Proc. of the Intl. Conf. on Advances In Bio-Informatics,Bio-Technology And Environmental Engineering-ABBE 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-009-5 doi: 10.15224/ 978-1-63248-009-5-104

Comparing Formal Specifications with Diagrammatic Notations: A Case-Study Approach

[Kobamelo Moremedi[1] and John Andrew van der Poll[2]]

[†] ***Abstract*—Formal specification techniques, e.g. Z have been applied in a variety of application areas to provide for clear and unambiguous specifications. Diagrams on the other hand have also been used in various areas and in software engineering they could be used to add a visual component to software specifications. It is plausible that diagrams may also be used to reason in a semi-formal way about the properties of a specification. In this paper we employ a case study approach to determine the extent to which diagrammatic notations can successfully be used to specify system properties. Comparisons on the merits of a diagrammatic notation are presented towards the end of the paper.**

***Keywords*—case study, diagrammatic notation, formal specification, Spider diagrams, Venn diagrams, Z**

I. Introduction

Z is a formal language that can be used in software engineering to construct clear and unambiguous specifications [9], [2], [15], [16]. It is based on first-order logic and a strongly-typed fragment of Zermelo-Fraenkel set theory [6] to model the properties of a system. The formal text is normally augmented with natural language prose to assist stakeholders and users in deciphering the often terse mathematical notation.

Diagrams have been applied in various areas and in software development they are used to specify and reason (mostly semi formally) about the system [14], [12]. They have also been applied in numerous examples such as specifying failures in safety critical hardware, database search queries, file system management, ontology representation and statistical data representation [4], [5], [10], [13].

Kobamelo Moremedi[1]
School of Computing
University of South Africa
South Africa

John Andrew van der Poll[2]
Graduate School of Business Leadership (SBL)
University of South Africa
South Africa

[†] This work is based on the research supported in part by the National Research Foundation of South Africa (Grant Number 78911). Any opinion, finding and conclusion or recommendation expressed in this material is that of the authors and the NRF does not accept any liability in this regard.

The purpose of this paper is to determine the merits of diagrammatic notations with respect to the established techniques of formal specifications, in particular the Z specification language. Our notation is a semi-formal one that could be used to deliver a specification that is accessible to a wide range of users [3], [13], [12]. Formal specification languages generally embody a fair amount of mathematics, requiring rigorous training and experience in order to comprehend the specification and gain the desired benefits. Our case study is the specification of a symbol table [7] from the arena of compiler construction.

The layout of the paper follows: Section II briefly introduces the case study to be used in our work, while various operations on the state are presented in Section III. An analysis of the merits of both specification techniques appears in Section IV and the conclusion and directions for future work in this area appear in Section V.

II. Symbol table

A symbol table (*ST*) maintains a set of symbols with corresponding values. Each symbol in the table is associated with a unique value. The usual operations performed on a symbol table are to *Add* a symbol with corresponding value, *Look up* the value associated with a given symbol, *Replace* the value of a symbol, and *Delete* a symbol and its associated value from the table.

The specification follows the Established Strategy for constructing a Z spec [2], augmented by a set of enhanced principles [17] to model the operations of a system. Three basic types are defined for our specification:

[*SYM*, *VAL*, *REPORT*]

SYM represents the set of all symbols that may ever find their way into the symbol table; *VAL* specifies the set of all allowable values, and feedback to a user of the specification is indicated by *REPORT*. In line with a design principle proposed in [17], communication with the user of the specification ought to be maximized. Subsequently, feedback to the user is defined and consists of (called a data type definition):

REPORT ::= *OK* | *Symbol_not_present* | *Symbol_exists* | *Symbol_not_found*

Further user communication may be defined but is beyond the scope of this paper.

Proc. of the Intl. Conf. on Advances In Bio-Informatics,Bio-Technology And Environmental Engineering-ABBE 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-009-5 doi: 10.15224/ 978-1-63248-009-5-104

III. States and operations

A. *Abstract state*

The Schema *ST* below denotes the abstract state of the system. The relationship between *SYM* and *VAL* is modeled by a partial function, *st*.

```
_ST_______________________________
| st : SYM ⇸ VAL
|_________________________________
```

The diagram in **Fig. 1** below is a graphical representation of the above abstract state. The name of the state and the basic types are shown at the top of the diagram. The closed curves called contours are used to represent sets [6], [11], [4]. The arrow pointing from one circle to the next represents a relation, in this case a partial function written below the curve, with name *st* indicated above the curve. A preliminary version of this notation was developed in [18].

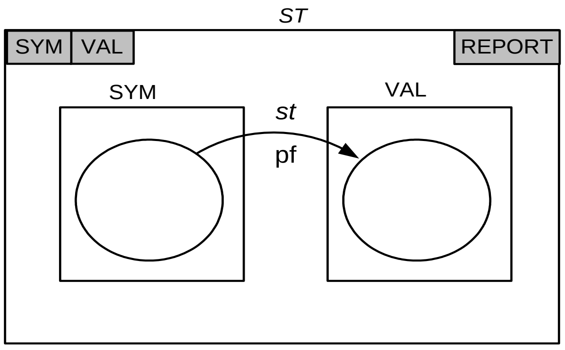

Figure 1. The abstract state of *ST*.

B. *Initial state*

The initial state, *Init_ST*, of the symbol table system appears below. Unless dictated otherwise (e.g. a schema involving numeric components), it is customary to start with empty sets as indicated: $st' = \emptyset$. System components are included above the short dividing line and relationships among components are given below the line.

```
_ Init_ST__________________________
| ST
|_________________
| st′ = ∅
|__________________________________
```

Fig. 2 captures *Init_ST* in a diagram. The shading of the closed curve is used to denote that the set is empty, in line with a particular version of the language of Venn diagrams [19]. Our operation diagrams are divided into two parts. The top half of the larger box is called a *before* diagram, while the lower part is coined the *after* diagram.

Notice a slight deviation from the information in schema *Init_ST*: In the formal notation we specify an empty function; in the diagram we explicitly show that the domain of *st*' is empty, leading to a proof obligation $st' = \emptyset$ as far as the diagram is concerned.

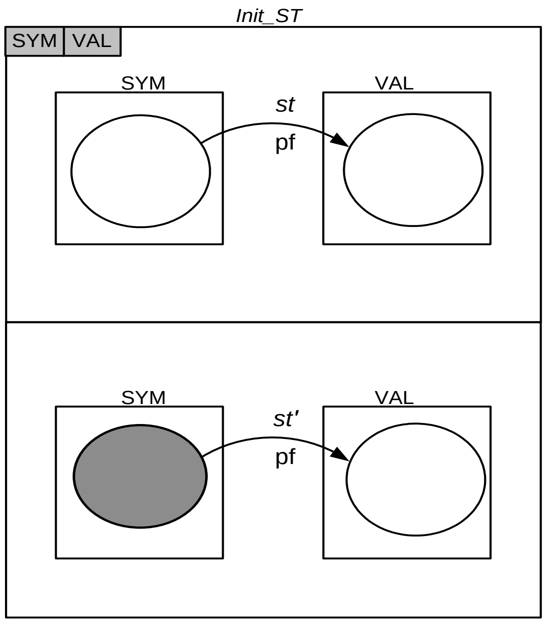

Figure 2. Initial state of the symbol table.

C. *Operations on the symbol table*

The following schema specifies the operation to add a new symbol in the symbol table. A precondition is that the symbol to be added should not be already in the table.

```
_Add_______________________________
| ΔST
| s? : SYM; v? : VAL
| rep! : REPORT
|__________________
| s? ∉ dom st
| st′ = st ∪ { s? ↦ v? } ∧ rep! = OK
|__________________________________
```

Δ*ST* denotes a possible change in the after value of *st* (i.e. *st*′). The operation receives the inputs *s*? and *v*?, denoting the new symbol and its associated value respectively to be added to the symbol table. Feedback to the user is indicated by *rep*! (Input- and output variables are decorated by '?' and '!' respectively). For a correct *Add* operation, the new symbol ought not to be in the symbol table already – *s*? ∉ dom *st*. The after state contains the new symbol and its associated value. The user is informed of a successful addition to the table.

The diagram in **Fig. 3** represents the above *Add* operation in appropriate before- and after diagram notation. A possible

Proc. of the Intl. Conf. on Advances In Bio-Informatics,Bio-Technology And Environmental Engineering-ABBE 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-009-5 doi: 10.15224/ 978-1-63248-009-5-104

state change is indicated in the top right hand corner of the before diagram.

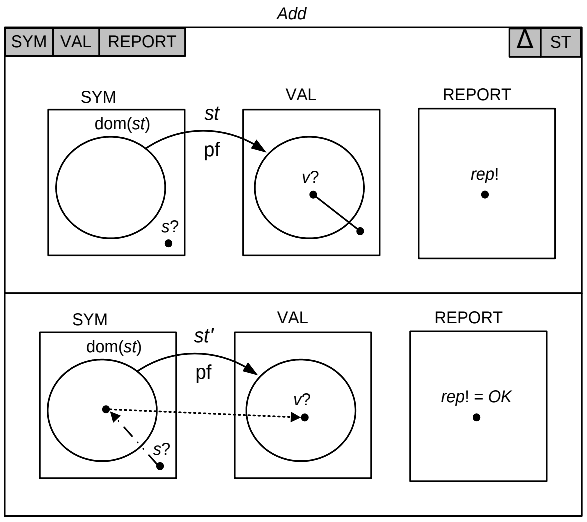

Figure 3. The *Add* operation of ST.

In the before diagram, *s*? represents an input variable that is not yet in the symbol table (indicated as being outside the circle which represents the domain of *st*).

A ● (dot) in our diagrams represents the existence of an element [6], [14]. The straight line which joins the two dots in the before diagram indicates that it is immaterial whether *v*? is already a value in the symbol table or not. Notice this deviation, giving more information in the diagram than what is available in the schema.

Strictly speaking the component *rep!* of type *REPORT* does not exist in the before state (diagram); it only comes into 'existence' as part of the postcondition of the schema. However, looking ahead at refinement into executable code, variable *rep*! would presumably be a global variable in a programming language and would therefore be declared, and exist in a program before an operation (like *Add*) would be invoked. Hence we made it part of our before diagram. Note that the Z schema notation is not specifically clear about this aspect.

The after diagram indicates that *s*? has 'moved' to be part of the symbol table and is related to its value *v*? Appropriate feedback is conveyed to the user of the specification.

The *LookUp* operation is used to determine the current value associated with a symbol. ΞST indicates that the state of the system remains invariant. Input to the operation is represented by *s?* and output is specified by *v*! and *rep*!.

LookUp

$\Xi\, ST$

$s? : SYM;\ v! : VAL;\ rep! : REPORT$

$s? \in \mathrm{dom}\ st$

$v! = st\ (s?) \wedge rep! = OK$

Fig. 4 is a diagrammatic representation of operation *LookUp*.

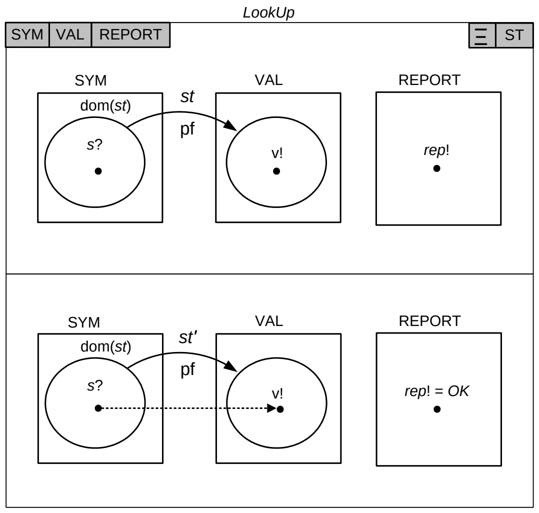

Figure 4. The *LookUp* operation.

Variable *s*? ought to exist in the before diagram. Naturally it is related to a value (according to our *Add* operation), but such value is not known beforehand. The after diagram states that *s*? is linked to its value *v*!. Feedback to the user is specified.

The schema below describes an operation to replace the value of a symbol already in the table. The *Replace* operation may also change the state of the system just like in operation *Add*, hence the notation Δ*ST*. The symbol '⊕' in the predicate is the overriding function, indicating that any existing value associated with *s*? is replaced by *v*? The postcondition $st' = st \oplus \{s? \mapsto v?\}$ denotes that *st*′ is *st* overwritten by the symbol associated with a new value.

Replace

$\Delta\, ST$

$s? : SYM;\ v? : VAL;\ rep! : REPORT$

$s? \in \mathrm{dom}\ st$

$st' = st \oplus \{s? \mapsto v?\} \wedge rep! = OK$

The diagram in **Fig. 5** models the above *Replace* operation.

The symbol whose value is to be replaced ought to exist in the table. As before, it is immaterial whether the associated value is already present in the range of the function, or not. Afterwards, the value of *s*? is known to be *v*?.

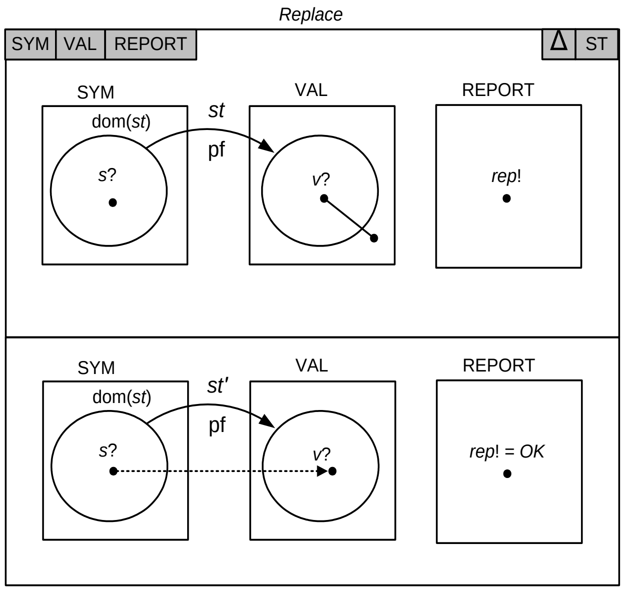

Figure 5. The *Replace* operation.

The symbol whose value is to be replaced ought to exist in the table. As before, it is immaterial whether the associated value is already present in the range of the function, or not. Afterwards, the value of *s*? is *v*?.

A symbol may also be deleted from the symbol table. For a correct deletion, we would require the symbol to exist in the table beforehand. The following schema specifies the operation to delete a symbol. The *Delete* operation introduces the domain subtraction operator (⩤) [1], [9], [8], used to exclude *s*? from the domain of *st*′. A proof obligation of *Delete* is to show that *s*? does not exist in the after state of *st*.

Delete

$\Delta\, ST$

$s? : SYM;\ rep! : REPORT$

$s? \in \operatorname{dom} st$

$st' = \{s?\} \lhd st \wedge rep! = OK$

The diagram below captures operation *Delete*. The after diagram indicates that *s*? is not in the domain of *st*′. For the sake of clarity, one could show that *s*? was related to some value in its range and that such value may continue to exist, or may not exist anymore (cf. the notation in figures 3 and 5) in the range of *st*′. But, since schema *Delete* is silent about such information, our diagram follows suit. One could argue that the indication of such tautological information would indeed strengthen the visual characteristics of the diagram.

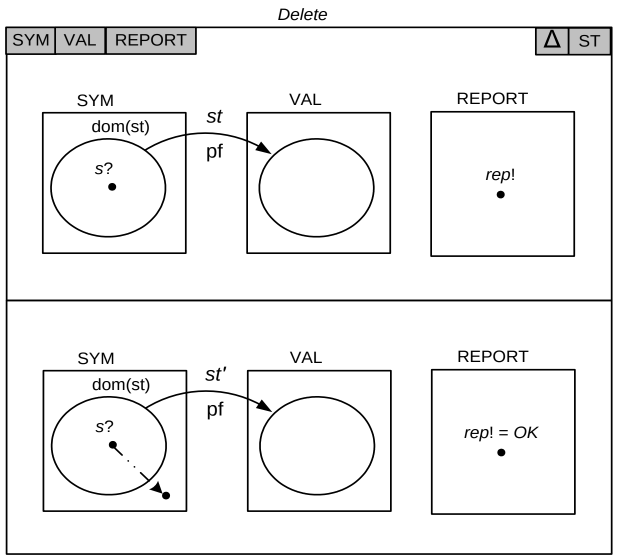

Figure 6. *Delete* operation.

So far in this paper we showed partial and correct versions of our operations. If any of the preconditions are not satisfied, error conditions arise together with the appropriate feedback to the user. An example is *NotPresent* in conjunction with *LookUp*.

NotPresent

$\Xi\, ST$

$s? : SYM;\ rep! : REPORT$

$s? \notin \operatorname{dom} st$

$rep! = Symbol\_not\_present$

A diagrammatic specification of *NotPresent* is given in **Fig. 7**. It shows that the symbol enquired about is not present in the table (outside dom(*st*)). The condition prevails in the after diagram, hence the no change in the system state.

Proc. of the Intl. Conf. on Advances In Bio-Informatics,Bio-Technology And Environmental Engineering-ABBE 2014.
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-1-63248-009-5 doi: 10.15224/ 978-1-63248-009-5-104

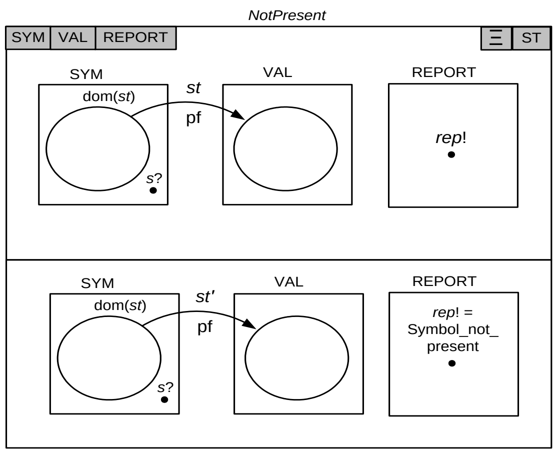

Figure 7. Representation of schema *NotPresent* .

IV. Comparisons

A comparison of the differences and similarities between a formal notation as embedded in Z, with diagrammatic notations introduced in this paper appears in TABLE 1.

TABLE 1. COMPARISON OF FORMAL- AND DIAGRAMMATIC NOTATIONS

Attribute	**Specification Style**	
	Formal specification	***Diagrammatic***
Precision	A formal specification is per definition precise and unambiguous.	Diagrams may suffer from imprecision and ambiguity.
Conciseness	Formal specifications (e.g. Z) are generally concise.	Diagrams tend to be verbose and time consuming to construct.
Clarity	A formal specification is clear, but only to the mathematically literate.	Diagrams are comprehensible to non mathematicians owing to their visual character.
Level of detail	Schema *Init_ST* specifies *st*′ = ∅. Information about the domain and range are to be inferred indirectly.	**Fig. 2** which represents schema *Init_ST* specifies the domain of *st*′ to be empty. This gives more detail than the schema predicate.
Additional information	Schemas leave tautological information up to the user to determine.	Tautological information (e.g. *v*? ∊ ran *st* or not) is shown explicitly (e.g. **Fig. 3**).
Variables in precondition	Output variables in the header of a schema presumably exist as part of the precondition.	Output variables are explicitly shown to exist in a before diagram.

V. Conclusions

In this paper, formal specifications as defined in Z were compared to corresponding visual notations as captured by Venn-like diagrams. A case study from the literature was used as the vehicle of comparison. Formal specifications are generally concise and precise, while the corresponding diagrammatic notation is more verbose and takes up more space as e.g. a Z schema. In some instances however, a diagram may convey information more directly, e.g. when specifying the domain of a function to be empty instead of stating the function to be empty. Other aspects relate to specifying tautological information and the presence of output variables as part of the precondition of a schema or a before diagram. A diagram may also be more easily interpreted than the corresponding mathematical text.

Future work in this area may be pursued along a number of lines. The feasibility of specifying diagrammatically more comprehensive structures will be investigated. Examples include arbitrary unions, bags (as defined in Z), etc. Part of this development will be the specifying of schema calculus operations as diagrams and the investigation of their space complexities. For example, a robust operation of *LookUp* could be *RobustLookUp* ≙ *LookUp* ⋁ *NotPresent*. Investigating the scalability of our approach and tool support are further items on the agenda.

References

[1] A. Diller, “ Z: An Introduction to Formal Methods,” Wiley, Chichester, 2nd ed., 1994.
[2] B. Potter, J. Sinclair, and D. Till, “An Introduction to Formal Specification and Z,” Prentice Hall, Upper Saddle River, UK, 1996.
[3] F. Dau, “Types and tokens for logic with diagrams,” Lecture Notes in Computer Science. vol. 3127/2004, pp. 62-93, 2004.
[4] G. Stapleton, “A Survey of Reasoning Systems Based on Euler diagrams,” Proceedings of the First International Workshop on Euler Diagrams, Brighton, UK, 1 June, vol. 134, pp. 127 – 151, 2005.
[5] G. Stapleton, P. Rodgers, J.Howse, and J. Taylor, “Properties of Euler diagrams,” Layout of (Software) Engineering Diagrams, vol. 7, pp. 1 – 15, 2007.
[6] H. B. Enderton, “Elements of Set Theory,” Academic Press Inc, 1977.
[7] I Hayes, “Specification Case Studies,” Prentice Hall, UK, 1992.
[8] J. B. Wordsworth, “Software Development with Z,” Addison Wesley, IBM United Kingdom, 1992.
[9] J. Bowen, “Formal Specification and Documentation Using Z – A case study approach” C.A.R. Hoare. pp. 3 – 11, 2003.
[10] J. Gil and J. Howse, “Formalizing Spider Diagrams,” IEEE Symposium on Visual Languages, pp. 130 – 137,1999.
[11] J. Howse, F. Molina, and J. Taylor, “Reasoning with Spider Diagrams,” IEEE Symposium on Visual Languages, pp. 138 – 145, 1999.
[12] J. Howse, J. Taylor, and G. Stapleton, “Spider diagrams,” LMS Journal of Computation and Mathematics, vol 2980/2004, pp. 154 – 194, 2005.
[13] J. Howse, J. Taylor, G. Stapleton, and T. Simpson, “ The Expressiveness of Spider Diagrams Augmented with Constants,” vol 20, pp. 30 – 49, 2009.
[14] J. Howse, J. Taylor, G. Stapleton, and T. Simpson, “What Can Spider Diagrams Say?;” In: Blackwell, A., Marriott, K. and Shimojima, A., eds. Diagrammatic Representation and Inference: Third International conference, Diagrams. Cambridge, pp. 112 – 127, 2004.
[15] J. M. Spivey, “Z Notation: A Reference Manual,” 2nd ed, Prentice Hall, Oxford, 1992.
[16] J. Woodcock and J. Davies, “Using Specification, Refinement and Proof,” Prentice-Hall, 1996.
[17] J. A. van der Poll and P. Kotze, “Enhancing the Established Strategy for Constructing a Z Specification,” South African Computer Journal (SACJ), Number 35, pp. 118 – 131, 2005.

ISBN: 978-1-63248-009-5 doi: 10.15224/ 978-1-63248-009-5-104

[18] K. P. Moremedi and J. A. van der Poll, "Transforming Formal Specification Constructs into Diagrammatic Notations," The 3rd International Conference on Model & Data Engineering," (MEDI). *Lecture Notes in Computer Science* (LNCS), No 8216, pp 212 – 224, 2013. ISBN 978-3-642-41365-0.

[19] S. Chow and F. Ruskey, "Drawing Area-Proportional Venn and Euler diagrams," Lecture Notes in Computer Science. vol. 2912/2004, pp. 466 – 477, 2004.

About the Authors:

Kobamelo Moremedi is busy with his MSc in Computer Science at the University of South Africa. His research interests are in formal specification, sem-formal specification techniques and combining formal- and semi-formal notations.

John van der Poll is a Professor in Computing at the Graduate School for Business Leadership (SBL) at the University of South Africa. His research interests are in Automated Reasoning, Specification formalisms and the application of Formal Methods in industrial and business applications.

Diagrams have been applied in various areas and in software development they are used to specify and reason (mostly semi formally) about the system. They have also been applied in numerous examples such as specifying failures in safety critical hardware, database search queries, file system management, ontology representation and statistical data representation.