Automatic Guidance for Refinement Based Formal Methods*

Maria Teresa Llano
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, UK
mtl4@hw.ac.uk

Gudmund Grov
School of Informatics
University of Edinburgh
Edinburgh, UK
ggrov@inf.ed.ac.uk

Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, UK
a.ireland@hw.ac.uk

ABSTRACT

Refinement is a technique used to model systems at different abstraction levels to handle the complexity of large systems. It is used in many different methods, and independent of the method applied, many users find it difficult to identify the correct level of abstraction and steps of refinements. To achieve a correct refinement, the step has to be justified – often by formal proof. Such proofs represent an additional challenge, typically requiring a user to understand the relationship between modelling and reasoning. To address these challenges, we introduce the notion of *refinement plans*, a technique that combines proof and modelling patterns of refinement, with the aim of automatically providing modelling guidance when refinement fails.

Keywords

Formal modelling, correctness by construction, model based refinement, proof planning, Event-B

1. INTRODUCTION

While some argue that the use of formal methods is actually cheaper than conventional software development methods[1], the overhead of formal modelling and verification remains a barrier to the wider adoption of formalism within industry.

Correctness by construction (C-by-C) [14] approaches aim at address this issue by promoting step-wise construction, where a system is incrementally developed by reducing the abstraction level or augmenting the behaviour at each refinement step. Importantly, C-by-C helps identify defects and build formal guarantees at an early stage of a development.

There are two major approaches to achieving provably correct refinement. Firstly, within the *rule-based* approach, refinement proceeds by applying a set of pre-defined and verified rules. The soundness of the rules ensures the correctness of a refinement. An example of this type of techniques is found in [16]. Secondly, there is the *posit-and-prove* approach where a user "posits" a refined model, and then formally justifies the correctness of the refinement. This is typically achieved by discharging a set of proof obligations (POs). Examples of posit-and-prove approaches are VDM [15], Z [18], B [1] and Event-B [2].

The advantage of the rule-based approach is that the burden of proof is kept to a minimal. However, a users freedom to explore the modelling space is restricted by the set of available refinement rules. In contrast, posit-and-prove promotes a high degree of freedom in terms of modelling styles. However this freedom results in a greater burden at the level of proof.

Many problem solving activities rely on patterns, both in terms of to increasing productivity and improving the reliability of products[2]. The use of pattern has also been suggested for formal methods, either as part of the modelling process [3] or in the reasoning process [6]. Here we introduce *refinement plans*, a computational technique that combines modelling and reasoning patterns. A refinement plan can be viewed as a way of describing "good" modelling practice. However, crucially we see the technique as supporting the posit-and-prove approach to refinement, i.e. when failure arises refinement plans provide the basis for automatically generating user guidance. Whilst the approach discussed here is generic with respect to posit-and-prove, we focus on its application within the context of Event-B.

In the following sections we present how we believe refinement plans can be used during the modelling process. Section 2 starts by presenting an example of how refinement is performed in Event-B. Section 3 describes our notion of refinement plans. A concrete example of a refinement plan and how it can be used to analyse flawed designs is presented in section 4. Section 5 gives a brief description of the cur-

*The research reported in this paper is supported by EPSRC grants EP/F037058, EP/E005713 and EP/E035329. Maria Teresa Llano is partially supported by a BAE Studentship.

[1]The argument is that the extra resources required to get it right the first time, more than covers the time spent on testing and reworking (due to bugs found in testing) in conventional disciplines – and the end product is often better! See [4] for more details.

[2]One example is *design patterns* [9] used for the development of object-oriented programs.

rent implementation of our approach, while sections 6 and 7 describe some related and future work, respectively.

2. EVENT-B REFINEMENT BY EXAMPLE

An Event-B specification is structured into *models* and *contexts*. A context describes the static part of a system, e.g. *constants* and their *axioms*, while a model describes the dynamic part. Models are themselves composed of three parts: *variables*, *events* and *invariants*. Variables represent the state of the system, events are guarded actions that update the variables and invariants are constraints on the behaviour described by the events.

Consider a simple example of a logging system that records whether a resource R is allocated or unallocated. Figure 1 shows a fragment of an abstract model for this system in which the state of the resources is handle using sets. The sets *allocated* and *unallocated* are variables that store resources according to their corresponding state. In this way, every time a resource is moved from one set to the other, the logging system should write to the log file with the details of the respective change. This is shown in Figure 1 with the event *allocate*, which registers in the log file the movement of a resource from set *unallocated* to set *allocated*.

Variables: $resources, allocated, unallocated, ...$
Invariants: ...
$allocated \subseteq resources$
$unallocated \subseteq resources$
$allocated \cap unallocated = \emptyset$
Events ...
Event *allocate* $\widehat{=}$
any r
where $r \in unallocated$
then $unallocated := unallocated \setminus \{r\}$
$allocated := allocated \cup \{r\}$
$logContent := logContent \cup \{writeLog(r)\}$
end

Figure 1: Change of state through set manipulation.

Refinement of Event-B specifications can be achieved by adding more detailed information about the behaviour of the system or by introducing incremental changes towards a concrete model that can be implemented. One such refinement for this logging system is to refine the state sets into a function that returns the state of a given resource (which may later be implemented as an array).

Figure 2 shows a refinement of the model shown in Figure 1. This model uses the function *resourceState* in order to update the state of a resource.

In order to prove the correctness of the refinement, the correspondence between the abstract and the concrete specifications must be specified. For instance, in Event-B we have to prove that the guards of the concrete event imply the guards of the abstract event, $G_C \rightarrow G_A$. In order to establish this correspondence, the relation between the states of both models should be stated. In Event-B, this relation is formalised with a *gluing invariant*. In this case, the gluing invariants state the relationship between the sets of the abstract model and the function in the concrete model. The gluing invariants for the logging system are formalised in Figure 3.

Sets: $Status$
Constants: $UNALLOCATED, ALLOCATED$
Axioms:
$partition(Status, ALLOCATED, UNALLOCATED)$

Variables: $resources, resourceState, ...$
Invariants: ...
$resourceState \in resources \rightarrow Status$
Events ...
Event *allocate* $\widehat{=}$
refines *allocate*
any r
where $resourceState(r) = UNALLOCATED$
then $resourceState(r) := ALLOCATED$
$logContent := logContent \cup \{writeLog(r)\}$
end

Figure 2: Change of state through a function.

$allocated = resourceState^{-1}[\{ALLOCATED\}]$
$unallocated = resourceState^{-1}[\{UNALLOCATED\}]$

Figure 3: Gluing invariants.

3. REFINEMENT PLANS

A *refinement plan* represents a common pattern of refinement by combining modelling and proof knowledge. Initially, one of our aims is to give refinement plans a role for explaining refinements. In other words, we want to be able to communicate *why* a particular pattern works. However, ultimately we want to use patterns to explain *how* to overcome a failed refinement. We aim to achieve this by means of a critics style exception handling mechanism, analogous to proof critics [12]. Note that while the analysis of failure and generation of guidance is automatic, the decision as to whether or not to take the guidance on offer will be left to the user.

More concretely a *refinement plan* consists of the following components:

$$\begin{aligned} \textit{refinement plan} &= \textit{refinement patterns} \\ &+ \textit{proof methods} \\ &+ \textit{exceptions} \end{aligned}$$

A *refinement pattern* describes a refinement in terms of one (abstract) specification into another (concrete) specification, including the correspondence between these models. Moreover, particular patterns of refinements, may have particular patterns of reasoning associated with them – these are captured by the *proof methods* [5] component of a refinement plan. Finally, there are cases where a refinement partially matches a refinement pattern or in which a proof

method fails. These cases may indicate some mistakes in the models or for example the need for a lemma in order to discharge a proof. To handle such cases the *exceptions* component of the refinement plan is applied to suggest the appropriate changes.

3.1 Syntax

The syntax used for refinement plans is the following:

1. First an explanation about the purpose of the pattern analysed in the refinement plan is given.
2. Then a template of the modelling part of the refinement pattern is introduced in order to give a more concrete representation of the patterns contained in the abstract and concrete models.
3. Then the schema of the plan is given. This schema is shown in Figure 4.

Method_Name
INPUTS:
PO_SET {*POs*}
MODELS AM, CM
PRECONDITIONS:
1. *precondition ..*
....
I. *precondition ..*
PROOF METHODS:
1. *Proof_Method ..*
....
N. *Proof_Method ..*
EXCEPTIONS:
1. CRITIC: *Critic_Name*(*arguments*)
....
M. CRITIC: ...

Figure 4: Refinement plan schema.

The *INPUTS* of the plan are the set of proof obligations (successful and failed) as well as the abstract and the concrete models. The slot *PRECONDITIONS* is used to specify the preconditions that are used to identify if the models in the input are an instance of the refinement pattern analysed in the plan. The *PROOF METHODS* capture the patterns of reasoning that are associated with the refinement pattern. These methods are used in the case in which a refinement fully matches with the pattern but it has a set of undischarged POs. Proof methods thus provide proof automation. Finally, partial matches and failures of the proof methods are managed by the *EXCEPTIONS* slot. Here all the possible failures that can arise by the evaluation of the preconditions and the proof methods are listed alongside with a corresponding critic that handles the fault. Explanations about the preconditions, the proof methods and the exceptions are also provided.

4. Critics have the schema shown in Figure 5. The inputs slot adds an optional component RP_ELEMENTS, these are the elements found in the refinement plan schema that may be related to the critic. This slot is optional since a critic may be triggered by a proof method instead. The *PRECONDITIONS* determine the applicability of the critic and the *OUTPUT* may suggests either a patch of a proof or a change to the model. Explanations about the preconditions and the suggestions are provided.

Critic_Name
INPUTS:
PO_SET {*POs*}
MODELS AM, CM
RP_ELEMENTS {*Elements*} (*optional*)
PRECONDITIONS:
1. *precondition 1*
....
N. *precondition N*
OUTPUTS:
PATCH *patch description* (*optional*)
GUIDE *guidance description* (*optional*)

Figure 5: Critic schema.

A fragment of the syntax of the language used to express pre-conditions is given in Appendix A (This fragment is later used for example purposes).

4. REFINEMENT PLAN EXAMPLE

In this section we give an example of a refinement plan and how it can be used to analyse failures in a refinement and to provide high-level modelling guidance.

Refinement plan - From sets to a function: This method defines how a particular pattern of set manipulation (abstract) can be refined to a function (concrete). Figure 6 presents the schema of the modelling part of this pattern.

As can be seen in Figure 6, in the abstract model of this refinement some sets are defined in order to determine the states in which a particular type of elements can be placed. These sets are manipulated in the events by moving elements from one set to another set. In the concrete model, on the other hand, these states are determined through the use of a function that returns the current state of an element. The events of this model modify the state of an element by updating the relation defined by the function.

Refinement plan schema: A simple schema for this refinement plan is shown in Figure 7. In order to simplify discussion of the plan, we omit the proof plans component and present only one example of the kind of failures that we intent to handle.

The preconditions of the plan state the following:

P1: This precondition expresses the requirement for a partition in the set of variables of the abstract model. This partition represents the state sets and their type. For instance, in the logging system example this partition is formed by the

Abstract Context	Concrete Context
Sets $OBJECT$	**Sees** $AbstractContext$ **Sets** $Status$ **Constants** $STATE_1, STATE_2, ..., STATE_N$ **Axioms** $Status = \{STATE_1, STATE_2, ..., STATE_N\}$ $partition(Status, \{STATE_1\}, \{STATE_2\}, \{STATE_N\})$
Abstract Model	**Concrete Model**
Variables $object, state_1,$ $state_2, ..., state_N$ **Invariants** $object \subseteq OBJECT$ $state_1 \subseteq object,\ ...,\ state_N \subseteq object$ $disjoint(state_1, state_2, ..., state_N)$ **Events ...** **Event** $AbstractEvent \widehat{=}$ **any** o **where** $o \in state_1 \wedge ...$ **then** $state_1 = state_1 \setminus \{o\} \parallel$ $state_2 = state_2 \cup \{o\} \parallel ...$ **end**	**Refines** $AbstractModel$ **Variables** $object, objectStatus$ **Invariants** $objectStatus \in object \rightarrow Status$ $state_1 = objectStatus^{-1}[\{STATE_1\}]$ ... $state_N = objectStatus^{-1}[\{STATE_N\}]$ **Events ...** **Event** $ConcreteEvent \widehat{=}$ **refines** $AbstractEvent$ **any** o **where** $objectStatus(o) = STATE_1 \wedge ...$ **then** $objectStatus(o) := STATE_2 \parallel ...$ **end**

Figure 6: Pattern template : *From sets to a function*

sets *resources*, *allocated* and *unallocated*; more specifically, *allocated* and *unallocated* partition the set *resources*.

P2: This precondition states that the abstract model must contain events that manipulate the state sets by moving elements from one set to another one. In the logging system example the event *allocate* represents one of these events.

P3: This precondition identifies a partition in the set of constants of the concrete context. The constants in this partition enumerate the state sets found in the abstract model. Constants $ALLOCATED$ and $UNALLOCATED$ partition set *Status* in the concrete context of the logging system example.

P4: In this precondition the function that maps elements to their state in the concrete model is identified. In the case of the logging system, variable *resourceState* represents this function.

P5: This precondition verifies the relation between the state sets in the abstract model and the function of the concrete model. This verification consists in examining if each of the state sets can be obtained from the image of the inverse function evaluated over the corresponding enumeration. This relation between the state sets and the function represents the gluing invariants.

P6: This precondition states that the concrete model must contain events that manipulate the state function by updating the state of an element from a state constant to another state constant. In the logging system example the event *allocate* represents one of these events.

Refinement plan critic: Figure 8 shows an example of a critic used by the sets to function refinement plan. In this critic it is analysed the failure produced by the omission of a gluing invariant.

The first 2 preconditions of the critic validate which preconditions of the refinement plan should have succeeded and which preconditions should have failed for the critic to be applicable. In this case, preconditions 1 to 4 and 6 should succeed and precondition 5 should fail. In our example of the logging system this would mean a failure with the definition of the gluing invariants (For simplicity of discussion we will only analyse the critic in the case in which the gluing invariants are absent from the model, to analyse wrongly defined invariants other critics are applied).

The third precondition asserts the type of failed PO that should appear when the gluing invariant is missing. The failed PO should have the following shape:

$$\begin{array}{l} \textit{Guards of the concrete event}, \\ objectStatus(o) = STATE_i \\ \vdash \\ o \in state_i \end{array}$$

This PO is used to prove that the guards of the concrete event imply the guards of the abstract event. For the event *allocate* in the logging system, the failed PO is:

$$\begin{array}{l} resourceState(r) = UNALLOCATED \\ \vdash \\ r \in unallocated \end{array}$$

If a PO of this type is found, preconditions 4 and 5 evaluate if the failed PO is associated with the pattern. This is done

Sets_2_Function_RP

INPUTS:

PO_SET $\{POs\}$

MODELS AM, CM

PRECONDITIONS:

1. $\exists v_{states}, v_{type}.(v_{states} \subseteq distinctVariables(AM, CM) \wedge$
   $v_{type} \in sharedVariables(AM, CM) \ \wedge \ partition(v_{type}, v_{states}))$
2. $\exists e_{am}.(e_{am} \subseteq E(AM) \ \wedge \ \forall e \in e_{am}, \exists s_1, s_2 \in v_{states}.(setSetsEvent(e, s_1, s_2))$
3. $\exists s_{status}, c_{states}.(s_{status} \in distinctSets(CC, AC) \ \wedge \ c_{states} \subseteq distinctConstants(CC, AC)$
   $\wedge \ partition(s_{status}, c_{states}))$
4. $\exists v_{function}.(v_{function} \in distinctVariables(CM, AM) \ \wedge \ function(v_{function}, v_{type}, s_{status}))$
5. $inverse(v_{function}, v_{states}, c_{states})$
6. $\exists e_{cm}.(e_{cm} \subseteq E(CM) \ \wedge \ \forall e \in e_{cm}, \exists c_1, c_2 \in c_{states}.(setFunEvent(e, v_{function}, c_1, c_2))$

EXCEPTIONS:

1. CRITIC: $Missing\_Gluing\_Critic(\{POs\}, AM, CM, \{v_{function}, v_{states}, c_{states}\})$

Figure 7: Refinement plan schema.

Missing_Gluing_Critic:

INPUTS:

PO_SET $\{POs\}$

MODELS AM, CM

RP_ELEMENTS $\{stateFunction, stateSets, stateConstants\}$

PRECONDITIONS:

1. Preconditions 1-4, 6 succeed.
2. Precondition 5 fails.
3. $\exists Fpo \in \{\langle Evn, Typ, Po\rangle \in POs | failure(Po)\}.$
   $Fpo = (\Delta, stateFunction(x) = Y \vdash x \in \{z\}))$
4. $Y \in stateConstants$
5. $z \in stateSets$
6. $provable\ (\{\mathbf{z}\} = \mathbf{stateFunction^{-1}}[\{\mathbf{Y}\}], \Delta, stateFunction(x) = Y \ \vdash \ x \in \{z\})$

OUTPUTS:

PATCH *empty*

GUIDE $add\_invariant(NewInv, CM)$
   where $NewInv : \{z\} = stateFunction^{-1}[\{Y\}]$

Figure 8: Critic associated to the omission of a gluing invariant.

by checking if the elements used in the PO correspond to elements found by the preconditions of the refinement plan. In the logging system example precondition 4 verifies if *UNALLOCATED* is a member of the set of state constants, while precondition 5 verifies if *unallocated* is a member of the set of state sets.

Finally, precondition 6 asserts if the failed PO can be discharged by adding the gluing invariant to the hypotheses. If the PO is provable after the addition of the hypothesis, the suggestion given to the user is the addition of a gluing invariant to the concrete model. In our example $unallocated = resourceState^{-1}[\{UNALLOCATED\}]$ is added as a hypothesis of the failed PO. This discharges the PO. Then the suggestion given to the user is the addition of the new hypothesis as a gluing invariant in the concrete model.

5. TOOL IMPLEMENTATION

The work reported here is currently being implemented in OCaml. The tool integrates a set of refinement plans that are used in the analysis of refinement-based developments. In this process, the first step is to identify the type of refinements that are applied in a development, this involves pattern matching. If failure arises during this process and a partial match is found (or several partial matches are found), the tool applies the respective critics and automatically generates suggestions that are presented to the user. The decision of applying or not a suggestion is left to the user. Currently our development is independent of any other applications; however, our final goal is to integrate the tool as a plug-in of the Rodin toolset.

6. RELATED WORK

There are several techniques that have been proposed for the enhancement of the posit-and-prove approach. In gen-

eral, these can be seen as incorporating rule-based features. For instance, the BART tool for classical B [17] has incorporated rules in order to automate the process of refining specifications into implementations. In their work, a series of small rules are applied to specific elements of a model (e.g. a rule to transform a variable) and as a result one or more refined models are provided (because the application of one rule may split complex refinements into several machines). However, currently BART rules are not verified, i.e. each application of a rule gives rise to a set of POs. The ZRC refinement calculus for Z [7] introduces a refinement method for Z. However, it s strongly linked with Z Schema calculus, and we will thus not discuss it further.

In [10, 11, 3, 8] refinement patterns are introduced for Event-B. In their methodology, a pattern is represented by a normal Event-B development, where an initial model is refined (over potentially many steps). Where a pattern matches a user model, the same refinement steps, and associated proof steps, are applied to the user development.

The ultimate target of all these approaches is to increase productivity by automating common refinements. We will call this *forward application* of refinement methods. Naturally, due to the application, these approaches have a rather operational description. Although such forward application is possible for our notion of refinement plans, it is not our target. We want to understand the refinement in order to provide modelling guidance from failures. To achieve this an operational description becomes too low level – we need a higher abstraction level which describes the *intention* behind the refinement. Such abstractions will give the flexibility required for failure analysis, but also for other approaches which we will discuss below. By analogy to theorem proving, the work described in [17, 10, 11, 3, 8, 7] is at the tactic level, whilst we are working at the proof planning level.

In [13], the notion of *reasoned modelling critics* was introduced. Reasoned modelling critics extend the concept of proof critics by adding modelling suggestions – i.e. common patterns of proof are used in order to analyse proof failure and then corresponding critics are used in order to suggest a patch to the proof or a change to the model (like a missing invariant). Refinement plans incorporates the reasoned modelling critics into its framework, and the work discussed here can thus be seen as an extension of [13].

7. FUTURE WORK

Besides using refinement plans to analyse failures in user refinements, we envisage a number of other ways they can enhance the application of formal methods, i.e.

Proof automation: In a complete match of a refinement plan the associated proof methods can be applied to (automatically) discharge corresponding proof obligations.

Forward application: This is similar to the use of rules and refinement patterns described in the previous section: the abstract specification matches, and from this the concrete specification (and corresponding relationship is inferred) and the proof obligations are verified using the proof plans.

Backward application: An abstract model (or a generalisation) may be obtained from a more concrete model. In this case, the concrete specification matches and the the abstract specification (and relationship) is inferred. The proof plans can be used to verify that the refinement is indeed correct. Example of such application is when the initial model is too concrete to model certain aspect of the system.

Decomposition: Complex and flawed refinements can be decomposed into more manageable and correct refinements.

8. CONCLUSIONS

We have presented the notion of refinement plans. This is a technique that combines proof and modelling patterns for the analysis of refinement. Particularly, refinement plans aim at providing modelling guidance by automatically analysing flawed specifications that lie just outside a known pattern of refinement. Note that while the analysis of failure and generation of guidance is automatic, the decision as to whether or not to take the guidance on offer will be left to the user. We believe that this technique will exploit synergies that arise between formal modelling and reasoning. Moreover, we also believe that this approach will enable us to turn low-level proof-failures into high-level modelling guidance.

Acknowledgements

Thanks go to Rob Pooley, Alan Bundy and members of the Mathematical Reasoning Group at Edinburgh University for their feedback and encouragement with this work.

9. REFERENCES

[1] J.-R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] J.-R. Abrial. *Modelling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To be published.

[3] J.-R. Abrial and T. S. Hoang. Using Design Patterns in Formal Methods: an Event-B Approach. In *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[4] P. Amey. Correctness by Construction: Better Can Also be Cheaper. *CrossTalk - The Journal of Defense Software Engineering*, Mar. 2002.

[5] A. Bundy. A science of reasoning. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[6] A. Bundy, G. Grov, and C. B. Jones. Learning from experts to aid the automation of proof search. In *AVoCS'09*, CSR-2-2009, pages 229–232. Swansea University, 2009.

[7] A. Cavalcanti and J. Woodcock. Zrc - a refinement calculus for z. *Formal Asp. Comput.*, 10(3):267–289, 1998.

[8] A. Fürst. Design Patterns in Event-B and Their Tool Support. Master's thesis, ETH Zürich, 2009.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] A. Iliasov. Refinement Patterns for Rapid Development of Dependable Systems. In *EFTS '07:*

Proceedings of the 2007 workshop on Engineering Fault Tolerant Systems, page 10, USA, 2007. ACM.

[11] A. Iliasov. *Design Components*. PhD thesis, Newcastle University, July 2008.

[12] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 1992.

[13] A. Ireland, G. Grov, and M. Butler. Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. In *Proceedings of ABZ'10*, number 5977 in Lecture Notes in Computer Science, pages 189–202. Springer, 2010.

[14] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39:93–95, 2006.

[15] C. B. Jones. *Systematic Software Development using VDM (second edition)*. Prentice Hall, 1990.

[16] C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[17] A. Requet. Bart: A tool for automatic refinement. In *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 345. Springer, 2008.

[18] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

APPENDIX

A. REFINEMENT PLANS LANGUAGE

In this appendix we present a fragment of the refinement plans language. This fragment describes the sets, functions and predicates used in the example shown in this paper.

A.1 Definitions

DEFINITION A.1. *An event e is a tuple* (P, G, A) *where* P *is a set of parameters,* G *is a set of guards, and* A *is a set of actions.*

DEFINITION A.2. *A model* M *is a tuple* (V, I, E) *where* V *is a set of variables,* I *is a set of invariants, and* E *is a set of events.*

DEFINITION A.3. *A context* C *is a tuple* (S, C, Ax) *where* S *is the set of carrier sets,* C *is a set of constants, and* Ax *is a set of axioms.*

A.2 Notation

NOTATION A.4. *We write* $P(e)$, $G(e)$ *and* $A(e)$ *for the sets of parameters, guards and actions of an event* e*, respectively.*

NOTATION A.5. *We write* $V(M)$, $I(M)$ *and* $E(M)$ *for the set of variables, invariants and events of a model* M*, respectively.*

NOTATION A.6. *We write* $S(C)$, $C(C)$ *and* $Ax(C)$ *for the set of carrier sets, constants and axioms of a context* C*, respectively.*

NOTATION A.7. *We also write* M_i *for models,* C_i *for contexts,* e_i *for events and* P, Q, X *and* Y *for any set. We omit the subscript whenever possible.*

A.3 Predicates

For each predicate, only the true case is presented; it is always false otherwise.

1. $\mathbf{refines}(\mathbf{M_1}, \mathbf{M_2})$: M_2 is a refinement of M_1.
2. $\mathbf{extends}(\mathbf{C_1}, \mathbf{C_2})$: C_2 is an extension of C_1.
3. $\mathbf{totalFunction}(\mathbf{f}, \mathbf{P}, \mathbf{Q})$: $f \in P \rightarrow Q$.
4. $\mathbf{inverse}(\mathbf{f}, \mathbf{P}, \mathbf{Q})$: This predicate evaluates if there is a relation between sets P and Q through function f, such that:
   - ▷ $f \in X \rightarrow Y$
   - ▷ $P \subseteq X$
   - ▷ $Q \subseteq Y$
   - ▷ $\forall p_i \in P,\ \exists q_i \in Q.(p_i = f^{-1}[\{q_i\}])$.
   - ▷ $\forall p_i \in P, q_1, q_2 \in Q.(f^{-1}[\{q_1\}] = p_i \ \wedge \ f^{-1}[\{q_2\}] = p_i \ \Rightarrow \ q_1 = q_2)$.
5. $\mathbf{setSetsEvent}(\mathbf{e}, \mathbf{P}, \mathbf{Q})$: This predicate evaluates if event e is responsible for moving elements from set P to set Q. This implies:
   - ▷ $P \subseteq X \ \wedge \ Q \subseteq X$.
   - ▷ $\exists p, g, a_1, a_2.(p \in P(e) \wedge g \in G(e) \wedge a_1 \in A(e) \wedge a_2 \in A(e) \ \wedge \ g = (p \in P) \ \wedge \ a_1 = (P := P \setminus p) \ \wedge \ a_2 = (Q := Q \cup p))$.
6. $\mathbf{setFunEvent}(\mathbf{e}, \mathbf{f}, \mathbf{oldValue}, \mathbf{newValue})$: This predicate evaluates if event e modifies function f by changing the codomain of one of its ordered pairs from $oldValue$ to $newValue$. This implies:
   - ▷ $f \in X \rightarrow Y$.
   - ▷ $oldValue \in Y$.
   - ▷ $newValue \in Y$.
   - ▷ $\exists p, g_1, g_2, a.(p \in P(e) \wedge g_1 \in G(e) \wedge g_2 \in G(e) \wedge a \in A(e) \wedge g_1 = (p \in X) \wedge g_2 = (f(p) = oldValue) \wedge a = (f(p) := newValue))$.
7. $\mathbf{partition}(\mathbf{P}, \mathbf{Q})$: This predicate evaluates if the elements of Q partition set P. This implies:
   - ▷ $P = \{q_1 \cup q_2 \cup ... \cup q_n\}$.
   - ▷ $\forall q_i, q_j \in Q.(q_i \neq q_j \Rightarrow q_i \cap q_j = \emptyset)$.

A.4 Functions

1. $\mathbf{distinctVariables}(\mathbf{M_1}, \mathbf{M_2})$: Returns the variables that appear in M_1 but that do not appear in M_2. This implies:
   - ▷ $refines(M_1, M_2) \vee refines(M_2, M_1)$.
   - ▷ $v = \{v_i \mid v_i \in M_1 \ \wedge \ v_i \notin M_2\}$.
2. $\mathbf{sharedVariables}(\mathbf{M_1}, \mathbf{M_2})$: Returns the variables that appear in both model M_1 and model M_2. This implies:
   - ▷ $refines(M_1, M_2) \vee refines(M_2, M_1)$.
   - ▷ $v = \{v_i \mid v_i \in M_1 \ \wedge \ v_i \in M_2\}$.
3. $\mathbf{distinctConstants}(\mathbf{C_1}, \mathbf{C_2})$: Returns the constants that appear in context C_1 but that do not appear in context C_2. This implies:
   - ▷ $extends(C_1, C_2) \vee extends(C_2, C_1)$.
   - ▷ $c = \{c_i \mid c_i \in C_1 \ \wedge \ c_i \notin C_2\}$.
4. $\mathbf{distinctSets}(\mathbf{C_1}, \mathbf{C_2})$: Returns the carrier sets that appear in context C_1 but that do not appear in context C_2. This implies:
   - ▷ $extends(C_1, C_2) \vee extends(C_2, C_1)$.
   - ▷ $s = \{s_i \mid s_i \in C_1 \ \wedge \ s_i \notin C_2\}$.