Transforming Formal Specification Constructs into Diagrammatic Notations

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

[1] School of Computing, University of South Africa, Pretoria, South Africa
kobamelomoremedi@yahoo.com
[2] Graduate School of Business Leadership, University of South Africa, Midrand, South Africa
vdpolja@unisa.ac.za

Abstract. Specification plays a vital role in software engineering to facilitate the development of highly dependable software. Various techniques may be used for specification work. Z is a formal specification language that is based on a strongly-typed fragment of Zermelo-Fraenkel set theory and first-order logic to provide for precise and unambiguous specifications. While diagrammatic specification languages may lack precision, they may, owing to their visual characteristics be a lucrative option for advocates of semi-formal specification techniques. In this paper we investigate to what extent formal constructs, e.g. Z may be transformed into diagrammatic notations. Several diagrammatic notations are considered and combined for this purpose.

Keywords: Diagrammatic notation, Formal specification, Euler diagrams, Spider diagrams, Venn-Pierce diagrams, Z.

1 Introduction

The correctness of software has a significant impact in controlling and delivering the essential, and often safety critical services that we depend on, such as health care; transport (airlines and railways); and telecommunication [1]. Specification is a vital activity aimed at producing a system that will meet the user requirements stated during the initial stages of software development. The resultant specification is used in software development to provide a clear communication of requirements documentation and system objects among stakeholders involved in the software project. Hence, it is desirable that a specification be accessible to intended users in order to facilitate the development of quality software.

Various specification techniques have been developed to specify software systems. The Z notation is a formal text-based language that has a successful history of being able to provide for precise specifications in the development of critical systems [2]. The IBM CICS system is one of the large projects in which Z was used successfully [17]. The use of Z increased the quality and reliability of the system [21]. Z is based on first-order logic and a strongly typed fragment of Zermelo-Fraenkel set theory [19]. Its basic construct is the schema which is used to structure the specification. System operations are collected into schemas to describe the state of the system and how it changes [6], [20], [21].

A. Cuzzocrea and S. Maabout (Eds.): MEDI 2013, LNCS 8216, pp. 212–224, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Diagrams, as a semi-formal notation are widely applicable in conveying important ideas, and in Computer Science they can be used to specify software. For example, ‘spider diagrams’ have been used in the specification of failures of safety critical systems, ontology representations, database search queries and file system management [5], [7], [10], [12].The familiar Venn diagram has often been used as a heuristic tool in mathematics and logic, facilitating the formalization of the relevant idea. However, Shin challenged the view that diagrams could not yield formal specifications by developing two sound and complete reasoning systems of Venn diagrams [4], [7].

Although diagrams (as a semi-formal notation) lack the precision of a formal notation, e.g. spider diagrams, their value has been recognized recently, in aspects of software specification, reasoning and information visualization. Consequently, this paper is aimed at investigating the extent to which diagrams can capture the structures and operations of discrete structures omnipresent in Z specifications. Translating semi-formal notations (e.g.) UML to variants of Z have been done before [18], but since UML may be viewed as being at a 'higher' level than the core set-theoretic structures and operations on which a Z specification is based, our translations are based on closed-curve constructs, namely, Euler-, Venn-, Spider- and Pierce diagrams.

The layout of the paper follows: Different types of diagrams are discussed briefly in Section 2 and a small Z example is given in Section 3. Section 4 identifies a number of set-theoretic structures and operations in Z and we show how these may be specified using diagrams. Section 5 presents an analysis and directions for future work in this area.

2 Diagrammatic Notations

Different types of diagrammatic languages can be used for specifying software and in this paper we focus on diagrams based on closed curves.

2.1 Euler Diagrams

Euler diagrams were introduced in the 17th century by Leonard Euler. This notation uses ‘contours’ to represent the relationship between sets [7], [15]. A contour is a closed circle used in a diagram to represent a set. Most diagrammatic languages emerged from Euler diagrams [12]. **Fig.1** denotes that sets A and B are disjoint and C is a subset of A. The non-existence of elements is used to indicate an empty set. For example, no elements are indicated for set C, hence it's empty.

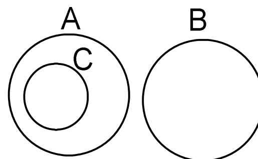


Fig. 1. An Euler diagram

2.2 Venn Diagrams

In 1880, John Venn developed Venn diagrams that are similar to Euler diagrams. Instead of missing elements, Venn diagrams use shading to denote an empty set. Venn diagrams use overlapping circles for representing relationships among sets [3]. **Fig.2** shows an example of Venn diagrams drawn with three overlapping contours. For example, the region C – (A ∪ B) could be shaded, indicating it is empty. Venn diagrams may become hard to interpret or draw once the diagram contains more than three contours.

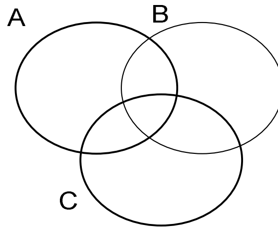


Fig. 2. A 3-contoured Venn diagram

2.3 Pierce Diagrams

Pierce introduced existential graphs by adding X-sequences on Venn diagrams to represent disjunctive information [7], [9]. Pierce diagrams, also known as 'Venn-Pierce diagrams', extend Venn diagrams by adding syntax that represents existential statements in diagrams. Pierce used 'x' instead of the existence of elements, and 'o' instead of shading, to represent an empty set [13]. The Pierce diagram in **Fig.3** shows that $A - B = \emptyset$ and $A \cap B \neq \emptyset$.

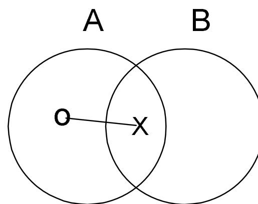


Fig. 3. A Pierce diagram

2.4 Spider Diagrams

Spider diagrams were inspired by Pierce-, Venn- and Euler diagrams. A 'spider' denotes the presence of one or more elements in a set. Spiders are nodes connected with straight lines [7]. Spider diagrams use non-overlapping contours of Euler diagrams, spiders, which generalize Pierce's X-sequences and shading from Venn diagrams [5], [9], [11]. The spider diagram in **Fig.4** has three contours labeled A, B and C. The contours are represented as A ∪ C and B ⊂ A.

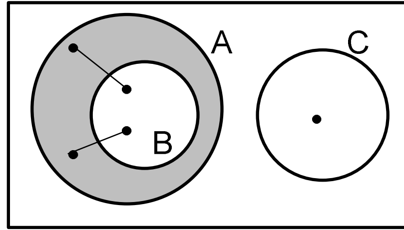


Fig. 4. A Spider diagram

3 The Z Notation

Z was developed by the Programming Research Group (PRG) at the University of Oxford. Its basic construct is the schema, containing mathematical text and being surrounded by natural language prose. Basic types to be used are specified early on in the specification document.

Below is an example specification showing two basic types, a state space (*File*) and one partial operation (*FileRead*) on the state. The example is modeled on specifications in [19] and [8].

The basic types are:

[KEY, RECORD]

The abstract state of the file system is shown below:

File

$file : KEY \mapsto RECORD$

The relationship between *KEY* and *RECORD* is defined by a partial function ($\mapsto$). State variables (*file* above) in Z are known as components.

The below schema specifies an operation on the state. Ξ *File* specifies that the *Read* operation will not change the state of the system (in contrast, a 'Δ' before a state name is used to indicate a possible state change). The operation receives the input *k*? and produces output *r*! The symbols '?' and '!' are used to decorate input and output variables respectively. Predicates are specified below the short dividing line in a schema and further constrain the state components and any additional variables. The *key*? should be known to the system and a record (*r*!) is returned for a correct key.

FileRead

$\Xi File$

$k? : KEY$

$r! : RECORD$

$k? \in \mathrm{dom}\, file \wedge r! = file\ k?$

4 Specification Structures and Operators

In this section we present a number of Z specifications from the literature and transform each such construct into a diagram. The specifications shown stem mainly from [8]. The first operation considered is domain restriction, indicated by $\lhd$.

4.1 Domain Restriction

Consider the above file system. **Fig.5** below gives a diagrammatic representation of the state, *File*. The 'rectangles' in the diagram are used to indicate the basic types in the specification. Closed circles called contours, represent sets in the specification. The curved arrow connecting two contours denotes a relation. The name of the relation (*file*) appears above the curve, and its type is labeled below the curve. It is a partial function (*pf*).

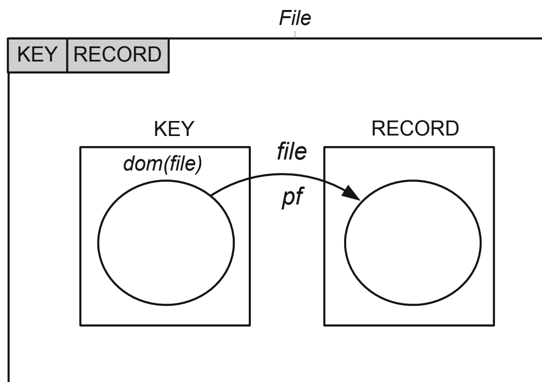


Fig. 5. The abstract state: File system

Next we consider an operation, *SelectRecord* to restrict the file system to just one record for which a key ($k?$) is provided:

SelectRecord

$\Delta File$

$k? : KEY$

$k? \in \operatorname{dom} file$

$file' = \{k?\} \lhd file$

The file system is changed to just the record matching $k?$ Note, in practice one would define a variable for this purpose instead of removing all other records from the state.

Fig.6. shows how the above operation may be translated into a contoured diagram.

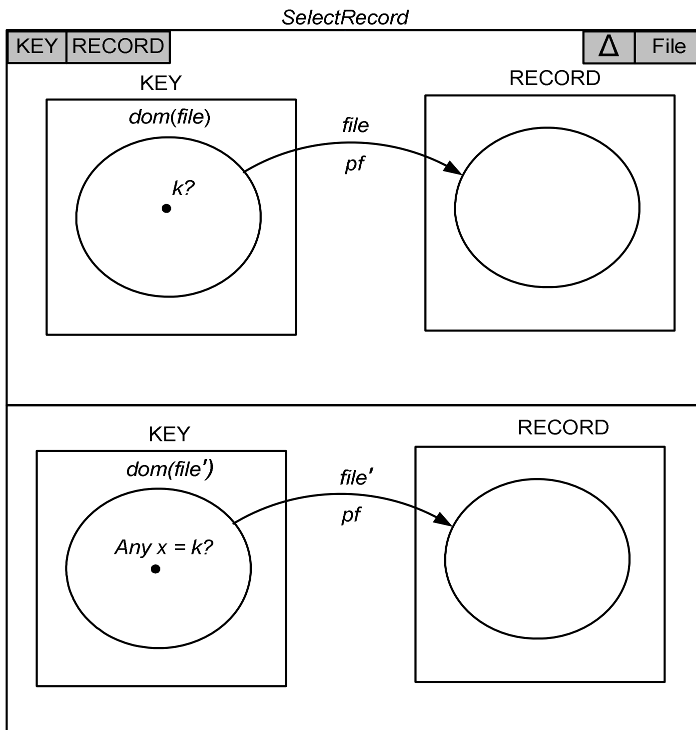


Fig. 6. Operation *SelectRecord*

The top part of the diagram (called a *before* diagram) represents the precondition to the operation. It indicates that the key *k*? should exist in the *file* domain. The after (bottom) diagram specifies that *k*? is the only key left in the file. The dot • indicates that there is at least one element in the set (a syntax taken from spider diagrams). Having restricted the domain of *file* to just {*k*?}, leaves but one record in the file. The key of any such record equals *k*? In the absence of further information one assumes *file*′ (*k*?) = *file* (*k*?), being a traditional proof obligation arising from the specification.

Note that our diagrammatic notation allows us to abstract away from the set connotation {*k*?} specified in the schema, simply because we are working with a singleton, and the only element of the singleton is explicitly instantiated.

4.2 Overriding Operator

Consider a symbol table containing symbols with associated values. *SYM* and *VAL* are basic types used to represent, respectively, the set of symbols and associated values. The state, *ST*, consists of one component, *st*, a partial function.

$$\begin{array}{|l} \hline \textit{ST} \\ st : SYM \nrightarrow VAL \\ \hline \end{array}$$

Fig.7 below gives a diagrammatic representation of the above state. Note that we may omit the denotation '*dom*(*st*)', since it may be inferred from the diagram.

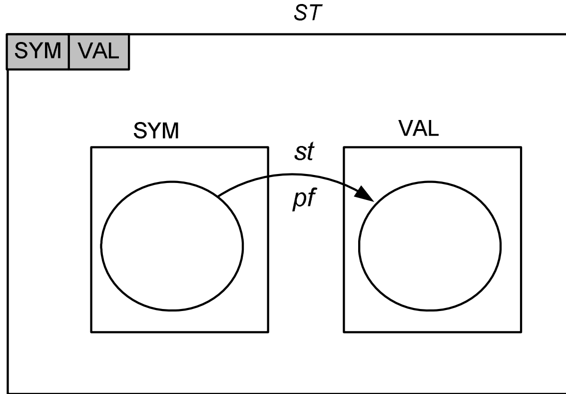


Fig. 7. The abstract state of symbol table

The following operation associates a value *v*? with symbol *s*? The operation gives feedback to the user.

$$\begin{array}{|l} \hline \textit{Replace} \\ \Delta\, st \\ s? : SYM \\ v? : VAL \\ rep! : REPORT \\ \hline s? \in \operatorname{dom} st \\ s' = st \oplus \{ s? \mapsto v? \} \\ rep! = OK \\ \hline \end{array}$$

The overriding operator '⊕' is used to replace the value (if any) of a variable in the symbol table with a new value. Its definition for any two relations $R : X \leftrightarrow Y$ and $S : X \leftrightarrow Y$ (say) is given by: $R \oplus S = ((\operatorname{dom} S) \lhd R) \cup S$.

Fig.8 denotes the operation to update a symbol in the table in line with the above schema. The before diagram indicates that *s*? is to exist in the symbol table, while *v*?, the input to the system, may either be in the range of *st,* or not. The after diagram indicates that *s*? maps to *v*? and variable *rep*! has the value 'Ok' after the operation.

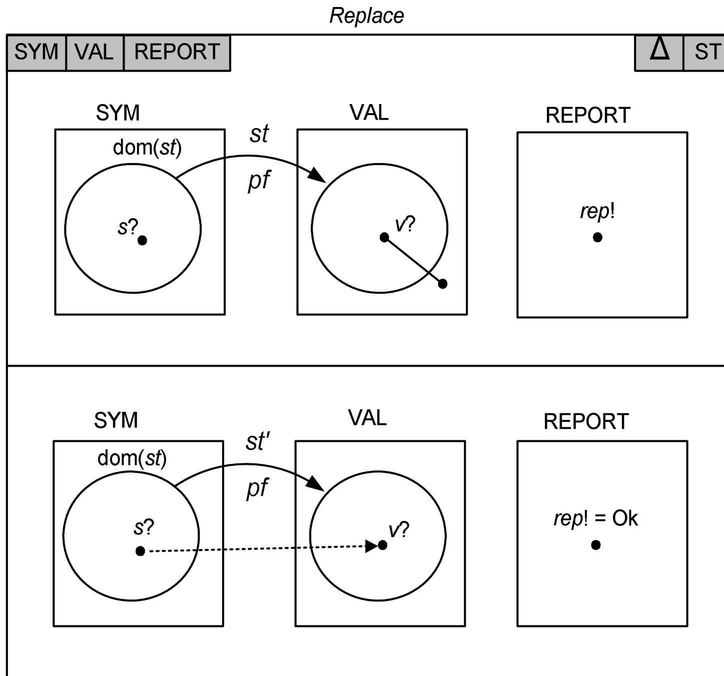
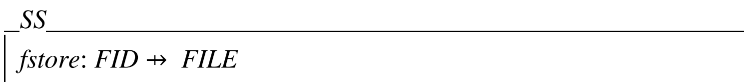


Fig. 8. The Replace operation

4.3 Domain Subtraction

Consider the next higher level of the above file system to model file identifiers mapped to files. Each file has a unique identifier. The schema below depicts the state of such a file storage system (*SS*). The abstract state denotes a partial function from *FID* to *FILE* [8].

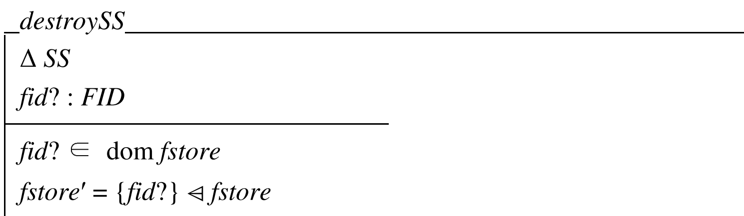


SS

$fstore : FID \pfun FILE$

Fig.9 shows the abstract state of SS. It specifies *fstore* as a partial function.

The schema below specifies the operation of deleting a file [8]. Only files that exist in the system can be deleted.



destroySS

$\Delta\, SS$

$fid? : FID$

$fid? \in \operatorname{dom} fstore$

$fstore' = \{fid?\} \ndres fstore$

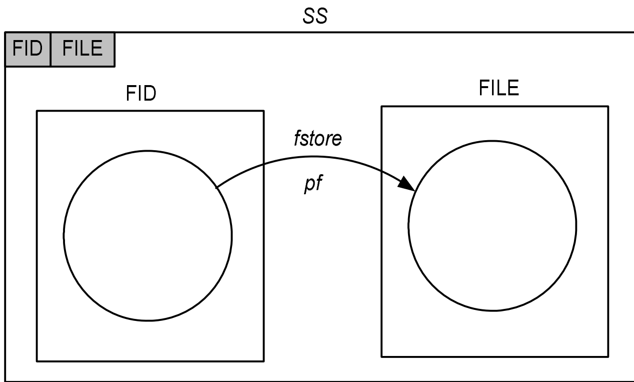


Fig. 9. The abstract state of *SS*

The domain subtraction operator '⩤' is used to remove *fid*?; the state of the system is changed as indicated. After the operation, *fid?* no longer exists as a valid file identifier in the system.

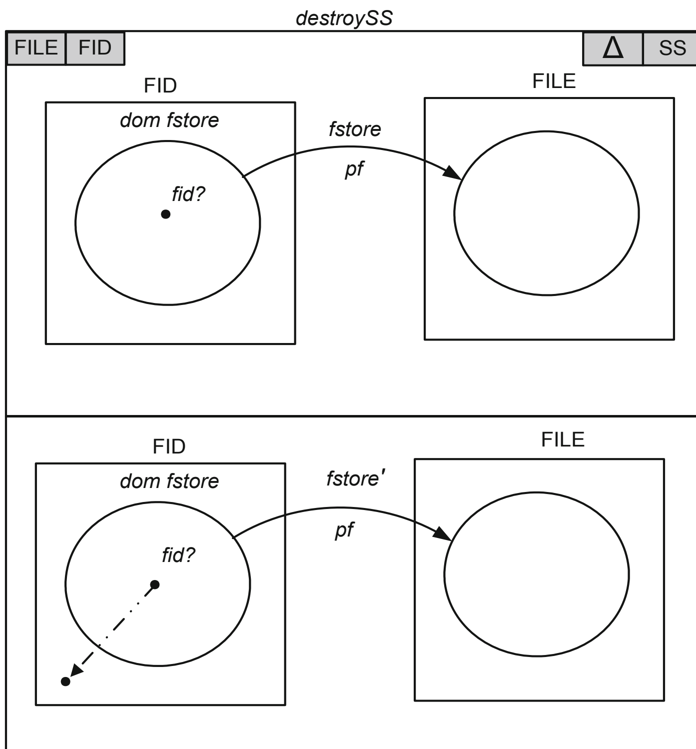


Fig. 10. The *destroySS* operation of the file storage system

Fig. 10 captures operation *destroySS*. The before diagram specifies that the file to be deleted should exist in the system and the after diagram states that the file identifier has been removed from the set of valid file identifiers. A dashed line indicating the movement is used for this purpose.

4.4 Range Subtraction

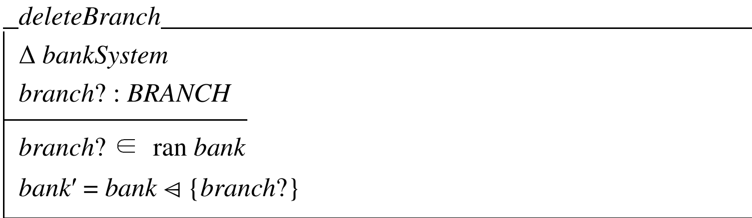
A simplified banking system stores the details of customers with the corresponding branches they belong to. A customer can be registered with only one branch. The state of the system is given by *bankSystem*.

bankSystem

$bank : CUSTOMER \pfun BRANCH$

The contour diagram for the above state is similar to that of the other operations shown above.

An operation to delete an entire branch from the system is similar to the domain subtraction operation shown earlier, and is given by:



deleteBranch

$\Delta\ bankSystem$

$branch? : BRANCH$

$branch? \in \operatorname{ran} bank$

$bank' = bank \rhd \{branch?\}$

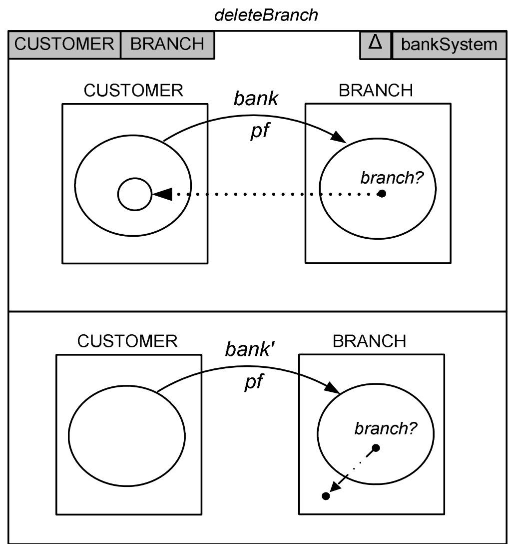


Fig. 11. The *deleteBranch* operation

The above schema is a simplified version of a real life situation. In practice customers would be moved to another branch before their branch is closed. The corresponding diagram follows.

4.5 Specifying Non-Singleton Sets

So far we have removed from a set, or restricted the domains or ranges of relations to a set containing one element only. We were able to abstract away from the complexities of sets and showed in such cases a single item only, instead of a singleton containing only that item.

The following operation removes a set containing an unspecified number of items from a domain and also overrides the relation with one of the same type. The state of the system is given in Section 3 and the operation is specified by *FileUpdate* below.

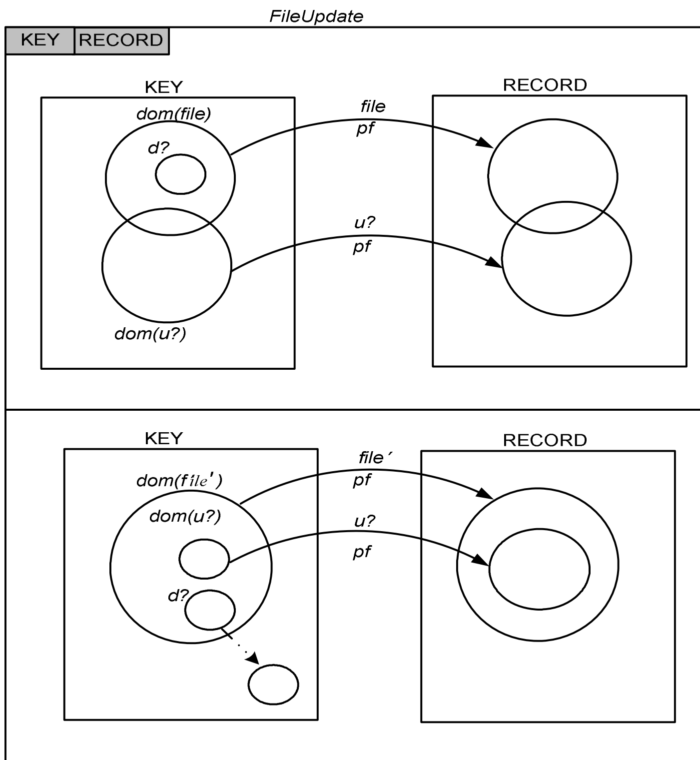


Fig. 12. FileUpdate operation

FileUpdate

Δ *File*

$d? : \mathbb{P} KEY$

$u? : KEY \nrightarrow RECORD$

$d? \subseteq \operatorname{dom} file$

$d? \cap \operatorname{dom} u? = \{\}$

$file' = (d? \lhd file) \oplus u?$

The set of keys to be deleted is specified by $d?$ Only valid keys may be deleted. The predicate $d? \cap \operatorname{dom} u? = \{\}$ indicates that a record cannot be deleted and updated at the same time. The file is updated as indicated.

FileUpdate is modeled by Fig. 12.

5 Conclusions and Future Work

This paper considered the feasibility of translating Z constructs to the language of contour diagrams. The formality of Z lends itself to precise specifications and it has been applied successfully to specify systems where the quality and reliability are critical. Z may also be used as a documentation tool to increase a specifier's understanding of system operations. A possible disadvantage of a formal notation is that specialist knowledge of the underlying mathematics is required before the real benefits of formal specification can be realized [2]. This steep learning curve is often the reason cited why formal notations are not used more widely in the software industry.

Diagrams model a system by using contours to represent the relationships between mathematical structures. The use of diagrammatic languages is perceived as a way whereby software specifications are made more accessible to stakeholders and potential users of the system [7]. In the past diagrams were often excluded as contenders of formality; however the research done by Shin challenged the view that diagrams could not be used in the arena of formal specification work [4].

As part of future work, our notation will be applied to more complex operations and structures, e.g. distributed unions, bags, etc. The feasibility of reasoning about the properties of our diagrams has to be considered and the scalability of the notations has to be investigated. To this end, tools for industrial applications have to be further developed. We also plan to combine Z constructs with our diagrams to generate a comprehensive specification language to cater for clear specifications that may also be accessible to a wide range of users.

References

1. Alagar, V.S., Periyasamy, K.: Specification of Software Systems, pp. 3–14. Springer, New York (1998)
2. Bowen, J.: Formal Specification and Documentation using Z – A Case Study Approach, pp. 3–11 (2003); C.A.R. Hoare Series Editor
3. Chow, S., Ruskey, F.: Drawing Area-Proportional Venn and Euler Diagrams. In: Liotta, G. (ed.) GD 2003. LNCS, vol. 2912, pp. 466–477. Springer, Heidelberg (2004)
4. Dau, F.: Types and Tokens for Logic with Diagrams. In: Wolff, K.E., Pfeiffer, H.D., Delugach, H.S. (eds.) ICCS 2004. LNCS (LNAI), vol. 3127, pp. 62–93. Springer, Heidelberg (2004)
5. Delaney, A., Stapleton, G.: On the Descriptional Complexity of a Diagrammatic Notation. In: Proceedings of the 13th International Conference on Distributed Multimedia Systems, September 6-8 (2007)
6. Diller, A.: Z: An Introduction to Formal Methods, 2nd edn. Wiley, Chichester (1994)
7. Gil, J., Howse, J.: Formalizing Spider Diagrams. In: IEEE Symposium on Visual Languages, pp. 130–137 (1999)
8. Hayes, I.: Specification Case Studies. Prentice Hall International, UK (1992)
9. Howse, J., Molina, F., Taylor, J.: Reasoning with Spider Diagrams. In: IEEE Symposium on Visual Languages, September 13-16, pp. 138–145 (1999)
10. Howse, J., Taylor, J., Stapleton, G., Simpson, T.: The Expressiveness of Spider Diagrams Augmented with Constants. Journal of Visual Languages and Computing 20, 30–49 (2009)
11. Howse, J., Taylor, J., Stapleton, G., Simpson, T.: What Can Spider Diagrams Say? In: Blackwell, A.F., Marriott, K., Shimojima, A. (eds.) Diagrams 2004. LNCS (LNAI), vol. 2980, pp. 112–127. Springer, Heidelberg (2004)
12. Howse, J., Taylor, J., Stapleton, G.: Spider Diagrams. LMS Journal of Computation and Mathematics 2980, 154–194 (2005)
13. Molina, F.: Reasoning with Extended Venn-Pierce Diagrammatic Systems. PhD Thesis, University of Brighton (2001)
14. Potter, B., Sinclair, J., Till, D.: An Introduction to Formal Specification and Z, 2nd edn. Prentice Hall, Upper Saddle River (1996)
15. Stapleton, G., Rodgers, P., Howse, J., Taylor, J.: Properties of Euler diagrams. Layout of (Software) Engineering Diagrams 7, 1–15 (2007)
16. Stapleton, G.: A Survey of Reasoning Systems Based on Euler Diagrams. In: Proceedings of the First International Workshop on Euler Diagrams, Brighton, UK, June 1, vol. 134, pp. 127–151 (2005)
17. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall (1992)
18. Kim, S.-K., Carrington, D.A.: A Formal Mapping between UML Models and Object-Z Specifications. In: ZB Conference, pp. 2–21 (2000)
19. Van der Poll, J.A.: Formal Methods in Software Development: A Road Less Travelled. South African Computer Journal (SACJ) (45), 40–52 (2010)
20. Wordsworth, J.B.: Software Development with Z. Addison-Wesley, IBM United Kingdom (1992)
21. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice-Hall (1996)