In Michael Leuschel & Heike Wehrheim, editors: *IFM*, *Lecture Notes in Computer Science* 5423, Springer, pp. 183–199, doi:10.1007/978-3-642-00255-7.

[5] C. Bolton & J. Davies (2002): *Refinement in Object-Z and CSP*. In M. Butler, L. Petre & K. Sere, editors: *IFM 2002: Integrated Formal Methods*, *LNCS* 2335, pp. 225–244.

[6] M. J. Butler (1992): *A CSP approach to Action Systems*. DPhil thesis, Oxford University.

[7] M. J. Butler (2000): *csp2B: A Practical Approach to Combining CSP and B*. In: *FACS*, pp. 182–196.

[8] J. Derrick & E. A. Boiten (2001): *Refinement in Z and Object-Z*. Springer-Verlag, doi:10.1007/978-1-4471-0257-1.

[9] J. Derrick & E.A. Boiten (2003): *Relational Concurrent Refinement*. *Formal Aspects of Computing* 15(2-3), pp. 182–214, doi:10.1007/s00165-003-0007-4.

[10] Formal Systems (Europe) Ltd.: *The FDR Model Checker*. `http://www.fsel.com/` (accessed 8/3/11).

[11] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall.

[12] A. Iliasov (2009): *On Event-B and Control Flow*. Technical Report CS-TR-1159, School of Computing Science, Newcastle University.

[13] C. Métayer, J.-R. Abrial & L. Voisin (2005): *Event-B Language*. RODIN Project Deliverable 3.2, `http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf`, accessed 25/5/10.

[14] C. Morgan (1990): *Of wp and CSP*. In: *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, Springer, pp. 319–326.

[15] E-R. Olderog & H. Wehrheim (2005): *Specification and (property) inheritance in CSP-OZ*. *Sci. Comput. Program.* 55(1-3), pp. 227–257, doi:10.1016/j.scico.2004.05.017.

[16] A.W. Roscoe (1998): *Theory and Practice of Concurrency*. Prentice-Hall.

[17] S. Schneider (1999): *Concurrent and Real-time Systems: The CSP approach*. Wiley.

[18] S. Schneider & H. Treharne (2005): *CSP theorems for communicating B machines*. *Formal Asp. Comput.* 17(4), pp. 390–422, doi:10.1007/s00165-005-0076-7.

[19] S. Schneider, H. Treharne & H. Wehrheim (2010): *A CSP Approach to Control in Event-B*. In Dominique Méry & Stephan Merz, editors: *IFM*, *Lecture Notes in Computer Science* 6396, Springer, pp. 260–274, doi:10.1007/978-3-642-16265-7.

[20] S. Schneider, H. Treharne & H. Wehrheim (2011): *Bounded Retransmission in Event-B||CSP: a Case Study*. Technical Report CS-11-04, University of Surrey.

[21] S. Schneider, H. Treharne & H. Wehrheim (2011): *Stepwise refinement in Event-B||CSP*. Technical Report CS-11-03, University of Surrey.

[22] J. Woodcock & A. Cavalcanti (2002): *The Semantics of Circus*. In D. Bert, J. P. Bowen, M. C. Henson & K. Robinson, editors: *ZB*, *Lecture Notes in Computer Science* 2272, Springer, pp. 184–203. Available at `http://link.springer.de/link/service/series/0558/bibs/2272/22720184.htm`.

Perspicuity and Granularity in Refinement

Eerke Boiten
School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E.A.Boiten@kent.ac.uk

This paper reconsiders refinements which introduce actions on the concrete level which were not present at the abstract level. It draws a distinction between concrete actions which are "perspicuous" at the abstract level, and changes of granularity of actions between different levels of abstraction.

The main contribution of this paper is in exploring the relation between these different methods of "action refinement", and the basic refinement relation that is used. In particular, it shows how the "refining skip" method is incompatible with failures-based refinement relations, and consequently some decisions in designing Event-B refinement are entangled.

Keywords: Refinement, action refinement, stuttering steps, ASM, Event-B, Z, internal operations, weak refinement, granularity, perspicuity, divergence.

1 Introduction

This paper discusses how different ways of introducing "extra" actions in refinement (such as weak refinement, action refinement, stuttering steps) relate to the underlying refinement relations used (e.g. trace refinement, failures refinement). In particular, we aim to show how the choices in those two dimensions are interdependent. The paper is not intended to be polemic ("my formalism/refinement relation is better than yours") nor is it really meant to be a first introduction to the topic. Where it appears to state the obvious, this is in an attempt to ensure that commonalities, differences, and design decisions in refinement relations are exhibited in an unambiguous and uncontroversial way.

Before describing the issues in detail, we consider an example. The example is presented in Z, but the notation used is not essential to what follows in this paper. In general, most of what is described in this paper could be expressed in ASM [18], (Event-)B [1], Z [19], binary relations [11], UTP [15] or many other state-based formalisms; for the moment we make no assumptions about what refinement relation is "in force".

This example is due to Carroll Morgan, who presented it during an enlightening conversation at the 2009 Dagstuhl seminar "Refinement Based Methods for the Construction of Dependable Systems". The abstract specification is essentially a priority queue, stored as a bag, so taking out an element involves selecting the minimum of the bag. Obvious specifications of functions *min* on bags and (later) *sorted* on sequences are omitted. The schema *AS* describes system states, *AInit* initial states, and the schemas *Ain* and *Aout* the operations of adding and removing an element. The precondition $b \neq [\![]\!]$ is included explicitly in *Aout*, in recognition of it having to be an explicit guard in alternative notations such as

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 155–165, doi:10.4204/EPTCS.55.10

Event-B.

AS

$b : \mathrm{bag}\,\mathbb{N}$

AInit

AS'

$b' = [\![\,]\!]$

Ain

ΔAS

$x? : \mathbb{N}$

$b' = b \uplus [\![x?]\!]$

Aout

ΔAS

$x! : \mathbb{N}$

$b \neq [\![\,]\!]$

$b = b' \uplus [\![x!]\!]$

$x! = min(b)$

The concrete specification uses a sequence to represent the queue. Removing an element is only possible when the sequence is non-empty and sorted, in which case the element to be removed is at the head of the sequence. The schema *Sort* describes the sorting of the sequence. The schema *Cycle* is mostly a red herring[1] and not part of Morgan's original example.

CS

$s : \mathrm{seq}\,\mathbb{N}$

CInit

CS'

$s' = \langle\,\rangle$

Cin

ΔCS

$x? : \mathbb{N}$

$s' = s \frown \langle x? \rangle$

Cout

ΔCS

$x! : \mathbb{N}$

$s \neq \langle\,\rangle$

$sorted(s)$

$s = \langle x! \rangle \frown s'$

Sort

ΔCS

$items\, s = items\, s'$

$sorted(s')$

Cycle

ΔCS

$s = \langle\,\rangle \land s' = \langle\,\rangle \lor$

$s' = (tail\, s) \frown \langle head\, s \rangle$

This paper discusses the many ways in which one may consider the concrete specification to refine the abstract one, possibly after a slight modification, or possibly not at all, depending on the notions of refinement and action refinement employed. Before we move on to that level of complication, consider the composed schema $SortOut == Sort \fatsemi Cout$, whose meaning is given by

SortOut

ΔCS

$x! : \mathbb{N}$

$s \neq \langle\,\rangle$

$\exists s'' : \mathrm{seq}\,\mathbb{N} \bullet items\, s = items\, s'' \land sorted(s'') \land s'' = \langle x! \rangle \frown s'$

[1] One might use it to represent the non-determinism in a distributed implementation where individual clients have no control over the access pointer in a cyclical list, ... maybe.

Then, uncontroversially, in most sensible refinement relations, the operation *Aout* is refined by *SortOut* (or more precisely: the data type $(AS, AInit, Ain, Aout)$ is refined by $(CS, CInit, Cin, SortOut)$) under the retrieve relation $b = items\, s$. In fact, this is normally an equivalence: refinement also holds in the reverse direction[2].

The rest of this paper is structured as follows. In Section 2 we describe different basic refinement notions. Then in Section 3 we discuss the various methods in which "extra" operations may appear in refinement steps. In Section 4 we compare how these methods can be used to model the decomposition of actions into smaller grained ones, and how this impacts on the various basic refinement notions. Finally, Section 5 presents some conclusions.

2 Basic Notions of Refinement

We have given detailed fully formal descriptions and comparisons of the different basic notions of refinement for state-based and concurrent systems in many previous papers, e.g. [6, 11, 5]. Rather than repeating this and thereby fixing a formalism or even introducing a new one, we remain informal here, using various formalisms and their refinement notions as illustrations.

In basic data refinement, systems (or machines or abstract data types) are compared which have identical alphabets (or sets of labels of operations (or actions or events)). Apart from conditions on initial and possibly final states, and other details which depend on what observations can be made of these systems, operations are compared in pairs of an abstract and a concrete operation, with refinement conditions being some subset of the following properties:

(1) Consistency The effect of the concrete operation is one that is allowed by the abstract operation.

(2) Enabledness When operations can be invoked in the abstract state, they can be invoked in the concrete state as well.

(3) Restricted consistency In states where the abstract operation is enabled, the effect of the concrete operation is one that is allowed by the abstract operation.

Property (1) or its weaker variant (3) represents the essence of refinement: that a client would be unable to observe conclusively that they are using the concrete rather than the abstract system. Property (2) ensures that the client is indeed able to perform the same "experiments" on both systems. Property (1) obviously implies (3), and also a converse of (2): where concrete operations are enabled (leading to an "effect"), their abstract counterparts should be enabled, too (in order to allow comparison of effects). The properties leave out detail about what an effect is, are purposefully vague on "can be invoked" in (2) to allow a variety of interpretations, and leave any linking between abstract and concrete states implicit. They are also somewhat biased towards downward simulation. A few examples should make all this clearer. The refinement relations described below will be refered to in later sections.

Traditional (downward simulation) *Z refinement* [19, 11] is characterised by properties (2) and (3), with "can be invoked" in a state computed as individual operations' preconditions, i.e. whether their defining predicates can be satisfied for some after-state. Condition (2) is called "applicability" and typically formulated as

$$\mathrm{pre}\, AOp \wedge R \Rightarrow \mathrm{pre}\, COp$$

[2] A refinement linking *Ain* to $Cin \fatsemi Sort$ instead is equally possible but would require strengthening the concrete state invariant to sorted sequences; $Cin \fatsemi Sort$ then simplifies to the insert operation of insertion sort.

where $\mathrm{pre}\, AOp == \exists AS' \bullet AOp$ denotes the computed precondition. Condition (3) is called "correctness", and typically formulated as

$$\mathrm{pre}\, AOp \wedge R \wedge COp \Rightarrow \exists AS' \bullet R' \wedge AOp$$

We have sometimes called this refinement relation the "contract" model of refinement as it constrains the implementation only within the original precondition.

Trace refinement is characterised by (1) only, only requiring that anything that *does* happen in the concrete specification is consistent with the abstract one. As such, it represents preservation of safety properties only, "nothing bad happens". No concrete operations being enabled at all, for example, is an acceptable trace refinement.

Basic *Event-B refinement* (called simple refinement in [1, Ch. 14]) is characterised by (1), with (optionally) a weak alternative to (2): if the concrete state deadlocks (i.e. no events are enabled), then so should the abstract state. Enabledness of events is given by explicitly specified guards, with a "feasibility" proof obligation ensuring that they are at least as strong as any computed precondition. Abrial [1, p. 429] states that condition (2) could be imposed, but "this happens to be sometimes too strong". (We will return to this.)

Failures-based variants of refinement are characterised by (1) and (2), where (2) considers individual operations for "blocking Z refinement" and singleton failures refinement, or sets of concurrently enabled operations for failures refinement as in CSP. We refer to [6, 17, 5] for detailed discussion of these refinement relations and the finer distinctions between them, which are not relevant in the current paper.

Note that a refinement relation characterised by property (3) without property (2) is nonsensical as it is not transitive: preconditions or guards can be strengthened (lack of (2)) and then weakened (by (3)), but the composition of such steps does not respect (3).

3 Adding Operations in Refinement

The basic refinement rules described above deal only with the situation where the abstract and concrete specifications have the same alphabet of operations. There are many ways in which one could allow a refined specification to have "extra" operations – we discuss a number of them. First, we mention alphabet extension and alphabet translation [11, Ch. 14] for completeness. Then, we get to the core of this paper: stuttering steps, the introduction of internal operations, and action refinement, and how these sometimes get conflated.

3.1 Alphabet Extension and Translation

The simplest way of allowing new operations in refinement is *alphabet extension*: to just accept them without any further constraints. If we make the intuitive step of identifying a non-existent operation with one that is never enabled, alphabet extension should be perfectly acceptable in traditional Z refinement: it means we allow implementors to provide functionality that we had not asked for. In a process algebra context alphabet extension is typically not allowed, and indeed that would make sense in our intuitive view: it would go against refinement property (1), by having no matching abstract behaviour for some concrete behaviour.

In *alphabet translation*, a single abstract operation is implemented by multiple concrete ones, which requires an explicit mapping, recording for every concrete operation which abstract operation it represents, and thus which operation's behaviour it needs to correspond with. (If this mapping is not required

to be total, alphabet extension is subsumed.) A typical example for this would be an abstract two-dimensional grid specification with a "move" operation, which is refined into "moveNorth", "moveEast", etc. Alphabet translation is allowed in Event-B, where it is called "splitting" an abstract event.

The semantic property established in alphabet translation is: every concrete trace (with its corresponding observations) is consistent with an abstract trace that relates to it by the given mapping (applied elementwise) with its corresponding observations.

3.2 Perspicuous Operations

State-based systems potentially change state when operations are executed. When no operation is invoked, the state does not normally change. Some formalisms take this into account by including explicitly so-called stuttering steps in their semantics: steps where the state does not change between two observations, due to no event having taken place. In the light of that, it is intuitively obvious to accept the introduction of additional concrete events as refinements of the identity operation (a.k.a. *skip*) on the abstract state. We will call these *perspicuous* concrete events, to be distinguished from "internal events" (see below) which incur additional assumptions and requirements. In particular, in subsequent refinement steps, perspicuous operations do *not* have a different status from operations that were present earlier.

Abrial [1] presents a similar motivation for the introducion of new events in Event-B, analogous to how this is done in action systems [3], and refers to it as "observing our discrete system in the refinement with a finer grain than in the abstraction". Event-B is explicit about the introduction of such events as being refinements of *modelling*: introducing not just aspects of a solution, but more detail of the model. Indeed, where refinement is viewed as only moving from a complete description of a problem to its solution, the introduction of perspicuous operations which achieve nothing in the abstract world can hardly be useful by itself[3]. Both action systems and Event-B include a relative deadlock freedom condition with this kind of refinement: the new system should deadlock (i.e., terminate, in the action systems view) no more often than the old one. The semantic relation established by this kind of generalised refinement is: for every concrete trace with its corresponding observations, an abstract trace constructed by crossing out all perspicuous actions is consistent with it.

In the running example, under most refinement relations and with the obvious retrieve relation $items\, s = b$ both concrete operations *Sort* and *Cycle* are candidate perspicuous operations, as they satisfy $items\, s = items\, s'$ and thus relate identical abstract states. They are both applicable in every concrete state and thus are refinements of an abstract *skip* even when property (2) is imposed.

For perspicuous operations, the notion of *divergence* comes into the picture. A collection of perspicuous operations is divergent if infinitely often in succession, from some state, one of its members can be invoked. In a trace-based view, where perspicuous operations could be inserted at arbitrary points between "normal" operations, non-divergence is necessary to ensure that a finite trace cannot get extended into an infinite one by that process. This is how Abrial [1] explains it[4]. With additional assumptions, such as that a system might perform perspicuous operations independently, divergence becomes a practical as well as a theoretical problem. Butler [9] explains the non-divergence requirement in Event-B by saying "The new events introduced in a refinement step can be viewed as hidden events not visible to the environment of a system and are thus outside the control of the environment" which would suggest

[3] This is *not* intended to be a controversial statement or implicit criticism on Event-B: the crux is in the phrase *by itself*, and this should become clearer later when we compare the different ways of encoding action refinement.

[4] His use of the term "reachable" is a bit unfortunate, though – this tends to be an existential property (some path is finite) rather than the required universal (all paths are finite) property required.

these are not just perspicuous events, but even *internal* events as we will discuss next. In action systems [3], which are viewed as a main inspiration for Event-B, all actions could be considered to be internal (even if the variables they modify are not), which conforms more with Abrial's explanation than with Butler's[5]. A typical method of proving non-divergence is by establishing a variant (well-founded, strictly decreasing function) on newly introduced (collections of) perspicuous operations [8, 12, 1]. If refinement is based on property (1) rather than property (3), i.e., an action cannot gain behaviour in refinement, then non-divergence is preserved by subsequent refinement steps.

In the example, both perspicuous operations are divergent. This is obvious from the fact that they are enabled in *every* concrete state. *Sort* allows an infinite sequence of invocations of which only the first does not necessarily correspond to a concrete *skip*. For formalisms that use infinite traces and allow stuttering steps, such as TLA, this may not be a problem. Removing divergence on each of the operations can be done using several possible small modifications. The divergence problem for *Sort* could be fixed by including a guard $\neg sorted(s)$, but this makes it a refinement of *skip* only if property (2) is not imposed and guards can be strengthened. Another way would be to add a flag that ensures *Sort* is invoked exactly once after every occurrence of *Cin* or *Cycle* (possibly also preventing the next *Cin* until after sorting). A counter could be used to remove divergence in *Cycle*, with each of the other operations (excluding *Sort*) setting the counter to fix the maximal number of occurrences of *Cycle* to follow it, and *Cycle* decrementing it at every step until it is 0. None of those modifications would retain the property that *Sort* or *Cycle* refines *skip* if the prevalent refinement relation respects (2).

3.3 Internal Operations

An internal operation is a perspicuous operation with a special status: it is assumed to be invisible to the environment, and under internal control of the system only. In process algebras, internal operations naturally occur in a number of ways. In CSP [14] they arise from channels being hidden, for example encapsulating an internal communication channel when considering a system of communicating subsystems. They may also be used, for example in LOTOS [7], to encode internal choice when only external choice is available as a basic operator. Butler first considered the introduction of internal events in B refinement [8], and based on this approach we introduced "weak refinement" for Z [12, 10], which was analysed and compared to ASM refinement in detail by Schellhorn [18].

The requirements imposed in this context are inspired by how process algebras deal with internal actions, for example in defining "weak" bisimulation: where standard refinement conditions refer to a single action, their "weak" equivalents consider the same action possibly prefixed and postfixed by occurrences of internal actions. Thus, the refinement consistency property, e.g., will state that for every concrete action, with internal concrete behaviour before and after, its effect is consistent with the abstract action, possibly also pre- and postfixed with (abstract) internal behaviour. E.g. in [12] the restricted consistency (correctness) condition for weak refinement in Z (downward simulation) is phrased as

$$\mathrm{pre}(Int_A \fatsemi AOp) \wedge R \wedge (Int_C \fatsemi COp \fatsemi Int_C) \Rightarrow \exists AS' \bullet R' \wedge (Int_A \fatsemi AOp \fatsemi Int_A)$$

where Int_C is arbitrary internal behaviour in the concrete state, i.e. the transitive reflexive closure of the union of internal operations, and similar for Int_A. Taking this process algebra inspired approach has a few consequences:

[5] Note however that Abrial [1] does recognise (on page 414) a different class of operation that "*is not part of the protocol: it corresponds to a "daemon" acting . . .*".

- internal actions have a special status which goes beyond the refinement step where they are introduced. They can not only be introduced this way, but must also be taken into consideration or can even be removed in subsequent refinement steps.
- there is an assumption that if internal actions are necessary for progress, they will "eventually" happen, so external operations are viewed as "enabled" if their before-state is reachable through internal behaviour; in timed process algebras in particular, internal actions are often taken as "urgent" meaning they happen as soon as they are enabled.
- there need not be independent refinement conditions for internal operations: all internal behaviour is viewed in the context of its composition with external behaviour. Thus, internal operations need *not* be refinements of *skip*. Of course, all internal operations being perspicuous, with external operations corresponding as normal, is *one* way of satisfying the refinement conditions like the one above, but it is not the only way. In fact, in some refinement relations, it may not be a viable way, see below.

The approaches for B and Z mentioned above only included *prevention* of divergence in weak refinement steps. A more general approach, also consistent with the process algebraic view, is to *preserve* or *reduce* any divergence that was already present in the abstract specification. This is worked out in detail in [6], and the impact of differing notions of "livelock" or divergence is discussed in [4]. The semantic relation established in this case is roughly that for every concrete trace, an abstract trace exists that is consistent with it, with both traces' subsequences of *external* actions being identical[6].

3.4 Action Refinement

Alphabet translation described above allows for arbitrary matchings of an occurence of an abstract action with the occurrence of a single concrete action. The most explicit way of changing the granularity of actions is to allow for matchings between *sequences* of abstract and concrete actions. This has been called "action refinement" [2] or "non-atomic refinement" [10]. In its most[7] general form, action refinement corresponds to ASM 1-to-*n* diagrams with *n* possibly greater than 1 [18], generalising the normal commuting simulation diagram to one where the concrete effect is achieved in *n* steps, without requiring a relation between abstract and *intermediate* concrete states. In this view, all concrete operations resulting from the decomposition are of the same status, with only their order having an impact on refinement conditions. This is also the view we took in definining non-atomic refinement for Z [10], work which was continued by Derrick and Wehrheim [13]. This kind of action refinement is even possible without changing the state space involved. It requires an explicit matching between abstract actions and concrete action sequences, which also extends to traces. The semantic relation aimed for is that concrete traces are consistent with abstract traces under this extended matching relation. The concrete and the abstract models end up having different interfaces with this approach – this may be exactly what is required, though. For example, [11, Ch. 13] has an example of a watch which in the abstract model has a *ResetTime* operation, which in the concrete model is represented by a series of executions of *ButtonA* and *ButtonB* operations.

Considering for simplicity now only the case that $n = 2$, the refinement requirements are like the introduction of sequential composition in refinement calculus [16]. Splitting an operation in two means

[6] In fact it is a somewhat more subtle matching: non-determinism included in a single operation on one abstraction level may be represented through a different choice of sequence of internal actions on the other level, so it is really a relation between sets of abstract vs. concrete traces with the same external subsequence.

[7] Avoiding for now the generalisation to *m*-to-*n* diagrams with $m \neq 1$.

finding an intermediate state (predicate) such that the first "half" lands in the intermediate state, and the second "half" moves from the intermediate to the original after-state. The problematic issue is what is or is not allowed to happen in the intermediate state. In a concurrent context, this comes under the heading of "interference" – when the first "half" of an operation has been executed, should other operations be disabled (non-interference, as e.g. discussed for action systems in [3]), or should their execution cancel out the effect of this one? This is a well-known problematic area, discussed also in [10], which we will not focus on here, as it is orthogonal to the issues discussed: when an action is split with part of it being perspicuous or internal, that also creates an intermediate state with the same potential interference problems.

4 How to Reduce Granularity in Refinement

From the discussion above, it should be clear that there are at least three semantic models for reducing the granularity of actions in refinement:

- by introducing perspicuous actions that take on some of the "work" – possibly requiring non-divergence;
- by introducing internal actions to the same effect – either using the limited refinement rules for perspicuous actions, or by using the more general "weak refinement" rules;
- by giving explicit decompositions of actions in which all parts have the same status.

We limit ourselves for now to the case where we are decomposing an action into two actions, where the first part could be viewed as "prepatory work", and the second part as the "real work" – in other words, the situation in our example of refining *Aout* into *Sort* and *Cout*, where we expect *Sort* to be executed before *Aout*. However, in order to concentrate on the general situation, let us consider refining *AWork* into *Prepare* and *CWork*.

For the methods of reducing granularity by refining *skip*, we aim for *Prepare* to be perspicuous, and for *CWork* to be a refinement of *AWork*. Now consider an abstract state in which the operation *AWork* was applicable. If in every corresponding concrete state it would be possible to apply *CWork*, then we have a degenerate situation: we are introducing a new action *Prepare* whose contribution is unnecessary in all situations (i.e., it might as well be a *concrete skip*, too). Thus, in any relevant case of reducing granularity, *CWork* can be applicable in only a subset of the corresponding concrete states – namely those where *Prepare* has nothing (left) to do. Indeed, because *Prepare* is a refinement of an abstract *skip*, if its before-state is linked to a particular abstract state, then so should its after-state. Again in order to ensure that *Prepare* does something useful in some circumstances, there should be some abstract states linked to the before-states of *Prepare*.

This is where the prevalent notion of refinement makes a difference. If condition (2) ("enabledness") is in force, we have made it impossible for *CWork* to be a refinement of *AWork*, because *CWork* is only applicable in a strict subset of the corresponding concrete states. This holds a fortiori for stronger versions of condition (2) such as failures refinement.

Thus, condition (2) excludes reduction of granularity by introducing perspicuous actions. It also excludes reduction of granularity by introducing internal actions using the "perspicuous actions" conditions. However, the more general "weak refinement" rules can be used in combination with condition (2), as we have shown in [6] in a context with condition (1) in force, and in [10] with condition (3) in force. This is explained by not being constrained to considering the concrete operation in isolation, but rather only considering it in the context of possible internal concrete behaviour.

The other way in which condition (2) is problematic for the refinements of *skip* is any requirements for perspicuous actions to be non-divergent. If they are refinements of *skip* respecting condition (2), then they are by definition applicable in all states and thus always applicable "again" and by definition divergent.

Returning to the example, ignoring *Cycle* for now, refinement reducing granularity is possible in several ways:

- by having *Sort* perspicuous, and guarded by $\neg sorted(s)$ if it is also required to be non-divergent. This works for trace refinement (just (1)), Event-B refinement, but not the other forms.
- by having *Sort* internal, provided it is guarded by $\neg sorted(s)$. This works according to the rules for Event-B, establishing normal Event-B refinement. However, it can also work for stronger refinement relations respecting condition (2), but then the more general weak refinement rules need to be used to establish it. In particular, it would mean that *Aout* is compared for refinement with $Sort^* \fatsemi Cout$.
- for explicit action refinement of *Aout* by *Sort* followed by *Cout*, there is no requirement for *Sort* to be guarded (compare the watch example referred to above: as conceptionally the user presses *ButtonB*, there is no guard preventing the user from doing that infinitely often), and refinement can be any kind, including relations respecting property (2) or even (3). In fact, including a guard on *Sort* would disallow the combined concrete output operation on states which are already sorted, and thus be unacceptable if the refinement relation obeys property (2).

5 Conclusion

The paradox that led to the discussion with Carroll Morgan referred to earlier was the following. If the work of one abstract operation is split between two concrete ones, and one of the concrete operations makes no progress that can be detected abstractly[8], why do we need this action at all? And if we do need it, how can the other concrete operation, achieving some but not all of the work of its abstract counterpart, be a refinement of the abstract one? The answer is hopefully somewhat clarified above. It requires a notion of refinement that allows for guards to be strengthened. The underlying issue may well have been known in "folklore" but it is not presented in any published papers we are aware of.

Coming back to Event-B specifically, two of its design decisions are thus closely entangled:

- to have essentially a trace semantics with only global deadlock prevention;
- to use stuttering step refinements for reducing granularity.

Both lead to relatively simple refinement obligations, which is attractive. In order for Event-B to strengthen refinement to preserve stronger properties such as encoded in various refusal-based semantics, it would also have to give up its simple notion of reduction of granularity. It could do this in at least two ways: either by going the way of ASM and having explicit recipes for decomposing operations with their corresponding conditions, or by going the way of process algebra, and giving certain operations explicit "internal" status which they then would need to retain subsequently. In either case, the price of gaining semantic strength is a considerable amount of complication of refinement conditions, which may be too big a price to pay, particularly for a formalism which now has so much (automated) proof tool support available. Would that be what Abrial had in mind when he wrote that (condition (2)) "happens to be sometimes too strong"?

[8] Thus, some degree of data refinement is implied: a refinement of *skip* on the *same* state really cannot make any progress.

Postscript

Finally, returning to the running example once more, a last word on the *Cycle* operation. It makes no useful progress whatsoever, but the constraints put upon this completely irrelevant operation in refinement in any "stuttering steps" approach (namely: taming its divergence), have been no more and no less than on the supposedly enormously useful *Sort* operation. Surely that is somewhat disappointing.

Acknowledgements

To Carroll Morgan for his explanations, to Michael Butler, John Derrick, Steve Dunne and Gerhard Schellhorn for useful discussions, and to the reviewers for their comments.

References

[1] J.-R. Abrial (2010): *Modelling in Event-B*. CUP.

[2] L. Aceto (1992): *Action Refinement in Process Algebras*. CUP.

[3] R.J.R. Back (1993): *Refinement of Parallel and Reactive Programs*. In M. Broy, editor: *Program Design Calculi*, pp. 73–92.

[4] E.A. Boiten & J. Derrick (2009): *Modelling divergence in Relational Concurrent Refinement*. In M. Leuschel & H. Wehrheim, editors: *IFM 2009: Integrated Formal Methods*, *LNCS* 5423, Springer Verlag, pp. 183–199, doi:10.1007/978-3-642-00255-7-13.

[5] E.A. Boiten & J. Derrick (2010): *Incompleteness of Relational Simulations in the Blocking Paradigm*. *Science of Computer Programming* 75(12), pp. 1262–1269, doi:10.1016/j.scico.2010.07.003.

[6] E.A. Boiten, J. Derrick & G. Schellhorn (2009): *Relational Concurrent Refinement Part II: Internal Operations and Outputs*. *Formal Aspects of Computing* 21(1-2), pp. 65–102, doi:10.1007/s00165-007-0066-z.

[7] T. Bolognesi & E. Brinksma (1988): *Introduction to the ISO Specification Language LOTOS*. *Computer Networks and ISDN Systems* 14(1), pp. 25–59, doi:10.1016/0169-7552(87)90085-7.

[8] M. Butler (1997): *An approach to the design of distributed systems with B AMN*. In J.P. Bowen, M G. Hinchey & D. Till, editors: *ZUM'97: The Z Formal Specification Notation, Lecture Notes in Computer Science* 1212, Springer-Verlag, pp. 223–241, doi:10.1007/BFb0027291.

[9] M. Butler (2009): *Decomposition Structures for Event-B*. In M. Leuschel & H. Wehrheim, editors: *IFM, Lecture Notes in Computer Science* 5423, Springer, pp. 20–38, doi:10.1007/978-3-642-00255-7-2.

[10] J. Derrick & E.A. Boiten (1999): *Non-atomic refinement in Z*. In J.M. Wing, J.C P. Woodcock & J. Davies, editors: *FM'99, Lecture Notes in Computer Science* 1708, Springer-Verlag, Berlin, pp. 1477–1496, doi:10.1007/3-540-48118-4_28.

[11] J. Derrick & E.A. Boiten (2001): *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT, Springer Verlag, doi:10.1007/978-1-4471-0257-1.

[12] J. Derrick, E.A. Boiten, H. Bowman & M.W.A. Steen (1998): *Specifying and Refining Internal Operations in Z*. *Formal Aspects of Computing* 10, pp. 125–159, doi:10.1007/s001650050007.

[13] J. Derrick & H. Wehrheim (2003): *Using coupled simulations in non-atomic refinement*. In D. Bert, J. Bowen, S. King & M. Walden, editors: *ZB 2003: Formal Specification and Development in Z and B, Lecture Notes in Computer Science* 2651, Springer, pp. 127–147, doi:10.1007/3-540-44880-2-10.

[14] C.A R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall.

[15] C.A.R. Hoare & He Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall.

[16] C.C. Morgan (1994): *Programming from Specifications*, 2nd edition. International Series in Computer Science, Prentice Hall.

[17] S. Reeves & D. Streader (2008): *Data refinement and singleton failures refinement are not equivalent*. *Formal Aspects of Computing* 20(3), pp. 295–301, doi:10.1007/s00165-008-0076-5.

[18] G. Schellhorn (2005): *ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison*. *Theoretical Computer Science* 336(2-3), pp. 403–436, doi:10.1016/j.tcs.2004.11.013.

[19] J.C.P. Woodcock & J. Davies (1996): *Using Z: Specification, Refinement, and Proof*. Prentice Hall.

Concurrent Scheduling of Event-B Models *

Pontus Boström, Fredrik Degerlund, Kaisa Sere and Marina Waldén
Åbo Akademi University, Dept. of Information Technologies
Turku Centre for Computer Science (TUCS)
Joukahainengatan 3-5, FIN-20520 Åbo, Finland
{pontus.bostrom, fredrik.degerlund, kaisa.sere, marina.walden}@abo.fi

Event-B is a refinement-based formal method that has been shown to be useful in developing concurrent and distributed programs. Large models can be decomposed into sub-models that can be refined semi-independently and executed in parallel. In this paper, we show how to introduce explicit control flow for the concurrent sub-models in the form of event schedules. We explore how schedules can be designed so that their application results in a correctness-preserving refinement step. For practical application, two patterns for schedule introduction are provided, together with their associated proof obligations. We demonstrate our method by applying it on the dining philosophers problem.

1 Introduction

Event-B [1, 18] is a state-based modelling framework with its roots in the guarded command language and the Action Systems formalism [3, 4]. It advocates proof-based correct-by-construction design, abstraction, stepwise refinement and model decomposition as its main development strategies.

In an Event-B model, events are chosen non-deterministically for execution following the interleaving principle and assuming atomicity of events. Much of the effort in the refinement approach, especially down in the refinement chain, is about the modeller aiming at diminishing the non-determinism in the model and introducing more deterministic ways of choosing events for execution. In an extreme case we can think of the modeller encoding this by using explicit program counters in the events. Work on introducing more deterministic *schedules* of events to Event-B has been studied extensively recently [8, 11, 14, 20]. The goal has been to avoid explicitly coding this scheduling information into the events. We base our approach on [8], which concerns sequential systems, and extend it to concurrent programs.

When models become large, decomposition strategies are used to focus on specific parts of the model. To be practical, such strategies need to support compositional verification in the sense that the modeller can locally reason about properties of a decomposed part of the model even though the underlying Event-B assumption is that events are chosen for execution from the entire set of events in the model. Relying on the atomicity requirement for events and the interleaving semantics for Event-B models the distinct parts can be interpreted as concurrently executing models [12]. We show here how the scheduling approach of Boström [8] can be extended so that we can apply it in a compositional manner focusing only on part(s), or sub-model(s), of the model. We turn these sub-models into tasks, giving each of them a schedule of its own. The main addition to the original approach for sequential programs is to handle the possible interferences the concurrently executing tasks might exhibit. This can also be seen as an extension, with explicit schedules, of the Hoang-Abrial approach [12] to development of concurrent programs.

To facilitate practical use of our method, the schedules are introduced stepwise into a model via patterns. The patterns have associated proof obligations needed for ensuring the correctness of the refine-

*This research was supported by the EU funded FP7 project DEPLOY (214158). http://www.deploy-project.eu

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 166–182, doi:10.4204/EPTCS.55.11

ment step. As a result of the schedules, the scheduling information contained in events can be expressed explicitly in the schedules.

In this paper, we focus on developing concurrent programs following the stepwise refinement approach. Apart from the introduction of explicit schedules, concurrent programs are modelled within Event-B in a normal manner [1, 12]. While Event-B models can be executed as such using a non-deterministic scheduler ("animation"), our approach is designed to be close to traditional programming languages and results in models that are more efficient to execute on a computer, since more control flow information is explicitly stated in the schedule than using only Event-B [8]. The approach can also be used to replace parts of event behaviour with scheduling information as the scheduling concept as such is more general than what the focus is here. The schedules actually give a process-oriented specification style for Event-B modeller complementing its state-based style [9, 17].

The rest of this paper is structured as follows. In section 2, we present the foundations needed to understand our approach. We discuss set transformers (predicate transformers), the Event-B formalism and model decomposition. In section 3, we introduce a dining philosophers [13] Event-B model, which serves as a running example. Section 4 presents our main contributions. We introduce a scheduling language, show how schedules and tasks can be introduced, and demonstrate how it is possible to tackle the problem of interference from interleaving tasks. In section 5, we show how our framework can be applied on the dining philosophers example model. Finally, we sum the paper up in section 6, where we also discuss related work and future perspectives.

2 Foundations

2.1 Event-B

Event-B [1, 10] is a state-based modelling language. Models in Event-B consist of a dynamic and a static part, referred to as *machines* and *contexts*, respectively. The most important parts of a machine are *variables*, an *invariant* and *events*. Contexts contain parts such as *constants*, which can be referred to from machines. The state space is made up of the variables v_1, ..., v_n of types Σ_1, ..., Σ_n, and can be modelled as the cartesian product $\Sigma = \Sigma_1 \times ... \times \Sigma_n$. The events E_1, ..., E_m modify the state space, and can be written in the following general form [10], where $k \in 1..m$:

$$E_k \mathrel{\hat{=}} \textbf{when } G_k(v,c) \textbf{ then } v : |A_k(v,v',c) \textbf{ end}. \tag{1}$$

Here, v represents the variables, c the constants seen by the machine, and the *action* $v : |A_k(v,v',c)$ is the nondeterministic assignment assigning v any such values v' for which $A_k(v,v',c)$ holds. $G_k(v,c)$ represents the *guard*, which is a condition that must hold in order for the action to take place. An event is said to be *enabled* when its guard holds. Each machine also contains a special event *Initialisation* $\mathrel{\hat{=}} v : |A_0(v',c)$ that initialises the state space. Unlike other events, it is unguarded and does not depend on a previous state. Events can be classified as *ordinary*, *convergent* or *anticipated*. This will be further explained in section 2.4. The invariant $I(v,c)$ is a predicate constraining the values of the variables.

2.2 Set transformers

The events in Event-B can be viewed as set transformers [10]. Our presentation of events as set transformers is similar to the presentation in [10].

Consider a state space Σ. A set transformer is a function $\mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma)$ that tranforms a set of states into another set of states. A weakest precondition set transformer S applied to a set q returns the largest set p from which S is guaranteed to reach a state in q.

We have the following definitions to give a set transformer semantics to Event-B models:

$$\begin{array}{lcl} \Sigma & = & \{v|\top\} \\ i & = & \{v|I(v,c)\} \\ g_k & = & \{v|G_k(v,c)\} \\ a_k & = & \{v \mapsto v'|A_k(v,v',c)\} \\ a_0 & = & \{v'|A_0(v',c)\} \end{array} \tag{2}$$

The set i describes the subset of the state space where the invariant I holds. Similarly, the sets g_k ($k \in 1..m$) represent the state space subsets where guard G_k of the respective event E_k is true. The relation a_k describes the possible before-after states that can be achieved by the assignment of the respective event. Note that the initialisation results in a set a_0 instead of a relation, since it does not depend on the previous values of the variables. In this paper, we do not consider properties of constants c separately, as it is not important at this level of reasoning. The axioms that describe the properties of the constants are here considered to be part of the invariant.

Let g and q be subsets of Σ, and a be a relation. Furthermore, S, S_1 and S_2 are arbitrary set transformers. The variables of Σ are denoted v. We have the following set transformers:

$$[a](q) \mathrel{\hat{=}} \{v|a[\{v\}] \subseteq q\} \quad \text{(Nondeterministic update)} \tag{3}$$

$$[g]\,(q) \mathrel{\hat{=}} \neg g \cup q \quad \text{(Assumption)} \tag{4}$$

$$\{g\}(q) \mathrel{\hat{=}} g \cap q \quad \text{(Assertion)} \tag{5}$$

$$(S_1 \sqcap S_2)(q) \mathrel{\hat{=}} S_1(q) \cap S_2(q) \quad \text{(Nondeterministic choice)} \tag{6}$$

$$S_1; S_2(q) \mathrel{\hat{=}} S_1(S_2(q)) \quad \text{(Sequential composition)} \tag{7}$$

$$S^{\omega}(q) \mathrel{\hat{=}} \mu X.(S; X \sqcap \mathsf{skip})(q) \quad \text{(Strong iteration)} \tag{8}$$

$$S^{*}(q) \mathrel{\hat{=}} \nu X.(S; X \sqcap \mathsf{skip})(q) \quad \text{(Weak iteration)} \tag{9}$$

$$\mathsf{skip}(q) \mathrel{\hat{=}} q \quad \text{(Stuttering)} \tag{10}$$

$$\mathsf{magic}(q) \mathrel{\hat{=}} \mathsf{true} \quad \text{(Miracle)} \tag{11}$$

$$\mathsf{abort}(q) \mathrel{\hat{=}} \mathsf{false} \quad \text{(Aborting)} \tag{12}$$

Here, true and false are notations representing the sets Σ and $\emptyset$, respectively. This is because of convenience as well as the fact that the same notation is used in weakest precondition predicate transformers. We will also in general use predicate notation for describing subsets of the state space. (Nondeterministic) update is used to assign values to variables in the state space, of which the stuttering set transformer skip is a special case, which leaves the state unmodified. The set transformer magic achieves the desired postcondition (even false) from any state, whereas abort does not guarantee to achieve any postcondition q from any state. Not even termination is guaranteed. Assumption and assertion both behave as skip when g is true, but when false, assumption behaves as magic, whereas assertion behaves as abort. Nondeterministic choice represents demonic choice between set transformers, and sequential composition combines set transformers in a sequential manner. An important property of demonic choice is that miraculous behaviour is avoided whenever possible, whereas aborting behaviour is always preferred. This is demonstrated by the following theorems, which follow directly from the definitions:

$$\begin{array}{l} \mathsf{magic} \sqcap S = S \\ \mathsf{abort} \sqcap S = \mathsf{abort} \end{array} \tag{13}$$

The following properties can easily be derived, and the proofs can also be found in [5]:

$$\begin{array}{ll} \mathsf{magic}; S = \mathsf{magic} & \mathsf{abort}; S = \mathsf{abort} \\ \{g\}; [h] = \{g\} & [g]; \{h\} = [g] \\ \{g \cap h\} = \{g\}; \{h\} & [g \cap h] = [g]; [h] \end{array} \tag{14}$$

The iteration set transformers are used to achieve repeated execution. Iteration has been thoroughly discussed by Back and von Wright [5, 6], and is only shortly summarised here. In both strong and weak iteration (S^ω and S^*, respectively), the set transformer S is repeatedly executed a demonically chosen number of times. In strong iteration, the number of executions may be infinite, whereas for weak iteration it is guaranteed to be finite. Important theorems regarding iteration include the following *unfolding* rules:

$$\begin{array}{l} S^\omega = S; S^\omega \sqcap \mathsf{skip} \\ S^* = S; S^* \sqcap \mathsf{skip} \end{array} \tag{15}$$

The set of states in which a set transformer S does not behave miraculously is called the guard of S. The guard $\mathsf{g}(S)$ is given as:

$$\mathsf{g}(S) \mathrel{\hat{=}} \neg S(\mathsf{false}) \tag{16}$$

We can now interpret an event E_k from (1) as a set transformer. Using the definitions from (2), we can now give the set transformer $[E_k]$ for E_k as [10]:

$$[E_k] \mathrel{\hat{=}} [g_k]; [a_k] \tag{17}$$

For a set of events, $\{E_1, \ldots, E_m\}$, we will use the denotion $[E]$ for the expression $[E_1] \sqcap \ldots \sqcap [E_m]$.

2.3 Refinement

Refinement is an important concept in Event-B. In this paper, we are mainly interested in refinement on the set transformer level, where it can be defined as [5]:

$$S_1 \sqsubseteq S_2 \mathrel{\hat{=}} \forall s. S_1(s) \subseteq S_2(s) \tag{18}$$

Here, S_1 and S_2 are set transformers. The intuitive interpretation of $S_1 \sqsubseteq S_2$ is that if S_1 will reach a state in a set s, then so will S_2. We say that S_1 and S_2 are (refinement) equivalent if and only if $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$. The relation between the set transformer view of refinement and a proof obligations approach has been studied in [10].

A set transformed S is said to behave miraculously when executed in a state in the set $S(\mathsf{false})$, i.e. when the execution of S results in a post-state belonging to the empty set. We typically want to avoid introduction of more miraculous behaviour during refinement. Given a set transformer S_1 and a refinement S_2, S_2 does not exhibit more miraculous behaviour than S_1 if $S_1(\mathsf{false}) = S_2(\mathsf{false})$.

2.4 Behavioural semantics

We aim at using Event-B for construction of concurrent programs. Ultimately we like to show that a (concurrent) program S is correct given a precondition P and a postcondition Q. This correctness requirement is expressed in the Hoare triple:

$$\{P\}\ S\ \{Q\} \tag{19}$$

As the basis for our method, we use the development method for concurrent programs in [12]. In this approach, the concurrent programs are built from atomic events in the same way as sequential programs are constructed [1]. The program S is considered to consist of a collection of events. Note that there is no control flow other than non-deterministic choice of enabled events. Using the refinement based approach of Event-B, the program S that satisfies the pre/post-specification is derived stepwise. In order to use the refinement process to develop programs, the pre-/post-specification first has to be encoded into an initial Event-B model. This model has a specific structure [1]: it has an initialisation event *init*, progress events *prog* and a finalisation event *fin*. The events *prog* model (non-deterministically) the computation of the program, while *fin* models the post-condition Q as a guard. The precondition is encoded in an external context machine. The semantics of an Event-B model M specifying a sequential program is in this setting:

$$M \mathrel{\hat{=}} [init]\,;[prog]^*;[fin] \tag{20}$$

The system is first initialised, then *prog* is executed until the postcondition given by *fin* becomes true. The program can then terminate. The progress events *prog* are later refined to create a deterministic algorithm to reach the postcondition. We will also later need to show that the refinements E of *prog* terminate [1], i.e. $[E]^\omega = [E]^*$, as we are interested in total correctness. We assume that all Event-B models in the rest of the paper have this structure. Each event should maintain the invariant and therefore we assume that there is an invariant assertion $\{i\}$ implicitly given before and after each event.

We previously mentioned that events can be classified as *ordinary*, *convergent* or *anticipated*. This is relevant from a behavioural semantics point of view. Events are normally classified as ordinary, but it is sometimes necessary to prove that execution of events from a group will eventually terminate. All events belonging to this group should then be labelled as convergent. In practice, the termination property is proven by introducing a variant, and by showing that it is decreased by all convergent events. There is also the possibility of classifying events as anticipated. Labelling an event as anticipated indicates that it will be classified as convergent in a later refinement step, whereby the proof is postponed until further down the refinement chain. The notions anticipated or convergent should be for the events *prog* to guarantee that the model eventually terminates.

2.5 Decomposition

In order for a refinement based development method to be scalable there should be a way to decompose specifications into smaller parts that can be independently developed. The verification of refinement should thus be compositional, i.e., refinement of the individual parts should yield a refinement of the whole system.

Here we will use a decomposition approach based on shared variables [1, 2]. Following this approach, a model can be decomposed into sub-models that can themselves be further decomposed. The set of sub-models forms the complete system model.

Definition 1. *Sub-model. A sub-model is given as a 7-tuple* $(v,x,E,X,I,init,fin)$*, where v and x are sets of variables, E and X are sets of events, I the invariant, init the initialisation and fin the finalisation.*

The variables v are only visible inside the sub-model, and will be referred to as internal variables. Variables x are shared with other components and will be called external variables. The events E can refer to both v and x. Since they (also) manipulate the internal variables of the sub-model, they are denoted the internal events. The external events, X, are abstractions that only refer to the external variables x modelling the effects of events of other components. Hence, each event in X has a corresponding internal event in another component. The initialisation of a sub-model is given by event *init* and the loop

termination guard is given by event *fin*. Note that a traditional Event-B model can be seen as a sub-model where the sets of external events and external variables are empty. A sub-model $(v,x,E,X,I,init,fin)$ can be (further) decomposed into sub-models:

$$(v,x,E,X,I,init,fin) = (v_1,x_1,E_1,X_1,I_1,init_1,fin_1) \parallel (v_2,x_2,E_2,X_2,I_2,init_2,fin_2)$$

The parallel composition of the sub-models is defined as:

$$\begin{array}{ll} & (v_1,x_1,E_1,X_1,I_1,init_1,fin_1) \parallel (v_2,x_2,E_2,X_2,I_2,init_2,fin_2) \\ \widehat{=} & (v_1 \cup v_2,(x_1 \cup x_2)\backslash(v_1 \cup v_2),E_1 \cup E_2,(X_1 \cup X_2)\backslash(E_1 \cup E_2),I_1 \wedge I_2,init_1 \parallel init_2,\ fin_1 \parallel fin_2) \end{array} \quad (21)$$

The parallel composition of two events is given as:

$$\begin{array}{ll} & \textbf{when } G \textbf{ then } v : |S \textbf{ end} \parallel \textbf{when } H \textbf{ then } w : |R \textbf{ end} \\ \widehat{=} & \textbf{when } G \wedge H \textbf{ then } v,w : |S \wedge R \textbf{ end} \end{array} \quad (22)$$

The semantics $[M_1 \parallel M_2]$ of a the parallel composition $M_1 \parallel M_2$ is given as:

$$[M_1 \parallel M_2] \mathrel{\widehat{=}} init_1 \parallel init_2; ([E_1 \cup E_2 \cup ((X_1 \cup X_2)\backslash(E_1 \cup E_2))])^*; [\neg \mathrm{g}(fin_1 \parallel fin_2)] \quad (23)$$

The composition can be extended to arbitrary many components by recursively merging components pairwise. Since we want to do compositional proofs of refinement, we need to show that refinement of the individual sub-models lead to refinement of the entire system. First we need to prove that the external events provide abstractions of their internal counterparts $\{i_1 \cap i_2\}; [X_1] \sqsubseteq [E_2] \sqcap [X_2]$ and $\{i_1 \cap i_2\}; [X_2] \sqsubseteq [E_1] \sqcap [X_1]$. To compositionally prove the refinement $[M_1 \parallel M_2] \sqsubseteq [M_1' \parallel M_2]$, we then only need to prove the refinement $[M_1] \sqsubseteq [M_1']$, see [7].

We need to model that external events are executed a finite number of times, as they model the finite execution of their internal counterparts in other sub-models. Since these external events are not necessarily terminating by themselves, strong iteration cannot be used for describing behaviour of sub-models. The use of weak iteration can be seen as compositionally verifying partial correctness of a program, since termination is not ensured by set transformer refinement. However, we want to prove total correctness of the complete system. Since we in this approach [1, 12] label the events E as anticipated or convergent, we show that the model will eventually terminate. Hence, total correctness follows from partial correctness in combination with the Event-B proof obligations that ensure termination [5, 6].

3 Dining philosophers case study

3.1 Problem description

We are now ready to introduce a model of the dining philosophers [13], which will serve as a running example. In this section, we show the initial model, we refine it, as well as decompose it into sub-models. The dining philosophers scenario can be described as follows. There are four philosophers sitting around a round table. Each philosopher has a plate in front of him, and there is a fork placed between each pair of adjacent plates. Each philosopher always does one of two things: think and eat, but not both at the same time. Furthermore, in order to eat, a philosopher must pick up both of the two forks located next to his plate. A philosopher can also drop a fork back into its original position, but only after he has eaten.

The basic problem is that if the philosophers pick up the forks arbitrarily, there may be deadlocks. For example, if each philosopher picks up his right fork, there will not be any forks available anymore,

and no philosopher will have enough forks to eat. Since a philosopher will not drop a fork until he has eaten, there will be a deadlock. One well-known solution to this problem is to assign a number to each fork, and enforce that each philosopher picks up the adjacent fork with the lowest number first. In our case study we assume that we have four philosophers and number the forks as follows: Philosopher 1 can access forks 1 and 2, philosopher 2 accesses forks 2 and 3, philosopher 3 uses forks 3 and 4, while philosopher 4 has access to forks 1 and 4.

3.2 Modelling and refinement

Initially we model the scenario as an abstract Event-B machine, where the four philosophers eat in a non-deterministic order. We only model one round, so each philosopher will only eat once. We introduce the variables *ph1eaten* thru *ph4eaten*, to model whether each philosopher has eaten. The event *Intialisation* sets these variables to FALSE. The events *Ph1Eat* thru *Ph4Eat* for the four philosophers then represent the progress of the model. They model that a philosopher eats which has not yet eaten by setting the corresponding variable to TRUE. Finally, event *Finalisation* checks that all four philosophers have eaten. The *Initialisation* and *Finalisation* events are classified as ordinary events, whereas *Ph1Eat*, ..., *Ph4Eat* are convergent, since they correspond to the *prog* variables in (20). We now have:

variables
ph1eaten
ph2eaten
ph3eaten
ph4eaten

invariant
ph1eaten ∈ BOOL
ph2eaten ∈ BOOL
ph3eaten ∈ BOOL
ph4eaten ∈ BOOL

Initialisation (*ordinary*) ≙
begin
ph1eaten := FALSE
ph2eaten := FALSE
ph3eaten := FALSE
ph4eaten := FALSE
end

Ph1Eat (*convergent*) ≙
when
ph1eaten = FALSE
then
ph1eaten := TRUE
end

Finalisation (*ordinary*) ≙
when
ph1eaten = TRUE
ph2eaten = TRUE
ph3eaten = TRUE
ph4eaten = TRUE
then
skip
end

In the first refinement step we introduce the forks, which are modelled as variables *fork1* thru *fork4*. They are of type 0..4 to represent which philosopher that currently holds the fork. Value 0 represents the fork lying on the table. All forks are initialised to this value. There are 16 new events in this refinement step: two for each of the four philosophers getting their adjacent forks (e.g. *Ph3GetFork3* and *Ph3GetFork4*), and two events for each philosopher releasing the corresponding forks (e.g. *Ph3RelFork4* and *Ph3RelFork3*). Note that philosopher 4 uses forks 1 and 4.

In order to be able to prove that the new events will not take over the execution, we classify them as convergent and give a variant that they decrease. There is no variable that can be used as a variant, but when each new event is executed it will disable itself and it will not be enabled again. Hence, we define a function *v* as follows:

$$v = \{ \quad (FALSE, FALSE, FALSE) \mapsto 5,$$
$$(TRUE, FALSE, FALSE) \mapsto 4,$$
$$(TRUE, TRUE, FALSE) \mapsto 3,$$
$$(TRUE, TRUE, TRUE) \mapsto 2,$$
$$(TRUE, FALSE, TRUE) \mapsto 1,$$
$$(FALSE, FALSE, TRUE) \mapsto 0\}$$

The first and second dimension of the triple correspond to whether a philosopher is holding his left or right fork, respectively. The third one indicates whether he has already eaten or not. The variant is then formed as a sum of the values of function *v* applied on the variables of each philosopher. The refined model is now as follows:

variables
fork1
fork2
fork3
fork4
ph1eaten
ph2eaten
ph3eaten
ph4eaten

invariant
$fork1 \in 0..4$
$fork2 \in 0..4$
$fork3 \in 0..4$
$fork4 \in 0..4$
...
variant
$v(bool(fork1 = 1), bool(fork2 = 1), ph1eaten)$
$+v(bool(fork2 = 2), bool(fork3 = 2), ph2eaten)$
$+v(bool(fork3 = 3), bool(fork4 = 3), ph3eaten)$
$+v(bool(fork1 = 4), bool(fork4 = 4), ph4eaten)$

Initialisation (*ordinary*) $\widehat{=}$
begin
fork1 := 0
fork2 := 0
fork3 := 0
fork4 := 0
ph1eaten := FALSE
ph2eaten := FALSE
ph3eaten := FALSE
ph4eaten := FALSE
end

Ph1GetFork1 (*convergent*) $\widehat{=}$
when
fork1 = 0
ph1eaten = FALSE
then
fork1 := 1
end

Ph1GetFork2 (*convergent*) $\widehat{=}$
when
fork1 = 1
fork2 = 0
ph1eaten = FALSE
then
fork2 := 1
end

Ph1Eat (*convergent*) $\widehat{=}$
when
fork1 = 1
fork2 = 1
ph1eaten = FALSE
then
ph1eaten := TRUE
end

Ph1RelFork2 (*convergent*) $\widehat{=}$
when
fork2 = 1
ph1eaten = TRUE
then
fork2 := 0
end

Ph1RelFork1 (*convergent*) $\widehat{=}$
when
fork2= 0
fork1= 1
ph1eaten = TRUE
then
fork1 := 0
end

Finalisation (*ordinary*) $\widehat{=}$
when
fork1 = 0
fork2 = 0
fork3 = 0
fork4 = 0
ph1eaten = TRUE
ph2eaten = TRUE
ph3eaten = TRUE
ph4eaten = TRUE
then
skip
end

Note that when the *v* function is called, the fork variables are not directly passed as parameters. Instead, we check whether the currently evaluated philosopher holds the fork or not. The *bool* function is a technicality of Event-B that is needed to convert the result of the comparison into a value of BOOL.

The events corresponding to philosophers 2, 3 and 4 eating, as well as picking up and releasing their respective forks are analogous to the events of philosopher 1, and are thus not shown here. We now have a refined model for the four philosophers eating, and in the next subsection we will decompose this model.

3.3 Decomposition

In the decomposition step we separate the functionality of the four philosophers in such a way that each philosopher constitutes a sub-model of its own. The partitioning we achieve is shown in the table below. Since philosophers 2 and 4 share fork 2 and fork 1, respectively, with philosopher 1, the external events of sub-model 1 are Ph2GetFork2, Ph2RelFork2, Ph4GetFork1 and Ph4RelFork1. Analogous reasoning is used to find the external events of the other sub-models.

	Sub-model 1	Sub-model 2	Sub-model 3	Sub-model 4
Internal events	Ph1Eat	Ph2Eat	Ph3Eat	Ph4Eat
	Ph1GetFork1	Ph2GetFork2	Ph3GetFork3	Ph4GetFork1
	Ph1RelFork1	Ph2RelFork2	Ph3RelFork3	Ph4RelFork1
	Ph1GetFork2	Ph2GetFork3	Ph3GetFork4	Ph4GetFork4
	Ph1RelFork2	Ph2RelFork3	Ph3RelFork4	Ph4RelFork4
External events	Ph2GetFork2	Ph1GetFork2	Ph2GetFork3	Ph1GetFork1
	Ph2RelFork2	Ph1RelFork2	Ph2RelFork3	Ph1RelFork1
	Ph4GetFork1	Ph3GetFork3	Ph4GetFork4	Ph3GetFork4
	Ph4RelFork1	Ph3RelFork3	Ph4RelFork4	Ph3RelFork4

4 Concurrent programs

This far, we have considered model decomposition, resulting in sub-models that can be refined semi-independently. We are now ready to examine how these sub-models can be executed in a concurrent or parallel setting. This problem has been studied in [12], which is a case study showing how to decompose Event-B models into concurrently executing sub-models. Here we extend this approach by giving sub-models explicit flow control in the form of event schedules, instead of the traditional nondeterministic choice. An important concept in our approach is the concept of *tasks*, which we define as follows:

Definition 2. *Task. A task is an 8-tuple* $(v, x, E, X, I, init, fin, S)$ *where v are the internal variables, x the external variables, E the internal events, X the external events, I the invariant, init the initialisation,* fin *the loop termination condition, and S is a schedule conforming to the syntax in (24) concerning the internal events* E.

Since all coordinates, except for S, are the same as in a sub-model, a task can be seen as an extension of the sub-model concept. Whereas the events of traditional decomposed sub-models are executed non-deterministically, the internal events of a task are scheduled according to S. The schedule S may only consist of internal events, and the set of events in the schedule is denoted $\mathsf{e}(S)$. We assume that $E = \mathsf{e}(S)$, since if an internal event was not included in the schedule, it would never be executed.

4.1 Scheduling language

In order to describe schedules of events we give a small scheduling language [8], which adheres to the following syntax:

$$\begin{array}{lcl} S & ::= & PS \rightarrow S \mid PS \\ PS & ::= & \textbf{do } S \textbf{ od} \mid S_1 [] \ldots [] S_n \mid E \mid \{g\} \end{array} \tag{24}$$

Here $\rightarrow$ represents sequential composition, $[]$ non-deterministic choice, **do od** is a loop, E an event and $\{g\}$ is an assertion.

4.2 Semantics of tasks

The semantics of schedules is given using a function sched that maps each schedule to the corresponding set transformer as in [8]. However, when scheduling the events in a task we need to consider interference from other tasks. A goal of the scheduling language is to be able to express schedules of internal events in such a way that interference from external events does not have to be explicitly taken into account. Such interference freedom is instead proven separately. We now recursively define a function $\mathsf{sched}(S,X)$ where S is a schedule, X is the set of external events.

$$\begin{array}{lcl} \mathsf{sched}(PS \rightarrow S, X) & = & \mathsf{sched}(PS, X); \mathsf{sched}(S, X) \\ \mathsf{sched}(\textbf{do } S \textbf{ od}, X) & = & ([\mathsf{g}([\mathsf{e}(S) \cup X])]; \mathsf{sched}(S, X))^*; [\neg \mathsf{g}([\mathsf{e}(S) \cup X])] \\ \mathsf{sched}(S_1 [] \ldots [] S_n, X) & = & \mathsf{sched}(S_1, X) \sqcap \ldots \sqcap \mathsf{sched}(S_n, X) \\ \mathsf{sched}(E, X) & = & [X]^*; [E]; [X]^* \\ \mathsf{sched}(\{g\}, X) & = & \{g\} \end{array} \tag{25}$$

The scheduling function takes the schedule S, as well as the set of external events X as input and outputs a set transformer containing both internal and external events. An arbitrary (but finite) number of external events X can occur before and after an internal event E in a schedule. This is modelled by the set transformer $[X]^*$ on both sides of the event.

Consider a system consisting of two tasks $T_1 = (v_1, x_1, E_1, X_1, init_1, fin_1, S_1)$ and $T_2 = (v_2, x_2, E_2, X_2, init_2, fin_2, S_2)$. To find the complete system behaviour, we need to compose the tasks, i.e. obtain $T_1 \parallel T_2$. However, the number of interleavings of atomic set transformers grows exponentially with the length of the schedule [19]. Hence, we need an appropriate approach to reason about the interleavings in order to make refinement proofs manageable. Here we make the restriction that we only consider tasks where the set transformers obtained after scheduling can be decomposed into a loop containing the demonic choice of atomic set transformers. This is an extension of the approach used in [12], where the programs are built from atomic *events* that are chosen non-deterministically for execution. Composition of such tasks can be easily handled [7]. We have the following requirement for schedulability in our approach:

$$\exists S_{11}, \ldots, S_{1n} \cdot \mathsf{sched}(S_1, X_1) = (S_{11} \sqcap \ldots \sqcap S_{1n} \sqcap [X_1])^*; [fin_1] \tag{26}$$

where all S_{1i} are atomic compositions of internal events. Using these atomic set transformers we can now use the traditional parallel composition [7]. The semantics of the composition of the whole system $T_1 \parallel T_2$ is now given as:

$$[T_1 \parallel T_2] \mathrel{\hat{=}} [init_1 \parallel init_2]; ((\sqcap_i S_{1i}) \sqcap (\sqcap_j S_{2j}))^*; [fin_1 \parallel fin_2] \tag{27}$$

This approach thus extends the decomposition method in [2, 12] with the possibility to reason about groups of sequentially scheduled events, instead of only individual ones. However, to find the groups

$S_{11},\ldots,S_{1n}$ is in general non-trivial. Here we will give special cases encoded as *patterns* to make the verification of schedules manageable in practise.

4.3 Introduction of schedules

Schedules are introduced for the sub-models as a refinement step, in which we convert sub-models into tasks. The introduction of schedules has to constitute a refinement step in order to ensure that the properties we have already proved for the models before introduction of schedules are preserved. Note that we do not support scheduling of anticipated events, so they have to be turned into convergent ones before the introduction of schedules.

We now need to show for the two tasks $T_1 = (v_1, x_1, E_1, X_1, init_1, fin_1, S_1)$ and $T_2 = (v_2, x_2, E_2, X_2, init_1, fin_1, S_2)$:

$$[M_1 \parallel M_2] \sqsubseteq [T_1 \parallel T_2] \tag{28}$$

where sub-model M_i corresponds to task T_i as $M_i = (v_i, x_i, E_i, X_i, init_i, fin_i)$. As in the traditional decomposition method, we can use external events to perform compositional proofs of refinement. Here we rely on the property (26) to decompose schedule $\mathsf{sched}(S_i, X_i)$ into a loop consisting of atomic set transformers. We need to show that for all tasks T_i [7]:

$$\{i_1 \cap i_2\}; [X_{ij}] \sqsubseteq S_{kj} \tag{29}$$

$$([\mathsf{e}(S_i)] \sqcap [X_i])^*; [fin_i] \sqsubseteq \mathsf{sched}(S_i, X_i) \tag{30}$$

In (29) we assume that for any external event $X_{ij} \in X_i$, there is one corresponding atomic set transformer S_{kj} in another task T_k. To give a practical approach to the decomposition of schedules required by (26), we give patterns that give generic instantiations of the quantified variables. In the patterns we rely on special cases of scheduling constructs where we know we can prove (29) and (30). Patterns thus encode reusable schedule structures. One such case is when the introduction of sequential behaviour does not alter the behaviour of the sub-model. Another useful special case is when the introduction of sequential behaviour does not modify the externally visible behaviour of a sub-model. We use the same scheduling approach as in [8], where patterns are applied on schedules stepwise and we prove that each pattern application leads to a refinement of the previous application.

A pattern consists of a *precondition*, a *schedule*, a *result* and a number of *assumption*. The precondition predicate describes under which conditions the pattern is applicable. The schedule part describes what schedule the pattern is intended for, and the result part gives the set transformer that is produced when the pattern is applied. The assumptions are extra conditions that have be fulfilled in order to use the pattern.

Pattern 1 The first pattern, P_1, introduces sequential behaviour into a sub-model.

$$\begin{array}{lcl}
P_1(E_1, h, g, S, X) \mathrel{\hat{=}} & & \\
\text{Precondition} & : & h \\
\text{Schedule} & : & E_1 \rightarrow \{g\} \rightarrow S \\
\text{Result} & : & \{h\}; X^*; E_1; X^*; \{g\}; \mathsf{sched}(S, X) \\
\text{Assumption 1} & : & h \subseteq \neg \mathrm{g}(\mathsf{e}(S)) \\
\text{Assumption 2} & : & g \subseteq \neg \mathrm{g}(E_1) \\
\text{Assumption 3} & : & \{g\}; (X \sqcap \mathsf{e}(S)) \sqsubseteq (X \sqcap \mathsf{e}(S)); \{g\} \\
\text{Assumption 4} & : & \{h\}; X \sqsubseteq X; \{h\} \\
\text{Assumption 5} & : & E_1 = E_1; \{g\}
\end{array} \tag{31}$$

The first assumption states that the precondition h implies that the events following E_1 are disabled. The second assumption states that g ensures that E_1 is disabled. Context information cannot be propagated in schedules without taking interference into account. Hence we need assumptions 3 and 4 to state that g and h are invariant with respect to the environment. Furthermore, g should also be invariant for all events in the schedule S. The last assumption states that E_1 will establish g. We also directly use the event name E_1 instead of the set transformer $[E_1]$, as well as E instead of $[E]$.

In order to stepwise use patterns we need to show that each application of a pattern is correct, i.e. that (30) holds. In order to do that, we assume that $\mathsf{sched}(S,X)$ represents a yet unscheduled loop of events $\mathsf{sched}(S,X) = (\mathsf{e}(S) \sqcap X)^*; [\mathsf{g}(\mathsf{e}(S) \sqcap X)]$. We instantiate the existential quantifier in (26) with S_i as E_i. Hence, we then need to show that $\{h\}; \mathsf{sched}(E_1 \to \{g\} \to S) = \{h\}; X^*; E_1; X^*; \{g\}; \mathsf{sched}(S)$. Note that we also rely here on the properties (32)-(34) in Lemma 1. Note also that to ensure (30) we here assume $i \cap \neg\mathsf{g}(E \sqcap X) \subseteq \mathsf{g}(fin)$. The reason for formulating the pattern in this way is to be able to use the same verification approach also to nested loops.

Lemma 1. *Context preservation. If* $\{g\}; S \sqsubseteq S; \{g\}$ *then:*

$$\{g\}; S = \{g\}; S; \{g\} \tag{32}$$
$$\{g\}; S^* = \{g\}; S^*; \{g\} \tag{33}$$
$$\{g\}; S^* = (\{g\}; S)^* \tag{34}$$

The proofs of the properties in the lemma are straightforward and they are omitted for brevity. We can now prove the correctness of pattern P_1.

Proof.

$$\begin{array}{ll}
 & \{h\}; \mathsf{sched}(E_1 \to \{g\} \to S, X); [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Representation of } \mathsf{sched}(E_1 \to \{g\} \to S)\} \\
 & \{h\}; (E_1 \sqcap E \sqcap X)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Decomposition [6]} : (S \sqcap T)^* = (S; T^*)^*; T^*\} \\
 & \{h\}; X^*; (E_1 \sqcap E; X^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Distributivity}\} \\
 & \{h\}; X^*; ((E_1; X^*) \sqcap (E; X^*))^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Decomposition}\} \\
 & \{h\}; X^*; ((E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Unfolding (15)}\} \\
 & \{h\}; X^*; ((E_1; X^*); (E_1; X^*)^*) \sqcap \mathsf{skip}; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Assumption 3 and Property (33)}\}\} \\
 & \{h\}; X^*; \{h\}; (E_1; X^*); (E_1; X^*)^*) \sqcap \{h\}; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Distributivity, assumption } h \subseteq \neg\mathsf{g}(E) \text{ and disabledness of guard}\} \\
 & \{h\}; X^*; \{h\}; (E_1; X^*); (E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Assumption } E_1 = E_1; \{g\}\} \\
 & \{h\}; X^*; \{h\}; E_1; X^*; \{g\}; (E_1; X^*)^*; ((E; X^*); (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Assumption } g \subseteq \neg\mathsf{g}(E_1)\} \\
 & \{h\}; X^*; \{h\}; E_1; X^*; \{g\}; (E; X^*; (E_1; X^*)^*)^*; [\neg\mathsf{g}(E_1 \sqcap E \sqcap X)]
\end{array}$$

$$
\begin{array}{ll}
= & \{\text{Property (34) and } * \text{ below}\} \\
& \{h\};X^*;\{h\};E_1;\{g\};X^*;\{g\};(\{g\};E;X^*;\{g\})^*;[\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Leapfrog [6] : } S;(T;S)^* = (S;T)^*;S\} \\
& \{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};E;X^*)^*;\{g\};[\neg\mathsf{g}(E_1 \sqcap E \sqcap X)] \\
= & \{\text{Assumption } g \subseteq \neg\mathsf{g}(E_1) \text{ and } \{g\};[g] = \{g\}\} \\
& \{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};(E;X^*))^*;\{g\};[\neg\mathsf{g}(E \sqcap X)] \\
= & \{\text{Lemma 9(c) in [6] : } S^* = S^*;S^* \text{ and decomposition}\} \\
& \{h\};X^*;\{h\};E_1;\{g\};X^*;(\{g\};E \sqcap \{g\};X)^*;[\neg\mathsf{g}(E \sqcap X)] \\
= & \{\text{Property (33) and assumption 5}\} \\
& \{h\};X^*;E_1;X^*;\{g\};(\{g\};E \sqcap \{g\};X)^*;[\neg\mathsf{g}(E \sqcap X)] \\
= & \{\text{Representation of } \mathsf{sched}(S,X)\} \\
& \{h\};X^*;E_1;X^*;\{g\};\mathsf{sched}(S,X)
\end{array}
$$

The proof of step $*$ is:

$$
\begin{array}{ll}
& (\{\mathsf{g}\};E;X^*;(E_1;X^*)^*)^* \\
= & \{\text{Assumption 3 and Properties (32) and(33)}\} \\
& (\{\mathsf{g}\};E;X^*;\{\mathsf{g}\};(E_1;X^*)^*)^* \\
= & \{\text{Assumption 2}\} \\
& (\{\mathsf{g}\};E;X^*;\{\mathsf{g}\})^*
\end{array}
$$

□

Pattern 2 The second pattern, P_2, also introduces sequential behaviour. However, this time we show that we can group local behaviour E_2 to an arbitrary event.

$$
\begin{array}{lll}
P_2(E_1,E_2,h,g,S_1,X) \mathrel{\hat{=}} & & \\
\text{Precondition} & : & h \\
\text{Schedule} & : & E_1 \to E_2 \to \{g\} \to S \\
\text{Result} & : & \{h\};X^*;E_1;X^*;E_2;X^*;\{g\};\mathsf{sched}(S,X) \\
\text{Assumption 1} & : & h \subseteq \neg\mathsf{g}(\mathsf{e}(S)) \\
\text{Assumption 2} & : & g \subseteq \neg\mathsf{g}(E_1 \sqcap E_2) \\
\text{Assumption 3} & : & E_2;X = X;E_2 \\
\text{Assumption 4} & : & \{\mathsf{g}(E_2)\};X = X;\{\mathsf{g}(E_2)\} \\
\text{Assumption 5} & : & \{g\};(X \sqcap \mathsf{e}(S)) \sqsubseteq (X \sqcap \mathsf{e}(S));\{g\} \\
\text{Assumption 6} & : & \{h\};X \sqsubseteq X;\{h\} \\
\text{Assumption 7} & : & E_2 = E_2;\{g\}
\end{array}
\tag{35}
$$

The assumptions in pattern P_2 are similar to the ones in P_1. However, we additionally need assumptions that states that E_2 and X do not interfere with each other (assumptions 3 and 4). To prove the correctness of the pattern we need to show that

- By instantiation of (26) we get: $\{h\};X^*;E_1;X^*;E_2;X^*;\{g\};\mathsf{sched}(S,X) = \{h\};(E_1;E_2 \sqcap \mathsf{e}(S) \sqcap X)^*;[\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap \mathsf{e}(S) \sqcap X)]$
- Refinement (30): $\{h\};\mathsf{sched}(E_1 \to E_2 \to \{g\} \to S,X) \sqsubseteq \{h\};(E_1;E_2 \sqcap \mathsf{e}(S) \sqcap X)^*;[\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap \mathsf{e}(S) \sqcap X)]$
- Deadlock freedom: $\{h\};(E_1;E_2 \sqcap \mathsf{e}(S) \sqcap X)^*;[\neg\mathsf{g}(E_1 \sqcap E_2 \sqcap \mathsf{e}(S) \sqcap X)](\mathsf{false}) = \{h\};\mathsf{sched}(E_1 \to E_2 \to \{g\} \to S,X)(\mathsf{false})$

The deadlock freedom proof obligation ensures that the scheduling does not introduce new deadlocks. This was not needed in pattern P_1, as that pattern does not alter the behaviour of models. The proofs are straightforward using the assumptions in the pattern. This ensures that the scheduling does not introduce more deadlocks than in the original system.

5 Scheduling of dining philosophers

We now return to the running example introduced in section 3. Up till now, the dining philosophers model has been refined and split into sub-models. Now, we show how the sub-models can be turned into tasks by introducing schedules. In the scheduling process we use the patterns given in section 4.3. Correctness will be proven by checking the assumptions of the patterns. We will concentrate on how to derive a schedule for task 1. The schedules for task 2, 3 and 4 can be derived analogously.

Our approach is that the schedule should be formulated such that it fulfills the previously mentioned solution to the dining philosophers problem, i.e., that each philosopher should pick up the lower numbered fork first. Since we first want to pick up fork number 1, we wish to schedule *Ph1GetFork1* as the first event. The correct order of events will be *Ph1GetFork1*, *Ph1GetFork2*, *Ph1Eat*, *Ph1RelFork2*, *Ph1RelFork1*. This is captured by the following schedule:

$$\begin{aligned}&\text{Ph1GetFork1} \rightarrow \{g_1\} \rightarrow \text{Ph1GetFork2} \rightarrow \text{Ph1Eat} \rightarrow \{g_2\}\\ &\rightarrow \text{Ph1RelFork2} \rightarrow \{g_3\} \rightarrow \text{Ph1RelFork1} \rightarrow \{g_4\}\end{aligned}$$

The assertions in the schedule are needed to capture intermediate results and thereby enable verification of the schedule in smaller parts.

We now want to prove that it is correct to schedule *Ph1GetFork1* as the first event. To show this, we will follow pattern P_1 introduced in Section 4.3 and show that the assumptions 1 - 5 for the pattern are fulfilled. We instantiate pattern P_1 as $P_1(Ph1GetFork1, h_1, g_1, S_r, X_{t1})$, where $h_1 = (fork1 \neq 1 \wedge ph1eaten = FALSE)$, $g_1 = (fork1 = 1 \vee ph1eaten = TRUE)$, $S_r = \text{Ph1GetFork2} \rightarrow \text{Ph1Eat} \rightarrow \{g_2\} \rightarrow \text{Ph1RelFork2} \rightarrow \{g_3\} \rightarrow \text{Ph1RelFork1} \rightarrow \{g_4\}$ and $X_{t1} = \{\text{Ph2GetFork2, Ph4GetFork1, Ph2RelFork2, Ph4RelFork1}\}$.

We chose precondition h_1 so that it also is an invariant for the external events X_{t1}. Here, h_1 states that philosopher 1 does not hold his forks nor has he eaten. Moreover, we chose assertion g_1 to state that philosopher 1 has picked up fork 1 or eaten. This condition is an invariant for the events $\mathsf{e}(S_r) \cup X_{t1}$ and established by *Ph1GetFork1*. We now confirm that the assumptions for the pattern hold:

- $h_1 = (fork1 \neq 1 \wedge ph1eaten = FALSE)$ implies that events in $\mathsf{e}(S_r)$ are disabled. This holds, since they are only enabled when philosopher 1 holds fork 1 or has eaten.
- The assertion $g_1 = (fork1 = 1 \vee ph1eaten = TRUE)$ following event *Ph1GetFork1* ensures that *Ph1GetFork1* is disabled. Since g_1 is a negation of the guard of *Ph1GetFork1* the second assumption is fulfilled.
- g_1 is an invariant of the environment $\mathsf{e}(S_r) \cup X_{t1}$. This is fulfilled, since in the events of $\mathsf{e}(S_r)$ philosopher 1 holds fork 1 or has eaten. Moreover, the events in X_{t1} that share fork 1 are not enabled when philosopher 1 holds fork 1, and none of these events modify variable *ph1eaten*.
- h_1 is an invariant of the external events X_{t1}. Since none of the external events model that philosopher 1 picks up fork 1 or modify variable *ph1eaten*, this assumption holds.
- Event *Ph1GetFork1* establishes g_1. This holds trivially since *Ph1GetFork1* models that philosopher 1 picks up fork 1 ($fork1 := 1$).

To verify the complete schedule, we then apply pattern P_2 once, followed by three applications of P_1. In the last application of P_1, the schedule following the assertion is empty. This can be interpreted as a schedule with an event that is always disabled. When task 1 has been fully proven, the whole procedure is repeated to schedule tasks 2, 3 and 4 in the order shown in the table below (for simplicity, the assertions are not shown).

Task 1	Task 2	Task 3	Task 4
Ph1GetFork1	Ph2GetFork2	Ph3GetFork3	Ph4GetFork1
$\rightarrow$ Ph1GetFork2	$\rightarrow$ Ph2GetFork3	$\rightarrow$ Ph3GetFork4	$\rightarrow$ Ph4GetFork4
$\rightarrow$ Ph1Eat	$\rightarrow$ Ph2Eat	$\rightarrow$ Ph3Eat	$\rightarrow$ Ph4Eat
$\rightarrow$ Ph1RelFork2	$\rightarrow$ Ph2RelFork3	$\rightarrow$ Ph3RelFork4	$\rightarrow$ Ph4RelFork4
$\rightarrow$ Ph1RelFork1	$\rightarrow$ Ph2RelFork2	$\rightarrow$ Ph3RelFork3	$\rightarrow$ Ph4RelFork1

6 Conclusions and related work

In this paper, we have proposed a method of correct-by construction development of concurrent programs using Event-B. The programs are first developed as proposed by Hoang and Abrial [12]. From this development process we obtain a number of sub-models that communicate via shared variables, which represent the program. We then introduce explicit control flow in the form of schedules for each sub-model, so that each sub-model/schedule corresponds to exactly one task. The schedules are introduced as correctness preserving refinements. We use a set-transformer semantics for Event-B, as well as well known algebraic rules [6] for the analysis of correctness. The schedules are verified in a stepwise manner, and each step carries some related proof obligations. The schedules enable more efficient implementation of the Event-B models as more explicit control flow information is available than for pure event-B models. We can, e.g., use the transformations in [8] to introduce traditional control flow constructs, such as while loops and if-statements, as well as remove unnecessary guards. Furthermore, the schedules give a process-oriented specification of the behaviour of the models.

Our goal is to compositionally reason about concurrent programs. This has been a very active field of research [19]. Our approach directly extends the approach in [12] for development of concurrent programs with explicit schedules of events. Compositional reasoning in this setting goes back to the work of Owicki and Gries [16] and Jones' Rely-Guarantee reasoning [15]. The decomposition method based on shared variables in Event-B [2, 12] is based on these ideas. Essentially the same approach is also available for action systems using the refinement calculus [7]. The theory for decomposition in the set-transformer setting is largely based on that paper. Several approaches to introducing control flow into Event-B models have been developed. Hallerstede's approach in [11] to adding control flow only deals with sequential programs and it is thus more related to Boström's earlier work [8]. The scheduling approaches in [14, 20] can also handle concurrent schedules. In [14] the scheduling (referred to as *flows*) is expressed using a special purpose language, while in the approach [20] the scheduling is expressed in CSP. The latter approach can be seen as an extension of the former. Processes or flows are both considered to communicate via shared events. Our focus is on compositional verification and scheduling of concurrent programs that use shared variables for communication. However, in both approaches not all events need to be scheduled, but non-scheduled events are considered interleaved in the scheduled. This could be used to take into account external events, and thus be used for compositional verification of shared variable programs also. Our contribution is threefold: 1) Compared to purely event-based modelling, we consider explicit schedules of events that can be interleaved 2) We do all analysis on the level of set transformers, which gives convenient formalism to algebraically perform the needed analysis

of Event-B models 3) We provide patterns and a method to develop patterns for introducing control flow in a stepwise manner. This is important, since verifying that a certain event schedule is correct can be very challenging and reusable scheduling structures can significantly aid in this task.

Set-transformers give a powerful framework to reason about Event-B models on a high level of abstraction. They give a good basis for creating reusable patterns for scheduling, which are essential for practical applications. If schedules are introduced as a last refinement step, as in the example of this paper, existing tool support can be used for development up till, but not including, the scheduling step. Future work involves investigating tool support for schedule application. Generation of refinement proof obligations for scheduled models is also of interest, since that would allow for schedule intoduction earlier in the refinement chain.

References

[1] J. R. Abrial (2010): *Modeling in Event B: System and Software Engineering*. Cambridge University Press.

[2] J.-R. Abrial & S. Hallerstede (2007): *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae* 77(1-2), pp. 1–28.

[3] R.-J. R. Back & R. Kurki-Suonio (1983): *Decentralization of Process Nets with Centralized Control*. In: *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 131–142, doi:10.1145/800221.806716.

[4] R.-J. R. Back & K. Sere (1991): *Stepwise Refinement of Action Systems. Structured Programming* 12, pp. 17–30.

[5] R.-J. R. Back & J. von Wright (1998): *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Springer-Verlag.

[6] R.-J. R. Back & J. von Wright (1999): *Reasoning algebraically about loops. Acta Informatica* 36, pp. 295–334, doi:10.1007/s002360050163.

[7] R.-J. R. Back & J. von Wright (2003): *Compositional Action System Refinement. Formal Aspects of Computing* 15, pp. 103–117, doi:10.1007/s00165-003-0005-6.

[8] P. Boström (2010): *Creating sequential programs from Event-B models*. In: *IFM'10 Proceedings of the 8th international conference on Integrated formal methods, LNCS* 6396, Springer-Verlag, pp. 74–88, doi:10.1007/978-3-642-16265-7_7.

[9] M. Butler (2000): *csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing* 12(3), pp. 182–198, doi:10.1007/PL00003930.

[10] S. Hallerstede (2008): *On the purpose of Event-B proof obligations*. In: *Abstract State Machines, B and Z, LNCS* 5238, Springer-Verlag, pp. 125–138, doi:10.1007/978-3-540-87603-8_11.

[11] S. Hallerstede (2010): *Structured Event-B models and proofs*. In: *Abstract State Machines, B and Z, LNCS* 5977, Springer-Verlag, pp. 273–286, doi:10.1007/978-3-642-11811-1_21.

[12] T. S. Hoang & J-.R. Abrial (2010): *Event-B decomposition for parallel programs*. In: *ABZ2010, LNCS* 5977, Springer-Verlag, pp. 319–333, doi:10.1007/978-3-642-11811-1_24.

[13] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall.

[14] A. Iliasov (2009): *On Event-B and control flow*. Technical Report CS-TR-1159, School of Computing Science, Newcastle University.

[15] C. B. Jones (1983): *Tentative steps toward a development method for interfering programs. Transactions on Programming languages and systems* 5(4), pp. 596–619, doi:10.1145/69575.69577.

[16] S. S. Owicki & D. Gries (1976): *An axiomatic proof technique for parallel programs I. Acta Informatica* 6, pp. 319–340, doi:10.1007/BF00268134.

[17] J. Plosila, K. Sere & M. Waldén (2005): *Asynchronous system synthesis*. *Science of Computer Programming* 55, pp. 259–288, doi:10.1016/j.scico.2004.05.018.

[18] (2010): *Rodin Platform*. `http://www.event-b.org`.

[19] W. P. de Roever & et. al. (2001): *Concurrency Verification: Introduction to compositional and noncompositional methods*. Cambridge University Press.

[20] S. Schneider, H. Treharne & H. Wehrheim (2010): *A CSP Approach to Control in Event-B*. In: *Integrated Formal Methods 2010, LNCS* 6396, Springer-Verlag, pp. 260–274, doi:10.1007/978-3-642-16265-7_19.