The Design and Implementation of a Dynamic Verification System of Z

Jun Wang, Yi Zhuang(✉), and Siru Ni

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, People's Republic of China
nuaa_wangj@163.com, {zyl6,nisr}@nuaa.edu.cn

Abstract. Z notation can accurately describe static structures and operation specifications of software. Predicate calculus is used to perform validation and verification of Z models. The existing validation tools of Z are aimed to check the expected behavior of software model, while they cannot automatically verify the correctness and safety of the software. This paper proposes a software model named ZA (Z-Automata) to describe the behavior and data constraints of software, by combining the elements of Z notation and FA (Finite Automata). An extended temporal logic is defined, and a model checking algorithm ZAMC (Z-Automata Model Checking) is designed to perform the verification of ZA's expected properties. For the practical usage of our ZA model and ZAMC algorithm, we implement a prototype of Z Dynamic Verification System (ZDVS), which can be used as a modeling and verification tool for Z models. The illustration of the modeling and verification process of a case study shows that our ZA model can describe data constraints within the software behavior, and can automatically verify the expected properties of the Z model.

Keywords: Z notation · Finite automata · Model checking · Temporal logic

1 Introduction

Software, as carrier of information technology, bears the mission of interactions between human and computers. It has been widely used in people's daily life, as well as in industrial production, aviation, spaceflight and so on. The growing demand for software has led to its greater complexity and larger scale. At the same time, ensuring dependability of software is becoming a strong requirement in many fields. As a result, the software designers face with new challenges at the stages of design, implementation and maintenance of software development these years.

Z [1–5] is a formal specification language used for describing computer programs and computer-based systems in general. Theorem proving [6–8] and animation [9–11] are two main categories of analysis methods for Z notation. The former usually uses predicate logic reasoning to prove the static properties of a Z model. The latter employs a query approach to confirm the expected system behavior by accepting or denying the manually input operations. The decisions are made depending on whether there is any inconsistency between the specification of the operation input and the global constraints

Z. Huang et al. (Eds.): ICCCS 2015, LNCS 9483, pp. 344–355, 2015.
DOI: 10.1007/978-3-319-27051-7_29

of the Z model. These two categories of analysis methods are both unable to automatically verify the temporal properties of a given Z model.

Model checking [12], or property checking, is a technology dealing with the following problem: given a model of a state transition system, exhaustively and automatically check whether this model meets a given formula. Usually, the formula describes temporal properties such as safety and liveness [13]. Because of its conciseness and efficiency, model checking technology has been widely adopted to verify temporal properties of hardware and software systems for over three decades.

Because Z lacks the ability of describing system temporal behavior, building a proper state transition system is the primary step to study the model checking method on Z. A state transition system based on FA (Finite Automata) can describe the run-time behavior of software, but it is insufficient in describing data constraints, comparing with Z's syntax based on set theory and first-order logic. Some of the current studies on model checking Z are based on transforming Z to a middle language that can be model checked by existing tools [14]. Other related works use new structures such as ZIA [15, 16], which is a combination model of Z notation and interface automata targeted at component based systems. By combining Z and state transition structures, we can establish a more comprehensive system model, and study its model checking method accordingly. However, the existing researches have not implemented automatic model transformation between Z and those hybrid models, which is insufficient for industrial usage.

In this paper, we design and implement a prototype system ZDVS. Firstly, we define a formal model ZA (Z-Automata) combining Z and FA (Finite Automata). The generation algorithm from the basic structures of Z to ZA is studied to enhance the practical usage of our hybrid model. Further on, a model checking algorithm ZAMC is proposed to automatically verify the temporal/data constraints within the structure and behavior specified by ZA. Finally, a case study is used to illustrate the correctness and feasibility of ZDVS.

2 ZA Model

To verify dynamic temporal behavior of a system, we take the advantages of the predicate logic description ability of Z notation and the temporal description ability of FA. In this section, we give the formal definition of ZA model and design a generation algorithm to transform Z static model to ZA.

2.1 Formal Definition of ZA

In order to facilitate the study of model checking Z, and to establish a formal relationship between Z specification and ZA model, we present a formal definition of ZA. First of all, we define some useful notations in Definition 1, to help the definition of ZA. Secondly, we define a simplified structure of Z models as ZA_{static} in Definition 2, to delineate the element set we need to map to FA.

Definition 1. Assume that a Z schema declaresas $x_1, \ldots, x_n$ its variables, the invariants in the schema can be seen as an n-ary predicate. We use the form S_{p_n} to state the n-ary prediction. In particular, we denote the pre-conditions of an operation schema *Op* as pre_{Op}, and the post-conditions as $post_{Op}$.

Definition 2. A ZA Static Model

$$ZAstatic = (State, Operation) \tag{1}$$

consists of the following elements:

- *State* is a finite set of state spaces. Its elements correspond to the state schemas in a Z specification.
- *Operation* is a finite set of operations. Its elements correspond to the operation schemas in a Z specification.

We has designed and implemented a Z notation parser as a crucial component of ZDVS, to analyze an input Z model and transform it to a ZA_{static} model written by C++.

The formal definition of ZA is given as follows.

Definition 3. A ZA Model

$$\mathrm{ZA} = (S, S_0, \Sigma, \delta, F, \mathrm{M}) \tag{2}$$

consists of the following elements:

- S is a finite set of states.
- $S_0 \subseteq S$ is a finite set of initial states.
- Σ is a finite set of operations.
- $\delta \subseteq (S \times \Sigma \times S)$ is a finite set of state transitions. A state transition $(s, a, s') \in \delta$ represents that system transits from state s to s'.
- F is a finite set of end states.
- M denotes a binary relationship between S, Σ and the elements in ZA_{static}, such that $\mathrm{M} = (F_s, F_\Sigma)$. The definitions of its tuples are as follows:
  F_s is a mapping from S to *State*, i.e.,

$$F_s(s) = state, \text{ where } s \in S, \ state \in State \tag{3}$$

F_Σ is a mapping from Σ to *Operation*, i.e.,

$$F_\Sigma(a) = op, \text{ where } a \in \Sigma, \ op \in Operation \tag{4}$$

According to Definition 3, the prerequisite of a state transition $(s, a, s') \in \delta$ is that state s meets the pre-conditions, and s' meets the post-conditions of operation a, that is, $s| = pre_{F_\Sigma(a)}$ and $s'| = post_{F_\Sigma(a)}$ hold.

2.2 The Generation of ZA Model

After the formal definition of ZA model, we propose a generation algorithm which builds a ZA model from the ZA_{static} model. The input of the algorithm is ZA_{static} = (*State*, *Operation*), and the output is a ZA model.

The procedure of the generation is described as follows:

- *Step 1*. Initialization phase. Initialize s_0 and a stack St;
- *Step 2*. Add s_0 to S_0 and S. Build the one-to-one relationship between *Operation* and Σ, and the relationship between *State* and S.;
- *Step 3*. Push s_0 onto St;
- *Step 4*. Get the top element s of St, while St is not empty;
- *Step 5*. For each operation a_i in Σ, if $s \models pre_{F_\Sigma(a_i)}$ is true, perform a_i on s and get a new state s';
- *Step 6*. If $s' \models post_{F_\Sigma(a)}$ is true and it meets the global constrains, add (s, a_i, s') to δ;
- *Step 7*. If s' doesn't exist, add s' to S and push it onto St. If it's an end state, add it to F;
- *Step 8*. Go to *step 5*, until all operations in Σ are considered;
- *Step 9*. Go to *step 4*, until St is empty;
- *Step 10*. Output the ZA model.

Figure 1 gives the pseudo code of ZA model generation algorithm according to the above procedure.

3 Model Checking ZA

In this section, we define a set of temporal logic formulas called ZATL (Z-Automata Temporal Logic) to describe the expected temporal properties of the system. Further on, a model checking algorithm called ZAMC (Z-Automata Model Checking) is proposed to perform the verification towards such properties.

3.1 Temporal Logic Formula Towards ZA

First of all, we define some useful notations in Definition 4, to help the definition of ZATL.

Definition 4. A finite state sequence $\pi = (s_0, s_1, \ldots, s_n)$ is used to describe a state transition path in a ZA model, where $s_i \in S$ and $(s_i, a, s_{i+1}) \in \delta (i \in N,\ i < n)$. We define that π and each state s_i in π satisfy the relation $\pi[i] = s_i$. We denote $\Pi(s)$ as a set of paths starting from s, that is, $\prod(s) = \{\pi | \pi[0] = s\}$. By $[[\varphi]]_s$ we denote a set of states where formula φ holds. By $[[\varphi]]_\pi$ we denote a set of paths in which all states satisfy φ.

ZATL compliances with the following elements:

- $\Box$ is the always operator. $\Box\varphi$ means that φ is always true.
- $\diamond$ is the sometimes operator. $\diamond\varphi$ means that φ is sometimes true.

input: $ZM_{analyzed}$=(*State*,*Operation*) , *InitialInfo*
output : ZA = (S, S_0, Σ, δ, F, M)
Initialize ZAModel s_0←*InitialInfo;* S, S_0, Σ, δ, F← ϕ*; InitStack* <*Stack*>St;
$S_0 \leftarrow \{s_0\}$,$S \leftarrow S \cup \{s_0\}$
for each $op_i \in Operation$
    Define $F_\Sigma(a_i) = op_i$; $\Sigma \leftarrow \Sigma \cup \{a_i\}$
end for
<*Stack*>St← s_0
while ! *StackEmpty*(St)
    <*Stack*>St → s
    for each $a_i \in \Sigma$ do
        if $s| = pre_{F_\Sigma(a_i)}$
        $s' \leftarrow a_i(s)$
        if $s'| = post_{F_\Sigma(a_i)} \wedge s'| = F_S(s')_{p_n}$
            $\delta \leftarrow \delta \cup \{(s, a_i, s')\}$
                if $'\notin S$ $S \leftarrow S \cup \{s'\}$; <*Stack*>St ←$s'$;
                end if
                if s ' *is final state* then$F \leftarrow F \cup \{s'\}$;
                end if
            end if
        end if
    end for
end while

Fig. 1. The pseudo code of ZA model generation algorithm

- **A** is the all quantifier.
- **E** is the exist quantifier.

Definition 5. The syntax of ZATL is defined as follows:

- The atomic proposition φ_{p_n} is a ZATL formula.
- If φ and ψ are ZATL formulas, so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$.
- If φ is a ZATL formula, so are A $\Box$ φ, **E** $\Box$ φ, **A** $\diamond$ φ, **E** $\diamond$ φ.
- If and only if the above rules are used for limited times, we get a ZATL formula.

Definition 6. The semantics of ZATL is defined as follows:

- $[[\varphi_{p_n}]]_s = \left\{ s|\varphi| = F_S(s)_{q_m} \right\}$, where p and q are predicates, $n, m \in N$;
- $[[\neg\varphi]]_s = S - [[\varphi]]_s$;

- $[[\varphi_1 \wedge \varphi_2]]_s = [[\varphi_1]]_s \cap [[\varphi_2]]_s$;
- $[[\varphi_1 \vee \varphi_2]]_s = [[\varphi_1]]_s \cup [[\varphi_2]]_s$;
- $[[\mathbf{A} \Box \varphi]]_\pi = \{\pi | \forall \pi \in \prod(s_0) \cdot (\forall s \in \pi \cdot (s_{p_n} | = \varphi))\}$
- $[[\mathbf{A} \diamond \varphi]]_\pi = \{\pi | \forall \pi \in \prod(s_0) \cdot (\exists s \in \pi \cdot (s_{p_n} | = \varphi))\}$;
- $[[\mathbf{E} \Box \varphi]]_\pi = \{\pi | \exists \pi \in \prod(s_0) \cdot (\forall s \in \pi \cdot (s_{p_n} | = \varphi))\}$
- $[[\mathbf{E} \diamond \varphi]]_\pi = \{\pi | \exists \pi \in \prod(s_0) \cdot (\exists s \in \pi \cdot (s_{p_n} | = \varphi))\}$.

The first four formulas represent the composition mode of φ. The last four formulas are temporal logic formulas used to describe properties of system.

$\mathbf{A} \Box \phi$ denotes that all the states in the system satisfy φ. $\mathbf{A} \diamond \varphi$ denotes that there is at least one state satisfying φ in every path of the system. $\mathbf{E} \Box \varphi$ denotes that there is at least one path in which all the states satisfy φ. And $\mathbf{E} \diamond \varphi$ denotes that there is at least one state in the system satisfying φ. These formulas can be used to describe the dependability properties of system such as liveness and safety.

3.2 ZAMC Algorithm

In order to verify whether the given system satisfies the property described by a ZATL formula, we propose a model checking algorithm based on counterexample searching strategy. For each formula we design different searching strategies depending on its semantics.

(1) Input formula 1: $\varphi' = \mathbf{A} \Box \varphi$.

Formula 1 denotes that formula φ holds for all the states in the system. If the given ZA model meets this condition, the system satisfies formula φ'. Otherwise, the system doesn't satisfy φ'. The counterexample of formula 1 is a state on which formula φ doesn't hold. The verification procedure of formula 1 is as follows.

Step 1.1. Get an unprocessed element s in set S;
Step 1.2. If s doesn't satisfy φ, the system doesn't satisfy φ'. Output current state s as a counterexample, and the verification ends up with *false*;
Step 1.3. If s satisfies φ, go to *Step 1.1*;
Step 1.4. Output *true* of this verification.

(2) Input formula 2: $\varphi' = \mathbf{A} \diamond \varphi$.

Formula 2 denotes that there is at least one state satisfying φ in every path of the system. If the given ZA model meets this condition, the system satisfies formula φ'. Otherwise, the system doesn't satisfy φ'. The counterexample of formula 2 is a path in which none of the states satisfies φ. The verification procedure of formula 2 is as follows.

Step 2.1. If initial state s_0 satisfies φ, the system satisfies formula φ'. Verification process ends, and output *true*;
Step 2.2. Otherwise, put s_0 into queue Q;
Step 2.3. While Q is not empty, get state s from Q;
Step 2.4. If s doesn't satisfy φ, update current searching path. If $s \in F$, we find a counterexample. So the verification ends up with *false*;
Step 2.5. If s satisfies φ, put all unprocessed successor states of s into Q;
Step 2.6. Go to *Step 2.3*, until Q is empty;
Step 2.7. If we find a counterexample, output the counterexample according to searching path. Otherwise, output *true*.

(3) Input formula 3: $\varphi' = \mathbf{E} \Box \varphi$.

Formula 3 denotes that there is at least one path in which all the states satisfy φ. If the given ZA model meets this condition, the system satisfies formula φ'. Otherwise, the system doesn't satisfy φ′, that is, in every path of the system there is at least one state that doesn't satisfy φ. As a result, we can formalize it as $\mathbf{A} \diamond \neg\varphi$, thus the verification of formula 3 can use the verification procedure of formula 2. Firstly, we verify whether the system satisfy formula $\mathbf{A} \diamond \neg\varphi$. Then, invert the result as output.

(4) Input formula 4: $\varphi' = \mathbf{E} \diamond \varphi$.

Formula 4 denotes that there is at least one state in the system satisfying φ. If the given ZA model meets this condition, the system satisfies formula φ'. Otherwise, the system doesn't satisfy φ', that is, none of the states satisfies φ. The verification procedure of formula 4 is as follows.

Step 4.1. Get an unprocessed element s in set S;
Step 4.2. If s satisfies φ, the system satisfy φ'. And the verification ends up with *true;*
Step 4.3. If s doesn't satisfy φ, go to *Step 4.1*;
Step 4.4. Output *false* of this verification. Any state can be used as a counterexample.

Figure 2 gives the procedure of ZAMC.

4 Design and Application of ZDVS

Based on the ZA model and the ZAMC algorithm, we realize a prototype system called ZDVS (Z Dynamic Verification System). In this section, we present its framework and procedure. A case study is used to illustrate the correctness and effectiveness of our method.

```
input : ZA = (S, S0, Σ, δ, F, M), where M = (FS, Fδ), φ′ ∈ ZATL
output: ZA| = φ′?
Initialize checkpath ← ϕ
for each φ′ ∈ Sub(φ) do
  case φ′ = A □φ
    for each s ∈ S
        if s| = φ continue;
        else result ← 0; output s; break;
    end for
     result ← 1;
  case φ′ = A ◇φ
     if s0| = φ result ← 1; break;
     else   result ← -1
        <queue>Q ←s0
        while !QueueEmpty(Q)
            <queue>Q → s
            if ! (s| = φ) then
                if s ∈ F result ←0; output any π ∈ ∏(s0) ; break;
            for each s′ (s, a, s′) ∈ δ if s′ is not visited then
          <queue>Q ←s′
        end while
    end if
    if (result = -1)  result ← 1
  case φ′ = E □φ
    if ! (s0| = φ) then  result ← 0; output any π ∈ ∏(s0); break;
    else  result ← -1
        <queue>Q ←s0; update checkpath
        while !QueueEmpty(Q)
            <queue>Q → s
            if s| = φ then
                if s ∈ F result ← 1; break;
            for each s′ (s, a, s′) ∈ δ
                if s′ is not visited then <queue>Q ←s′ ; update
          checkpath
        end while
    end if
    if (result = -1)  result ← 0; output a counterexample according to
checkpath
  case φ′ = E ◇φ
    for each  s ∈ S
        if s| = φ result ← 1; break;
        else continue;
    result ← 0; output any s ∈ S
return result;
```

Fig. 2. The procedure of ZAMC

4.1 Framework of ZDVS

ZDVS has three main modules, including the modeling module, the verification module and the user interface module. Figure 3 gives the flow diagram of ZDVS.

Firstly, we transform Z specification to ZA_{static} model by implementing a parser in the modeling module, to analyze Z notation and transform it to ZA_{static} model written by C++. Then, we implement the ZA model generation algorithm in the modeling module to build the corresponding ZA model.

In the verification module, we design a command parser to obtain the input ZATL formulas (through the user interface module), and implement the ZAMC algorithm to verify the ZA model.

The user interface module provides the necessary interaction between user and system, such as the input of ZATL formulas, and the display of the counterexamples if the verification ends up with *false*.

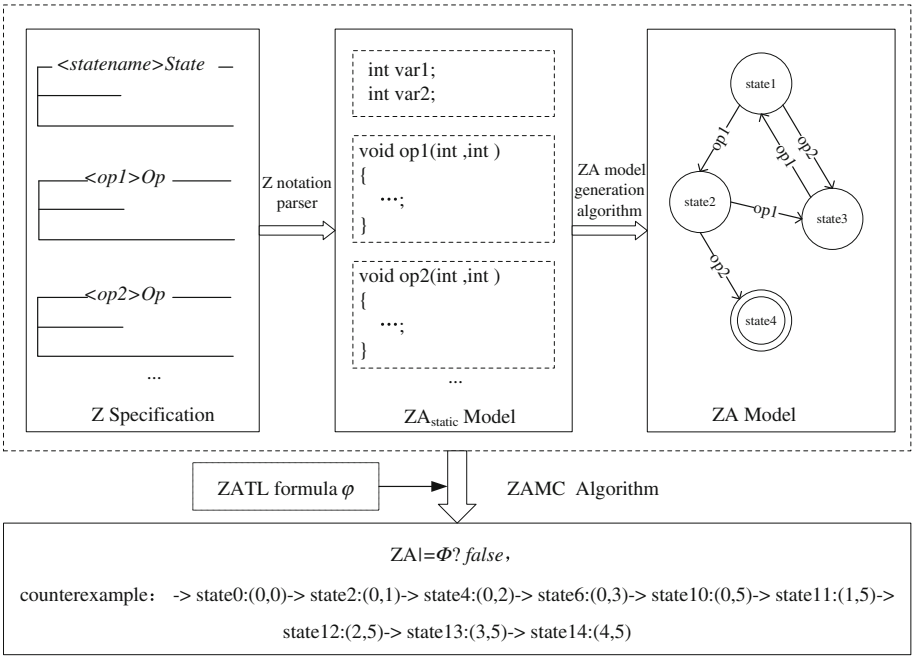

Fig. 3. The flow diagram of ZDVS

4.2 A Case Study

In this section, we describe the modeling and analysis procedure of a case study to illustrate the correctness and effectiveness of ZDVS. The inputs of ZDVS are Z specification represented by LaTeX and ZATL formula φ, while the output is the result of model checking. If the result is *false*, it also gives a counterexample.

The case study contains 4 schemas as shown in Fig. 4, including a state schema and 3 operation schemas. The corresponding ZA model generated by ZDVS contains 30 states and 3 actions.

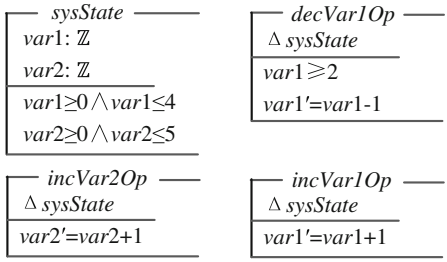

Fig. 4. Z specification of the instance

We use 4 temporal logic formulas as inputs of the model checking process, which are listed as follows.

A □ var1 ≤ 3, denotes that var1 in all the states of the system is equal or greater than 3.
A ◇ var1 > 4, denotes that for every path of the system there is at least one state in which var1 is greater than 4.
E □ var1 ≥ var2, denotes that there is at least one path in which all the states satisfy var1 ≥ var2.
A◇ var1 ≥ var2 && var2 > 2, denotes that there is at least one state in the system satisfying var1 ≥ var2 && var2 > 2.

Figure 5 shows the verification result of formula **A**◇ var1 > 4. Since the result is *false*, the system outputs a counterexample.

Illustration on the case study shows that the proposed ZA model can enhance the descriptive power by combining Z notation and FA. The prototype system ZDVS is able to correctly parse the input Z specification and generate the corresponding ZA model, accept and analyze the input ZATL formulas, and verify the system temporal properties automatically and effectively.

5 Discussion and Conclusion

In order to verify the temporal properties of Z model, this paper designs and implements a prototype system ZDVS to perform the model transformation and model checking on a proposed hybrid software model ZA. Our main contributions are as follows:

1. A formal software model called ZA is defined by combining Z notation and FA. The proposed ZA model can specify not only static structure and operation specifications, but also temporal constraints of the targeted system. A generation algorithm is designed to build ZA model from ZA_{static}, a simplified structure of Z specification.

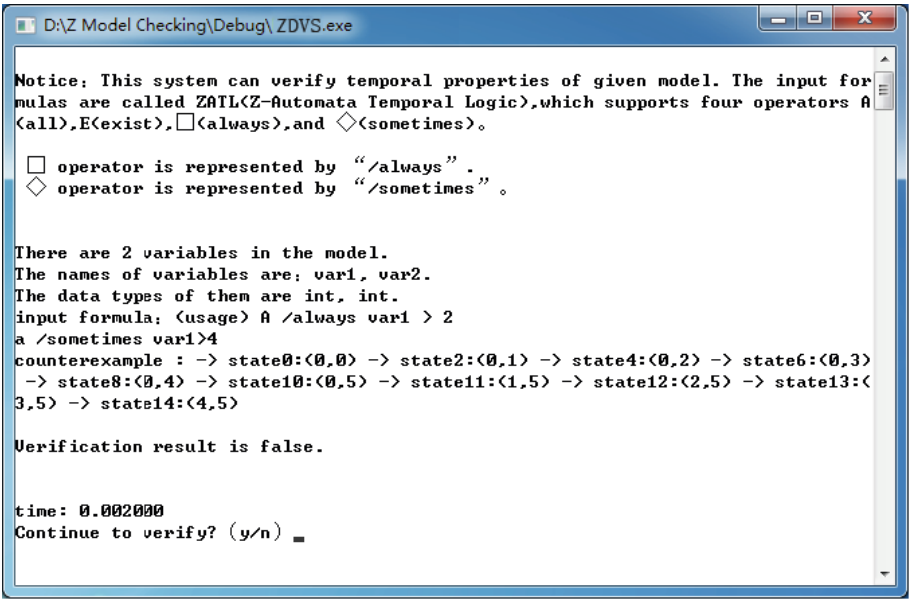

Fig. 5. Verification result

2. A model checking algorithm towards the ZA model is proposed. Firstly, a temporal logic called ZATL is defined to describe the temporal properties of the ZA model. Then, we propose a model checking algorithm called ZAMC to verify the ZATL formulas.
3. A prototype system called ZDVS is realized to perform our proposed methods. A case study is used to illustrate the correctness and effectiveness of ZDVS.

Future work of this paper includes further study on hybrid software models and their temporal logic, and the performance enhancement of their model checking algorithms.

References

1. ISO I. IEC 13568: 2002: Information technology–Z formal specification notation–Syntax, type system and semantics. ISO (International Organization for Standardization), Geneva, Switzerland (2002)
2. SPIVEY J M. The Z notation: a reference manual. In: International Series in Computer Science. Prentice-Hall, New York, NY (1992)
3. Wordsworth, J.B.: Software Development with Z: A Practical Approach to Formal Methods in Software Engineering. Addison-Wesley Longman Publishing Co. Inc, Reading (1992)
4. Ince, D.C., Ince, D.: Introduction to Discrete Mathematics, Formal System Specification, and Z. Oxford University Press, Oxford (1993)

5. Abrial, J.-R., Schuman, S.A., Meyer, B.: Specification language. In: McKeag, R.M., Macnaghten, A.M. (eds.) On the Construction of Programs: An Advanced Course. Cambridge University Press, Cambridge (1980)
6. Z/EVES Homepage (2015). http://z-eves.updatestar.com/
7. ProofPower Homepage (2015). http://www.lemma-one.com/ProofPower/index/
8. Martin, A.P.: Relating Z and first-order logic. Formal Aspects Comput. **12**(3), 199–209 (2000)
9. ProB Homepage (2015). http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page
10. Jaza Homepage (2005). http://www.cs.waikato.ac.nz/~marku/jaza/
11. Zlive Homepage (2015). http://czt.sourceforge.net/zlive/
12. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
13. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
14. Smith, G.P., Wildman, L.: Model checking Z specifications using SAL. In: Treharne, H., King, S., Henson, M.C., Schneider, Steve (eds.) ZB 2005. LNCS, vol. 3455, pp. 85–103. Springer, Heidelberg (2005)
15. Cao, Z., Wang, H.: Extending interface automata with Z notation. In: Arbab, F., Sirjani, M. (eds.) FSEN 2011. LNCS, vol. 7141, pp. 359–367. Springer, Heidelberg (2012)
16. Cao, Z.: Temporal logics and model checking algorithms for ZIAs. In: 2010 2nd International Conference on Proceedings of the Software Engineering and Data Mining (SEDM). IEEE (2010)