ISSN 2278-3091
Volume 8, No.1.4, 2019
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse1681.42019.pdf
https://doi.org/10.30534/ijatcse/2019/1681.42019

A Survey on Refinement in Formal Methods and Software Engineering

Muhammed Basheer Jasser, Mar Yah Said, Jamilah Din, Abdul Azim Abdul Ghani
Faculty of Computer Science and Information Technology University Putra Malaysia
43400 UPM Serdang, Selangor, Malaysia
mbjasser@m.ieice.org; mbjasser@gmail.com

ABSTRACT

In software engineering, formal methods allow the design, modelling and verification of hardware and software systems. Formal methods introduce preciseness, remove ambiguity in specifications, and support the verification of requirements and design properties. Methods and approaches are needed to manage the formal models and handle their complexity. Refinement has been carried out for system artefacts ranging from modelling and design levels like architectures, and state machines to implementation and programming levels like source code. Refinement is a significant way for building complicated systems starting from simple ones by adding features gradually. Refinement has to be understood carefully in the context of formal specification and verification. This article provides a survey on some refinement techniques and methods and in the context of formal methods and software engineering. We believe that this survey sheds a light on the research direction in regards to the refinement of formal methods. This survey also helps formal methods practitioners and users in observing and understanding the advantages and limitations of refinements methods and techniques of various studied formal methods. Accordingly, they can decide which formal method is to be used in modelling systems via refinement or which formal method is to be extended with new concepts and notions to support refinement.

Key words: Refinement, Formal Methods, Object-Oriented Formal Methods

1. INTRODUCTION

In software engineering, formal methods [1], [2] allow the design, modelling and verification of systems. Formal methods introduce preciseness, remove ambiguity in specifications, and support the verification of requirements and design properties. The specifications in formal methods could be viewed as mathematical models, which represent the intended behaviour of the systems and they are used to model several safety critical systems [3] such as: railway control systems, nuclear power plant control systems, aircraft control systems, intelligent transport systems, and medical systems. There exist different kinds of formal specifications and each has its own advantages and limitations. Some formal specifications are considered at the system modelling like (B-Method [2], Event-B [1], Z-Method [4] and VDM [5]), while another type is viewed as part of the system implementation level, in other words, the formal specification is added as supportive statements to the source code like Larch [6] and JML [7]. In this work, we concentrate on refinement in formal methods that are considered at modelling level.

Refinement is considered as obtaining a better version of software than the original one during the development process [8]. This is because refinement has been known as a familiar technique and methodology to deal with the changing and new requirements and to provide better concrete versions of the system artefacts at hand. This includes: vertical refinement in which the abstract requirements are refined into more detailed ones and horizontal refinement in which new requirements are handled in the refined model. Refinement has been carried out for system artefacts ranging from modelling and design levels like architectures, and state machines to implementation and programming levels like source code. Stepwise refinement is a powerful way for developing complicated systems using simple ones by adding features incrementally [9].

Several refinement methods and techniques have been introduced in formal methods and software engineering. In this article, we provide an overview of some refinement methods and techniques in the context of software engineering, formal methods and some object oriented formal methods. We believe that the overview may help formal methods users in understanding the refinements methods and techniques of various studies formal methods.

In this article, we deal with some well-known formal methods (B-Method, Event-B, Z-Method, VDM) and some object-oriented formal methods. The research questions that we handle in this work are: What are the main refinement methods and techniques proposed in software engineering and at which development phase/stage are they performed?, what are the refinement methods and techniques proposed for the formal methods studied in this work?. To answer these questions, we survey the research articles, technical reports, and theses from the known databases (ACM, IEEE Xplore, Science Direct) and universities repositories. We only focus on the sources that are related to the formal methods of interest to us. In future, we will extend our work to cover more formal methods and more refinement techniques.

This paper is organized as follows. Section 2 presents an overview on some formal methods (B-Method, Event-B, Z-Method, VDM) and some object-oriented formal methods. Section 3 presents some refinement methods in software engineering in general. Section 4 presents refinement methods and techniques in formal methods. Section 5 concludes the work.

2. FORMAL METHODS

In this section we provide an overview of some formal methods (B-Method, Event-B, Z-Method, VDM) and some object-oriented formal methods.

2.1. B-Method

B-method [2], [10] is a widely used formal method that allows developing correct-by-construction systems through different levels of abstraction. Specific conditions are determined and must be preserved through refinement of the abstract specification into more concrete specifications. The specifications in B-Method are called machines where abstract machines provide an abstract system view while the refinement machines provide a more concrete view. B-method is based on the predicates, set theory and first order logic.

Each B-Method machine consists of the following clauses: MACHINE, SETS, CONSTANT, PROPERTIES, VARIABLES, INVARIANT, INITIALIZATION, and OPERATIONS. MACHINE defines the machine name. SETS introduces the used sets in the machine. CONSTANT introduces the used constants in the machine. PROPERTIES contains the constants definition. VARIABLES includes the variables that represent the machine state. Variables are restricted by conditions called the invariants that are introduced in INVARIANTS. INITIALIZATION defines the machine initial state. OPERATIONS includes the operations that change the machine state.

2.2. Event-B

Event-B [1] is a variant of B-method and is based on action systems [11]. The mathematical notation used in Event-B is based on the set-theory [12]. One of the differences between B-Method and Event-B is that the latter differentiates the static and dynamic parts. An Event-B context contains the types, axioms and constants, while an Event-B machine represents the changes of the state variables via events. Machines contain variables, events and invariants. Variables v define the machine state, constrained by the invariants $I(v)$. The events change the state of the machine. They are described by guards $G(v,x)$, and actions $A(v,x,v')$. $G(v,x)$ represent the conditions under which $A(v,x,v')$ changes the value of v to a new value v'. x represents the parameters that are local variables of the event.

2.3. Z-Method

Z-method [4], [13], [14] is a formal specification language initiated by the programming research group at Oxford university to specify systems based on algebra set theory and predicate calculus.

Every specified system in Z is started with an abstract state and a sequence of operations which change the system state and result in the system evolution. The abstract state is represented by mathematical structures such as sets, relations, functions and sequences without considering the implementation mechanism, but focuses on making a system specification more readable to users. The abstract state and additional initial conditions should specify an initial state of the system. Z specifications are structured as related schemas which are mainly used to specify system state space, operations and invariants. State space is represented by the combination of system variables. Operations change the system state leading to the existence of the before and after states. Invariants are the general conditions, which must be preserved and should relate the before and after states for all the possible operations.

2.4. VDM

Vienna Development method VDM [5], [15], [16] is a formal development method invented by the researchers of IBM laboratories in Vienna. VDM is used for specification, modelling, and design of computer based systems. VDM started as a definition language at 1970s and evolved to development method at 1980s. The specification language of VDM is called VDM-SL [17] which is considered as a notation for system specification.

VDM-SL specification language is structured as modules. VDM-SL module consists of several basic clauses: *types*, *inv*, *state*, *init* and *operations*. *types* clause defines the basic types which are used for the system variables types definition. *inv* clause defines the general conditions which must be always preserved. *state* defines the system variables and their types which are defined in *types* clause. *init* initializes the system variables to their initial values. *operations* clause defines the several system functionalities which change the system state by changing the values of the system state variables.

2.5. Object-Oriented Formal Methods

Modelling systems using object oriented features benefits in the structuring, organizing and reuse. Many methods and languages are proposed to augment formal methods with object oriented features in order to get the advantages from both formal methods and object orientation.

Several variants of the conventional Z language have been proposed to augment Z with the object-oriented structuring

features [18]. Some of these are Object-Z [19], [20], [21], Z++ [22], OOZE [23], Hall's style [24], and Schuman and Pitt's variant [25]. Object-Z is considered the most applicable and the most supporting for object-oriented features [18]. Object-Z introduces the class construct to the conventional Z that encapsulates the state and operations schemas, and allows their inheritance.

VDM is extended in VDM++ [26] with class, object, inheritance, and a formalism feature to specify the methods invocation sequence.

Several studies have been proposed in the literature to combine the formal preciseness of B-Method and the object oriented features of UML. In [27], transformation rules are proposed to translate the UML behavioural diagrams to formal B specifications. This work is extended in [28] to support mapping UML class operations to B operations where a class operation and its related data are mapped to the same B abstract machine. In addition, the automatic derivation from UML behavioural diagrams into B specifications is addressed in [28]. The integration of UML and B is extended in [29] to support the transformation of UML object constraint language OCL into B considering the class invariants, guard conditions in state-machines and the OCL specifications in class operations. In [30], a more extensive account is provided completing the work in [27]-[29] where the transformation rules are formalized and the formal verification is introduced for behavioural elements of UML models into B abstract machines.

UML-B [31]-[34] is a graphical front end of Event-B. It shares similar properties with UML object oriented modelling language, but UML-B has its own meta-model. UML-B is supported by a tool which provides the user with an environment for drawing its diagrams. These diagrams are translated to Event-B in order to be verified using Rodin theorem provers. UML-B offers four types of diagrams which are package diagram, in which contexts and machines are represented with the interconnecting relationships, context diagram where static part of system is defined, class diagram where classes, variables, events and invariants are defined and state machine diagram which represents system state changes when executing transitions.

3. REFINEMENT IN SOFTWARE ENGINEERING

This section reviews some refinement techniques and methods in software engineering context covering different software development lifecycles. This section also compares the techniques and methods by their application level and the refinement proof type. The refinement proof is to prove that the refined model/ specification of the system refines correctly the abstract version.

System models could be represented by state space in [35]. A refinement mapping [35] could be considered between low level specification state space *Sm1* and high level specification state space *Sm2*. State machine behaviour is represented by transitions or steps allowed in different scenarios. Allowed behaviour by *Sm1* is mapped to allowed behaviour by *Sm2*, this research answers the question on how to ensure that the low level specification represented by *Sm1* is a correct implementation of high level one *Sm2*. A complete practical hierarchical specification method has been resulted, and it showed that, under some assumptions and circumstances about specifications, if low level specification *Sm1* is implementing high level specification *Sm2*, the existence of refinement mapping between the two specification levels is guaranteed by adding auxiliary variables.

A refinement is considered as system classes and operations changes during evolution [36]. It has also a concept where features are added incrementally. These features encapsulate individual characteristics, where they are used to distinguish programs among other different related programs. Most systems nowadays are collaborating individual's subcomponents with each other like: client-server architectures and tool-suites such as Microsoft Office. Several tools exist to compose feature refinements which are usually used to generate source code of individual programs. This study introduces AHEAD (Algebraic Hierarchical Equations for Application Design) model to show how step-wise refinement scales to synthesize several programs and non-code system representations and that software could have a mathematical structure represented as a set of equations. One individual program represented by source code is started with, and GenVoca model is used to show that this code representation could be expressed by an equation. Then, AHEAD model is introduced to handle multiple programs and generalize the equational specifications to their multiple representations. The proposed AHEAD model is related with other models like Aspect Oriented Programming and multidimensional separation of concerns. AHEAD model is supported by tools to show the applicability of this study.

Refinement could be considered a way for handling system programming complexity [37]. Better programming tools are needed to overcome the complexity of integrating large systems, so a tool program development system (PDS) to support the overall program production is introduced. This software production is covering the several system lifecycle levels starting from requirement and ending with the implementation and coding. PDS tool is a programming environment supports stepwise refinement allowing changes from high level specifications to be reflected at lower specification levels. Different levels of specifications are not necessarily created from abstract high levels to more concrete lower ones, but the order is not important and any requirement of the system at hand could be reflected directly at its corresponding system level.

As some refinement techniques focuses on specific implementation levels or the overall levels, other studies put the interest solely on software architectural level as base for creating the most concrete level later. Focusing on architecture refinement as in [38] ensures a good level of architectural integrity, consistency and quality. Step-wise

refinement is also considered for software architecture to cope the complexity of architecture conversion process from its abstract version to concrete one. Component-based refinement method, called refinement pattern, is proposed which is a framework for refining architecture. This method concentrates on components refinements with several steps. It starts with defining the architecture style, describing the abstract component which needs refinement, refining the component, and it ends with defining the resulted refined architecture. Refinement pattern uses novel design language π-ARL for architecture refinement considering date, port and component refinements.

Refinement may be used in model driven engineering context for object model [39]. An appropriate notation of object models refinement is discussed in this study. A formal support for model driven object oriented development is introduced in the objective of generation process for software artefacts from structural models and investigating applicability of data and refinement to object models.

Table 1 presents a comparison between the different refinement methods and techniques considering factors like: the software development life cycle where refinement is applied and the type of refinement proof.

Table 1: Refinement methods comparison

Study	Level/ Stage	Refinement Proof
"The existence of refinement mappings." [35]	Modelling- State Machine	Behavioural simulation of low level state machines to high ones
"Scaling step-wise refinement." [36]	Specification(Mathematical) and implementation (source code)	Source code automatic generation from mathematical specification
"A system for program refinement." [37]	several system lifecycle levels	Problem understanding of the system at a specific level
"A component-based method for software architecture refinement." [38]	Architectural level	Each level is decomposed to get a set of components which represents the later concrete level until no more component is decomposable
"Compositionality and refinement in model-driven engineering." [39]	Modelling (Object Models)	Formal proof support

4. REFINEMENT IN FORMAL METHODS

Several refinement methods and techniques are introduced in formal methods. In this section, we present refinement in formal methods (B-Method, Event-B, Z-Method, VDM) and object-oriented formal methods.

4.1. Refinement in B-Method

Refinement in B-Method [2], [10] allows capturing requirements in modelling gradually by a sequence of machines where an abstract machine may be refined by a refinement machine. This may have sets, concrete variables, constants and properties. Also, the refinement machine may include one or several machines.

There are two refinement types: data refinement and algorithmic refinement. In data refinement, new variables may be added to the refinement machine and they are linked to the variables of the abstract machine by gluing invariants. In algorithmic refinement, the operations of the abstract machine may be refined by more deterministic operations in the refinement machine.

4.2. Refinement in Event-B

Refinement allows modelling in Event-B gradually through an ordered sequence of models where each model refines its preceding one [1], [40], as in Figure 1. Two refinement types exist for Event-B: super-position and data refinement.

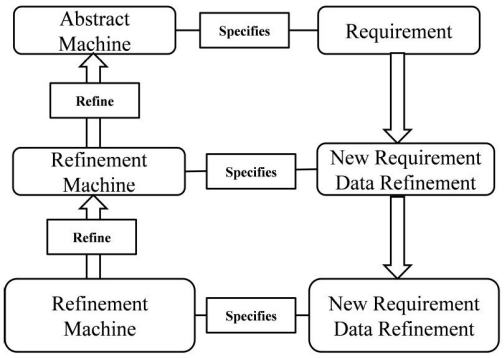

Figure 1: Refinement in Event-B

Super-position or so-called horizontal refinement is to extend the model with new requirements that corresponds to the model spatial extension focusing on the mathematical representation using the set-theoretic notation specifying the state-space invariants and its transitions. The state is expanded by adding new variables, strengthening events guards, adding new guards, and new events. This type of refinement stops when there is no more new requirement to be taken into account in the model.

Data-refinement or so-called vertical refinement is performed when no more new requirements are needed for consideration. The same state space variables and transitions are refined to more discrete details facilitating the model implementation

using some programming languages. In data-refinement, variables could be replaced by new ones where gluing invariants are required to relate the abstract and refined states. An example of data-refinement is refining a variable of the type integer number to a new one of the type natural number. In both super-position, and data refinement, proofs are required to show that the refinement steps do not violate the invariants of the abstract steps. Generally speaking, machines are refined in terms of variables and events while contexts are extended in Event-B models.

During the refinement, the state variables are extended by new super-position or data-refined variables. In the refinement machine, existing events are refined, new events that refines "skip" step are introduced using only the new variables.

Event-B events could be refined by retaining them, renaming or splitting into several cases. In the case of retention or renaming, event parameters could be added or replaced provided that a witness is introduced for every removed parameter. Event guards could be changed or added provided that the event overall guards are not weakened. New event actions may be added provided that they only modify new variables. Existing event actions may be modified correctly provided that they simulate the same behaviour in the abstract event specification. Event splitting is done when more than one event in the refinement specification is refining one abstract event which does not show the detailed cases provided in its refining events.

4.3. Refinement in Z-Method

Refinement in Z [41], [42] includes: data refinement, operations schemas refinement, simulation and functional refinement. An operation schema in Z corresponds to a relation on the states of the specified system. An operation is correctly refined when the relation is correctly refined. A retrieve relation *R* is defined to represent the relationship between abstract and concrete schemas. To decide if *R* is a simulation, operations of the abstract and concrete schemas have to be compared. Functional refinement is a special case that occurs when the relations used in refining Z specifications are functions.

4.4. Refinement in VDM

The refinement in VDM has been introduced in [43]. System development in VDM is a sequence of specifications starting from abstract specifications and gradually following with more concrete specifications. The concrete specification is a valid refinement of the abstract one if refinement proof obligations hold. The refinement in VDM is defined as data reification in which data objects are refined to the level of the machine or the language constructs and at this stage operation decomposition is carried out.

4.5. Refinement in Object-Oriented Formal Methods

Refinement methods and techniques are introduced for Object-Z in [42]. A method of refinement is introduced in [44] for the integration notation Object-Z and CSP (Communicating Sequential Processes). The method has two approaches: First is the failures approach, and second the state-based approach. The former is based on CSP refinement where the failures and divergences are calculated for two processes/classes *P1* and *P2* and it is said that *P2* is a refinement of *P1* if $failuresP2 \subseteq failuresP1$ and $divergencesP2 \subseteq divergencesP1$. The state-based approach enables the refinement to be verified at the specification level when calculating the failures is a difficult task. Two simulation-based refinement relations are introduced, called upward and downward simulations, where an object-Z class *C* is a simulation of a class *A* if a retrieve relation exists such that every abstract relation in *A* is recast in a concrete one in *C*. The work in [44] does not consider the situation where classes contain objects as state-variables. This is extended and considered in [45]. Two refinement techniques are proposed in [45] for Object and operation in Object-Z specifications changing their granularity. This provides flexibility when refining specifications by supporting refinement to various language notions. A class *C* may be split during refinement into interacting classes *C1,..Cn*. A class operation *C.Op* may be split to a sequence of concrete operations *C.Op1,...C.Opn*. It is either the case that one concrete operation *C.Opn* refines the abstract one *C.Op*, and the rest refines the stuttering step called skip, or all the concrete operations *C.Op1,...C.Opn* refines *C.Op*. A methodology for class composition refinement is introduced in [45]. The methodology considers the conjunction of operations from different classes supporting the refinement of individual classes compositionally by isolating constraints that couple the classes.

Refinement methods and techniques are proposed for VDM++. In [47], a method of sub-typing and sub-classing is proposed for VDM++. In [48], a method for annealing and data-decomposition is introduced. In this work, the main class may be decomposed into communicating classes. In [49], VDM++ is extended by VDM-R since VDM++ is limited in terms of the refinement consistency checking called inter-specification consistency verification. In this work, the relationships between specifications are formally verified by VDM-R annotations.

UML-B refinement has been introduced in [50], [51] and is based on Event-B refinement notion. Class and state machine refinement has been covered in [50]. Two main features exist for UML-B refinement: Data-refinement and event-refinement. Data-refinement is reflected in class-types, classes, attributes, states and variables. Class-types are introduced in contexts and retained in the extended context, and only the new features that are added to the class type are introduced. Classes in refinement may be refined (retained), newly introduced (super-position), or data-refined where the gluing invariants are necessary. Attributes in refinement

maybe inherited, newly introduced (super-position), or data-refined.

UML-B events refinement is similar. UML-B events are represented by: class events, state machines transitions and machine events. A class event may be retained, refined or split. Class event refinement may be done by adding or replacing event parameters, guards or actions to perform UML-B class data refinement. It is not necessary to preserve the containment feature for class event and they could be moved to different classes or to the machine level and the witness for the previous lost class instance parameter must be introduced. Considering the abstract class *C* that has class events: *ce1*, *ce2*, *ce3*, *ce4* and *ce5*. Five different class event refinement cases exist: First, *ce1* could be introduced in the refined class *RC* as refining event of abstract *ce1* (*ce1* refines *ce1*). Second, new namely event ce6 may introduced in *RC* as refining event of *ce2* (*ce6* refines *ce2*). Third, *ce3a* and *ce3b* class events could be introduced in *RC* as individual cases that refine the abstract event *ce3* (*ce3a* refines *ce3*, *ce3b* refines *ce3*). Fourth, *ce4* could be refined and transferred to another class *D*, but a witness must be provided in this case for the lost class *RC* instance parameter (Class *D* event *ce4* refines *ce4*). Fifth, class event *ce5* could be transferred to machine level as machine event, where an event parameter which reveals the event belonging of the abstract class and a witness for replacing the lost parameter are introduced (Machine event *ce5* refines *ce5*). State machine transitions are refined similarly as class events, but with a small difference in that transition source state or target cannot be modified since this is related to state transition guards and actions respectively and must consistent with their abstract version. State machines could be refined by detailed elaborating models. State transition may be refined by splitting in which several transitions representing the abstract state transition individual cases may be introduced. A state may be elaborated by a nested state machine which represents a more detailed behaviour of its abstract version.

5. CONCLUSION

Refinement is considered as obtaining a better version of software than the original one during the development process. This is because refinement has been known as a familiar technique and methodology to deal with the changing and new requirements and to provide better concrete versions of the system artefacts at hand.

Formal methods in the software engineering discipline allow the design, modelling, verification, and maintenance of hardware and software systems. Formal methods introduce preciseness, remove ambiguity in specifications, and support the verification of requirements and design properties.
Several refinement methods and techniques have been introduced in formal methods and software engineering. Refinement has to be understood carefully in the context of formal specification and verification. In this article, we provide an overview of some formal methods and refinement methods and techniques in the context of software engineering, formal methods and some object oriented formal methods. We believe that this survey sheds a light on the research direction in regards to the refinement of formal methods. This survey also helps formal methods practitioners and users in observing and understanding the advantages and limitations of refinements methods and techniques of various studies formal methods. Accordingly, they can decide which formal method is to be used in modelling systems via refinement or which formal method is to be extended with new concepts and notions.

In this article, we focus on refinement in formal methods at the modelling level. We intend to extend this work to cover more formal methods at other development levels such as implementation.

6. ACKNOWLEDGEMENTS

Thanks to the Faculty of Computer Science and Information Technology, UPM and the MOHE for the financial support via Fundamental Research Grant Scheme, Project Code: 08-02-13-1368FR.

REFERENCES

[1] J. R. Abrial. ***Modeling in Event-B: System And Software Engineering***, Cambridge University Press, 2010. https://doi.org/10.1017/CBO9781139195881

[2] J. R. Abrial. ***The B-book: Assigning Programs To Meanings***, Cambridge University Press, 2005. https://doi.org/10.1109/INFOCT.2019.8711369

[3] S. P. Nanda and E. S. Grant. **A survey of formal specification application to safety critical systems,** in *Proc. 2019 IEEE 2nd International Conf. on Information and Computer Technologies (ICICT)* IEEE, March 2019, pp. 296-302.

[4] *J.P. Bowen.* ***Formal Specification and Documentation Using Z: A Case Study Approach***, London: International Thomson Computer Press, 1996.

[5] D. Bjørner. **The Vienna Development Method (VDM),** in *Mathematical Studies of Information Processing***,** Lecture Notes in Computer Science, vol. 75, E.K. Blum, M. Paul and S. Takasu, Ed. Berlin, Heidelberg: Springer, 1979, pp. 326-359. https://doi.org/10.1007/3-540-09541-1_33

[6] S. J. Garland, J. V. Guttag and J. J Horning. **An Overview of Larch.** in *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Berlin, Heidelberg: Springer, 1993, pp.329-348. https://doi.org/10.1007/3-540-56883-2_15

[7] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and, W. Dietl. ***JML Reference Manual***, 2008.

[8] A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.). ***Refinement Techniques in Software***

Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004, November 23-December 5, 2004, Revised Lectures, vol. 3167. Recife, Brazil: Springer, 2006. https://doi.org/10.1007/11889229

[9] E. W. Dijkstra and E. U. Informaticien. ***A discipline of programming***, vol. 1, Englewood Cliffs: Prentice-hall, 1976.

[10] K. Lano. ***The B language and method: a guide to practical formal development***, London: Springe-Verlag, 1996. https://doi.org/10.1007/978-1-4471-1494-9

[11] R. J. Back and R. Kurki-Suonio. **Decentralization of process nets with centralized control**, *Distributed Computing*, vol. 3, no. 2, pp. 73-87, 1989.

[12] J. R. Abrial. **From Z To B And Then Event-B: Assigning Proofs To Meaningful Programs**, in *International Conference on Integrated Formal Methods*, Berlin, Heidelberg: Springer, 2013, pp. 1-15.

[13] J. P. Bowen. **Comp. specification. Z and Z FORUM frequently asked questions**, in *International Conference of Z Users*, Berlin, Heidelberg: Springer, 1998, September, pp. 407-416. https://doi.org/10.1007/978-3-540-49676-2_25

[14] J. P. Z. Bowen,: **A formal specification notation**, *in Software specification methods*, London: Springer, 2000, pp. 3-19.

[15] C. B. Jones. **Scientific decisions which characterize VDM**, in *Formal Methods, FM'99*, J.M. Wing, J. Woodcock and J. Davies, Ed. Berlin, Heidelberg: Springer, 1999, pp. 28-47.

[16] C. B. Jones. **Systematic Software Development Using VDM**, Vol. 2, Englewood Cliffs: Prentice Hall, 1990.

[17] V. S. Alagar and, K. Periyasamy. **Specification of Software Systems**. Springer Science & Business Media, 2011.

[18] S. Stepney, R. Barden, R. and D. Cooper. **A survey of object orientation in Z**. *Software Engineering Journal*, vol. 7, no.2, pp. 150-160, 1992.

[19] G. Smith. **An object-oriented approach to formal specification,** Ph.D. dissertation, University of Queensland, 1992.

[20] R. Duke, G. Rose and G. Smith. **Object-Z: A specification language advocated for the description of standards**, *Computer Standards & Interfaces*, vol. 17, no. 5-6, pp. 511-533, 1995. https://doi.org/10.1016/0920-5489(95)00024-O

[21] G. Smith. **An Object-Oriented Development Framework for Z**, in *Z User Workshop, Cambridge 1994*, J.P. Bowen, Ed. London: Springer, 1994.

[22] K. Lano. **Z++, An Object-Orientated Extension To Z**, in *Z User Workshop, Oxford 1990*, J.E. Nicholls, Ed. London: Springer, 1991. https://doi.org/10.1007/978-1-4471-3540-1_11

[23] A.J. Alencar and J.A. Goguen (1991) **OOZE: An Object Oriented Z Environment**. In: *America P. (eds) ECOOP'91 European Conference on Object-Oriented Programming. ECOOP 1991*. Lecture Notes in Computer Science, vol. 512. Berlin, Heidelberg: Springer, 1991.

[24] A. Hall. **Using Z as a specification calculus for object-oriented systems**, in *VDM '90 VDM and Z — Formal Methods in Software Development. VDM 1990*, Lecture Notes in Computer Science, vol. 428, D. Bjørner, C.A.R. Hoare and H. Langmaack, Ed. Berlin, Heidelberg: Springer, 1990, pp. 290-318.

[25] D. Carrington. **ZOOM Workshop Report**, in *Z User Workshop*, Workshop in Computing, J. E. Nicholls, Ed. London: Springer-Verlag, 1992, pp. 352-364. https://doi.org/10.1007/978-1-4471-3203-5_16

[26] E. Durr and J. Van Katwijk. **VDM++, a formal specification language for object-oriented designs**, in *Proc. Computer Systems and Software Engineering*, The Hague, Netherlands, 1992, pp. 214-219.

[27] H. Ledang and J. Souquieres. **Formalizing UML behavioral diagrams with B**, in *Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Tampa Bay, Florida, USA, October, 2001.

[28] H. Ledang and J. Souquières. **Modeling class operations in B: application to UML behavioral diagrams**, in *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 289-296.

[29] H. Ledang and J. Souquières. **Integration of UML and B specification techniques: Systematic transformation from OCL expressions into B**. in *Proc. Ninth Asia-Pacific Software Engineering Conference*, 2002, pp. 495-504.

[30] N. T. Truong and J. Souquieres. **Verification of behavioural elements of UML models using B**, in *Proc. 2005 ACM symposium on Applied computing, 2005*, pp. 1546-1552. https://doi.org/10.1145/1066677.1067024

[31] C. Snook and M. Butler. **UML-B: Formal modeling and design aided by UML**, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, pp. 92-122, 2006.

[32] C. Snook and M. Butler. **UML-B and Event-B: an integration of languages and tools**, in *Proc. IASTED International Conference on Software Engineering (SE '08)*, Claus Pahl (Ed.), California, USA, 2008, 336-341.

[33] C. Snook and M. Butler. **UML-B: A plug-in for the Event-B tool set**, in *Abstract State Machines, B and Z. ABZ 2008*, Lecture in Computer Science, vol. 5238: Springer-verlag, 2008.

[34] C. Snook, I. Oliver and M. Butler. **The UML-B profile for formal systems modelling in UML**, in *UML-B specification for proven embedded systems design*, Boston: Springer, 2004, pp. 69-84.

[35] M. Abadi and L. Lamport. **The existence of refinement mappings**, *Theoretical Computer Science*, vol. 82, no. 2, pp. 253-284, 1991.

[36] D. Batory, J. N. Sarvela and A. Rauschmayer. **Scaling step-wise refinement**. *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355-371, 2004. https://doi.org/10.1109/TSE.2004.23

[37] T. E. Cheatham, J. A. Townley and, G. H. Holloway. **A system for program refinement,** in *Proc. 4th international conference on Software engineering*, NJ, USA: IEEE Press, 1979, pp. 53-62.

[38] J. Zhang, X. Ban, Q. Lv, J. Chen and D. Wu. **A component-based method for software architecture refinement,** in *Proc. 29th Chinese Control Conference*, Dalian, China, July 2010, pp. 4251-4256.

[39] J. Davies, J. Gibbons, D. Milward and J. Welch. **Compositionality and refinement in model-driven engineering.** in *Formal Methods: Foundations and Applications. SBMF 2012*. Lecture Notes in Computer Science, vol. 7498, Berlin, Heidelberg: Springer, 2012, pp. 99-114.

[40] J. R. Abrial and S. Hallerstede **Refinement, decomposition, and instantiation of discrete models**: **Application to Event-B.** *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1-28, 2007.

[41] J. Woodcock and J. Davies. ***Using Z: Specification, Refinement and Proof***, UK: Prentice Hall, 1996.

[42] J. Derrick and E. A. Boiten. ***Refinement in Z and Object-Z: foundations and advanced* applications**, London: Springer-Verlag, 2014.

[43] C. B. Jones. ***Systematic software development using VDM***, vol. 2, Englewood Cliffs: Prentice Hall, 1990.

[44] G. Smith, and, J. Derrick. **Refinement and verification of concurrent systems specified in Object-Z and CSP**. In *Proc. First IEEE international conference on Formal engineering methods*, Hiroshima, Japan, 1997, pp. 293-302.

[45] J. Derrick and E. Boiten. **Refinement of objects and operations in Object-Z.** in *Formal Methods for Open Object-Based Distributed Systems IV. FMOODS 2000. IFIP Advances in Information and Communication Technology*, vol. 49, S.F. Smith and C.L. Talcott, Ed. Boston, MA: Springer, 2000, pp. 257-277. https://doi.org/10.1007/978-0-387-35520-7_13

[46] T. McComb and G. Smith. **Compositional class refinement in Object-Z.** In *Formal Methods. FM 2006*, Lecture Notes in Computer Science, vol. 4085. J. Misra, T. Nipkow and E. Sekerinski, Ed. Berlin, Heidelberg: Springer, 2006, pp. 205-220.

[47] K. Lano and S. J. Goldsack. **Refinement, Subtyping and Subclassing in VDM++.** in *Theory and Formal Methods*, pp. 341-363, 1994.

[48] S. J. Goldsack and K. Lano. **Annealing and data decomposition in VDM.** *ACM Sigplan Notices*, vol. 31, no. 4, pp. 32-38, 1996.

[49] Y. Kawamata, C. Sommer, F. Ishikawa and S. Honiden. **Specifying and checking refinement relationships in VDM++**, in *Proc. Seventh IEEE International Conference on Software Engineering and Formal Methods*, Vietnam, 2009, pp. 220-227.

[50] M.Y. Said, M. Butler and C. Snook. **Language and tool support for class and state machine refinement in UML-B.** in *FM 2009: Formal Methods. FM 2009*, Lecture Notes In Computer Science, vol. 5850, A. Cavalcanti and D.R. Dams, Ed. Berlin, Heidelberg: Springer, 2009, pp 579-595. https://doi.org/10.1007/978-3-642-05089-3_37

[51] M.Y. Said, M. Butler and C. Snook. **A method of refinement in UML-B,** *Software & Systems Modeling*, vol. 14, no. 4, pp. 1557-1580, October 2015. https://doi.org/10.1007/s10270-013-0391-z