Scientia Iranica D (2015) 22(3), 1001–1017

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
www.scientiairanica.com

A survey on formal, object-oriented program development approaches

M. Najafi, H. Haghighi* and T. Zohdi Nasab

Faculty of Electrical and Computer Engineering, Shahid Beheshti University G.C., Tehran, Iran.

Received 6 September 2014; received in revised form 19 March 2015; accepted 15 June 2015

KEYWORDS
Formal program development;
Object-orientation;
Formal specification;
Object-Z;
VDM;
VDM++;
B;
Event-B;
UML-B.

Abstract. Due to the popularity of object-oriented programming approaches, there is a growing interest in utilizing object-oriented concepts, such as encapsulation and reuse, when applying formal methods. The main contribution of this paper is to review and compare existing formal methods to develop object-oriented programs from formal specifications. The secondary contribution is providing a comparison between widely used object-oriented formal specification languages. The results of this paper can be utilized by researchers wishing to know what open problems are outstanding in the areas of formal, object-oriented specification and program development. Our findings are also useful for those who are looking for proper specification languages and program development methods to specify and develop object-oriented programs formally. In addition, the provided criteria are suitable for evaluating numerous object-oriented formal specification languages that are under development, either by extending existing formal approaches or formalizing informal OO-methods. As one consequence of this work, it can be mentioned that among formal specification languages, OZ and VDM++ support OO concepts more strongly in comparison to VDM++ and UML-B. Program development methods based on OZ have less tool support. Finally, most proposed methods for formal, object-oriented program development have been evaluated using only case studies, rather than employing formal approaches.

1. Introduction

In the late 1960s, formal methods based on mathematics were proposed as an option for providing software reliability [1]. Formal methods are used to uncover ambiguity, incompleteness and inconsistency in a system. These methods can be used at any stage of software development, from the initial statement of a customer's requirements to system implementation and verification. Usually, these methods are introduced to the software life-cycle by adding the formal specification stage to the stages of software projects. At this stage, we describe WHAT has to be done in the final software, instead of HOW it has to be done.

Formal program development is a process producing software program code in relatively high level programming languages, such as C and Java, from a given formal specification of the software. A formal program development process has two stages. First, programs are specified using formal specification languages, and are then developed from formal specifications applying one of the formal program development methods. Formal program development methods are categorized as animation, refinement, or constructive approaches.

There are two approaches which have been pro-

*. *Corresponding author. Tel.: +98 21 29904190; Fax: +98 21 29904181*
E-mail addresses: M.Najafi@sbu.ac.ir (M. Najafi); h_haghighi@sbu.ac.ir (H. Haghighi); t.Zohdinasab@sbu.ac.ir (T. Zohdi Nasab).

posed for animating a formal specification: direct execution and rapid prototyping. Direct execution means that the formal specification statements are executed directly and normally by interpretation. Rapid prototyping refers to any method that directly converts specifications to programs in a high-level language by using ad hoc rules [2,3]. By a refinement technique, one applies a set of well-defined refinement rules to alter a software specification into a more concrete mathematical model, i.e. its refinement, while maintaining the abstract properties involved in the initial specification [4]. Constructive approaches based on constructive mathematics [5] derive programs from correctness proofs of specifications.

Many researchers are interested in studying the OO paradigm at specification and program development levels because object orientation is well adapted to specifying interesting high level properties of dependable systems. Precisely, by formal, object-oriented program development approaches, we refer to those approaches presented to refine formal specifications into object-oriented code, or animate formal specifications with object-oriented code. While sharing almost the same objective, the existing related publications adopt different approaches for formal, object-oriented program development. They also use different formal specification languages at the beginning stage of the development process.

Najafi and Haghighi [6] presented a brief overview on some of the existing formal approaches to develop object-oriented programs from Object-Z, VDM (Vienna Development Method), VDM++, Event-B, and UML-B specifications. Furthermore, they provided a comparison between Object-Z, VDM++, and UML-B.

This paper presents a more complete review, which relies on the following advantages in comparison to our previous study [6]:

1. In this study, a larger set of criteria has been used in order to compare Object-Z, VDM++, and UML-B, as well-known object-oriented, formal specification languages. For instance, "polymorphism", "subtyping", "multiple inheritances" and "multiple subtyping" are new criteria considered in the present study.
2. A brief description of existing formal methods to develop object-oriented programs from Object-Z, VDM, VDM++, B, Event-B, and UML-B specifications has been provided.
3. Formal methods to develop object-oriented programs from Object-Z, VDM, VDM++, B, Event-B, and UML-B specifications have been compared according to a larger set of criteria. For instance, we have considered "interactivity", "similarity degree", "validation approach", and "refinement style" as new criteria.
4. We have summarized the strengths and weaknesses of formal methods to develop object-oriented programs based on the given criteria.

Our methodology to perform this survey is as follows:

1. The following eligibility criteria were first determined to include papers, theses, and books published in English:
   - Performing a survey of formal specifications or formal program development;
   - Proposing a method or tool under formal specifications, formal program development or code generation, regarding OO approaches and viewpoints.
2. To search potentially eligible research and find appropriate papers to satisfy the specified criteria, databases such as ACM digital library, arXiv, CiteSeer, Scientific Literature Digital Library, IEEE/IET Electronic Library, and Google scholar were used.
3. In a duplicate and independent manner, title and abstract screening, and full text screening were undertaken. Irrespective of discrepancies, all studies selected at a title and abstract level were included for the full text screening. Then, the full text of studies was investigated and relevant studies were selected, according to the mentioned eligibility criteria, and others were disregarded.
4. Selected research was categorized into two overlapped categories:
   - Research related to formal specification;
   - Research related to formal program development or code generation.
5. Languages, methods, and tools mentioned in each category were investigated and compared, based on the criteria introduced in the next subsections.
6. The strengths and weaknesses of formal methods to develop object-oriented programs were summarized, based on the performed investigation and comparison.

Section 2 introduces and compares well-known object-oriented formal specification languages. Section 3 reviews existing formal methods to develop object-oriented programs from Object-Z, VDM, VDM++, B, Event-B, and UML-B specifications. Section 4 compares these methods according to a set of criteria. Finally, the last section is devoted to conclusions and some directions for future studies.

2. Object-oriented formal specification languages

Many object-oriented specification languages, i.e. languages which cover known object-oriented concepts, have been so far developed, but we only focus on well-known languages that are based on the first-order predicate logic and set-theory. They are Object-Z, UML-B, and VDM++. Object-Z [7,8] was developed by researchers at Queensland University as an object-oriented extension of Z. Object-Z models systems as collections of independent classes and objects [9]. The structure of class schema, as a main new construct of Object-Z, is described in [7].

VDM++ [10,11] is an extension of VDM, which was developed since 1992, but the current notation of VDM++ is a part of the Afrodite project [11]. Using this language, one can model OO systems that have parallel and real-time behavior [12]. The major new construct in VDM++, in comparison to VDM, is a class whose structure is explained in the CSK Corporation [12]. Also, there is a further construct in VDM++ specifications that introduces the notion of inheritance and multiple inheritance [12].

Based on UML [13], Snook et al. introduced UML-B [14-16] as a graphical formal specification language. The old version of UML-B [14,15] relies on the B Method, but its current version [16] relies on Event-B. Also, UML-B was implemented by the Eclipse Modeling Framework (EMF) as a plug-in for RODIN toolkits. UML-B provides four kinds of diagram: package, class, context, and statemachine diagrams [16].

The package diagram indicates the relationships between machines and contexts. In a machine, one can define classes, variables, events, statemachines and invariants. Static data are modeled in the context part [17]. More precisely, the context diagram is drawn as a class diagram but has constant data represented by Class Type, attributes, constants and association. The dynamic part is modeled in a Class diagram and used to describe a machine [17]. Classes may contain a set of attributes, events, statemachines and invariants. A statemachine is used to model the behaviors of a system [17]. It can be defined in two ways: within a corresponding class and as a statemachine [17]; a statemachine is defined within a class in order to explain the changes in the class states. In contrast, if an object has to be represented by a statemachine, a machine statemachine is utilized.

2.1. Comparison of object-oriented formal specification languages

In addition to "polymorphism" and "correspondence with the typical object-oriented style", which we proposed as two special criteria, a set of criteria presented in [18] are used in Table 1 to demonstrate a comparison between UML-B, Object-Z and VDM++ (for more description of these criteria, see [18]). It is worth mentioning that we added column "UML-B" to Table 1, and the rest come from [18]. In Table 1, "Y" means that the related language bears that feature and "N" stands for lack of that feature.

According to Table 1, we conclude that:

- Unlike UML-B, Object-Z and VDM++ have specification styles which correspond to constructs of typical object-oriented programming languages;
- Object-Z and VDM++ support important concepts of object-orientation, such as "object" and "multiple inheritance" which have been supported weakly in UML-B.

3. Formal development of object-oriented programs

Formal methods are described to develop object-oriented programs by categorizing them into animation and refinement techniques. It must be stated that no constructive approach with the aim of developing object-oriented programs from formal specifications has to date been proposed.

3.1. Animation

As mentioned in Section 1, we consider both categories of approaches that have been proposed for animating a formal specification: direct execution and rapid prototyping. Direct execution means that the formal specification statements are executed directly, normally by interpretation. Rapid prototyping refers to any method which directly converts specifications to programs in a high-level language by using ad hoc rules.

To investigate animation techniques, we classify them based on their formal specification languages in the following subsections. However, prior to investigating these techniques, we should mention that no animation technique for mapping Event-B and UML-B specifications into object-oriented code has been proposed to date.

3.1.1. Object-Z

Rafsanjani and Colwill [20] introduced rules to translate constants, state variables, inheritance, multiple inheritance, and operations into C++ constructs. Also, they considered a null constructor, a copy constructor, a destructor, and an assignment operator for each class of C++. Johnston and Rose [21] proposed guidelines for manual conversion of class schema, visibility list, inheritance, type definition, state schema, initial state schema and operations to C++ constructs. Fukagawa et al. [22] augmented the work of Rafsanjani and Colwill [20] by considering constructor for types of constants and template classes for generic parameters.

Table 1. Comparing object-oriented specification languages.

Criterion	Object-Z	VDM++	UML-B
Encapsulation	Y	Y	Y
Object	Y	Y	Y: It does not support object diagram
Object identity	Y	Y	Y
Objects data structures	Y	Y	N
Inheritance	Y	Y	N: Inheritance represents sub-typing of a class
Sub-typing	Y	Y	Y [15]
Multiple inheritance	Y	Y	N
Mutiple sub-typing	Y	Y	N [4,16]
Inheritance $\neq$ sub-typing	Y	Y	N: It only supports sub-typing concept
Polymorphism	Y	Y [19]	N
Classes as templates	Y	Y	Y
Classes as object	N	N	N
Collection of objects	Y	Y	N
Genericity or parameterization of classes	Y	N	N
Intra-object concurrency	Y	Y	Y
Semantics	Y	Y	Partially Y: We have only found references which partially define the semantics of the current version of UML-B
Calculus	Y	Y [11]	Y
Correspondence with the typical object-oriented style. (Compatibility of constructs of the language with typical object-oriented constructs such as class, attributes, methods, inheritance and threads.)	Strong: The main construct of the formalism is the class which contains visibility list, inherited classes, local definitions, state, initial state and operations.	Strong: The main construct of the formalism is the class which contains values, instance variables, methods (operations and functions), threads, traces, synchronizations and inheritance clause.	Weak: UML-B has more than one major construct (instead of only having class) such as machine, context and statemachines which do not have equivalent constructs in the typical object-oriented style. Also, the class construct contains attributes, events, statemachines, invariants and theorems.

Griffiths [23] proposed a method which maps Object-Z specifications to Eiffel code. Ramkarthik and Zhang [9] developed a tool for animation of Object-Z specifications to Java code. This tool consists of a main control system, a graphical user interface, XML manager, and Java skeletal code generator. XML manager first generates a XML document for an Object-Z specification, and then a Java skeletal code generator generates Java code from developed XML documents. In this work, general mapping rules are described for

class schema, class constants, class variables, class invariants, initialization schema, operation schema and visibility lists.

Ni and Zhang [24] developed a tool for conversion of Object-Z specifications to Spec# code. Their tool provides a GUI which accepts and facilitates formal specification in Object-Z, converts the Object-Z class schemas to XML representations, and generates a Spec# skeletal code through processing of the XML representation. Spec# skeletal code generated through this tool includes state variables with their types, class constants and initial schema, in addition to class invariants, and all inputs, preconditions and postconditions for operations. Wang et al. [25] proposed transformation mechanisms for conversion from Object-Z specifications to Java code, which support the mapping of class, inheritance, polymorphism, and object.

Najafi and Haghighi [26] described some mapping rules from Object-Z specifications to C++ code. Their method supports the mapping of formal generic parameters, visibility list, local definitions, class union, object aggregation, type definitions, object, state schema, initial state schema, operations, inheritance, multiple inheritance, object containment, and promotion. In another publication, Najafi and Haghighi [27] presented general ideas (not specific rules and related code) for mapping some new constructs and cases of Object-Z specifications into C++ that have not been considered in [26]. Najafi and Haghighi [28] presented another much more comprehensive version of their method, which has advantages, such as covering more Object-Z constructs and proposing mapping rules in a much more detailed way in comparison to the previous work. In addition, they provided templates for constructors and destructors that have not been considered in any previous work. Finally, Najafi and Haghighi [29] presented new work that describes mapping rules formally and proves their correctness formally, too.

3.1.2. VDM and VDM++

Jackson [30] presented a method for systematic development of sequential Ada programs using VDM. His approach is based upon using the facilities of Ada for supporting parameterized abstract data types to implement the primitives of the VDM specification language. Chedgey et al. [31] described the use of VDM in the context of development of software to be targeted at the Ada programing language. Moulding and Newton [32] investigated the formal refinement of a VDM specification to an Ada implementation. Also, O'Neill et al. [33] proposed a semiautomatic translation from VDM specification language to Ada, including the translation of composite types, domain equations, and expressions.

Lou [34] described a methodology for deriving C++ implementations from VDM specifications. To derive an object-oriented design from a given VDM specification, the methodology tries to link classes, their attributes and member functions in the design, with data types, variables, and operations in the specification, respectively. This process is independent from target object-oriented language. There are four stages in the mentioned transformation: identifying the classes in the design, identifying the attributes within each class, deriving member functions for each class and deriving relationships between the classes. Lou [34] then proposed rules for deriving C++ code from the obtained object-oriented designs.

In addition, code generators exist that convert VDM specifications to Smalltalk and Ada95 code [35]. Albalooshi and Long [36] proposed a software development environment that supports transformation between VDM and Ada. This automatic transformation supports the mapping of VDM types, value definitions, state variables, and methods.

Chartan and Kans [37] proposed a method that maps VDM specifications to Java code. It supports the mapping of state, value clauses, state clauses, initialization clauses, invariants, operations, sets, sequences, composite objects, maps and operators. The CSK group [38] developed a C++ code generator for VDM specifications. Their code generator supports approximately 95% of all VDM++ constructs, such as classes, types, functions, operations, instance variables, values, expressions, and statements.

Besides work that maps VDM specifications into object-oriented code, there is also work that translates VDM++ specifications to object-oriented code. Bousquet [39] describes translation rules which map VDM++ specifications to Ada95 code. In addition, the CSK group has developed VDMTools [40,41] for mapping VDM++ specifications to Java and C++ code. They support the mapping of class, type definitions, inheritance, function and operation definitions, instance variables, value definitions, expressions, statements and class members.

3.2. Refinement

We categorize refinement techniques into approaches that refine specifications to designs, and methods that refine designs to code. The former is also known as refactoring, which is a process of extending a specification to contain design elements [42]. Of course, refactoring is a technique which has long been used by programmers to improve the design of their code once it became unreadable. In this way, refactoring is the term given to the process of remodeling object-oriented software to improve an existing design whilst preserving its behavior [43]. At the formal specification level, however, refactoring means the use of more

general rules, similar to refactoring rules presented by Fowler [43] to introduce designs from specifications rather than improve existing designs.

3.2.1. Refinement from specification to design

We categorize this group of refinement techniques based on their formal specification languages as follows. We only review those refinements from specification to design whose corresponding refinements from design to object-oriented code have been also proposed. Hence, we do not review existing refinements from Z or VDM specifications to design, because their refinements from design to object-oriented code have not yet been proposed.

3.2.1.1 Object-Z

Derrick and Boiten [44,45] proposed a rule, called downward simulation, to refine one class to another class without considering object references. They also proposed refinement of one class to another class in the case of decomposing a class to multiple interacting classes (i.e., by considering object references). More precisely, in comparison to their former work, their later work uses instantiated objects in retrieve relations. However, the needed proof obligations in the two approaches are the same. Smith [46] described a process for refinement of the value semantics of classes (a class is denoted as a set of values representing its objects) to the reference semantics of classes (a class is denoted as a set of pointers to values representing its objects).

McComb [47] proposed "annealing" and "coalescence" rules as refactoring rules for Object-Z specifications. The former enables us to decompose one class into two, and the latter allows two classes, A and B, to be replaced with one class, C. Although he proves that these two rules are behavior preserving, they are not powerful enough to derive MVC (Model-View-Controller), as the observer pattern cannot be implemented. The reason for this is that the two rules provide structure modification by reorganizing state information, operations, and classes, but offer no means for adding redundancy to a system or expanding a system at a structural level. Therefore, McComb and Smith [48] have introduced a new behavior preserving rule, called "reflection", which adds redundancy to a system or expands a system at a structural level. Also, they [49] proposed a compositional class refinement, which is introduced in order to overcome the limitation where coupling constraints between classes make class refinement non-compositional. Moreover, Ruhroth [50] described an approach to transfer refactoring techniques in programming languages to formal methods through introducing an "Extract method" rule. He proposed a schema for the correctness proof of refactoring rules.

In another publication, McComb and Smith [42] proposed the following four refactoring rules:

1. Introducing generic parameter: replaces instantiated types with new formal parameters;
2. Introducing polymorphism: creates a union of classes through dividing the behavior of a class into separate classes;
3. Introducing inheritance: assists in building an inheritance hierarchy from existing classes;
4. Introducing instances: performs a similar function to the annealing rule; however, it overcomes the restriction of the annealing rule which is limited to introducing only one instance of a new class into the specification. Thus, having this rule, object construction and disposal no longer introduce challenges.

McComb and Smith [51] present a minimal set of refactoring rules, namely "introduce generic parameters", "introduce inheritance" and "introduce polymorphism", and show that these rules, along with compositional class refinement and annealing, make designing in Object-Z completely possible. Furthermore, regarding references to objects of a class that is being refined, McComb and Smith [52] have shown how an arbitrary number of object instances can be introduced into a specification.

Liu and Zhu [53] present a set of refactoring rules which are more fine-grained than the refactoring rules presented by McComb and Smith. These rules are renaming, moving, removing entities (e.g., classes, variables, operations and parameters), adding new entities, replacing expressions with equivalent expressions, refactoring (with generic entities, generalization and specialization), separating qualifiers, simplifying expressions, explaining literal and simplifying schemas.

3.2.1.2 VDM++

Lano and Goldsack [54] proposed refinement, subtyping and subclassing in VDM++. Also, Goldsack et al. [55] described how verification, as understood in VDM, can be applied to VDM++. In another publication, Goldsack and Lano [56,57] built upon [58], which introduced data decomposition in VDM specifications, to formalise annealing for decomposing classes in VDM++. Since they concentrated on invariant distribution in the decomposition process, they did not propose a complete method for object-oriented design based on formal specifications. In this work, annealing is proposed in two forms. In both forms, a main class is divided into classes one of which is the client and the rest of which are servers. However, in one form, the client maintains references of instances of servers, while in the other, servers will be held in one class through multiple inheritance.

3.2.1.3 B

In B, refinement of an abstract machine is described using a REFINEMENT part. The structure of REFINEMENT is similar to the abstract machine structure with two differences: It does not have parameters and only contains SEES relations. Suppose that we have an abstract machine, M, and its refinement, N. The following properties must be satisfied in order to say that N is a refinement of M [59]:

- There is a composition of abstract and real states which satisfies the refinement relationship (the correspondence between abstract and real states) and invariants of the abstract machine;
- Real initial state is a refinement of abstract initial state under the conditions defined in both abstract and refinement machines;
- If we suppose that operation op in abstract and refinement machines is defined as follows, then, each execution of $Def_{op,N}$ (when $Pre_{op,M}$ is satisfied) has a corresponding execution of $Def_{op,\,M}$.

$y \leftarrow$ op (x) =	$y \leftarrow$ op (x) =
PRE $Pre_{op,\,M}$	PRE $Pre_{op,\,N}$
THEN $Def_{op,\,M}$	THEN $Def_{op,\,N}$
END	END

3.2.1.4 Event-B

Métayer et al. [60], and Abrial and Hallerstede [61] proposed the following refinements in Event-B:

1. Extending the list of state variables;
2. Adding new carrier sets and new constants to existing sets and constants;
3. Refining abstract events into corresponding concrete events;
4. Adding new events.

Abrial et al. [62] showed that the proposed method for refining events is not always possible in the development of large systems and, thus, applied some non-deterministic actions to preserve the invariant instead of refining events. Also, Métayer et al. [60] introduced the concept of generic instantiation. More precisely, a development "M" (i.e., a set of machines and their contexts) is said to be generic if it is parameterized by the carrier sets "s" and the constants "c" that have been accumulated in its contexts. Now, it is possible to instantiate development "M" by instantiating sets "s" and constants "c". In addition, to provide reusability through instantiation, the generic instantiation is also introduced to construct large models more easily.

Abrial [63] has described an Event-B development process and developed a tool on the Rodin Platform which supports Event-B refinement. Butler et al. [64] categorized Event-B refinement into feature augmentation (the refinements of existing model features are maintained, and additional features are added) and structural refinement (detailed design is added to the implementation).

Butler [65,66] modelled atomicity decomposition (i.e., event decomposition) by which more fine-grained atomicities could be obtained through the refinement of a coarse-grained atomicity by means of an event refinement diagram. Also, Butler proposed the notion of basic parallel composition (i.e., $\|$) and parallel composition with shared event operators. Finally, he described machine decomposition using the composition operator in reverse. Abrial [67] proposed an event model decomposition in which for decomposing an event model, M, firstly, M is split into several sub-models, say N, ..., P. Next, the events and then the variables of M are partitioned over sub-models. Then, sub-models are refined several times independently yielding, eventually, NR,..., PR.

The decomposition of machines is where an Event-B machine is separated into a number of smaller components that are easier to manage [4]. Machine decomposition in Event-B is in the style of either shared variable decomposition or shared event decomposition. The former style of machine decomposition is also proposed by Abrial [67], Métayer et al. [60], Abrial and Hallerstede [61] and also Jones [68] in which a machine is decomposed into two or more machines, based on an arbitrary shared variable. Shared variables are those accessed by the events of different sub-machines.

Shared event decomposition is also proposed by Butler [65]. In this style of machine decomposition, a machine is decomposed into an arbitrary number of sub-machines, based on a shared event in that machine. Also, Pascal and Silva [69] developed a tool for both styles of machine decomposition as a decomposition plug-in of the Rodin platform. Silva et al. [70] described a complete specification of the mentioned decomposition plug-in. Also, they developed context decomposition as a plug-in of the Rodin platform. Furthermore, Hoang and Abrial [71] proposed an approach to develop parallel programs using refinement and decomposition.

Poppleton [72] proposed Event-B model composition based on shared variables. More precisely, he introduced Event-B model composition by introducing the notion of event fusion and model fusion. In order to perform model fusion, he divided the variables of two models, which are considered to be composed into two lists; "actioned variables" and "skipping variables". Thus, the composed model has variables existing in the intersection of actioned and skipping variables of two models. Also, its events are obtained through applying an event fusion operator on the events of two models. Finally, contexts and invariants of the composed model

are a conjunction of the contexts and invariants of the two models.

Silva and Butler [73] proposed a shared event decomposition which is necessary to recompose decomposed machines. They have described this form of composition through adding the composed machine notation to Event-B, and also using the notion of a parallel composition operator for events. Moreover, they developed a tool to support composition in Event-B as a plug-in of the Rodin platform. In another related work, Silva and Butler [74] proposed a way to instantiate generic models, previously defined by Méteyer et al. [60]. More precisely, they introduced the notion of an instantiated machine that allows one to replace elements in the context(s) of generic development and to rename variables and events in the generic development.

Hallerstede et al. [75] provided a detailed description of refinement in Event-B. This description was then used to assist simultaneous animation of multiple levels of refinement. Finally, besides work refining Event-B specifications directly, there is work which refines Event-B specifications through B method refinements using Atelier-B [76]. In other words, Atelier-B supports the refinement of Event-B specifications through mapping of Event-B specifications into B method specifications, and then refinement of B method specifications.

3.2.1.5 UML-B

Said et al. [77] proposed rules for refining classes and state machines. Refinement of classes can be done by either removing or adding attributes. To refine a state machine, its structure should be elaborated either by replacing each transition by one or more transitions or by elaborating an abstract state by a nested state machine. In addition, a technique for moving class events is proposed in [77].

Also, Said [4] proposed a more complete approach for refinement of UML-B specifications. Rules given in [4] support the refinement of machines, classes, state machines and context diagrams. Rules for refining machines include "decomposition with a shared event approach", "composition" and also "machine refinement via refining its class diagram". Rules for refining class diagrams include machine variables, events and invariant refinements (similar to Event-B rules for refining these constructs), class and state machine refinements, adding new classes and dropping abstract classes. The followings are rules for class refinement:

1. Introducing new associations and attributes;
2. Dropping abstract associations and attributes;
3. Refinement of class events and invariants (similar to Event-B rules for refining these constructs).

Besides the two ways proposed in [77] for the refinement of state machines, Said [4] introduces a flattening state machine (refinement of a machine with nested state machines to a state machine without any nested state machines) and state grouping (adding a new structure or state to a state machine and nesting some of its states in the new structure). Rules for refining context diagrams include introducing new associations and attributes to the extended classtype and also introducing new classtypes to the refinement. Other rules support a moving event or transition in the refinement class or new class.

3.2.2. Refinement from specification to code

This group of refinement techniques is categorized based on their formal specification languages, as follows.

3.2.2.1 Object-Z

Besides Derrick and Boiten [44,45] who refine one class schema to another class schema, there are other researchers who refine Object-Z into code using Perfect Developer [78,79] and Spec# [80]. Stevens [78] and Kimber [79] demonstrated how an Object-Z specification can be expressed in Perfect and refined towards an implementation. Qin and He [80] described a linking process between Object-Z and Spec#.

In another publication, Najafi and Haghighi [81] described a set of mapping functions that map Object-Z constructs to Morgan's Refinement Calculus (MRC) [82] constructs. Using the provided mappings, the specifier can develop final programs by applying MRC rules to resulting MRC specification constructs.

3.2.2.2 VDM++

Based on our investigations, there is no work in VDM++ in this area.

3.2.2.3 B

B provides a notation for implementation which is a basis for translating abstract machine/refinement into code. The form of IMPLEMENTATION is as follows:

IMPLEMENTATION Name
REFINES component (machine or refinement)
SEES seen machines
IMPORTS imported machines
PROMOTES operations of imported machines (without any changes)
SETS local sets
CONSTANTS scalar local constants
VALUES local values
INVARIANT relationship between refined and imported states
OPERATIONS implementation of refined operations
END

A description of the above clauses is as follows:

- The IMPLEMENTATION clause introduces the name of implementation;
- The REFINES clause contains the name of the refined component (also called abstraction) for the refinement;
- The SEES clause consists of a list of instances of seen machines;
- The IMPORTS clause contains the declaration of the list of imported machine instances and creates concrete instances of the modules in a project. The implementation creates the imported abstract machine instance to use its data and operations to implement its own data and operations;
- The PROMOTES clause introduces a list of promoted operations of instances of included machines;
- The SETS clause introduces the sets which are used in the implementation;
- The CONSTANTS clause introduces the constants which are used in the implementation;
- The VALUES clause is used to assign values to the deferred sets and to the concrete constants;
- The INVARIANT clause indicates the relationship between refined and imported states. It consists of predicates separated by a conjunction operator;
- The OPERATIONS clauses are made up of concrete expressions or substitutions.

Some proof obligations have been defined to show that an implementation implements a refinement/abstract machine appropriately. In addition, Atelier-B [76] has been developed that supports code generation from B specifications to Ada and C++ code.

3.2.2.4 Event-B

Although ProB is an animator for Event-B, it does not support Event-B code generation; hence, we would like to mention some works approaching Event-B code generation.

Based on [83-85], one can link Event-B specifications with concurrent object-oriented programs using Object-oriented Concurrent-B (OC-B). In this way, specification of concurrency issues and reasoning about them in an abstract manner become possible. For this purpose, Edmunds and Butler interpreted details of concurrent features, such as processes and monitors, in Event-B. Then, they introduced an approach to refine Event-B specifications to their OC-B counterparts. Finally, they described the mapping of OC-B specifications to object-oriented code (for example, in Java). Also, they have developed a tool which is based on Eclipse, and maps Event-B specifications to object-oriented code.

As another related work, Edmunds and Butler [86] showed how one could develop the Ada source code from Event-B specifications. For this purpose, they introduced an extension of Event-B, called Tasking Event-B, which includes tasking and shared machines. Using this approach, one can convert Event-B models to Tasking Event-B models, and then convert Tasking Event-B models to Ada code using translation rules and decomposition rules defined for Event-B, and also extension rules defined for Tasking Event-B. Moreover, Edmunds, Rezazadeh, and Butler [87] describe a streamline process, where the abstract modelling artefacts are mapped to Ada language constructs using refinement, decomposition, and implementation annotations.

In addition, Méry and Singh [88] developed a set of software tools, i.e. EB2C, EB2C++, EB2J and EB2C#, that generate programming code in C, C++, Java, and C# from Event-B specifications, respectively. These tools perform code generation from Event-B models using Event-B grammar and through syntax-directed translation, code scheduling architecture and verification of an automatic code generation. EB2C, EB2C++, EB2J and EB2C# have been developed as a set of Rodin plug-ins under the Eclipse development framework. In comparison to [83,84], these tools support set theory based notations.

3.2.2.5 UML-B

The current code generation technique for UML-B specifications is based on using the Event-B code generation tool [83-85]. UML-B specifications are first converted to Event-B specifications using the U2B tool [89]. Then, an object-oriented code is generated using the Event-B code generation tool. Some conversion rules of the U2B tool are described in Table 2.

Table 2. Mapping rules from UML-B constructs to Event-B constructs.

UML-B construct	Event-B construct
Machine	Implicit context and machine
Class	Machine variable
Class attribute	Machine variable with an invariant which demonstrates attribute membership in a relationship between its class and its type
Class event	Machine event
Class invariant	Machine invariant
Class theorems	Machine theorems
Context	Context
ClassType	Context sets
ClassType association	Context constant
State machine transition	Machine event

4. Comparison

In this section, we first define a set of criteria used to compare overviewed formal, object-oriented program development techniques. Then, these techniques will be compared in terms of the given criteria.

4.1. Criteria definition

Table 3 shows the criteria that will be used for comparing formal, object-oriented program development techniques:

Remarks:

- Two main styles are considered for the "Refinement Style": "Posit-and-Prove" is where a refinement of specification is proposed and then justified against its abstract specification via the verification of a set of proof obligations, and "Transformational" refinement is where algorithms or rules are applied to a specification to generate a more concrete specification [4].
- For "Expressiveness" criterion, we determine whether "Mapping rules are proposed with enough details (and thus are informative enough to show various aspects and cases of the mapping) or not" based on our intuition.
- A high "Similarity Degree" makes it easier for the

Table 3. Criteria for comparing formal, object-oriented program development techniques.

Category	Criterion	Description of criterion
Context	Main Approach (MA)	Main approach of a technique may be Animation (A) or Refinement (R).
	Refinement Style (RS)	If the main approach is refinement, the style of a technique can be Transformational (T) or Posit-and-Prove (P); see [4] for more details about these styles.
	Expressiveness	How mapping rules are proposed? Are mapping rules proposed with enough details?
	Similarity Degree (SD)	The degree of similarity between constructs in the initial specification and constructs in the resulting code.
Content	Formal language coverage	The portion of the formal specification language grammar treated successfully by the technique.
	Programming language coverage	The portion of the programming language grammar that can be generated by using the technique
Execution	Interactivity	Does the technique have interaction with users? If yes, is interaction done to increase the flexibility of code generation (by considering user's opinion) and to allow making changes in specifications at any stage of the mapping process?
	Tool support	—
Reliability	Validation approach	Validation of the technique is done via a mathematical proof, a case study, or other approaches.
Reusability	Library development	Libraries which are developed by the technique.

developer to understand how some constructs in the final programming language correspond to some specification constructs and vice versa.

⁎ We regard "Programming language coverage" as a criterion because using some special constructs of the programming language (such as macro and the notion of operator overloading in C++) may provide more efficient programs and also reveal the distinguishing features of programming languages between them.

4.2. Comparison between techniques

4.2.1. Context

As Tables A.1 to A.3 of [90] show, 30% of the existing formal methods for object-oriented program development are animation, and 70% of them are refinement. Most of the existing animation techniques:

1. Propose their mapping rules using natural language, while formal presentation of mapping rules increases precision and provides the possibility of proving the correctness of the method (the correctness of the mapping rules) formally.
2. Do not consider special cases of constructs in the specification language when presenting their mapping rules; however, they describe these rules with enough level of detail. For example, in animation of Object-Z specifications, the mapping of a local abbreviation definition when its left hand side is a variable name, and its right-hand side is the class union, has never been covered in many related publications, such as [20-22].
3. Benefit from a degree of high similarity.

In contrast, most of the existing refinement techniques propose their refinement rules using both natural language and first order predicate logic. Also, they propose their rules with enough level of detail.

4.2.2. Content

As Tables B.1 to B.3 of [90] show, none of the existing animation techniques cover all constructs of the formal specification language. Instead, most of the existing animation and refinement techniques propose rules for mapping common constructs of the formal specification language. Moreover, they use a small set of programming language constructs in their mappings.

4.2.3. Execution

- Object-Z: Most of the techniques available for animating Object-Z specifications interact with users and are not fully automatic. Also, few existing refinement techniques for the mentioned language have tool support.
- VDM and VDM++: Most of the existing animation and refinement techniques for these languages, in papers to which we had access, have tool support. In addition, all of them, except [34,36], have interaction with users.
- B, Event-B and UML-B: Considering Table C.3 of [90], most of the refinement techniques for B, Event-B and UML-B specifications have tool support and interact with users.

In addition, all the existing techniques (either animation or refinement), except [27,28,34,36,88], regard interactivity only for the purpose of getting specifications and filling those parts of the code whose techniques cannot present any mapping and cannot confirm the correctness of the performed mapping with users. In this case, we put "Y: Weak", in Tables C.1 to C.3 of [90], because interaction with users can also be done for other reasons, such as:

- Increasing flexibility of techniques by considering user opinion when obtaining code from specifications;
- Requesting and making changes in specifications at any stage when executing the mapping process.

4.2.4. Reliability

As Tables D.1 to D.3 of [90] show, most existing works, except [29] try to validate their methods based on case studies, while using mathematical proof is a more powerful validation approach.

4.2.5. Reusability

As Tables E.1 to E.3 of [90] show, most existing techniques did not develop libraries for their mapping rules.

5. Conclusion and future work

Using object-orientation in formal program development has led to methods which have the advantages of both formal methods and object oriented approaches. In this paper, we first compared Object-Z, VDM++ and UML-B specification languages. Then, we reviewed existing formal, object-oriented program development from Object-Z, VDM, VDM++, B, Event-B and UML-B specifications. Finally, we compared these methods according to a set of criteria classified into categories of context, content, reliability, reusability, and execution.

Tables 4 and 5 show a summary of our investigation. Some interesting topics for future work can be extracted from these two tables, along with detailed information extracted from the tables shown in [90] in order to reduce the weaknesses of existing methods while preserving their strengths through presenting new formal methods for object-oriented program development.

Table 4. Strengths of existing formal methods for object-oriented program development.

Specification language	Strengths
Object-Z	• Most of the existing animation techniques have tool support. • Static and dynamic verifications have been considered in one of the animation techniques, i.e. [24]. • The formal specification of mapping rules and proving their correctness formally have been considered in one of the animation techniques, i.e. Najafi and Haghighi (2013) [29]. • Animation of Object-Z with C++, Java, C# and Eiffel has been proposed. • Animation of Object-Z with C++, Java, C# and Eiffel has tool support.
VDM++	• Animation of VDM++ with C++ and Java has been proposed. • All of the animation techniques have tool support. • Help manuals for these animation techniques are well-documented.
VDM	• Animation of VDM with C++, Ada, Ada95, Smalltalk and Java has been proposed. • In animation of VDM with C++, interaction with the purpose of getting specification, filling parts of the code for which the technique could not present any mapping, confirming the correctness of the performed mapping with the user and improving obtained object-oriented designs is considered. • In animation of VDM with Ada, interaction with the purpose of getting specification, filling parts of the code for which the technique could not present any mapping, confirming the correctness of the performed mapping with the user and increasing the flexibility of the technique by requesting and making changes to specifications at any stage is considered. • As far as we know, all of the proposed techniques have tool support.
B	• All of the proposed refinement techniques for this language have tool support. In addition, help manuals for these refinement tools are well-documented. • Refinement of B Method to Ada and C++ has been proposed.
Event-B	• All of the proposed refinement techniques, except "machine composition with shared variable", have tool support.
UML-B	• State grouping and flattening rules for state machine refinement have been proposed formally. • All of the proposed refinement rules have tool support.

Table 5. Weaknesses of existing formal methods for object-oriented program development.

Specification language	Weaknesses
Object-Z	• None of the animation techniques covers mapping of some important constructs of Object-Z, such as distributed operators, free type definitions (when constructors are used in the definition) and recursion. • Only one of the existing animation techniques proposes mapping rules formally. • Only one of the existing animation techniques proves the correctness of their mapping rules. • Few animation techniques consider special cases of Object-Z constructs in their mapping rules. For example, in animation of Object-Z specifications, the mapping of a local abbreviation definition, when its left hand side is a variable name, and its right-hand side is class union, has never been covered in many related publications, such as [20-22]. • Case studies used in animation techniques are not large enough to cover all of the proposed rules. • Most animation techniques use only a small set of programming language constructs. • No tool exists to support direct refinement of Object-Z specifications.
VDM++	• Few animation techniques cover mapping of some important constructs of VDM++, such as function composition, function iteration and equality. • None of the existing animation techniques proposes mapping rules formally. • None of the existing techniques considers user interaction with the purpose of providing flexibility (see Subsection 4.2.3). • Most of the animation techniques use only a small set of programming language constructs. • No tool exists to supports refinement of VDM++ specifications. • None of the existing animation techniques proves the correctness of their mapping rules. • No refinement technique has been proposed to refine VDM++ specifications to final code.
VDM	• None of the animation techniques covers mapping of some important constructs of VDM, such as compose, iterate and equality for functions. • None of the existing animation techniques proposes mapping rules formally. • None of the existing animation techniques proves the correctness of their mapping rules.
B	• Proving the correctness of refinements is a tedious and time-consuming task. • Having a large number of refinement rules makes the identification and selection of appropriate rules complex. Thus, it is necessary to have an intuition of the final program to perform a successful refinement.
Event-B	• Composition and decomposition rules have not been proposed explicitly for contexts. • Decomposition rules have not been proposed formally. • No tool exists to support machine composition with a shared variable approach. • It is necessary to have an intuition of the final program to perform a successful refinement.
UML-B	• Some refinement rules, such as annealing and introduced sub-typing, have not been proposed for UML-B specifications, while they are well-known refinement rules in other object-oriented formal specification languages. • Only state grouping and flattening rules for mapping state machines have been described formally. • None of the existing techniques considers user interaction with the purpose of providing flexibility (see Subsection 4.2.3).

References

1. Naur, P. and Randell, B., Eds. "Software engineering", *Report on a Conference Sponsored by the NATO Science Committee*, Scientific Affairs Division (1969).
2. Mirian-Hosseinabadi, S.H. "Constructive Z", Ph.D. Thesis, University of Essex, Essex, UK (1997).
3. Kemmerer, R.A. "Testing formal specifications to detect design errors", *Software Engineering, IEEE Transactions* (1985).
4. Said, M.Y. "Methodology of refinement and decomposition in UML-B", Ph.D. Thesis, University of Southampton, Southampton, UK (2010).
5. Beeson, M.J., *Foundations of Constructive Mathematics: Mathematical Studies*, Springer-Verlag (1985).
6. Najafi, M. and Haghighi, H. "Formal methods to develop object-oriented programs: A survey", *Second World Conference on Information Technology*, Antalya, Turkey (2011).
7. Smith, G., *The Object-Z Specification Language*, Kluwer Academic Publishers, USA (2000).
8. Duke, R. and Rose, G., *Formal Object-Oriented Specification Using Object-Z*, Macmillan, UK (2000).
9. Ramkarthik, S. and Zhang, C. "Generating java skeletal code with design contracts from specifications in a subset of object-Z", In *Fifth IEEE/ACIS Int. Conf. on Computer and Information Science*, Honolulu, HI, pp. 405-411 (2006). [doi: 10.1109/ICIS-COMSAR.2006.41]
10. Dürr, E. and Katwijk, J.V. "VDM++: A formal specification language for object-oriented designs", *Computer Engineering and Software Systems (CompEuro'92)*, pp. 214-219 (1992). [doi: 10.1109/CMPEUR.1992.218511]
11. Fitzgerald, P., Larsen, P.G., Mukerji, P., Plat, N. and Verhoef, M., *Validated Designs For Object-Oriented Systems*, Springer-Verlag, USA (2004).
12. CSK Corporation, The VDM++ Language Manual 1.0, Available at: http://www.vdmtools.jp/en/ modules/tinyd2/index.php?id=2/langmanpp_a4E.pdf (2010).
13. Booch, G., Jacobson, I. and Rumbaugh, J., *The Unified Modeling Language - A Reference Manual*, Addison Wesley (1998).
14. Snook, C., Butler, M. and Oliver, I. "The UML-B profile for formal systems modeling in UML", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **15**(1), pp. 92-122 (2004).
15. Snook, C. and Butler, M. "UML-B: Formal modeling and design aided by UML", *ACM Transactions on Software Engineering and Methodology*, **15**(1), pp. 92-122 (2006).
16. Snook, C. and Butler, M. "UML-B and event-B: An integration of languages and tools", In *The IASTED Int. Conf. on Software Engineering*, pp. 1-6 (2008). [doi: 10.1016/j.infsof.2007.10.010]
17. Joochim, T. "Bringing requirements engineering to formal methods: timing diagrams for event-B and KAOS", Ph.D. Thesis, University of Southampton, Southampton, UK (2010).
18. Guelfi, N., Biberstein, O., Buchs, D., Canver, E., Gaude, M.C., Henke, F.V. and Schwier, D. "Comparison of object-oriented formal methods", *Technical Report of the Esprit Long Term Research Project 20072 "Design for Validation"*, University of Newcastle Upon Tyne, Department of Computing Science (1997).
19. Chrietensen, T.J.H. "Extending the VDM++ formal specification language with type inference and generic classes", M.S. Thesis, University of Aarhus, Aarhus, Denmark (2007).
20. Rafsanjani, G. and Colwill, S.J. "From object-Z to C++: A structural mapping", In *Z User Meeting (ZUM'92)*, pp. 166-179 (1992).
21. Johnston, W. and Rose, G. "Guidelines for the manual conversion of object-Z to C++", *SVRC Technical Report No. 93-14*, University of Queensland, Canada (1993).
22. Fukagawa, M., Hikita, T. and Yamazaki, H. "A mapping system from object-Z to C++", In *First Asia-Pacific Software Engineering Conference (APSEC94)*, pp. 220-228 (1994). [doi: 10.1109/APSEC.1994.465258]
23. Griffiths, A. "From object-Z to Eiffel: A rigorous development method", *Technology of Object-Oriented Languages and Systems: TOOLS 18*, Prentice-Hall (1995).
24. Ni, X. and Zhang, C. "Converting specifications in a subset of object-Z to skeletal spec# code for both static and dynamic analysis", *Journal of Object Technology*, **7**(8), pp. 165-185 (2008). [doi: 10.5381/jot.2008.7.8.a6]
25. Wang, Z., Xia, M. and Zhao, Y. "Transform mechanisms of object-Z based formal specification to java", In *Computational Intelligence and Software Engineering (CISE), Wuhan*, pp. 1-4 (2009). [doi: 10.1109/CISE.2009.5365403]
26. Najafi, M. and Haghighi, H. "An animation approach to develop C++ code from object-Z specifications", *CSI International Symposium on Computer Science and Soft-ware Engineering (CSSE 2011)*, Tehran, Iran, pp. 9-16 (2011). [doi: 10.1109/CSICSSE.2011.5963990]
27. Najafi, M. and Haghighi, H. "An approach to develop C++ code from object-Z specifications", In *Second World Conference on Information Technology*, Antalya, Turkey (2011).
28. Najafi, M. and Haghighi, H. "An approach to animate object-Z specifications using C++", *Scientia Iranica*, **19**(6), pp. 1699-1721 (2012). [doi: 10.1016/j.scient.2012.06.021]
29. Najafi, M. and Haghighi, H. "A formal mapping from object-Z specification to C++ code", *Scientia Iranica*, **20**(6), pp. 1953-1977 (2013).

30. Jackson, M.I. "Developing Ada programs using the Vienna development method (VDM)", *Software: Practice and Experience*, **15**(3), pp. 305-318 (1985).

31. Chedgey, C., Kerney, S. and Kugler, H. "Using VDM in an object-oriented development method for Ada software", *VDM-Europe Symposium 1987 on VDM'87: VDM - A Formal Method at Work*, LNCS 252, pp. 63-76 (1987). [doi: 10.1007/3-540-17654-3_5]

32. Moulding, M.R. and Newton, A.R. "Rapid prototyping from VDM specifications using Ada", *IEEE Colloqium on Automating Formal Methods for Computer Assisted Prototyping*, London, UK, pp. 11-22 (2002).

33. O'Neill, D., Bloomfield, R., Marshall, L. and Jones, R. "VDM development with Ada as the target language", *VDM'88-The Way Ahead*, LNCS 328, pp. 116-123 (1988). [doi: 10.1007/3-540-50214-9_11]

34. Lou, Y. "VDM/C++: A design and implementation framework", M.S. Thesis, Concordia University, Montreal, Canada (1994).

35. Katwijk, J.V., Dürr, E. and Goldsack, S. "Hybrid object-oriented real-time software development", In *First IEEE Int. Conf. on Formal Engineering Methods, Hiroshima*, Japan, pp. 17-26 (1997). [doi: 10.1109/ICFEM.1997.630393]

36. Albalooshi, F. and Long, F. "Multiple view environment supporting VDM and Ada", *IEEE Proceedings Software*, **146**(4), pp. 203-219 (2002). [doi: 10.1049/ip-sen:19990487]

37. Chartan, Q. and Kans, A., *Formal Software Development: from VDM to Java*, Palgrave Macmillan, China (2004).

38. CSK Corporation, *The VDM-SL to C++ Code Generator Manual 1.0*, Available at: http://www.vdmtools.jp/en/modules/tinyd2/index.php?id=2/ cg-mansl_a4E.pdf (2010).

39. Bousquet, F. "VDM++ to Ada95 translation rules", Technical Report VICE-MBDF-17, MatraBAe Dynamics, Rue Grange Dame Rose 20/22, 78141 Velizy-Villacoublay, France (2000).

40. CSK Corporation, *The VDM++ to C++ Code Generator Manual 1.0*, Available at: http://www.vdmtools.jp/en/modules/tinyd2/index.php?id=2/ cg-manpp_a4E.pdf (2010).

41. CSK Corporation, *The VDM++ to Java Code Generator Manual 1.1*, Available at: http://www. vdmtools.jp/en/modules/tinyd2/index.php?id=2/ javacg-manpp_a4E.pdf (2010).

42. McComb, T. and Smith, G. "Refactoring object-oriented specifications: A process for deriving designs", Technical Report SSE-2006-01, University of Queensland, Aus-tralia (2006).

43. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison Wesly (1999).

44. Derrick, J. and Boiten, E.A. "Refinement of objects and operations in object-Z", *Int. Conf. on Formal Methods for Open Object-Based Distributed Systems IV, IFIP Advances in Information and Communication Technology*, Stanford, California, USA, pp. 257-277 (2000). [doi: 10.1007/978-0-387-35520-7_13]

45. Derrick, J. and Boiten, E.A. "Refinement in Z and object-Z: Foundations and advanced applications", 1st Ed., *Formal Approaches to Computing and Information Technology*, Springer-Verlag (2001).

46. Smith, G. "Introducing reference semantics via refinement", In *Fourth Int. Conf. on Formal Engineering Methods: Formal Methods and Software Engineering*, pp. 588-599 (2002).

47. McComb, T. "Refactoring object-Z specifications", *Fundamental Approaches to Software Engineering*, Barcelona, Spain, LNCS 2984, pp. 69-83 (2004). [doi: 10.1007/978-3-540-24721-0_5]

48. McComb, T. and Smith, G. "Architectural design in object-Z", *Australian Software Engineering Conference (ASWEC)*, pp. 77-86 (2004). [doi: 10.1109/ASWEC.2004.1290460]

49. McComb, T. and Smith, G. "Compositional class refinement in object-Z", In *FM2006: Formal Methods Conference*, Hamilton, Canada, LNCS 4085, pp. 205-220 (2006). [doi: 10.1007/11813040_15]

50. Ruhroth, T., *Refactoring Object-Z Specifications*, Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi= 10.1.1.106.2651.pdf (2006).

51. McComb, T. and Smith, G. "A minimal set of refactoring rules for object-Z", *Formal Methods for Open Object-Based Distributed Systems*, Oslo, Norway, LNCS 5051, pp. 170-184 (2008). [doi: 10.1007/978-3-540-68863-1_11]

52. McComb, T. and Smith, G. "Introducing objects through refinement", In *FM2008: Formal Methods Conference*, Turku, Finland, LNCS 5014, pp. 358-373 (2008). [doi: 10.1007/978-3-540-68237-0_25]

53. Liu, H. and Zhu, B. "Refactoring formal specifications in object-Z", In *International Conference on Computer Science and Software Engineering*, Wuhan, Hubei, pp. 342-345 (2008). [doi: 10.1109/CSSE.2008.260]

54. Lano, K. and Goldsack, S.J. "Refinement, subtyping and subclassing in VDM++. computing", *Workshop on Theory and Formal Methods*, pp. 341-363 (1994).

55. Goldsack, S.J., Dürr, E.H. and Plat, N. "Object reification in VDM++", *ICSE 17: Workshop on Formal Methods Application in Software Engineering Practice*, pp. 194-201 (1995).

56. Goldsack, S.J. and Lano, K. "Annealing and data decomposition in VDM++", *ACM SIGPLAN*, **13**(4), pp. 32-38 (1996).

57. Lano, K. and Goldsack, S. "Refinement of distributed object systems", *Workshop on Formal Methods for Open Object-based Distributed Systems*, Chapman and Hall (1996).

58. Lu, J. "Introducing data decomposition into VDM for

tractable development of programs", *ACM SIGPLAN Notices*, **30**(9), pp. 41-50 (1995).

59. Lano, K. "Specification in B: An introduction using the B toolkit", *World Scientific Publishing Company*, Imperial College Press (1996).
60. Métayer, C., Abrial, J.R. and Voisin, L. "Event-B language", Technical Report, Deliverable 3.2, EU Project IST-511599-RODIN (2005).
61. Abrial, J.R. and Hallerstede, S. "Refinement, decomposition, and instantiation of discrete models: Application to event-B", *Fundamenta Informaticae*, **77**(1-2), pp. 1-28 (2007). [doi: 10.1145/362575.362577]
62. Abrial, J.R., Cansell, D. and Méry, D. "Refinement and reachability in event-B", In *Fourth International Conference of B and Z Users*, Guildford, UK, LNCS 3455, pp. 222-241 (2005). [doi:10.1007/11415787_14]
63. Abrial, J.R. "A system development process with event-B and the rodin platform", In *Ninth International Conference on Formal Methods and Software Engineering*, LNCS 4789, pp. 1-3 (2007). [doi: 10.1007/978-3-540-76650-6_1]
64. Butler, M., Abrial, J.R., Damchoom, K. and Edmunds, A. "Applying event-B and Rodin to the file store", *Abstract State Machines, B and Z*, London, UK, ASRNET (2008).
65. Butler, M. "Synchronization-based decomposition for event-B", *RODIN Deliverable D19 Intermediate Report on Methodology* (2006).
66. Butler, M. "Incremental design of distributed systems with event-B", *Engineering Methods and Tools for Software Safety and Security*, Marktoberdorf Summer School 2008, IOS Press, pp. 131-160 (2008). [doi: 10.3233/978-1-58603-976-9-131]
67. Abrial, J.R. "Event model decomposition", *Technical Report*, **626**, ETH Zurich (2009)
68. Jones, C.B. "RODIN deliverable D19", *Intermediate Report on Methodology*, Available at: http://rodin.cs.ncl.ac.uk/deliverables/D19.pdf, Technical report, University of Newcastle-upon-Tyne, UK (2006).
69. Pascal, C. and Silva, R. "Event-B model decomposition", *DEPLOY Plenary Technical Workshop* (2009).
70. Silva, R., Pascal, C., Hoang, T.S. and Butler, M. "Decomposition tool for event-B", *Workshop on Tool Building in Formal Methods- ABZ Conference* (2010).
71. Hoang, T.S. and Abrial, J.R. "Event-B decomposition for parallel programs", In *Second Int. Conf. on Abstract State Machines, Alloy, B and Z*, pp. 319-333 (2010). [doi: 10.1007/978-3-642-11811-1_24]
72. Poppleton, M. "The composition of event-B models", In *First Int. Conf. on Abstract State Machines, B and Z*, London, UK, pp. 209-222 (2008). [doi: 10.1007/978-3-540-87603-8_17]
73. Silva, R. and Butler, M. "Supporting reuse mechanisms for developments in event-B: Composition", *Technical Report ECS*, University of Southampton, Southampton, UK (2009).
74. Silva, R. and Butler, M. "Supporting reuse of event-B developments through generic instantiation", In *Eleventh Int. Conf. on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM)*, pp. 466-484 (2010). [doi: 10.1007/978-3-642-10373-5_24]
75. Hallerstede, S., Leuschel, M. and Plagge, D. "Refinement-animation for event-B-towards a method for validation", In *Second International Conference on Abstract State Machines, Alloy, B and Z*, LNCS 5977, pp. 287-301 (2010). [doi: 10.1007/978-3-642-11811-1_22]
76. Atelier-B, Available at: http://www.atelierb.eu/en/atelier-b-tools/atelier-b-4-0/ (2011).
77. Said, M.Y., Butler, M. and Snook, C. "Language and tool support for class and state machine refinement in UML-B", *FM 2009: Formal Methods*, pp. 579-595 (2009). [doi: 10.1007/978-3-642-05089-3_37]
78. Stevens, B. "Implementing object-Z with perfect developer", *Journal of Object Technology*, **6**(2), pp. 189-202 (2006). [doi: 10.5381/jot.2006.5.2.a5]
79. Kimber, T.G. "Object-Z to perfect developer", M.S. Thesis, Imperial College Lon-don, London, UK (2007).
80. Qin, S. and He, G. "Linking object-Z with spec#", In *Twelfth IEEE Int. Conf. on Engineering Complex Computer Systems*, Auckland, New Zealand, pp. 185-196 (2007). [doi: 10.1109/ICECCS.2007.27]
81. Najafi, M. and Haghighi, H. "Refinement of object-Z specifications using Morgan's refinement calculus", In *Int. Conf. on Software Engineering and Technology (ICSET2011)*, Venice, Italy, pp. 1735-1744 (2011).
82. Morgan, C., *Programming from Specifications*, Prentice-Hall (1990).
83. Edmunds, A. and Butler, M. "Linking event-B and concurrent object-oriented programs", *Thirteenth BAC-FACS Refinement Workshop (REFINE 2008)*, ENTSC 214, pp. 159-182 (2008). [doi: 10.1016/j.entcs.2008.06.008]
84. Edmunds, A. and Butler, M. "Tool support for Event-B code generation", *Workshop on Tool Building in Formal Methods*, Québec, Canada (2010).
85. Edmunds, A. "Providing concurrent implementations for event-B developments", Ph.D. Thesis, University of Southampton, Southampton, UK (2010).
86. Edmunds, A. and Butler, M. "Tasking event-B: An extension to event-B for generating concurrent code", *Programming Language Approaches to Concurrency and Communication-Centric Software*, Saarbrucken, Germany (2011).
87. Edmunds, A., Rezazadeh, A. and Butler, M. "Formal

modelling for Ada implementations: Tasking event-B", In *Ada-Europe*, LNCS, **7308**, pp. 119-132 (2012).

88. Méry, D. and Singh, N.K. "Automatic code generation from event-B models", In *Second Symposium on Information and Communication Technology*, ACM Press, pp. 179-188 (2011). [doi: 10.1145/2069216.2069252]

89. Snook, C. and Butler, M., *U2B- A Tool for Translating UML-B into B, UML-B Specification for Proven Embedded Systems Design*, Chapter 5, Springer-Verlag (2004).

90. Appendix, Available at: http://ticksoft.sbu.ac.ir/?page_id=5116 (2015).

Biographies

Mehrnaz Najafi received both her MS and BS degrees in Computer Engineering-Software from Shahid Beheshti University, Iran, in 2012 and 2010, respectively. Her research interests are formal program development and formal verification.

Hassan Haghighi received his PhD degree in Computer Engineering-Software from Sharif University of Technology, Iran, in 2009, and is currently Assistant Professor in the Faculty of Electrical and Computer Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods in the software development life cycle.

Tahereh Zohdi Nasab received her BS degree in Computer Engineering Software from the University of Tehran, Iran, in 2009. Her research interest is formal program development.