Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciências da Computação

Trabalho de Graduação

Automatic Z Data Refinement

por

André Luís Ribeiro Didier

Orientador: Alexandre Cabral Mota

Recife, 2006

To my parents and my wife

Acknowledgements

I am really thankful to my supervisor, Alexandre, who has been teaching so many things about science and life. I remember when I was still working in other area - as a technician in electronics - a few years ago, and he said that it is not a good idea to accommodate ourselves in a company for so many years. It's true as well as the lesson about always keep learning.

To my mother, Cléa, who has given me so many advices, like "take some rest" or "go watch some movie" or even "stay at home this weekend". She really wants all the best to me.

To my main investor, Luciano, who gave me my first computer when I was fifteen; and then my first job. My father showed me where I would find food: working. He also taught me to love beach when I was able to walk. He told me that life is not easy, but we must live with happiness.

To my wife, Juliana, who has shared with me many important moments, like the graduation on the technical course, the vestibular[1] exams pass and now, the graduation on Computer Science. She helped me on the decision to choose the Computer Science as the profession to me.

To my grandparents Zezé and Roberto who have always motivated me and do not forget their grandson on their prays.

To my mother in law, Conceição, an example of strength and wisdom.

To my colleagues at Pitang who have been asking me for the graduate diploma so many times. It is a subliminal message to say go ahead and achieve it. Special thanks to Malu, Aninha, Kika, Daniela Pompílio, Pedro and Geovani.

[1] Exam to enter an undergraduate course

Contents

Abstract

The goal of Formal Methods is to deliver high quality software using models and precise language. Z and CSP are languages used for this task. Z deals with states, operations, data types and system properties. CSP is used to modelling concurrency. The idea behind a data refinement is to achieve code refining the specifications. The proof obligations required by the data refinement is always a tedious task that requires a work by hand and the current tools requires strong human intervention. The tools to increase productivity on the use of Formal Methods have been demanded. On this work is presented an approach to automate the proof obligations for a data refinement for both forwards and backwards simulations by capturing the Z specifications as CSP processes and use the FDR model-checker to execute a deadlock-freedom test. It is presented a case study for each of the simulations to illustrate the automation processes.

List of Symbols and Abbreviations

Abbreviation	Description	Definition
CSP	Communicating Sequential Processes	page 19
CSP_M	The machine-readable version of CSP	page 26
FDR	Failures-Divergence Refinement	page 25

List of Figures

List of Tables

Chapter 1

Introduction

Software is everywhere. Unfortunately software has bugs. Formal methods is one alternative to tackle bugs. The idea is to have a specification and refine it to achieve code. Refinement needs theorem proving which is interactive and difficult to deal with. Our proposal is an automatic refinement approach. This proposal is based on model checking which is an automatic technique to prove properties of finite-state systems. We show in this work that we can formulate the proof obligations of data refinement in terms of a deadlock-freedom test. This is accomplished by rewrite the predicates of data refinement as relational operations on sets.

Formal methods is an area of Software Engineering whose goal is to deliver high quality software systems. By high quality software we mean software without bugs (or with a minimum). Formal methods use precise (formal) languages to describe the specification of these systems. There are some languages[1] with this intent, eg., EVES, B, Z and CSP. Normally they are based on predicate logic and algebra.

The Z language, presented in Chapter 2, is used to describe operations, their data types, states and properties. It defines previous conditions and post conditions of their execution, as we show in Section 2.2.

An execution of a program is associated with a process or a set of processes in a computer. In the first case, the process behaviour is quite predictable if its code is correctly written. But with two processes or more, if they communicate, the behaviour is difficult to predict. The CSP language, presented in Chapter 3, is used to describe processes interactions and their communications (see Section 3.1 and 3.2).

In a software development process we begin with a description in natural language and finish with a program in machine language. We should take some steps

[1] http://vl.fmnet.info/

between these extremes. For example, we may describe a state of a program as set of values, but the implementation should use a sequence. With this semantics over states, each of these steps are called *data refinement* . The secure transition between a more abstract state to a more concrete one is achieved using formal methods languages, such as Z. The Z data refinement is presented in Section 2.3.

There are two ways to validate a data refinement: by a forwards simulation and/or by a backward simulation. A simulation is a representation of one state by another. The forwards simulation relates an abstract state to a concrete one and backwards simulation relates these states in the opposite direction, relating a concrete state to an abstract one. To validate a data refinement we must prove some rules. The simulations and their rules are described in Section 2.3.1 and an example of each is presented is Section 2.3.2.

The mathematical proofs needed for a data refinement in Z is a tedious hand-work task. Everyone who has ever been put in contact with the technique has argued it. It would be usefull if we could automate the proofs. There are some works to do this, as we can see in [Bol05, BDW99]. In Z, there are some tools with graphical interface like Possum[2], which is used to verify or validate the Z specifications, but requires strong human intervention.

The FDR model-checker is used to validate the failures or divergences that may occur in a CSP specification. The failures occurs when a process refuses some events after a sequence of events, as we show in Section 3.1.3, and a divergence is a trace after which the process behaves chaotically. Such behaviour and the definition of divergence are presented in Section 3.1.4. The FDR model-checker can interpret only the machine-readable version of CSP, called CSP_M. We present a brief description and usage of the FDR in Section 3.4 and the description of the CSP_M and the equivalence to the human readable version in Section 3.4.2.

The objective of this work is to use the combination of Z and CSP to analyse the CSP deadlock-freedom property and use the FDR model-checker to automatically prove the validity of a data refinement. We present the CSP process refinements and the deadlock-freedom test in Section 3.3. A case-study for the automation of the proof obligations of the data refinement rules for each of the simulations is presented in Chapter 5. In Chapter 6 we present our conclusions, limitations, further works and related works.

The contributions of this work are:

- Convert operations on relations to CSP_M;
- Translate Z data refinements (for both forwards and backwards simulations) to CSP_M and
- Present a case-study to use the FDR model-checker.

[2]http://citeseer.ist.psu.edu/hazel97possum.html

Chapter 2

Z overview

2.1 Formal methods

Nowadays a sensible work is spent during and after software development in documentation, such as user guides, reference manuals and so forth. In despite of that, there is software with unmet requirements and undesirable properties, exposing data inconsistency. Formal methods have came to achieve full software functionality and prevent reworking, avoiding (bad) surprises. The produced documentation improves correctness in the development process. It is a means of ensuring, mathematically, the software quality.

The purposes of a formal method are: "to add precision, to aid understanding, and to reason the properties of a design" [WD96, p. 2]. But not every project manager expect to spent men work on those techniques; some prefer to run the risks. Costs also must be in mind and our work is intended to reduce them.

2.2 The Z notation

The Z language is a formal language used to specify systems properties and requirements. Almost every operation defines preconditions and postconditions. The language is based upon set theory and predicate logic.

The Z notation defines system states, data input and output, and conditions over them. The operations properties are represented in the schema notation using states, variables and the predicate logic. The schema definition may represent a state change or simply an output calculus, without state change. The Z notation can be found in [Spi89] and in [WD96]. In a higher level, our usage will be restricted to the schema notation and data refinement.

2.2.1 Schema notation

In [WD96, p. 161] is stated that "The schema language is used to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for re-use." If one do not use the schema notation the result may be a huge amount of unintelligible data from which no one could extract information. Also in [WD96, p. 161], is stated that "By identifying and sharing common components, we keep our descriptions both exible and manageable". A common component may be, for instance, a mean operation or a sum of a set of numbers.

The schemas may be defined inline like:

$$S \mathrel{\widehat=} [declaration \mid predicate]$$

Or as a box like:

$$\begin{array}{|l} \hline S \\ declaration \\ \hline predicate \\ \hline \end{array}$$

The declaration part defines variables such as sets - which may represent the system state -, inputs and outputs. The predicate part expresses conditions over which some operation is defined. The following schema may represent the valid dates (extracted from [WD96, p. 153]):

$$Month = \{jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec\}$$

$$\begin{array}{|l} \hline Date \\ month : Month \\ day : 1..31 \\ \hline month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \\ month = feb \Rightarrow day \leq 29 \\ \hline \end{array}$$

Another simple example represents a birthday book (extracted from [Spi89, p. 3]):

$$\begin{array}{|l} \hline BirthdayBook \\ known : \mathbb{P}\, NAME \\ birthday : NAME \pfun DATE \\ \hline known = \mathrm{dom}\, birthday \\ \hline \end{array}$$

The *mathematical toolkit* presented in [Spi89, pp. 86-127] defines operations over sets, relations, functions, sequences and bags. The Table 2.1 shows some examples.

Operator	Description	Example
$\frown$	Sequence concatenation	$\langle a, b\rangle \frown \langle c\rangle = \langle a, b, c\rangle$
$\#$	Sequence length	$\#\langle a, b\rangle = 2$
$\subseteq$	Subset inclusion	$A \subseteq B = \{x \mid x \in A \Rightarrow x \in B\}$
$\cup$	Union	$\{a, b, c\} \cup \{c, d, e\} = \{a, b, c, d, e\}$
dom	Domain	$\mathrm{dom}\, fact = \mathbb{N}$
ran	Range	$\mathrm{ran}\, fact = \mathbb{N}$

Table 2.1: Some math operations used in Z

2.2.2 Schemas as types

In [WD96, p. 152] is shown how schemas may be defined as types. A simple example is shown bellow:

SchemaOne

$a : \mathbb{Z}$

$c : \mathbb{P}\,\mathbb{Z}$

The declaration s : *SchemaOne* introduces a variable s of type *SchemaOne* which means that the schema variables a and c may be accessed as $s.a$ and $s.c$ with types $\mathbb{Z}$ and $\mathbb{P}\,\mathbb{Z}$, respectively

2.2.3 Schema operations

Two schemas may be combined together if they are type compatible. Being *type compatible* means that [Spi89, p. 31] "each variable common to the two has the same type in both of them". The operation results in a new schema with the union of the declarations and the predicate joined with the same operation as the schemas operation. It means that the schema operation must be a logical operation. Thus, the schema operations $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ are valid. The following example [Spi89, p. 32] illustrates an operation:

Given the schemas:

Aleph

$x, y : \mathbb{Z}$

$x < y$

Gimel

$y : \mathbb{Z}$

$z : 1..10$

$y = z * z$

Then, Aleph $\wedge$ Gimel is:

AlephAndGimel

$x, y : \mathbb{Z}$

$z : 1..10$

$x < y \land y = z * z$

Renaming

The renaming operation simply substitutes a variable name within a schema. The general form is $Schema[v^1_{new}/v^1_{old}, \ldots, v^n_{new}/v^n_{old}]$. Example: Given the schema:

S

$a : T_a$

$b : T_b$

The schema $S2 \mathrel{\widehat=} S[c/a, d/b]$ is the schema:

S2

$c : T_a$

$d : T_b$

Decoration

The decoration definition is presented in [WD96, p. 168]. To describe an operation upon a schema state we must have two copies of the state: the state before and the resulting state of performing the operation. The after state is "decorated" with an $'$. Then, an operation may be represented as:

Operation

State

$State'$

$in? : T_{in}$

$out! : T_{out}$

---

predicate

The definition also describes two schemas $\Delta State$ and $\Xi State$:

$\Delta State$

State

$State'$

$\Xi State$

State

$State'$

$State' = State$

The schema $\Xi State$ may be used in those operations that do a calculus and just outputs a value, for example.

Composition

Another usefull operator is the schema composition operator. It represents intermediate states that are the "end" of the first operand and the "beginning" of the second operand. We present the operator definition from [Spi89, p. 78]:

> The schema $S \fatsemi T$ has all the components of S and T , except for the components x of S and x of T , where x is a matching state variable. If $State : Exp$ is a schema containing just the matching state variables, then $S \fatsemi T$ is defined as:
>
> $$\begin{array}{l} \exists\, State'' \bullet \\ \quad (\exists\, State' \bullet [S;\ State'' \mid \theta State' = \theta State'']) \land \\ \quad (\exists\, State \bullet [T;\ State'' \mid \theta State = \theta State'']) \end{array}$$

There is a similar operator for relations. Given two relations $R : X \leftrightarrow Y$ and $S : Y \leftrightarrow Z$, the composition $R \fatsemi S : X \leftrightarrow Z$ is defined to be:

$$x \mapsto z \in R \fatsemi S \Leftrightarrow \exists\, y : Y \bullet x \mapsto y \in R \land y \mapsto z \in S$$

The composition $U \fatsemi S$, where $U : Y$ is an unary relation and $S : Y \leftrightarrow Z$, may be defined as follows:

$$z \in U \fatsemi S \Leftrightarrow \exists\, y : Y \bullet y \in U \land y \mapsto z \in S$$

Hiding

The hiding operation is defined to be an insertion of an existential operator ($\exists$) in the schema predicate. In formulas [WD96, p. 181]:
Given the schema:

S

$$\begin{array}{|l} a : A \\ b : B \\ \hline P \end{array}$$

The schema $S2 \mathrel{\widehat=} S \backslash \{b\}$ is:

$S2$

$$\begin{array}{|l} a : A \\ \hline \exists\, b : B \bullet P \end{array}$$

Parallel

The parallel operator relates two relations in the following way [WD96, p. 251]:

$$[W, X, Y, Z]$$
$$\_ \parallel \_ : (W \leftrightarrow Y) \times (X \leftrightarrow Z)$$
$$\forall \rho : W \leftrightarrow Y;\ \sigma : X \leftrightarrow Z;\ x : X;\ y : Y;\ w : W;\ z : Z \bullet$$
$$(w, x) \mapsto (y, z) \in \rho \parallel \sigma \Leftrightarrow w \mapsto y \in \rho \land x \mapsto z \in \sigma$$

2.2.4 The identity relation

The special identity relation relates an element of a set to itself. It is defined in [WD96, p. 89]:

$$\mathrm{id}\, X == \{x : X \bullet x \mapsto x\}$$

2.2.5 Preconditions

The precondition is defined to be the condition over which an operation is guaranteed to occur. Outside the precondition nothing can be said about it. We present the formal definition found in [WD96, p. 203]: Given an *Operation*, with state $State : Exp$ and a list of outputs *outputs*, its precondition is:

$$\mathrm{pre}\ Operation = \exists\, State' \bullet Operation \backslash outputs$$

The formal definition can also be found in [Spi89, p. 77].

In [WD96, p. 206] is presented a recipe to calculate the precondition. We present the example. Given the schemas:

S

$$a : \mathbb{N}$$
$$b : \mathbb{N}$$
$$a \neq b$$

T

$$S$$
$$c : \mathbb{N}$$
$$b \neq c$$

And the operation schema:

Increment

$$\Delta T$$
$$in? : \mathbb{N}$$
$$out! : \mathbb{N}$$
$$a' = a + in?$$
$$b' = b$$
$$c' = c$$
$$out! = c$$

The recipe gives us, three steps:

The first step, divides the declaration in three parts: Before, After and Mixed, representing the state before, the state after and a mixed definition:

$$Before = \{in? : \mathbb{N}\}$$
$$After = \{out! : \mathbb{N}\}$$
$$Mixed = \{\Delta T\}$$

The second step, slices the mixed definition into the state before and state after:

$$Before = \{in? : \mathbb{N},\ T\}$$
$$After = \{out! : \mathbb{N},\ T'\}$$
$$Mixed = \{\}$$

And then the last step defines the schema:

pre *Increment*

$$T$$
$$in? : \mathbb{N}$$

$$\exists\, out! : \mathbb{N},\ T' \bullet$$
$$a' = a + in?$$
$$b' = b$$
$$c' = c$$
$$out! = c$$

The resulting schema can be simplified, considering the invariants and definitions. The simplification steps can be found in [WD96] following the recipe. We present the final simplification:

pre *Increment*

$$T$$
$$in? : \mathbb{N}$$

$$a + in? \neq b$$

2.3 Data Refinement in Z

The Oxford Advanced Learner's Dictionary (6th edition) defines:

> **2 refinement of something** a thing that is an improvement on an earlier, similar thing.

The formal methods are all about specifying software. Data refinement is the process of removing undefinedess. The initial specifications may be as abstract as necessary to easily capture the requirements and properties of the software being designed, using the most convenient data types. Although, if the specifications are closer to the software implementation then the possibility of unexpected problems - such as performance - are avoided. In [WD96, p. 234] strengths that "several refinement steps may be performed, each removing another degree of uncertainty, until the specification approaches executable program code". Spivey in [Spi89, p. 137] says that data refinement is a step that relates an *abstract* data type to a *concrete* one, and that a concrete data type is, in fact, another abstract data type, considering the state space and schemas. Then, the refined operation (concrete) must be similar - as the refinement word definition says - to the original one (abstract) and, as formal methods mathematically specifies a system (or part of it), data refinement is a continuous process that removes undefinedess of such specifications.

To prove a data refinement, is necessary to define a link between the two states data types (abstract and concrete). Such link is sometimes called a *retrieve relation* [WD96, p. 258] or an *abstraction schema* [Spi89, p. 137]. A data refinement proof is a *simulation* [WD96, p. 244].

2.3.1 Simulations

If we imagine an execution of a program with a data type $State_A$ and then imagine the same program with a data type $State_C$, its execution should have the same number of steps (the number of operations), inputs and outputs. A simulation relates both program data types in a forward manner ($State_A \leftrightarrow State_C$) or in a backward manner ($State_C \leftrightarrow State_A$).

Before we present the simulations kinds, we must define the Z specification of such program in terms of its states, initialisations and variables. The abstract definition is subscripted as "A" and the concrete one is subscripted as "C".

Definitions:

$$State_A \mathrel{\widehat=} [a_1 : T_{a_1};\ \ldots;\ a_{k_a} : T_{a_{k_a}} \mid Inv_A(a_1, \ldots, a_{k_a})]$$
$$State_C \mathrel{\widehat=} [c_1 : T_{c_1};\ \ldots;\ c_{k_c} : T_{c_{k_c}} \mid Inv_C(c_1, \ldots, c_{k_c})]$$

These definitions of state are similar to the definitions found in [MS01].

The system states may have invariants which are represented as the predicates Inv_A and Inv_C. Each state may have any number of variables with their own types, captured with the definitions $a_n : T_{a_n}$ and $c_m : T_{c_m}$, with n varying from 1 to k_a and m from 1 to k_c. All these variables may be accessed from the declaration of a variable s_A of type $State_A$ and a variable s_C of type $State_C$ as presented in Section 2.2.2. From now on, we will use these variables.

Every system should have an initialisation schema and n operations which are generically obtained as:
Initialisations:

$$Sch_{Init_A} \mathrel{\widehat=} [s'_A : State'_A \mid Init_A(s'_A)]$$
$$Sch_{Init_C} \mathrel{\widehat=} [s'_C : State'_C \mid Init_C(s'_C)]$$

where $Init_A$ and $Init_C$ are unary relations defined as follows:

$$Init_A : State_A$$
$$Init_C : State_C$$

Operations:

$$Sch_{Op^i_A} \mathrel{\widehat=} [\Delta State_A;\ in? : T_{in};\ out! : T_{out} \mid Op^i_A((s_A, in?), (s'_A, out!))]$$
$$Sch_{Op^i_C} \mathrel{\widehat=} [\Delta State_C;\ in? : T_{in};\ out! : T_{out} \mid Op^i_C((s_C, in?), (s'_C, out!))]$$

where Op^i_A and Op^i_C are binary relations defined as follows:

$$Op^i_A : (State_A, T_{in}) \leftrightarrow (State_A, T_{out})$$
$$Op^i_C : (State_C, T_{in}) \leftrightarrow (State_C, T_{out})$$

the types T_{in} and T_{out} are the types of the input and output variables, that can be written as tuples. The index i varies from 1 to n, $\Delta State_A$ and $\Delta State_C$ are declared as:

$\Delta State_A$
$s_A : State_A$
$s'_A : State'_A$

$\Delta State_C$
$s_C : State_C$
$s'_C : State'_C$

Using the schema definitions with relations ($Init_A$, $Init_C$, Op^i_A and Op^i_C) is an important detail to achieve the simulations theorems as subset inclusion. Such representation is found in [WD96] and we present it in the following sections.

Forwards simulation

Naming the relation between the two datatypes $State_A$ and $State_C$ as *Retrieve*, the answer of the following questions - found in [WD96, p. 244] - are the rules which one must prove be a theorem to achieve a correct (forward) data refinement:

- Can any initialisation of $State_C$ be matched by taking an initialisation of $State_A$ and following it with *Retrieve*?
- Can any finalisation of $State_C$ be matched by preceding it with *Retrieve* and comparing it with the finalisation of $State_A$?
- Can any operation in $State_C$ be matched by the corresponding operation in $State_A$?

The retrieve relation takes an abstract state to a concrete state in the forwards simulation. Its predicate may be represented as the $Link_F$ relation:

$$Retrieve \mathrel{\widehat=} [s_A : State_A;\ s_C : State_C \mid Link_F(s_A, s_C)]$$

and

$$Link_F : State_A \leftrightarrow State_C$$

The forwards simulation rules of data refinement are shown in [WD96, p. 260] and in [Spi89, p. 138]. We present them, using our notation:

F-init: $\forall State'_C \bullet Sch_{Init_C} \Rightarrow \exists State'_A \bullet Sch_{Init_A} \land Retrieve'$
F-applic: $\forall State_A;\ State_C \bullet \text{pre } Sch_{Op^i_A} \land Retrieve \Rightarrow \text{pre } Sch_{Op^i_C}$
F-correct: $\forall State_A;\ State_C;\ State'_C \bullet \text{pre } Sch_{Op^i_A} \land Retrieve \land Sch_{Op^i_C} \Rightarrow \exists State'_A \bullet Sch_{Op^i_A} \land Retrieve'$

In [WD96] the applicability is part of the correctness theorem, separating them improves understanding.

The forwards simulation is chosen when the concrete operations are weakened considering the abstract operations. We will see that the backwards simulation is applied to those refinements that the concrete operations are stronger then the abstract operations. Being weaker means that the preconditions has more elements in the relation and being stronger means that the preconditions has less elements in the relation. The backwards meaning of strength can be found in [DH03, p. 4].

Backwards simulation

The idea in a backwards simulation is as if [WD96, p. 270] "the abstract system can simulate the concrete one by being able to anticipate its actions".

The rules can be found in [WD96, p. 270] and we write them in our notation:

B-init: $\forall State'_A;\ State'_C \bullet Sch_{Init_C} \land Retrieve' \Rightarrow Sch_{Init_A}$
B-applic: $\forall State_C \bullet (\forall State_A \bullet Retrieve \Rightarrow \text{pre } Sch_{Op^i_A}) \Rightarrow \text{pre } Sch_{Op^i_C}$
B-correct: $\forall State_C \bullet (\forall State_A \bullet Retrieve \Rightarrow \text{pre } Sch_{Op^i_A}) \Rightarrow \forall State'_A;\ State'_C \bullet Sch_{Op^i_C} \land Retrieve' \Rightarrow \exists State_A \bullet Retrieve \land Sch_{Op^i_A}$

with the retrieve relation defined as:

$$Retrieve \mathrel{\widehat=} [s_A : State_A;\ s_C : State_C \mid Link_B(s_C, s_A)]$$

and:

$$Link_B : State_C \leftrightarrow State_A$$

The retrieve in the backwards simulation relates a concrete state to an abstract one. The relations $Init_A$, $Init_C$, Op^i_A and Op^i_C are the same defined in the forwards simulation.

2.3.2 Simulations examples

Now that we have shown the theory of a data refinement, we present some examples. They are used in Chapter 5 to illustrate the presented theory.

We use two examples, one for the forwards simulation and one for the backwards simulation. We use the predicates version of the rules to be proved. In the automation process, the refinement is proved in the set relations version of the rules and then, we compare the results. To use the set relations we must use a finite state-space and the representation of infinite states as finite ones without leaks on semantics is not the focus of our work, but we say it is possible and we recommend the read of [Laz99] and the state-explosion avoidance with infinite state-spaces presented in [FMS02].

Forwards simulation

The program in this example must find the average of some natural numbers. The complete definition can be found in [WD96, p. 263]. The specification consists of two operations: an operation *AEnter* to enter a number to the data set and an operation *AMean* to calculate the arithmetic mean of the numbers entered so far. The state of the program is modelled using a sequence of natural numbers to represent the data set. The initial state and the operations are presented:

$$AMemory \mathrel{\widehat=} [s : \operatorname{seq} \mathbb{N}]$$
$$AMemoryInit \mathrel{\widehat=} [AMemory' \mid s' = \langle\rangle]$$

AEnter

$$\Delta AMemory$$
$$n? : \mathbb{N}$$
$$s' = s \frown \langle n? \rangle$$

AMean

$$\Xi AMemory$$
$$m! : \mathbb{R}$$
$$s \neq \langle\rangle$$
$$m! = \frac{\sum_{i=0}^{\#s}(s\ i)}{\#s}$$

Using the recipe to calculate the preconditions, we can write:

$$\text{pre } AEnter \mathrel{\widehat=} true$$
$$\text{pre } AMean \mathrel{\widehat=} s \neq \langle\rangle$$

Now we present the design of the arithmetic mean calculus:

$$CMemory \mathrel{\widehat=} [sum : \mathbb{N}, size : \mathbb{N}]$$
$$CMemoryInit \mathrel{\widehat=} [CMemory' \mid sum' = 0 \land size' = 0]$$

CEnter

$\Delta CMemory$
$n? : \mathbb{N}$

$sum' = sum + n?$
$size' = size + 1$

CMean

$\Xi CMemory$
$m! : \mathbb{R}$

$size \neq 0$
$m! = \dfrac{sum}{size}$

And the preconditions are:

pre $CEnter \mathrel{\widehat=} true$
pre $CMean \mathrel{\widehat=} size \neq 0$

The forwards simulation retrieve can be written as:

SumSizeRetrieve

$AMemory$
$CMemory$

$sum = \sum_{i=0}^{\#s}(s\ i)$
$size = \#s$

The rules to be proved are:
Initialisation:

$\forall\, CMemory' \bullet CMemoryInit \Rightarrow (\exists\, AMemory' \bullet$
$AMemoryInit \land SumSizeRetrieve')$

Applicability and **correctness** for the "**enter**" operation:

$\forall\, AMemory;\ CMemory;\ n? : \mathbb{N} \bullet \text{pre } AEnter \land$
$SumSizeRetrieve \Rightarrow \text{pre } CEnter$
$\forall\, AMemory;\ CMemory;\ CMemory';\ n? : \mathbb{N} \bullet$
$\text{pre } AEnter \land SumSizeRetrieve \land CEnter \Rightarrow$
$(\exists\, AMemory' \bullet AEnter \land SumSizeRetrieve')$

And finally, **applicability** and **correctness** for the "**mean**" operation:

$\forall\, AMemory;\ CMemory;\ m! : \mathbb{R} \bullet \text{pre } AMean \land$
$SumSizeRetrieve \Rightarrow \text{pre } CMean$
$\forall\, AMemory;\ CMemory;\ CMemory';\ m! : \mathbb{R} \bullet$
$\text{pre } AMean \land SumSizeRetrieve \land CMean \Rightarrow$
$(\exists\, AMemory' \bullet AMean \land SumSizeRetrieve')$

It's possible to assert mathematically that all rules are theorems.

Backwards simulation

We now present a simplified version of the example of backwards simulation found in [WD96, p. 274]. The original example has some other focus that are not interesting to this work: it is also an operation refinement. We reduced the number of operations on the design model to fit in a pure data refinement. Anyway, the operation refinement in this example is not its main matter; the author would like only to show it to treat in further chapters. The operation "vend" is our focus.

The program is a simplified version of the software of a vending machine which dispenses drinks in response to three-digits codes typed in by its users. The specification does not consider the digits being typed one-by-one, i.e., the digits are represented as an atomic sequence. The free-type *Status* is used to indicate an operation in progress or a success or failure on the vending operation. The *Digits* are numbers between 0 and 9 and $\text{seq}_3[X]$ is the sequence of Xs whose length is exactly 3.

$$Status ::= yes \mid no$$
$$Digit == 0..9$$
$$\text{seq}_3[X] == \{s : \text{seq}\, X \mid \#s = 3\}$$

The state of the specification has two boolean variables to indicate an operation in progress or a success or failure in the vending operation. The initialisation asserts a not busy status:

$$VMSpec \mathrel{\widehat=} [busy, vend : Status]$$

$$VMSpecInit \mathrel{\widehat=} [VMSpec' \mid busy' = no]$$

The two operations on the specification are *Choose* and *VendSpec*. We present them and their preconditions:

Choose

$\Delta VMSpec$
$i? : \text{seq}_3\, Digit$

$busy = no$
$busy' = yes$

VendSpec

$\Delta VMSpec$
$o! : Status$

$busy' = no$
$o! = vend$

pre $Choose \mathrel{\widehat=} busy = no$

pre $VendSpec \mathrel{\widehat=} true$

Note that the *vend* value on both *Choose* and *VendSpec* operations is nondeterministically chosen.

At design level, the digits are entered separately and all we actually need to record is the number of digits entered. When the first digit is entered, then the machine is busy. It continues accepting digits until the max size (3) is reached. The state and its initialisation are shown:

$$VMDesign \mathrel{\widehat=} [digits : 0..3]$$
$$VMDesignInit \mathrel{\widehat=} [VMDesign' \mid digits' = 0]$$

Now we present the operations on the design version and their preconditions. As we said before, they are modified from the original example. The modification is very simple; the original version has two operations to enter digits, one to enter the first digit (*FirstPunch*) and another the enter the second and the third digits (*NextPunch*). In our example, we omitted the operation *NextPunch*. This kind of refinement is an *operation refinement* and is not our focus. Omitting the refinement of the *NextPunch* operation turns the original refinement into a plain (backwards) data refinement as we will show.

FirstPunch

$\Delta VMDesign$
$d? : Digit$

$digits = 0$
$digits' = 1$

pre $FirstPunch \mathrel{\widehat=} digits = 0$

VendDesign

$\Delta VMDesign$
$o! : Status$

$digits' = 0$

pre $VendDesign \mathrel{\widehat=} true$

The $o!$ variable can have any value: *yes* or *no*.

The retrieve schema shown below can be used for both forwards and backwards simulation:

RetrieveVM

$VMSpec$
$VMDesign$

$busy = no \Leftrightarrow digits = 0$

Observing the forwards simulation rules of this specification, we can conclude that it fails on the correctness of the refinement of the operation "vend":

$$\forall\, VMSpec;\ VMDesign;\ VMDesign' \bullet \text{pre } VendSpec \land RetrieveVM \land VendDesign \Rightarrow \exists\, VMSpec' \bullet VendSpec \land RetrieveVM'$$

To prove it fails, the following must be *false* for some value:

$$\begin{array}{l} \forall\, busy, vend : Status;\ digits, digits' : 0..3;\ o! : Status \bullet \\ \quad busy = no \Leftrightarrow digits = 0 \land digits' = 0 \Rightarrow \\ \quad \exists\, busy', vend' : Status \bullet busy' = no \land \\ \quad o! = vend \land busy' = no \Leftrightarrow digits' = 0 \end{array}$$

Which is clearly false because nothing guarantees that the expression $o! = vend$ is true for all $vend$ and for all $o!$. The rules for the backwards simulation are:

Initialisation:

$$\forall\, VMSpec';\ VMDesign' \bullet VMDesignInit \land RetrieveVM' \Rightarrow VMSpecInit$$

Applicability and **correctness** for the "**choose**" operation:

$$\begin{array}{l} \forall\, VMDesign \bullet (\forall\, VMSpec \bullet RetrieveVM \Rightarrow \text{pre } ChooseSpec) \Rightarrow \\ \quad \text{pre } ChooseDesign \\ \forall\, VMDesign \bullet (\forall\, VMSpec \bullet RetrieveVM \Rightarrow \text{pre } ChooseSpec) \Rightarrow \\ \quad \forall\, VMSpec';\ VMDesign' \bullet ChooseDesign \land RetrieveVM' \Rightarrow \\ \quad \exists\, VMSpec \bullet RetrieveVM \land ChooseSpec \end{array}$$

And finally, **applicability** and **correctness** for the "**vend**" operation:

$$\begin{array}{l} \forall\, VMDesign \bullet (\forall\, VMSpec \bullet RetrieveVM \Rightarrow \text{pre } VendSpec) \Rightarrow \\ \quad \text{pre } VendDesign \\ \forall\, VMDesign \bullet (\forall\, VMSpec \bullet RetrieveVM \Rightarrow \text{pre } VendSpec) \Rightarrow \\ \quad \forall\, VMSpec';\ VMDesign' \bullet VendDesign \land RetrieveVM' \Rightarrow \\ \quad \exists\, VMSpec \bullet RetrieveVM \land VendSpec \end{array}$$

Chapter 3

CSP Overview

We may define CSP as [RHB97, p. 1]:

> CSP is a notation for describing concurrent systems (i.e., ones where there is more than one process existing at a time) whose component processes interact with each other by communication. Simultaneously, CSP is a collection of mathematical models and reasoning methods which help us understand and use this notation.

We use the same set of conventions presented in [Hoa85, p. 2]:

1. Words in lower-case letters denote distinct events, e.g.,

   $coin, choc, in2p, out1p$

   and so also do the letters, a, b, c, d, e.

2. Words in upper-case letters denote specific defined processes, e.g.,

   $VMS-$the simple vending machine
   $VMC-$the complex vending machine

   and the letters P, Q, R (occurring in laws) stand for arbitrary processes.

3. The letters x, y, z are variables denoting events.

4. The letters A, B, C stand for sets of events.

5. The letters X, Y are variables denoting processes.

6. The alphabet of process P is denoted αP , e.g.,

   $\alpha VMS = \{coin, choc\}$
   $\alpha VMC = \{in1p, in2p, small, large, out1p\}$

A channel has the same behaviour of an event, but communicates a value. It can be an input or output channel. The following example resumes the definition [RHB97, p. 18]:

$$COPY = left?x : T \rightarrow right!x \rightarrow COPY$$

The behaviour of process COPY is copying the value x, of type T, from left to right. So, the process contains two operations, left and right and its alphabet is $\{left, right\}$.

In CSP, the exactly timing is not relevant. The synchronisation between processes is one "catch" of the language. It is based on events. The events describes a process alphabet. Thus, a process is valid under a sequence of events and two processes are synchronised in one event (or a set of events) if it is in both alphabets. For example, we can define an alphabet $\alpha P = \{up, down\}$ and define a movement process P as:

$$P = (up \rightarrow down \rightarrow up \rightarrow STOP)$$

The process should make a move up, then down, then up again. The special process STOP never communicates, i.e., it indicates an execution termination. A simple synchronisation example is shown bellow, given two processes H and V supposed to be in a parallel execution:

$$V = up \rightarrow match \rightarrow down \rightarrow STOP$$
$$H = left \rightarrow match \rightarrow right \rightarrow STOP$$

They start their execution, synchronizes on the event "match" and then continues their executions.

Processes may have a recursive or looping behaviour. We present this definition in Section 3.2.2.

All the processes interaction and operators can be found in [Hoa85] and in [RHB97]. We present the relevant CSP processes semantics (in Section 3.1) and operations (in Section 3.2) to the Z data refinement automation.

In Section 3.3 and 3.4 we present the process refinements and the FDR - a model-check tool - its commands and the machine-readable version of CSP, which the tool can interpret.

3.1 Processes semantics

In this section we present some semantics of the processes. Some special processes have obvious semantics that are based on their definitions (e.g. $STOP$).

3.1.1 Traces

A trace is a sequence of events that a process engages. It is the "sequential record of the process behaviour up to some moment in time" [Hoa85, p. 5]. A trace to a CLK process might be:

$$\langle tick, tick, tick \rangle$$

The complete set of all possible traces of a process P is the function $traces(P)$. Some processes have special traces:

$$traces(STOP) = \{\langle\rangle\}$$

There are some operations over traces sequences that can be found in [Hoa85, p. 21]. On the failures definition we will need to use one trace operation: "after" (symbol: /). This binary operation relates a process to a trace and is defined to be a process behaviour after all the events of the trace [Hoa85, p. 32]. Some examples for the process $P = a \rightarrow b \rightarrow P$:

$P/\langle a \rangle = b \rightarrow P$ (P after a is b then P)
$P/\langle a, b \rangle = P$ (P after a and b is P)

3.1.2 Refusals

A refusal is a set of events that might cause a deadlock if offered to a process in its first step. The set of all refusals of a process P is denoted by $refusals(P)$ and the following law [Hoa85, p. 89] clarifies the refusal idea:

$$refusals(c \rightarrow P) = \{X \mid X \in (\alpha P - \{c\})\}$$

The law states that a process $c \rightarrow P$ refuses every set that does not contains the event c. The special process $STOP$ has the following refusals set:

$$refusals({STOP_A}^1) = \mathbb{P}\, A$$

If a process has a refusal set X then this process also refuses $Y \subseteq X$.

3.1.3 Failures

The failures of a process is a set of pairs. Such pair is formed from a trace and a refusal in the following way [Hoa85, p. 109]:

$$failures(P) = \{(s, X) \mid s \in traces(P) \wedge X \in refusals(P/s)\}$$

[1] The expression $STOP_A$ means the process $STOP$ under the alphabet A.

Some properties on the failures set

There are some properties in [RHB97, p. 214] that we might use for the data refinement:

Given a non-empty set of processes S:

$$failures(\sqcap S) = \bigcup\{failures(P) \mid P \in S\}$$

Given a boolean condition b:

$$failures(P \not< b \not> Q) = \begin{cases} failures(P), \text{ if } b \text{ evaluates to true} \\ failures(Q), \text{ if } b \text{ evaluates to false} \end{cases}$$

3.1.4 Divergences

Before we explain what is a divergence, we ought to define the *CHAOS* process. It is the most nondeterministic process, i.e., it is the [Hoa85, p. 106] "most unpredictable and most uncontrollable of processes. There is nothing that it might not do; furthermore, there is nothing that it might not refuse to do". In formulas:

$$traces(CHAOS_A) = A^*$$
$$refusals(CHAOS_A) = \mathbb{P}\, A$$

A *divergence* is a trace after which the process behaves cahotically. The set of all divergences is defined [Hoa85, p. 107]:

$$divergences(P) = \{s \mid s \in traces(P) \land (P/s) = CHAOS_{\alpha P}\}$$

Hoare also presents some laws:

$$s \in divergences(P) \land X \subseteq \alpha P \Rightarrow X \in refusals(P/s)$$
$$divergences(STOP) = \{\}$$
$$divergences(CHAOS_A) = A^*$$
$$divergences(P \sqcap Q) = divergences(P) \cup divergences(Q)$$

3.2 CSP operators

CSP is a rich language to model concurrency. There many operators and we present in this section some of them that are useful to model-check Z data refinements.

3.2.1 Prefix

The prefix is the simpler operation involving a process. It defines a process engagement on an event and then the process behaviour is like the suffixed process. The prefix is the operation "then":

$x \rightarrow P$ (read: x then P)

This operation takes a process on the right and an event on the left. The following definitions are syntactically incorrect [Hoa85, p. 4]:

$P \rightarrow Q$
$x \rightarrow y$

The last definition should be written as:

$x \rightarrow y \rightarrow STOP$

3.2.2 Recursion

Recursion in CSP is the ability of a process to enter a loop behaviour. A very simple recursive process is a clock:

$\alpha CLK = \{tick\}$
$CLK = tick \rightarrow CLK$

A valid sequence of events in a clock would be:

$CLK = tick \rightarrow CLK$
$CLK = tick \rightarrow (tick \rightarrow CLK)$ (Substitution)
$CLK = tick \rightarrow (tick \rightarrow (tick \rightarrow CLK))$ (Substitution again)

And its behaviour:

$tick \rightarrow tick \rightarrow tick...$

3.2.3 Conditional choice

The conditional choice operator is a CSP definition for the traditional if-then-else operator. Its definition is [Hoa85, p. 168]:

$P \not< b \not> Q$ (if b then P else Q)

The guard operator `b & P` [RHB97, p. 534] is a shorthand to:

$P \not< b \not> STOP$

It's only available on the machine-readable version of CSP.

3.2.4 Choice

There are three types of choice (between processes) in the CSP notation:

1. Choice: |
2. Internal Choice: $\sqcap$
3. General Choice (or external choice): $\Box$

The *choice* operator represents a process that can behave like either process depending of the first event that happens. If the event is the same in both options, the choice is nondeterministic.

The *internal choice* - also known as a "nondeterministic or" [Hoa85, p. 82] - is a binary operator that denotes a process that behaves like either of the processes in a nondeterministic way without the knowledge of the external environment. Thus the process P may behave as Q or R in:

$P = Q \sqcap R$, then
$P = Q$ or
$P = R$

The *general choice*, which generalizes the *internal choice* and the *choice* operators has the following behaviour [Hoa85, p. 86]:

$$a \rightarrow P \,\Box\, b \rightarrow Q = a \rightarrow P \mid b \rightarrow Q \text{ (if } a \neq b)$$
$$a \rightarrow P \,\Box\, b \rightarrow Q = a \rightarrow P \sqcap b \rightarrow Q \text{ (if } a = b)$$

The internal choice of two processes may result in a refusal if exists at least one refusal in one of them [Hoa85, p. 90]:

$$\mathit{refusals}(P \sqcap Q) = \mathit{refusals}(P) \cup \mathit{refusals}(Q)$$

The deadlock-freedom property (see Section 3.4.1) would pass only if the refusals set is empty.

3.2.5 Indexation

The use of index is as usual as the math indexing. Some examples can be found in [Hoa85, p. 190]. We present one:

$$\sqcap_{i \leq n} P_i = (P_0 \sqcap P_1 \sqcap \ldots \sqcap P_n)$$

3.3 Process refinement

A process refinement ($P \sqsubseteq Q$) means that the refined process P does all operations that the resulting process Q does and maybe some more operations. There are three kinds of refinement:

- Traces refinement [RHB97, p. 46]:

$$P \sqsubseteq_T Q \equiv traces(Q) \subseteq traces(P)$$

- Failures refinement:

$$P \sqsubseteq_F Q \equiv traces(Q) \subseteq traces(P) \land failures(Q) \subseteq failures(P)$$

  In words [RHB97, p. 95], "Q can neither accept an event nor refuse one unless P does".

- Failures-divergences refinement [RHB97, p. 97]:

$$P \sqsubseteq_{FD} Q \equiv P \sqsubseteq Q \equiv failures_{\perp}(Q) \subseteq failures_{\perp}(P) \land \\ divergences(Q) \subseteq divergences(P)$$

  where

$$failures_{\perp}(P) = failures(P) \cup \{(s, X) \mid s \in divergences(P)\}$$

3.3.1 Deadlock-freedom

The deadlock-freedom property is satisfied when a process P can never refuse a set of events in its alphabet, i.e., when there is always something it can do. It can be done either as a failure or failures-divergences refinement. In formulas it means that [RHB97, p. 98]:

$$\forall\, s.(s, \Sigma) \notin failures(P)$$

It's well known that model-checking a failures-divergences refinement can be slower than the failures refinement. Seems plausible that [RHB97, p. 99] "In general we often know that processes are divergence-free for independent reasons." We can conclude that for our data refinement process we only need to check its deadlock freedom property using the failures refinement because we know it does not diverge based on some properties of the $\sqcap$ operator and the divergences of $STOP$ (i.e., $STOP$ never diverges).

3.4 FDR

The FDR is a tool to model-check state machines specified in the CSP language. It directly supports three refinement methods [FDR]. To automate the data refinement we use the failures method.

Operation	CSP	CSP_M
Prefix	$a \rightarrow P$	`a -> P`
Input prefix	$a?x \rightarrow P$	`a?x -> P`
Output prefix	$a!x \rightarrow P$	`a!x -> P`
Conditional choice	$P \not< b \not> Q$	`if b then P else Q`
Guard	$P \not< b \not> STOP$	`b & P`
Internal choice	$P \sqcap Q$	`P |~| Q`
Indexing	$\sqcap_{i \in A} P_i$	`|~| i:A @ P(i)`
Local declaration	$P(f(s))$	`let s' = f(s) within P(s')`
Subset inclusion	$A \subseteq B$	`A <= B`

Table 3.1: Equivalence between CSP and CSP_M

3.4.1 FDR deadlock-freedom command

The command is available on one of FDR's tab (see Figure 3.1) and in its process context menu. The FDR tries to find out a failure upon a process definition and if so, indicates the trace, case else, it asserts the deadlock-freedom property. The seek process is based on expanding all possible states.

3.4.2 The machine readable version of CSP

The machine readable version of CSP is available in [RHB97, pp. 519-539]. It is denoted as CSP_M and the Table 3.1 presents the equivalence to the human readable version of CSP that are relevant to this work. The machine readable version can be loaded into the FDR model-checker.

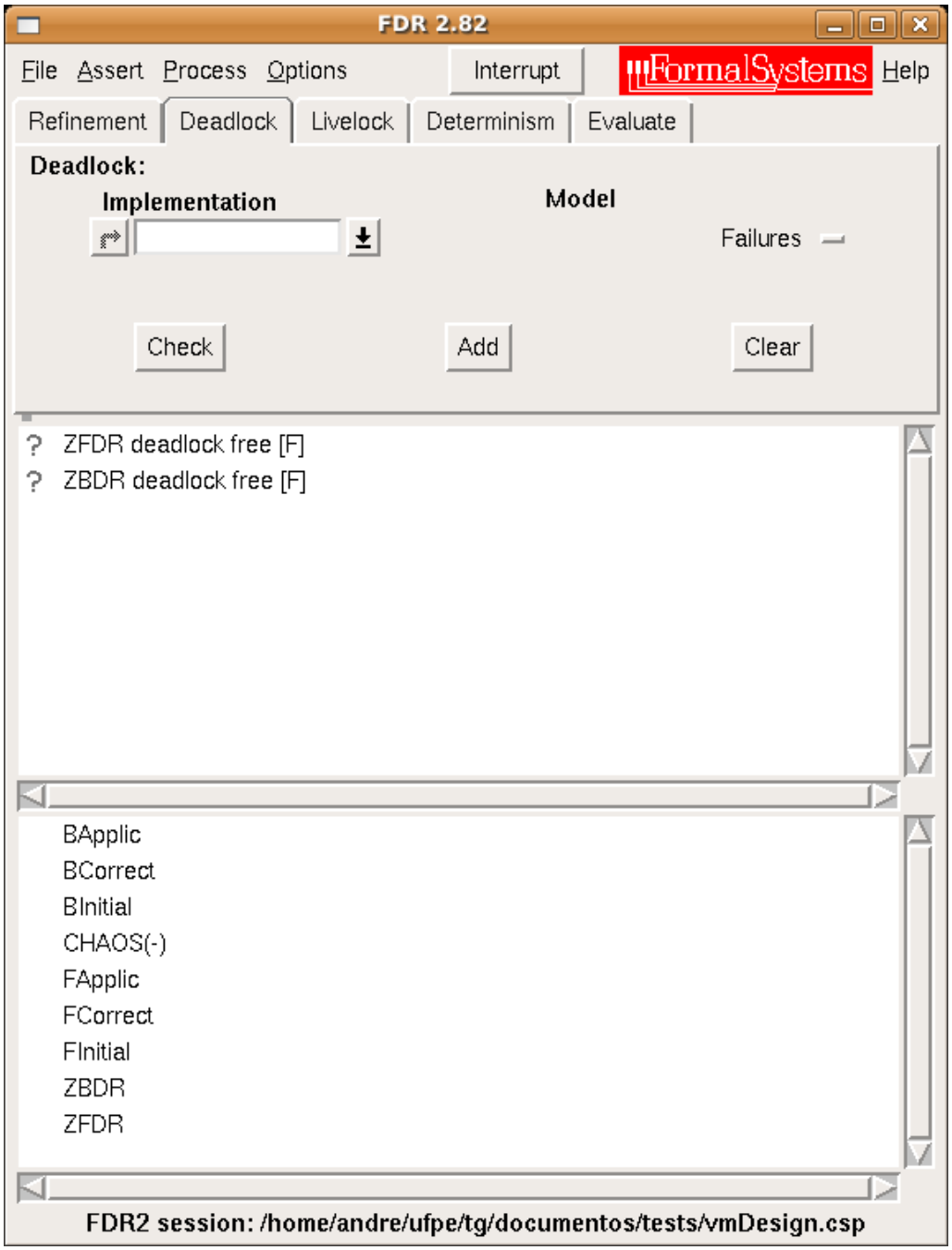

Figure 3.1: FDR deadlock tab

Chapter 4

Capturing Z data refinement through deadlock analysis

In this chapter we present how it is possible to automate the Z data refinement proof obligations. In Section 4.1 we present the CSP_M processes to validate the Z data refinement for both forwards and backwards simulations.

There are some works to semantically join together CSP and Z specifications (see [MS01] and its references). In this work, we use a simpler specification, because there is only the need to check the deadlock freedom, not considering the operations execution themselves. We are focused on translating Z to CSP_M to model-check data refinements. In [MS01] schemas are captured as functions and we will use a similar approach.

4.1 Z data refinement

The Z data refinement proof obligations can be proved as a subset inclusion of relations [WD96, p. 255]. They are summarized and written in our notation in Tables 4.1 and 4.2. We now present how these rules can be translated into CSP_M, indicating how each operator is translated into CSP_M, and how a process can be defined to check these rules and if the refinement is invalid, indicate where the rules fails.

Observing the rules of both simulations we can summarize the Z relations operators used. They are:

- $\fatsemi$: Relational composition;
- $\|$: Parallel composition;
- id: The identity relation;

F-init	$Init_C \subseteq Init_A \fatsemi Link_F$
F-applic	$\text{ran}((\text{dom}\ Op^i_A) \lhd (Link_F \parallel \text{id}\ T_{in})) \subseteq \text{dom}\ Op^i_C$
F-correct	$(\text{dom}\ Op^i_A) \lhd (Link_F \parallel \text{id}\ T_{in}) \fatsemi Op^i_C \subseteq Op^i_A \fatsemi (Link_F \parallel \text{id}\ T_{out})$

Table 4.1: Forwards simulation rules as subset inclusion

B-init	$Init_C \fatsemi Link_B \subseteq Init_A$
B-applic	$\overline{\text{dom}\ Op^i_C} \subseteq \text{dom}((Link_B \parallel \text{id}\ T_{in}) \rhd (\text{dom}\ Op^i_A))$
B-correct	$\text{dom}((Link_B \parallel \text{id}\ T_{in}) \rhd (\text{dom}\ Op^i_A)) \lhd Op^i_C \fatsemi (Link_B \parallel \text{id}\ T_{out}) \subseteq (Link_B \parallel \text{id}\ T_{in}) \fatsemi Op^i_A$

Table 4.2: Backwards simulation rules as subset inclusion

- dom: Domain operator;
- ran: Range operator;
- $\lhd$: Domain restriction operator;
- $\rhd$: Range subtraction operator and
- $\lhd$: Domain subtraction operator.

4.2 Z data refinement process

To model-check a Z data refinement we ought define the behaviour of the CSP process. To aid us to this task, we present some questions:

- If any of the theorems fails what is the process behaviour?
- Case else, if noone fails, what is the process behaviour?

In the first case, if any of the rules fails, the process must have a failure, indicating a deadlock. In the second case, the process should have no failure. Based on the semantics of the process, presented in Section 3.1, a generic Z data refinement process may be defined as:

```
ZDR = Initial |~| Applic |~| Correct
```

where `Initial`, `Applic` and `Correct` are processes capturing the initialisation, applicability and correctness rules of the corresponding data refinement simulation. Using this definition we can analyse the deadlock behaviour in the following way: if one of the processes behaves like `STOP`, then the process `ZDR` has a failure,

thus indicating a deadlock. We will define in the following sections the processes in a way that we can track back the events that causes a deadlock so that we can find out which rule has made the data refinement invalid.

4.3 Translating Z relational operators to CSP_M

We now present how each of the operators used in the simulation rules can be translated into CSP_M definitions.

In the translation process, the relations and the operators becomes sets. The operators also becomes sets, but they are parametrized.

4.3.1 Relational Composition

Given two binary relations $R : X \leftrightarrow Y$ and $S : Y \leftrightarrow Z$, the result of $R \fatsemi S$ may be represented as:

```
R = { (x,y) | ... }
S = { (y,z) | ... }
comp(R,S) = { (x,z) | (x,y1)<-R, (y2,z)<-S, y1==y2 }
```

In the case of an unary relation $U : Y$ and a binary relation $S : Y \leftrightarrow Z$, the result of $U \fatsemi S$ may be represented as:

```
U = { y | ...}
S = { (y,z) | ... }
comp_un(U,S) = { z | y1<-U,(y2,z)<-S,y1==y2 }
```

4.3.2 Parallel composition

Given two relations $R : W \leftrightarrow Y$ and $S : X \leftrightarrow Z$, the result of $R \parallel S$ may be represented as:

```
R = { (w,y) | ... }
S = { (x,z) | ... }
prll(R,S) = { ((w,x),(y,z)) | (w,y)<-R, (x,z)<-S }
```

The idea here is to rearrange the variables in a way that the parameters are properly applied to the corresponding relations.

4.3.3 The identity relation

The identity relation may be defined as follows:

```
id(T) = { (t,t) | t<-T }
```

4.3.4 The domain operator

For a relation $R : X \leftrightarrow Y$, the domain may be defined in the following way:

```
R = { (x,y) | ... }
dom(R) = { x | (x,y)<-R }
```

4.3.5 The range operator

Similarly to the domain definition, the range definition can be defined as:

```
R = { (x,y) | ... }
ran(R) = { y | (x,y)<-R }
```

4.3.6 Domain restriction operator

Given a relation $R : X \leftrightarrow Y$ and a set $A \subseteq X$, the result of $A \lhd R$ may be represented as:

```
R = { (x,y) | ... }
dres(A,R) = { (x,y) | (x,y)<-R, member(x,A) }
```

4.3.7 Domain subtraction operator

Given a relation $r : X \leftrightarrow Y$ and a set $A \subseteq X$, the result of $A \lhd\!\!\!- r$ may be represented as:

```
R = { (x,y) | ... }
ndres(A,R) = { (x,y) | (x,y)<-R, not member(x,A) }
```

4.3.8 Range subtraction operator

The translation for the range subtraction $r \rhd B$ of a relation $r : X \leftrightarrow Y$ and a set $B \subseteq Y$ would be:

```
R = { (x,y) | ... }
nrres(R,B) = { (x,y) | (x,y)<-R, not member(y,B) }
```

4.4 Forwards simulation process

We now present the processes in CSP_M that represents the rules of a forwards simulation.

4.4.1 Initialisation

The initialisation rule

$$Init_C \subseteq Init_A \fatsemi Link_F$$

can be translated into `Finit`

```
Finit = FHypInit <= FConsInit
FHypInit = InitC
FConsInit = comp_un(InitA,LinkF)
```

4.4.2 Applicability

The applicability rule

$$\text{ran}((\text{dom } Op^i_A) \lhd (Link_F \parallel \text{id } T_{in})) \subseteq \text{dom } Op^i_C$$

may be translated into `Fapplic` in:

```
Fapplic(op) = FHypApplic(op) <= FConsApplic(op)
FHypApplic(op) = ran(dres(dom(schA(op)), prll(LinkF,id(Tin))))
FConsApplic(op) = dom(schC(op))
```

4.4.3 Correctness

The correctness rule

$$(\text{dom } Op^i_A) \lhd (Link_F \parallel \text{id } T_{in}) \fatsemi Op^i_C \subseteq Op^i_A \fatsemi (Link_F \parallel \text{id } T_{out})$$

can be translated into `Fcorrect` in:

```
Fcorrect(op) = FHypCorrect(op) <= FConsCorrect(op)
FHypCorrect(op) = comp(dres(dom(schA(op)),prll(LinkF,id(Tin))),
    schC(op))
FConsCorrect(op) = comp(schA(op),prll(LinkF,id(Tout)))
```

The `op` variable represents the current operation being evaluated. It is of type `OP` which must be defined as a datatype representing all the operations. The `schA` and `schC` are the relations that represents the schemas. The indexation is based on the `op` variable. `InitA` and `InitC` defines the values of the initialisation, as a relation. And finally the `LinkF` relation, representing the retrieve schema.

As we have shown in Section 4.2, a generic data refinement process is given by:

```
ZDR = Initial |~| Applic |~| Correct
```

Then the forwards simulation of a data refinement is defined to be:

```
ZFDR = FInitial |~| FApplic |~| FCorrect
```

The `FInitial`, `FApplic` and `FCorrect` should give us informations about the validity of the data refinement, so we should define the following channels:

```
channel initOk, initErr
channel applicOk, applicErr: OP
channel correctOk, correctErr: OP
```

The channels for the applicability and correctness should output the operation that has been either successfully refined or marked as invalid. Considering these prerequisites, we define the forwards simulation rules processes:

```
FInitial = if Finit then initOk -> FInitial
               else initErr -> STOP
FApplic = |~| op: OP @ if Fapplic(op)
               then applicOk!op -> FApplic
               else applicErr!op -> STOP
FCorrect = |~| op: OP @ if Fcorrect(op)
               then correctOk!op -> FCorrect
               else correctErr!op -> STOP
```

As we have shown in Chapter 3, for the operators $\sqcap$ and the conditional choice, if one of the conditions `Finit`, `Fapplic` or `Fcorrect` fails for one value, then the main process `ZFDR` may fail the deadlock freedom test. The FDR model-checker may indicate such refusal with the corresponding "Err" suffixed channel and the output "op" value.

4.5 Backwards simulation process

In this section we present a `ZBDR` process that represents the backwards simulation process. The definition is similiar to the forwards simulation in the previous Section.

4.5.1 Initialisation

The initialisation rule

$$Init_C \fatsemi Link_B \subseteq Init_A$$

becomes `Binit` in:

```
Binit = BHypInit <= BConsInit
BHypInit = comp_un(InitC,LinkB)
BConsInit = InitA
```

4.5.2 Applicability

The applicability rule

$$\overline{\text{dom } Op_C^i} \subseteq \text{dom}((Link_B \parallel \text{id } T_{in}) \rhd (\text{dom } Op_A^i))$$

becomes `Bapplic` in:

```
Bapplic(op) = BHypApplic(op) <= BConsApplic(op)
BHypApplic(op) = { (sc,in) | sc<-StateC,in<-Tin,
    not member((sc,in), dom(schC(op))) }
BConsApplic(op) = dom(nrres(prll(LinkB,id(Tin)),
    dom(schA(op))))
```

4.5.3 Correctness

The correctness rule

$$\text{dom}((Link_B \parallel \text{id } T_{in}) \rhd (\text{dom } Op_A^i)) \lhd Op_C^i \fatsemi (Link_B \parallel \text{id } T_{out}) \subseteq (Link_B \parallel \text{id } T_{in}) \fatsemi Op_A^i$$

is translated into `Bcorrect` in:

```
Bcorrect(op) = BHypCorrect(op) <= BConsCorrect(op)
BHypCorrect(op) = comp(ndres(BConsApplic(op),schC(op)),
    prll(LinkB,id(Tout)))
BConsCorrect(op) = comp(prll(LinkB,id(Tin)),schA(op))
```

Notice the equivalence of the correctness hypothesis part to the applicability consequence part. The first is abbreviated to reuse the second one.

We now define the main process of the backwards simulation as:

```
ZBDR = BInitial |~| BApplic |~| BCorrect
```

We again must know where the refinement either is successful or fails and we use the same channels defined for the forwards simulation. Then we can define the backwards simulation processes as:

```
BInitial = if Binit then initOk -> BInitial
               else initErr -> STOP
BApplic = |~| op: OP @ if Bapplic(op)
               then applicOk!op -> BApplic
               else applicErr!op -> STOP
BCorrect = |~| op: OP @ if Bcorrect(op)
               then correctOk!op -> BCorrect
               else correctErr!op -> STOP
```

Chapter 5

Case study

To illustrate our data refinement processes we present the automation of the data refinement proofs for the specifications shown in Section 2.3.2.

The conversion recipe to CSP_M is straightforward; we divide it into two parts of enumerated steps:

Part 1: Translate

1. Define the datatypes, constants and functions to be used by the schemas definitions, such as max values, set of sequences creator and so on;
2. Define the States as sets;
3. Define the initialisation schemas as sets which are, in fact, subsets of the States previously defined;
4. Create relations that represent the schemas as sets and
5. Create a retrieve relation over the types of the States sets also as set.

Part 2: Link

1. Define the datatype OP;
2. Define the generic forwards or backward elements to use the relations defined in Part 1. The elements are described in Section 4.1: `StateA`, `StateC`, `InitA`, `InitC`, `Tin`, `Tout`, `schA`, `schC`, `LinkF` and `LinkB`.

5.1 Average calculus: forwards simulation

We present the specification in CSP_M of the example shown in Section 2.3.2. Let's make use of the recipe:

Part 1: Translate

1. Datatypes, constants and functions. An optimization has been made here. We are considering sequences of max size 2 and the natural number from 0 to 2. The representation of infinite sets as finite ones is possible. We recommend the read of [FMS02].

```
maxSize = 2
len = 2
Nat = { 0..len }
Real = { 0..len }

BSeq(values,0) = { <> | el<-values }
BSeq(values,n) = union({<>},{ <el>^s | el<-values,
                s<-BSeq(values,n-1) })

sumOf(<>) = 0
sumOf(<a> ^ s) = a + sumOf(s)
```

2. States

```
AMemory = { s | s<-BSeq(Nat, maxSize) }
CMemory = { (sum, size) | sum<-{ 0..len * maxSize },
                size<-{ 0..maxSize } }
```

3. Initialisation schemas

```
AMemoryInit = { sa | sa<-AMemory, sa == <> } = { <> }
CMemoryInit = { (sum, size) | (sum, size)<-CMemory,
                sum == 0, size == 0 } = { (0,0) }
```

4. Schemas

```
AEnter = { ((s,in),(s',out)) | s<-AMemory,s'<-AMemory,in<-Nat,
    out<-Real,#s < maxSize, (s' == s ^ <in>) }
AMean = { ((s,in),(s,out)) | s<-AMemory,in<-Nat,out<-Real,
    s != <>, out == sumOf(s) / #s}

CEnter = { (((sum,size),in),((sum',size'),out)) |
    (sum,size)<-CMemory,(sum',size')<-CMemory,in<-Nat,
    out<-Real,
    size < maxSize,
    sum' == sum + in, size' == size + 1}
```

```
CMean = { (((sum,size),in),((sum,size),out)) |
    (sum,size)<-CMemory,
    in<-Nat,out<-Real, size != 0, out == sum / size}
```

5. Retrieve

```
Link = { (s, (sum,size)) | s<-AMemory,(sum,size)<-CMemory,
    sum == sumOf(s), size == #s}
```

Now, we present the *link* part:

1. The `OP` datatype

```
datatype OP = Enter|Mean
```

2. Linking the definitions

```
StateA = AMemory
StateC = CMemory
InitA = AMemoryInit
InitC = CMemoryInit
Tin = Nat
Tout = Real
LinkF = Link
LinkB = inv(LinkF)

inv(R) = { (y,x) | (x,y)<-R }

schA(Enter) = AEnter
schA(Mean) = AMean

schC(Enter) = CEnter
schC(Mean) = CMean
```

The complete CSP_M definition is in Appendix A.

5.2 Vending machine: backwards simulation

We now present the CSP_M translation of vending machine specification shown in Section 2.3.2. We again use the recipe:

Part 1: Translate

1. Datatypes, constants and functions

```
datatype Status = yes|no

inputLen = 3

FSeq(values, 1) = { <el> | el<-values }
FSeq(values, size) = { s^<el> | el<-values,
            s<-FSeq(values,size-1) }
```

2. States

```
VMSpec = { (busy, vend) | busy<-Status, vend<-Status }
VMDesign = { digits | digits<-{0..inputLen} }
```

3. Initialisation schemas

```
VMSpecInit = { (busy, vend) | (busy, vend)<-VMSpec,
                busy == no) }
        = { (no, no), (no, yes) }
VMDesignInit = { digits | digits<-VMDesign, digits == 0 }
        = { 0 }
```

4. Schemas

```
Choose = { (((busy,vend),in),((busy',vend'),out)) |
    (busy,vend)<-VMSpec, (busy',vend')<-VMSpec,in<-Tin,
    out<-Tout,
    busy == no, busy' == yes}

VendSpec = { (((busy,vend),in),((busy',vend'),out)) |
    (busy,vend)<-VMSpec, (busy',vend')<-VMSpec,in<-Tin,
    out<-Tout,
    busy' == no, out == vend }

FirstPunch = { ((digits,in),(digits',out)) |
    digits<-VMDesign, digits'<-VMDesign,in<-Tin,
    out<-Tout,
    digits == 0 and digits' == 1 }

VendDesign = { ((digits,in),(digits',out)) |
    digits<-VMDesign, digits'<-VMDesign,in<-Tin,
    out<-Tout,
    digits' == 0 }
```

5. Retrieve

```
LinkVM = { ((busy,vend),digits) | (busy,vend)<-VMSpec,
    digits<-VMDesign,
    (busy == no and digits == 0) or (busy != no and digits != 0)}
```

Part 2: Link

1. The `OP` set. Here the only two operations are defined to be `First` and `Vend`. The original definition has another operation, but we must not consider as we have already shown.

```
datatype OP = First | Vend
```

2. Linking the definitions. We have made an optimization here. The input set is made of only the digits 0 and 1, thus the valid numbers are, for example, 100, 010 and 111, i.e., the sequences $\langle 1,0,0\rangle$, $\langle 0,1,0\rangle$ and $\langle 1,1,1\rangle$.

```
StateA = VMSpec
StateC = VMDesign
InitA = VMSpecInit
InitC = VMDesignInit
Tin = FSeq({0..3}, inputLen)
Tout = Status
LinkF = LinkVM
LinkB = inv(LinkF)

schA(First) = Choose
schA(Vend) = VendSpec

schC(First) = FirstPunch
schC(Vend) = VendDesign
```

The complete CSP_M definition is in Appendix B.

5.3 Results analysis

To check the examples, we executed both in a PC, using the following configuration:

- Processor: Pentium IV, 2.8 GHz
- RAM: 512 MB, 1 GB Swap

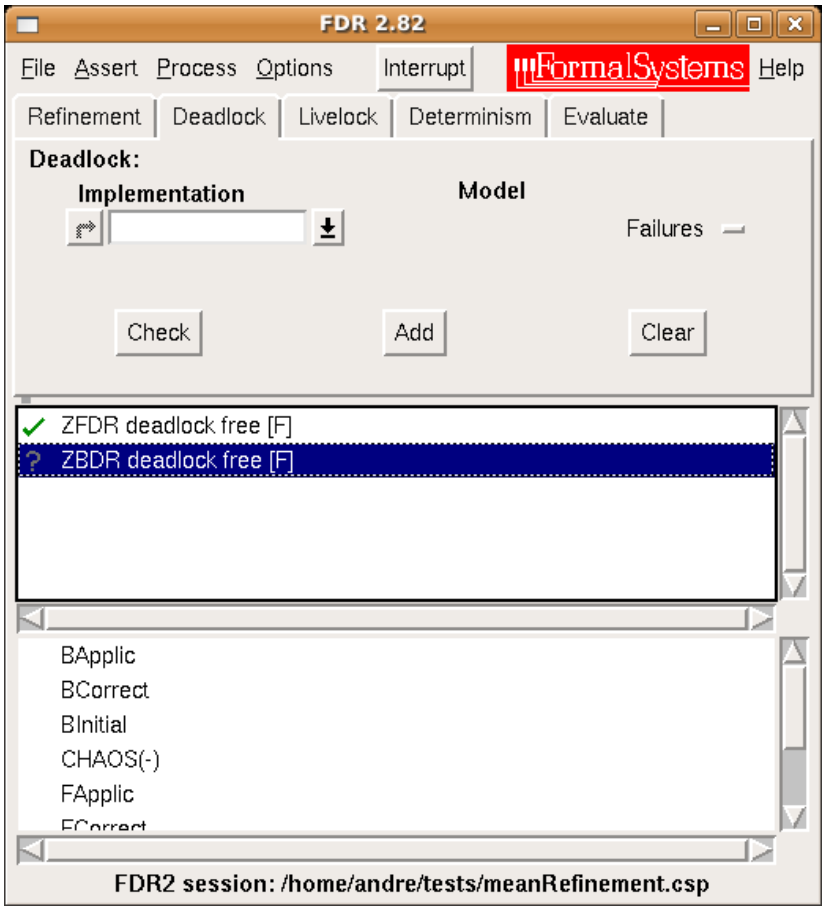

Figure 5.1: Valid average calculus forwards simulation

- OS: Linux Ubuntu
- FDR: version 2.82

We took two measures for each case-study. The executions and their parameters for the Average Calculus example are shown in Table 5.1 and for the Vending Machine example are shown in Table 5.2. The results show that as state enlarges, the FDR takes more time to terminate.

The Average Calculus example terminated successfully in the second execution, proving that the forwards simulation is correct. The screenshot of the FDR is shown in Figure 5.1.

For the second execution in the Vending Machine example we took two validations: firstly trying to prove the forwards simulation rules and secondly, trying to prove the backwards simulation rules. In the first case the FDR successfully pointed out the failure on the "Vend" operation correctness rule, as shown in

Execution #	Parameter	Value

1	`maxSize`
	`len`
		`5`
		`10`
		Total execution time: more than ten minutes - we interrupted the execution

2	`maxSize`
	`len`
		`2`
		`5`
		Total execution time: 2-4 seconds

Table 5.1: Average Calculus Example Executions

Execution #	Parameter	Value
1	`Tin`	`FSeq({0..9}, inputLen)` Total execution time: more than ten minutes - we interrupted the execution
2	`Tin`	`FSeq({0..3}, inputLen)` Total execution time: 2-4 seconds

Table 5.2: Vending Machine Example Executions

Figures 5.2 and 5.3. For the backwards simulation rules, the FDR terminated successfully, proving that the rules are correct as shown in Figure 5.4.

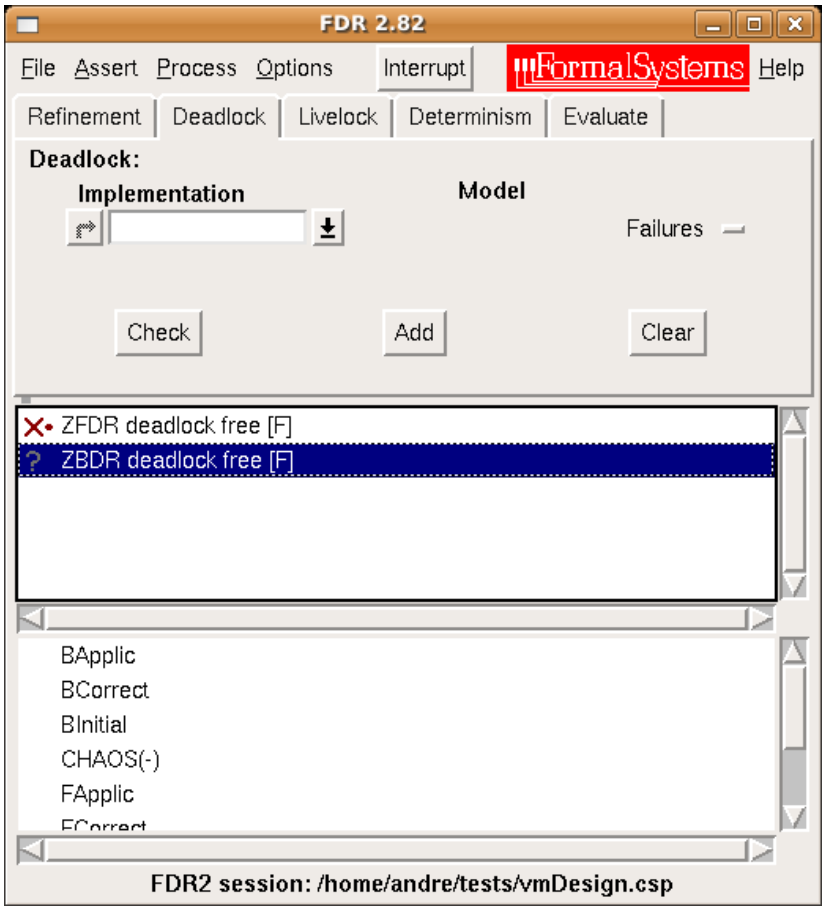

Figure 5.2: Vending machine forwards simulation failure

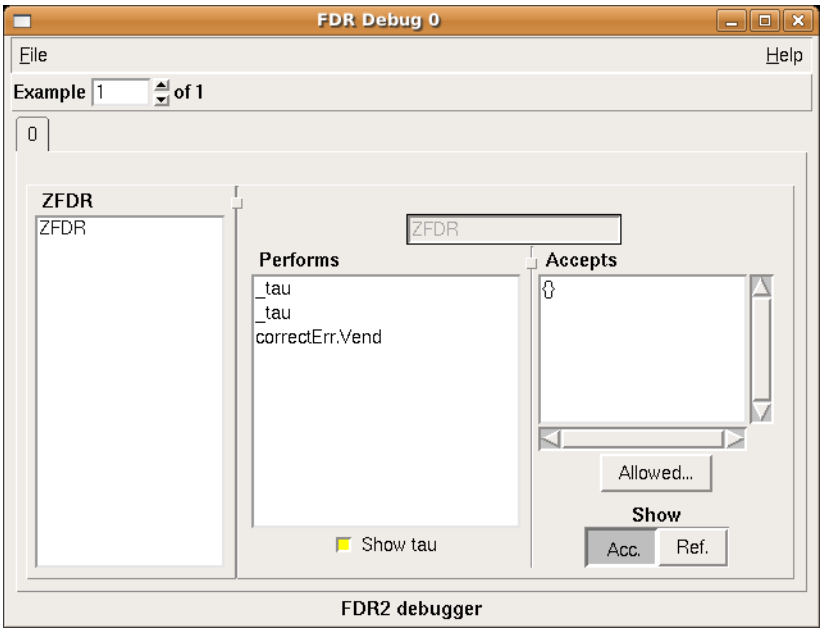

Figure 5.3: Debug screen of the vending machine forwards simulation failure: "correctErr.Vend"

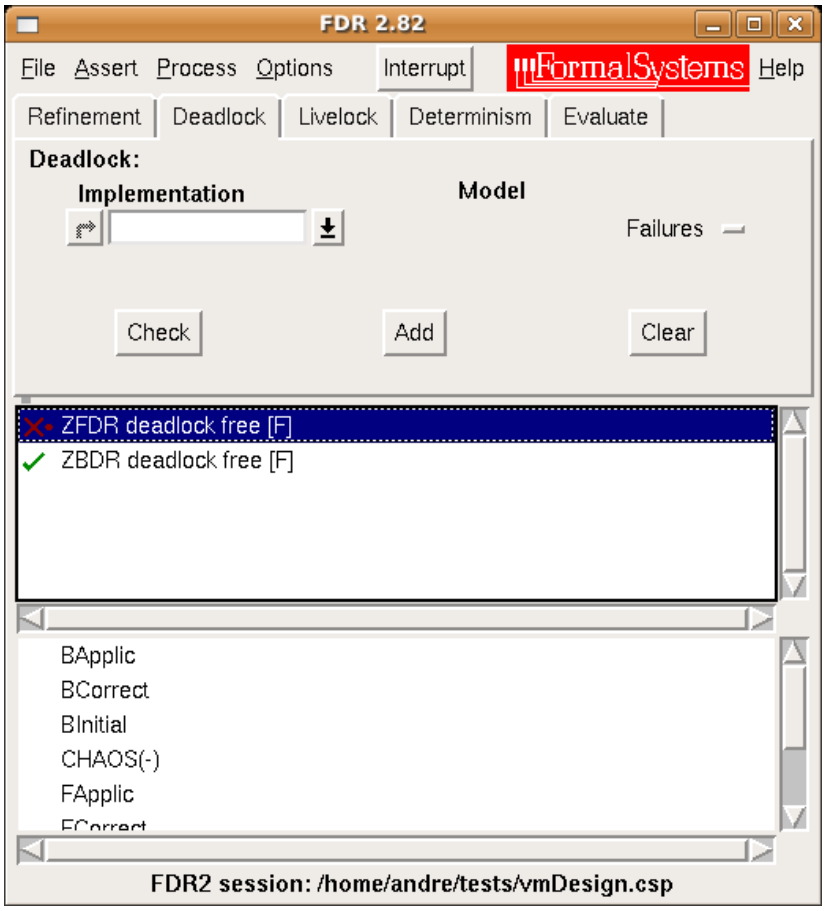

Figure 5.4: Valid vending machine backwards simulation

Chapter 6

Conclusion

In this work we presented an approach to automate the proof obligations concerning a data refinement. The approach consists in translating a Z specification as a CSP one. The proof obligations are captured as relational operators over sets in a convenient CSP process which must be deadlock-free iff the Z data refinement is valid and present some deadlock otherwise.

The events are carefully defined to indicate these situations: the valid refinement traces have all events suffixed with "Ok" and the invalid have exactly one event suffixed with "Err". The rules that involve operations have events with outputs to indicate the operation being evaluated. The events are defined as:

- initOk and initErr, for the validation of the initialisation rules;
- applicOk and applicErr, for the validation of the applicability rules and
- correctOk and correctErr, for the validation of the correctness rules.

The case-study for each of the simulation has illustrated the automation process. Particularly the vending machine example has shown the uncorrectness of the forwards simulation for this case and the validity of the backwards simulation as previewed by [WD96], the authors of the example.

6.1 Limitations

Our approach to model-check Z data refinements has some limitations. The state-explosion is intrinsic to the model-checking approaches and the automation is based on the proof obligations, not in the simulations automation. Thus, if the automatic design of the retrieve relations is possible, it is out of the scope of this work.

6.1.1 State-explosion

To model-check the specifications, the FDR expands the states whenever it is necessary. Thus, we can not use infinite states or even large states in this approach. In [FMS02] is presented a technique to solve this problem. The presented algorithm either avoids a state being expanded twice or avoids an expansion when it finds out an "infinite stable behaviour". This behaviour occurs when a trace is repeated.

6.1.2 Not a fully automated task

Although the proof automation is possible, some tasks may be executed by the designer:

- Convert Z to CSP_M, i.e., convert the Retrieve relations, schemas and state-spaces. It is the same difficulty found in [MS01, p. 69] and
- Concern about finiteness of the states in CSP_M which are dealt, for example, in [FMS02].

6.2 Future works

It is possible and would be usefull if we join the data refinement specification with the dealing of larger or infinite states presented in [FMS02].

A tool to convert from CSP-Z to CSP_M has been presented in [FMS01]. It would be possible to either reuse part of it or extend it to add the data refinement functionality shown in this work.

6.3 Related works

A proposal of data refinement automation is presented in [Bol05]. The approach captures Z schemas and data types as Alloy atoms and types. The simulations rules are captured as functions of the defined state-spaces. The proposal has the same problem of state-explosion and our advantage is that the CSP language is more powerful than Alloy.

Bibliography

[BDW99] Christie Bolton, Jim Davies, and Jim Woodcock. On the refinement and simulation of data types and processes. In *IFM*, pages 273–292, 1999.

[Bol05] Christie Bolton. Using the alloy analyzer to verify data refinement in z. *Electr. Notes Theor. Comput. Sci.*, 137(2):23–44, 2005.

[DH03] M. Deutsch and M. Henson. Four theories for backward simulation data-refinement, 2003.

[FDR] *FDR: User Manual and Tutorial, version 2.28.*

[FMS01] Adalberto Farias, Alexandre Mota, and Augusto Sampaio. From cspz to cspm: A transformational java tool. In *WMF*, pages 1–10, October 2001.

[FMS02] Adalberto Farias, Alexandre Mota, and Augusto Sampaio. Efficient analysis of infinite cspz processes. In *WMF*, pages 113–128, October 2002.

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

[Laz99] R.S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking.* PhD thesis, Oxford University Computing Laboratory, April 1999.

[MS01] A. Mota and A. Sampaio. Model checking cspz: strategy, tool support and industrial application. *Science of Computer Programming*, 2001.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[Spi89] J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[WD96] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Appendices

Appendix A

Average calculus: complete CSP_M specification

The full CSP_M definition for the average calculus data refinement is presented here. The file containing this specification is available. Contact us to receive it.

```
-----------------------------------------------------------------------
------ Definitions
-----------------------------------------------------------------------
maxSize = 2

len = 5

Nat = { 0..len }
Real = { 0..len }

BSeq(values,0) = { <> | el<-values }
BSeq(values,n) = union({<>},{ <el>^s | el<-values,
                                s<-BSeq(values,n-1) })

sumOf(<>) = 0
sumOf(<a> ^ s) = a + sumOf(s)

-----------------------------------------------------------------------
------ State
-----------------------------------------------------------------------
AMemory = { s | s<-BSeq(Nat, maxSize) }
CMemory = { (sum, size) | sum<-{ 0..len * maxSize },
```

```
            size<-{ 0..maxSize } }

AMemoryInit = { <> }
CMemoryInit = { (0,0) }

-----------------------------------------------------------------
------ The retrieve relation and the OP datatype
-----------------------------------------------------------------

Link = { (s, (sum,size)) | s<-AMemory,(sum,size)<-CMemory,
sum == sumOf(s), size == #s}

datatype OP = Enter|Mean

-----------------------------------------------------------------
------ Schemas
-----------------------------------------------------------------
AEnter = { ((s,in),(s',out)) | s<-AMemory,s'<-AMemory,in<-Nat,
    out<-Real,#s < maxSize, (s' == s ^ <in>) }
AMean = { ((s,in),(s,out)) | s<-AMemory,in<-Nat,out<-Real,
s != <>, out == sumOf(s) / #s}


preAEnter(s) = #s < maxSize
preAMean(s) = s != <>


CEnter = { (((sum,size),in),((sum',size'),out)) |
    (sum,size)<-CMemory,(sum',size')<-CMemory,in<-Nat,out<-Real,
    size < maxSize,
sum' == sum + in, size' == size + 1}

CMean = { (((sum,size),in),((sum,size),out)) | (sum,size)<-CMemory,
in<-Nat,out<-Real, size != 0, out == sum / size}

preCEnter(p) = let (sum,size) = p within size < maxSize
preCMean(p) = let (sum,size) = p within size != 0

-----------------------------------------------------------------
------ Z Data Refinement
-----------------------------------------------------------------
```

```
StateA = AMemory
StateC = CMemory
InitA = AMemoryInit
InitC = CMemoryInit
Tin = Nat
Tout = Real
LinkF = Link
LinkB = inv(LinkF)

schA(Enter) = AEnter
schA(Mean) = AMean

schC(Enter) = CEnter
schC(Mean) = CMean

-----------------------------------------------------------------
------ Declaring types, channels and processes: specification
-----------------------------------------------------------------
comp(R,S) = { (x,z) | (x,y1)<-R, (y2,z)<-S, y1==y2 }
comp_un(U,S) = { z | y1<-U,(y2,z)<-S,y1==y2 }
prll(R,S) = { ((w,x),(y,z)) | (w,y)<-R, (x,z)<-S }
id(T) = { (t,t) | t<-T }
dom(R) = { x | (x,y)<-R }
ran(R) = { y | (x,y)<-R }
dres(A,R) = { (x,y) | (x,y)<-R, member(x,A) }
ndres(A,R) = { (x,y) | (x,y)<-R, not member(x,A) }
nrres(R,B) = { (x,y) | (x,y)<-R, not member(y,B) }
inv(R) = { (y,x) | (x,y)<-R }


channel initOk, initErr
channel applicOk, correctOk: OP
channel applicErr, correctErr: OP

-----------------------------------------------------------------
------ Forwards simulation
-----------------------------------------------------------------
ZFDR = FInitial |~| FApplic |~| FCorrect
assert ZFDR :[ deadlock free [F] ]
```

```
FInitial = if Finit then initOk -> FInitial else initErr -> STOP
FApplic = |~| op: OP @ if Fapplic(op)
        then applicOk!op -> FApplic else applicErr!op -> STOP
FCorrect = |~| op: OP @ if Fcorrect(op)
        then correctOk!op -> FCorrect else correctErr!op -> STOP

------------------------------------------------------------------
------ Initialisation
------------------------------------------------------------------
Finit = FHypInit <= FConsInit
FHypInit = InitC
FConsInit = comp_un(InitA,LinkF)
------------------------------------------------------------------
------ Applicability
------------------------------------------------------------------
Fapplic(op) = FHypApplic(op) <= FConsApplic(op)
FHypApplic(op) = ran(dres(dom(schA(op)), prll(LinkF,id(Tin))))
FConsApplic(op) = dom(schC(op))
------------------------------------------------------------------
------ Correctness
------------------------------------------------------------------
Fcorrect(op) = FHypCorrect(op) <= FConsCorrect(op)
FHypCorrect(op) = comp(dres(dom(schA(op)),
prll(LinkF,id(Tin))),schC(op))
FConsCorrect(op) = comp(schA(op),prll(LinkF,id(Tout)))

------------------------------------------------------------------
------ Backwards simulation
------------------------------------------------------------------
ZBDR = BInitial |~| BApplic |~| BCorrect
assert ZBDR :[ deadlock free [F] ]

BInitial = if Binit then initOk -> BInitial else initErr -> STOP
BApplic = |~| op: OP @ if Bapplic(op)
        then applicOk!op -> BApplic else applicErr!op -> STOP
BCorrect = |~| op: OP @ if Bcorrect(op)
        then correctOk!op -> BCorrect else correctErr!op -> STOP
------------------------------------------------------------------
------ Initialisation
------------------------------------------------------------------
Binit = BHypInit <= BConsInit
```

```
BHypInit = comp_un(InitC,LinkB)
BConsInit = InitA

-----------------------------------------------------------------
------ Applicability
-----------------------------------------------------------------
Bapplic(op) = BHypApplic(op) <= BConsApplic(op)
BHypApplic(op) = { (sc,in) | sc<-StateC,in<-Tin,
    not member((sc,in), dom(schC(op))) }
BConsApplic(op) = dom(nrres(prll(LinkB,id(Tin)),
    dom(schA(op))))

-----------------------------------------------------------------
------ Correctness
-----------------------------------------------------------------
Bcorrect(op) = BHypCorrect(op) <= BConsCorrect(op)
BHypCorrect(op) = comp(ndres(BConsApplic(op),schC(op)),
    prll(LinkB,id(Tout)))
BConsCorrect(op) = comp(prll(LinkB,id(Tin)),schA(op))
```

Appendix B

Vending machine: complete CSP_M specification

Here we present the full specification of the data refinement of the vending machine. A file with this specification is also available; please contact us to receive it.

```
datatype Status = yes|no

inputLen = 3


FSeq(values, 1) = { <el> | el<-values }
FSeq(values, size) = { s^<el> | el<-values,
            s<-FSeq(values,size-1) }
-----------------------------------------------------------------
----- Specification
-----------------------------------------------------------------
VMSpec = { (busy, vend) | busy<-Status, vend<-Status }

VMSpecInit = { (no, no), (no, yes) }

Choose = { (((busy,vend),in),((busy',vend'),out)) |
    (busy,vend)<-VMSpec, (busy',vend')<-VMSpec,in<-Tin,out<-Tout,
    busy == no, busy' == yes}

VendSpec = { (((busy,vend),in),((busy',vend'),out)) |
    (busy,vend)<-VMSpec, (busy',vend')<-VMSpec,in<-Tin,out<-Tout,
```

```
    busy' == no, out == vend }


-----------------------------------------------------------------
------ Design
-----------------------------------------------------------------
VMDesign = { digits | digits<-{0..inputLen} }
VMDesignInit = { 0 }

FirstPunch = { ((digits,in),(digits',out)) | digits<-VMDesign,
    digits'<-VMDesign,in<-Tin,out<-Tout,
    digits == 0 and digits' == 1 }

VendDesign = { ((digits,in),(digits',out)) | digits<-VMDesign,
    digits'<-VMDesign,in<-Tin,out<-Tout,
    digits' == 0 }

-----------------------------------------------------------------
------ The retrieve relation and OP datatype
-----------------------------------------------------------------
LinkVM = { ((busy,vend),digits) | (busy,vend)<-VMSpec,
    digits<-VMDesign,
    (busy == no and digits == 0) or (busy != no and digits != 0)}

datatype OP = First | Vend

-----------------------------------------------------------------
------ Z Data Refinement
-----------------------------------------------------------------
StateA = VMSpec
StateC = VMDesign
InitA = VMSpecInit
InitC = VMDesignInit
Tin = FSeq({0..3}, inputLen)
Tout = Status
LinkF = LinkVM
LinkB = inv(LinkF)

schA(First) = Choose
schA(Vend) = VendSpec
```

```
schC(First) = FirstPunch
schC(Vend) = VendDesign

-----------------------------------------------------------------------
------ Declaring types, channels and processes: specification
-----------------------------------------------------------------------
comp(R,S) = { (x,z) | (x,y1)<-R, (y2,z)<-S, y1==y2 }
comp_un(U,S) = { z | y1<-U,(y2,z)<-S,y1==y2 }
prll(R,S) = { ((w,x),(y,z)) | (w,y)<-R, (x,z)<-S }
id(T) = { (t,t) | t<-T }
dom(R) = { x | (x,y)<-R }
ran(R) = { y | (x,y)<-R }
dres(A,R) = { (x,y) | (x,y)<-R, member(x,A) }
ndres(A,R) = { (x,y) | (x,y)<-R, not member(x,A) }
nrres(R,B) = { (x,y) | (x,y)<-R, not member(y,B) }
inv(R) = { (y,x) | (x,y)<-R }

channel initOk, initErr
channel applicOk, correctOk: OP
channel applicErr, correctErr: OP

-----------------------------------------------------------------------
------ Forwards simulation
-----------------------------------------------------------------------
ZFDR = FInitial |~| FApplic |~| FCorrect
assert ZFDR :[ deadlock free [F] ]

FInitial = if Finit then initOk -> FInitial else initErr -> STOP
FApplic = |~| op: OP @ if Fapplic(op)
        then applicOk!op -> FApplic else applicErr!op -> STOP
FCorrect = |~| op: OP @ if Fcorrect(op)
        then correctOk!op -> FCorrect else correctErr!op -> STOP

-----------------------------------------------------------------------
------ Initialisation
-----------------------------------------------------------------------
Finit = FHypInit <= FConsInit
FHypInit = InitC
FConsInit = comp_un(InitA,LinkF)
-----------------------------------------------------------------------
------ Applicability
```

```
-----------------------------------------------------------------
Fapplic(op) = FHypApplic(op) <= FConsApplic(op)
FHypApplic(op) = ran(dres(dom(schA(op)), prll(LinkF,id(Tin))))
FConsApplic(op) = dom(schC(op))
-----------------------------------------------------------------
------ Correctness
-----------------------------------------------------------------
Fcorrect(op) = FHypCorrect(op) <= FConsCorrect(op)
FHypCorrect(op) = comp(dres(dom(schA(op)),
prll(LinkF,id(Tin))),schC(op))
FConsCorrect(op) = comp(schA(op),prll(LinkF,id(Tout)))

-----------------------------------------------------------------
------ Backwards simulation
-----------------------------------------------------------------
ZBDR = BInitial |~| BApplic |~| BCorrect
assert ZBDR :[ deadlock free [F] ]

BInitial = if Binit then initOk -> BInitial else initErr -> STOP
BApplic = |~| op: OP @ if Bapplic(op)
        then applicOk!op -> BApplic else applicErr!op -> STOP
BCorrect = |~| op: OP @ if Bcorrect(op)
        then correctOk!op -> BCorrect else correctErr!op -> STOP
-----------------------------------------------------------------
------ Initialisation
-----------------------------------------------------------------
Binit = BHypInit <= BConsInit
BHypInit = comp_un(InitC,LinkB)
BConsInit = InitA

-----------------------------------------------------------------
------ Applicability
-----------------------------------------------------------------
Bapplic(op) = BHypApplic(op) <= BConsApplic(op)
BHypApplic(op) = { (sc,in) | sc<-StateC,in<-Tin,
    not member((sc,in), dom(schC(op))) }
BConsApplic(op) = dom(nrres(prll(LinkB,id(Tin)),
    dom(schA(op))))

-----------------------------------------------------------------
------ Correctness
```

```
-----------------------------------------------------------------
Bcorrect(op) = BHypCorrect(op) <= BConsCorrect(op)
BHypCorrect(op) = comp(ndres(BConsApplic(op),schC(op)),
    prll(LinkB,id(Tout)))
BConsCorrect(op) = comp(prll(LinkB,id(Tin)),schA(op))
```

Index

Data e assinaturas

Recife, 6 de outubro de 2006

André Luís Ribeiro Didier
(Autor)

Alexandre Cabral Mota
(Orientador)