American Journal of Computer Science and Technology
2018; 1(2): 44-54
http://www.sciencepublishinggroup.com/j/ajcst
doi: 10.11648/j.ajcst.20180102.12

The Examination of Shall Be Impossible Situations for Verification During Execution

Rimantas Seinauskas

Software Department, Informatic Faculty, Kaunas University of Technology, Kaunas, Lithuania

Email address:
rimantas.seinauskas@ktu.lt

To cite this article:
Rimantas Seinauskas. The Examination of Shall Be Impossible Situations for Verification During Execution. *American Journal of Computer Science and Technology*. Vol. 1, No. 2, 2018, pp. 44-54. doi: 10.11648/j.ajcst.20180102.12

Received: February 18, 2018; **Accepted**: March 5, 2018; **Published**: March 23, 2018

Abstract: Runtime verification is looking for violations of the properties of the system functioning. Finding and describing the system properties that indicate behavioural disorders is a complex and labour-intensive process that needs to be automated. This article describes how system properties can be determined automatically during the correct functioning. Inspection of the combinations of fulfilled properties makes it possible to detect more system problems. Three methods of handling property combinations are offered. The methods are based on the examination of the input sequences and output results. In order to increase the volume of the properties of the system under consideration, only the possible pairs of properties are analysed. Pairs are formed from the output properties, as well as from the input conditions and output properties, and the maximum possible number of property pairs is evaluated. The available property pairs are captured during the operation. Impossible combinations of properties that never occur during the execution highlight situations that are not possible during proper functioning. Capture of impossible property pairs during verification indicates system problems. During the experiment, five types of disorders and three detection methods were considered. Experimental results show that there is no single best method for detecting disorders. Therefore, it is appropriate at the same time to use several methods to detect disorders. The experiment shows how much of the disorders can detect the proposed approach.

Keywords: Runtime Verification, System Properties, Capturing of Properties, Inspection of Proper Functioning

1. Introduction

The system design is based on a specification that describes the operations as a function of all possible input data. Verification and testing verifies whether only the possible situations are received. Runtime verification checks the occurrence of described impossible situations. Impossible situations allow us to see the system from a different perspective. Impossible and possible situations represent a set of all existing situations. Runtime verification of situations that are impossible allows additional increase of system reliability. Inspection may be performed during service or before.

Verification is based on inspection of the expected results when the input stimuli are known in advance. A verification test case indicates the expected results for the selected input stimuli. The runtime verification decides the correctness of functioning during the execution. Input stimuli and the expected results are not known in advance. The decision about the correctness of functioning is taken on the basis of the characteristics (properties) of the results obtained.

The goal of runtime verification is to increase the reliability of existing systems. Software system reliability is increased in various ways. The key is to carry out thorough verification and testing. Automatic test generation faces the problem of how to determine the expected reaction (the oracle problem) using the test data. One way to do this is to independently design and implement two versions of the programs. Comparison of the results of the two program versions can be used instead of using separate oracle. The probability that bugs will be the same in both program versions is limited. A results mismatch with the same input data shows that there is an error in one of the versions. But it does not solve problems due to errors in the specification. Two independently derived specifications can also be considered.

Duplication of work is very expensive. Therefore, instead of using version of the application, a model of properties that have to be fulfilled for all the input data can be generated. The independent creation of such a model can be less labour

intensive compared to the creation of other versions. Drawing up a model of properties that allow the detection of program bugs may require considerable of effort as well. The question of whether such a property model that can expose all the bugs in the program can be created remains open. The property model makes it possible to automate test generation, to use randomly generated data, and to carry out runtime verification.

Design errors, software bugs, hardware defects, and human contingency condition system failure. Failure detection is based on the description of the correct functioning. Ageing defects in hardware and software changes can occur during operation. In this case, incorrect functioning can be captured during a service.

Input data that are used during runtime verification are not known in advance. Evaluation of the results is based on the expected results to be calculated. Determining the correct expected results mainly consumes resources during runtime verification, but these calculations can be carried out in parallel with the main execution calculations.

Verification and runtime verification become similar if the behavioural model determines the correctness of results. Working input stimuli are used during runtime verification, while artificial input stimuli can be used during verification. The calculation time and the necessary computer resources are very important for the runtime verification, because correctness of results is carried out in real time. Correctness of results is carried out after each step and may be executed in parallel with the next step. Parallelization is not possible if the decision should be taken into account before the next step. Therefore, correctness of results time is a critical parameter and the minimization of the time taken by the decision is a very important goal with regard to runtime verification.

An exact behaviour model like the finite-state machine [FSM] model enables one to obtain and compare the output results. A simplified behavioural model makes it possible to check only some properties like possible transitions between states or possible consistency of events in the execution. Impossible transitions between states or impossible consistency in the execution of the events indicates a failure. The possible transitions between states or possible execution sequences of events must be calculated and made available in advance. If this is not feasible, the possible transitions between states and the execution sequences of events can be registered in service until saturation is obtained. In this case, the answer to the question of how much a simplified analysis reduces the probability of detection of problems is very important.

Simplification of the exact behaviour model is also possible by indicating undefined output values (such as x) when the model cannot accurately define them. Only precisely defined values are compared during verification. Such a principle may allow the detection of more failures compared with inspection only of impossible properties. In addition, the establishment of such a model may be simpler.

The internal states are not always available during service, nor is other internal information. There is always a target for verification using only input and output data. Rating input sequences like integrated units partially compensate for the loss of information about the inner states.

In general, it is difficult to identify distinct properties of results so that the malfunction can be detected. Not all problems lead to nonfulfillment of such properties. Examination of combinations of properties allows the detection of more failures. A combination of properties can indicate failure, but when taken in isolation, properties do not indicate failure. The objective of the study is to find the combination of properties that captures failures during execution. We are considering a situation in which we do not have a functioning model and the potential properties are extracted using a program in the working environment.

The main drawback of this method is that nonfulfillment of properties extracted using a program in the working environment does not guarantee system malfunction. It may be that usage has highlighted a new property. Basically, this is just a warning that the results must be carefully analysed. A possible combination of properties is supplemented if the analysis indicates the absence of failure.

2. Related Work

Verification of the system that is being developed includes an assessment with respect to the specification. Validation involves an inspection system in relation to the user's needs. Testing provides quality assurance regarding the system. Runtime verification examines the fulfilment of properties during execution.

Runtime verification can rely on a behavioural model or a property model. The behavioural model provides the expected output results. Comparison of the expected results and the results obtained during service makes it possible to detect disturbances.

Detection of system failures requires additional resources. Input data are used at run time. In general, a sequence of input stimuli comprises input data. A model of behaviour can determine the expected reaction of the input data. Simulation of the input data can be carried out in parallel with the real execution. This is demonstrated in Figure 1. The duration of the service does not change if the simulation is faster than the actual implementation.

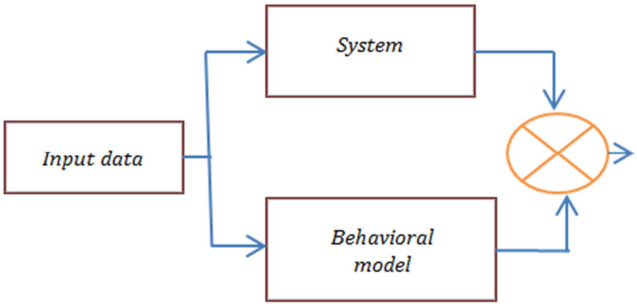

Figure 1. *Runtime verification with behavioural model.*

The probability of detecting system malfunction increases

if a behavioural model is created independently from the specification. In this case, it is less likely that the specification and behaviour model will have the same mistakes. The implemented system or behaviour model may have a mistake when the results do not match.

Comparison of the simulation and execution results enables us to detect system malfunctions but lengthens the duration of data processing. The comparison will not affect the duration of functioning of the system if it is carried out in parallel with the processing of the following input data. Verification should not last longer than the processing of input data. In this case, the solution of the verification action will be available later. This is demonstrated in Figure 2.

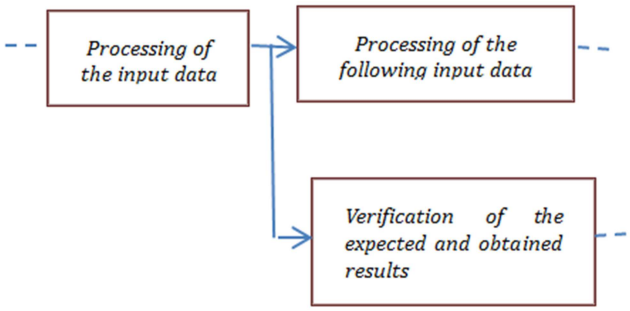

Figure 2. *Parallel execution of verification and of processing of the input data.*

Possession of the internal state of the system increases the problem-detection capabilities. However, monitoring of the system's internal states may affect its performance. An answer to the question of how much internal state tracking increases the detection of problems would be very helpful.

The exact behaviour of the system model is difficult to describe. Simplification of the model facilitates this work. An undefined value is indicated when the model cannot accurately determine the output values. The comparison of non-simulated values is not carried out. In this case, the detection of certain system failures may be lost. A compromise between model simplicity and non-detection of problems during runtime verification is necessary.

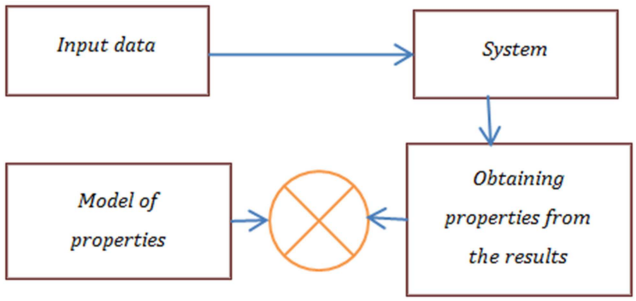

Figure 3. *Runtime verification with model of properties.*

Runtime verification with the model of properties is shown in Figure 3. Properties are obtained from the operational results and compared with the model of properties. Nonfulfillment of any properties indicates the presence of a failure in the system. Flair states that comparison of the results with the behavioural model should allow the detection of more system failures than comparison of the results with the model of the properties. The answer to the question of how many more failures would be detected would be helpful in any case. The property model raises many doubts about its usefulness due to lack of validity experiments.

Runtime verification checks whether the behaviour of the system complies with properties derived from the specification [1]. A model of properties taking into account the wishes of the user or other aspects of the ontology can highlight specification errors as well.

The properties are the basis for determining whether the system is compatible with the required behaviour. Properties can be expressed in the form of temporal logic [2] or in the form of finite automata [3]. Security properties are easier to express using automata rather than a logic-based formalism [4]. Linear temporal logic [5] allows the creation of properties, which links the events and their sorting. Possibilities of formal descriptions of properties have already been explored well. Property violations are detected by the monitor, which analyses the results and finds discrepancies between the observed and expected system behaviours. Runtime monitors are often derived directly from the properties, and properties are often simply written by the designer [6].

Runtime verification requires additional resources during execution and this may affect the functioning of the system. Fragmentation of additional costs in relation to place and time can reduce the overall impact on the runtime verification as claimed in [7]. Recently, much attention has been paid to comparing runtime verification tools [8]. However, there is currently insufficient attention to the automatic determination of properties during execution. Stochastic models [9] and stream processing applications [10] is a promising trend in solving runtime verification problems.

Runtime verification can increase the system reliability, but consumers always want to know how much it can improve by and how much it will cost. The answers to these questions are the hardest ones.

3. Conditions and Properties

Input stimuli are fed into the inputs, and output results are obtained during execution of the object. The output results of each stimulus satisfy or do not satisfy some properties. The situation can be displayed as a vector p = <p1, p2,..., p_i,..., p_m >, where p_i = 1 when the property is fulfilled and p_i = 0 when the property is not fulfilled. Similarly, the input stimuli can fulfil some of the data conditions, whereas some data conditions cannot be fulfilled. This can be represented by another vector C = <c1, c2,..., c_i,..., c_n >, where c_i = 1 when the condition is fulfilled and c_i = 0 if the condition is not fulfilled. Vectors C and P describe the situation after the execution of the input stimulus.

What indicates system failure? In the simplest case, nonfulfillment of at least one of the properties indicates system failure. Often system failure is defined as nonfulfillment of several properties or as nonfulfillment of properties when another property is fulfilled. In this case, there is talk about the combination of fulfilled properties. Combinations of fulfilled properties enable the identification of more system failures. Similarly, we can talk about combinations of fulfilled input conditions. Combinations of fulfilled input conditions give more options and flexibility in describing the occurrence of system failures.

Input data can have at most 2^n combinations of fulfilled conditions. The output results can have at most 2^m combinations of fulfilments of properties. Object behaviour can limit the amount of combinations of fulfilled properties and some combinations will be unavailable. Monitoring of shall be impossible combinations of properties on the object output indicates malfunctions. Let us say that we have a set F of possible combinations of properties and a set U of impossible combinations of properties, where $|F| + |U| = 2^m$. Only the combinations of the set F are obtained during correct operation. A malfunctioning case, instead of the expected combinations of the set F, may be another combination of sets F and U. Obtaining combinations of the set U unambiguously shows that malfunctioning has occurred. This feature can be used for identification of problems in system execution.

Each combination of input conditions is related to set combinations of possible properties of outputs. Possession of such information enables the detection of more failures. A failure that is replacing a combination of the set F with a different combination of the set F can be found in this case.

Similarly, the functional environment of the object may limit the amount of combinations of input conditions. Linking the input conditions and output properties creates preconditions for obtaining more invalid situations that indicate a failure.

Let us consider a small example. Let us say that the relevant function has input data of four numeric variables and that three numerical variables are obtained at the output after the execution of the function. All variables can be both positive and negative. If it is known that a certain output variable can only be positive, then the output variable taking a negative value, indicates system failure. Such a strict property may seldom be valid. A situation that takes output variables that can be both positive and negative separately is more likely to occur, but let us say that a situation in which the first variable is positive, the second negative, and third again positive cannot occur. Therefore, examination of the property combinations is more flexible.

The following describes the properties p1, p2, and p3 of outputs, which have the value of 1 when the property is fulfilled and the value 0 otherwise. Accordingly, the value of 1 marks the fulfilment of the conditions c1, c2, c3, and c4 of inputs and the value 0 otherwise. Possible combinations of fulfilled conditions and properties are shown in Table 1, as an example. In general, different properties can be obtained under the same input conditions. This is demonstrated in rows 8 and 9.

Table 1 shows that the set F = {000, 001, 010, 100, 110} has five possible property combinations and the set U = {011, 101, 111} has three impossible combinations of properties. Combinations of properties of the set U compose the identification function of errors.

Table 1. *Possible combinations of conditions and properties.*

No.	c1	c2	c3	c4	p1	p2	p3
1	0	0	0	0	0	0	1
2	0	0	0	1	0	0	1
3	0	0	1	0	0	0	1
4	0	0	1	1	0	0	1
5	0	1	0	0	0	0	0
6	0	1	0	1	0	0	0
7	0	1	1	0	0	0	0
8	0	1	1	1	0	1	0
9	0	1	1	1	0	0	0
10	1	0	0	0	0	0	0
11	1	0	0	1	0	0	0
12	1	0	1	0	0	0	0
13	1	0	1	1	0	0	0
14	1	1	0	0	0	0	0
15	1	1	0	1	0	0	0
16	1	1	1	0	1	0	0
17	1	1	1	1	1	1	0

We will consider how the sets F and U can be used in practice for verification during the execution. During verification, it is necessary to determine what conditions the input data fulfil and what properties the results obtained fulfil and to determine whether the properties of the obtained results are possible according to the table of combinations of conditions and properties. How can a table of conditions and possible properties be obtained? In general, it follows from the specification. However, in practice it is impossible to write out all possible combinations of conditions and properties. This can be find during a long operation of the correctly functioning system. However, a table of possible conditions and properties can be overwhelming. It is possible to reduce it by listing only possible combinations of

properties (F set). The inverse of the set U should be used for verification. This simplification can significantly reduce the verification accuracy, because there is a loss of connection to the combinations of input conditions. In general, the set U can be empty and cannot be used in such cases. Another problem is that the set U can be enormous and in practice, it cannot be listed as well. In this case, it is necessary to look for processing techniques of the set U that are acceptable in practice. Finding the set U can require a lot of resources as well. In addition, it is still necessary to answer the question of what part of the failures causes the appearance of combinations of the set U. We will try to discuss this.

Functional disturbances can greatly change the output properties in different ways. One way of analysing the situation includes changing the table that describes the conditions and properties with the corresponding logic circuit and its fault studies. A stuck-at fault imitates system disorder. This is done only for demonstration purposes. Table 1 becomes similar to the truth table when line 9 is discarded. Properties assigned to outputs and conditions assigned to the inputs allow us to get the logical functions of outputs (p1 = c1c2c3; p2 = c2c3c4; p3 = $\overline{c1}\ \overline{c2}$). A synthesized circuit is shown in Figure. 4. For demonstration purposes, we will consider how stuck-at faults of the synthesized circuit affect outputs. Eleven stuck-at faults of the synthesized circuits are listed in Table 2. The third column shows how stuck-at faults are changing the properties of outputs. Fault c1 (1) ≡ 1 corresponds to the fixing of a value as one in the input of element (1), which is connected to the input c1. In case of this fault, the sample circuit behaviour changes as though the value of the input vector conditions c1 value were always equal to one. In Table 1, we can see how the properties of the output changes after the amendment of condition c1 always to one. The output values 000 change to the values 100 (010 → 110) for the fault c1 (1) ≡ 1. Similarly, the output property values 000 change to values of 100 (000 → 110) at the input condition values c1c2c3c4 = 1010 and 1011 (lines 12 and 13). We see that only two faults (c2 (2) ≡ 1, p3 (3) ≡ 1) cause combinations of the set U (highlighted) and other faults only cause combinations of the set F. Detection of these failures requires information about the possible properties of outputs for the relevant input conditions. In this case, it is necessary to consider the combinations of pairs of input conditions and output properties. The number of possible combinations of pairs is equal to 2^{m+n}.

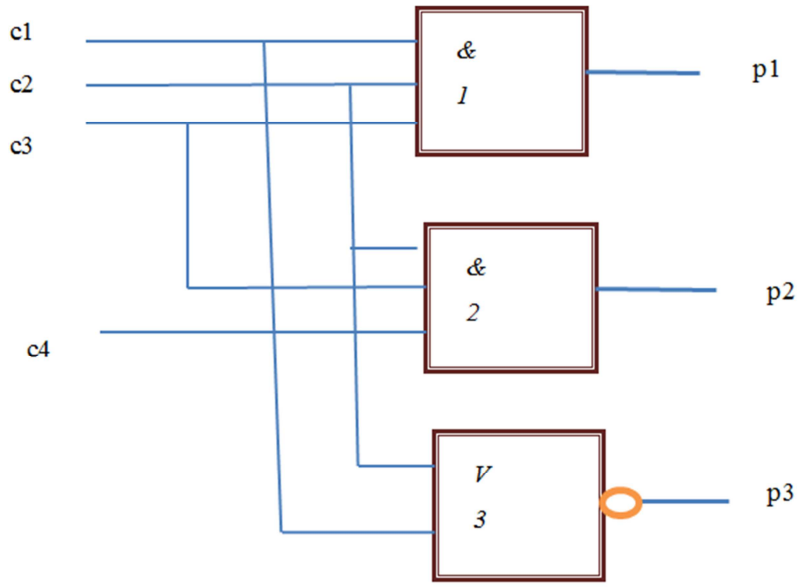

Figure 4. *Synthesized circuit according to Table 1.*

Table 2. *Influence of stuck-at faults on the output results.*

No.	Faults	Output changes	OO options	IO options
1	c1 (1) ≡ 1	000→100 010→110		c1↔ p1 (01)
2	c2 (1) ≡ 1	000→100 000→110		c2↔ p1 (01)
3	c3 (1) ≡ 1	000→100 000→110		c3↔ p1 (01)
4	p1 (1) ≡ 0	100→000 110→010		
5	c2 (2) ≡ 1	000→110 001→011	(p2, p3) – (11)	c2↔ p2 (01)
6	c3 (2) ≡ 1	000→010 000→110		c3↔ p2 (01)
7	c4 (2) ≡ 1	000→010 100→110		c4↔ p2 (01)
8	p2 (2) ≡ 0	010→000 110→100		
9	c1 (3) ≡ 1	001→000 001→010		
10	c2 (3) ≡ 1	001→000 001→010		
11	p3 (3) ≡ 1	000→001 010→011 100→101 110→111	(p2, p3) – (11) (p1, p3) – (11)	c1↔ p3 (11) c2↔ p3 (11)

Processing of large sets of combinations is practically impossible. To simplify the calculations, we can restrict ourselves to checking only the amount of all possible pairs of set. The scope of pairs of a set of N elements is equal to N (N – 1)/2. Here is the estimated symmetry of the element pairs. The pairs of elements (e1, e2) and (e2, e1) denote the same information. Each pair of elements of the sets has four options for the fulfilment of the conditions or properties: 00, 01, 10, and 11. Notation 00 shows that the condition or property is not fulfilled by both elements. Similarly, the notation 01 indicates that the condition or property of the first element is not fulfilled and that of the second element is fulfilled.

A set of N elements can have up to 2 ** N possible combinations. The number of possible combinations decreases to 4 * N (N – 1)/2 = 2* N (N–1) if we restrict ourselves to examining not all combinations, but merely examining the combinations of element pairs. Expression 2 * N (N – 1) shows that, in practice, real-size sets can be processed in this case. However what impact this has on the accuracy of the examination must be assessed. This simplification only makes sense when it is impossible to treat all possible combinations.

Table 3 shows combinations pairs of outputs, which are found from Table 1. Symmetrical pairs like (p2, p1) and pairs with yourself like (p1, p1) are not listed. We find all four possible embodiment properties (00, 01, 10, 11) in columns p1 and p2 of Table 1. Columns p1 and p3 do not have the combination (11) where both properties have been satisfied. Unavailable combinations are shown in bold (red). Combination (11) indicates an inpossible situation. The fourth column of Table 2 shows that in this case we can detect only two faults.

Table 3. *Combinations of pairs of fulfilled output properties.*

	p1	p2	p3
p1		00 01 10 11	00 01 10 11
p2			00 01 10 11
p3			

Similarly, we can consider pairs of inputs and outputs. The results are shown in Table 4, and the detection of stuck-at faults is shown in the last column of Table 2. We see that in this case, four faults remain unidentified. In any case, the question is always whether input/output pairs detect all defects, which detect the output/output pairs. Otherwise, it is appropriate to use both output/input pairs and output/output pairs.

Table 4. *Combinations of pairs of fulfilled input/output properties.*

	p1	p2	p3
c1	00 10 11 01	00 01 10 11	00 01 10 11
c2	00 10 11 01	00 10 11 01	00 01 10 11
c3	00 10 11 01	00 10 11 01	00 01 10 11
c4	00 01 10 11	00 10 11 01	00 01 10 11

The case study showed that the unavailable combinations of properties of pairs between the outputs and between the inputs and outputs allow some possible failures of the system to be detected.

4. Input Sequences

So far, we determined, in a simplified fashion, that the output properties depend only on the input conditions. In reality, the examined object can have internal states and the output reactions are dependent on the previous state and on previously filled input stimuli. It is therefore necessary to examine the sequences of input stimuli and output reactions. The sequence is of fixed length and is composed of a clock cycle t = 1, 2, 3,..., T. The initialization stimulus is at the beginning of the sequence. The reactions of different stimuli can be examined separately and can vary greatly in different clock cycles of the sequence. This makes it possible to obtain more combinations of properties that are unavailable and thus to increase the chances that disorders can be identified.

The input sequence can be displayed as a matrix $S = \| s_{t,j} \|_{T, n+m}$, where $s_{t,j} = 1$ when $j <= n$ corresponds to the fulfilled input conditions at clock cycle t and $s_{t,j} = 1$ when $n < j <= n + m$ corresponds to the fulfilled output properties at clock cycle t. Otherwise $s_{t,j} = 0$. An example matrix is illustrated in Figure 5.

T	c1	c2	c3	c4	p1	P2	p3
0	0	0	0	0	0	1	0
1	0	1	0	1	1	0	1
2	1	0	1	0	0	1	1
3	0	0	1	1	0	0	1

Figure 5. *Matrix S example.*

The pairs of output sequence properties can be represented by a matrix $Z = \|z_{k,d}\|_{2*m*T,2*m}$, where $z_{k,d} = 1$, $k = 2 * j + s_{t,j+n} + t * 2 * m$, and $d = 2 * i + s_{t,i+n,}$ where t = 0, 1, 2,... T, j = 0,..., m – 1, i = j + 1,..., m. This description is formally correct but difficult to understand. The calculation procedure is more clearly demonstrated in Figure 6. Procedure SPP of making matrix Z demonstrates a calculation algorithm for the values of output sequence property pairs.

```
for t = 0 (1) T
    for j = 0 (1)  m -1
        for i = j + 1 (1) m
            k =  2 * j + s [t ],[j + n] + t  * 2 * m;
            d = 2 * i + s[t ],[i + n];
            z[k],[d] = 1;
        end for
    endfor
endfor
```

Figure 6. *Procedure SPP of making matrix Z.*

The external loop of the variable t examines all rows of matrix S and each cycle forms a part of the value of the matrix Z. The property pairs of line t are examined during internal cycles. All possible j and i pairs where i > j are analysed. The

indices k and d of matrix Z are calculated for each pair. There are four possible property pairs combinations of the matrix S: 00, 10, 01,1. Depending on the property pairs, the indexes k and d designate one of the adjacent cells (2 * j + s [t] [j + n], 2 * i + s [t], [i + n]), where one is recorded. The second index of matrix S has increased in size by n, since only outputs are examined. In this way, each combination of a property pair has a separate cell of the matrix Z. The k index is shifted by the amount 2 * t * m so that the values of the various rows are written into different matrix areas.

The matrix Z formed in accordance with the matrix S is shown in Figure 7. The matrix has four sections corresponding to the line t of matrix S. Each property has two rows and two columns in all parts. One row or column is intended for the case when the property is not fulfilled (zero value), and the next row or column is for the case when the property is fulfilled (one value). Let us say that we have a pair of properties (p1 = 0, p2 = 1) of the matrix S and line t = 0. Four cells are at the intersection of rows and columns of these properties in the matrix Z. Accordingly, we assign one of them a value of one. The meaning is indicated for all possible pairs of properties. All values are initially zero. Cells in which the values are not preceded by procedure are marked with an asterisk. The first two lines and the first three columns of the matrix Z are shown for demonstration purposes.

t			p1		p2		p3	
			0	1	0	1	0	1
0	p1	0	*	*	0	1	1	0
		1	*	*	0	0	0	0
	p2	0	*	*	*	*	0	0
		1	*	*	*	*	1	0
	p3	0	*	*	*	*	*	*
		1	*	*	*	*	*	*
1	p1	0	*	*	0	0	0	0
		1	*	*	1	0	0	1
	p2	0	*	*	*	*	0	1
		1	*	*	*	*	0	0
	p3	0	*	*	*	*	*	*
		1	*	*	*	*	*	*
2	p1	0	*	*	0	0	0	1
		1	*	*	0	0	0	0
	p2	0	*	*	*	*	0	0
		1	*	*	*	*	0	1
	p3	0	*	*	*	*	*	*
		1	*	*	*	*	*	*
3	p1	0	*	*	1	0	0	1
		1	*	*	0	0	0	0
	p2	0	*	*	*	*	0	1
		1	*	*	*	*	0	0
	p3	0	*	*	*	*	*	*
		1	*	*	*	*	*	*

Figure 7. *Matrix Z example.*

Different input sequences occupy different cells of matrix Z. A sample matrix can occupy a maximum of 12 × 4 = 48 cells. According to the rules of formation, cells just above the diagonal can be occupied.

The ones in the sample matrix Z are indicated in accordance with the input sequence of the matrix S. The other input sequence can add new ones in the cells of the matrix Z. The matrix Z is filled with the maximum number of ones after consideration of all possible input sequences. The remaining zeros indicate impossible combinations of properties.

The zeros of matrix Z only show likely impossible combinations of properties if not all possible input sequences are examined. The probability of impossible combinations of properties is higher when more input sequences are examined.

5. The Assessment Criteria of the Input Sequence

Input stimuli determine the output response. The maximal number of pairs of output properties is equal to 4 * m * (m – 1)/2. Disorders of runtime can be identified only if not all pairs of properties are available at the output. Only impossible pairs of properties may identify the failure. The probability of identification of failure is proportional to the number of pairs of properties that are impossible. It is therefore appropriate to seek to evaluate more such situations.

```
for t = 0 (1) T - 1
    for j = 0 (1)  m -1
        for i = j + 1 (1) m
            sk = 0;
                if (s[t],[ j + n ] == 1): sk = sk + 2;
                if (s[t + 1],[j + n ] == 1): sk = sk + 1;
             sd = 0
                if (s[t],[i + n] == 1): sd = sd + 2;
                if (s[t + 1],[i + n] == 1): sd = sd + 1;
            k = j * 4  + sk + t  * 4 * m ;
            d = i * 4  + sd ;
            z[k],[d] = 1;
        end for
    endfor
endfor
```

Figure 8. *Procedure APP of making matrix Z.*

The number of situations is enhanced if the property pairs of adjacent stimuli of the output sequences are considered. Properties of the adjacent stimuli may have the variations 00, 01, 10, and 11, and property pairs will already have 4 × 4 = 16 variations. The calculation procedure is shown in Figure 8. Compared with the SPP procedure for the external cycle, T – 1 lines are analysed, because only adjacent rows are considered. The indices k and d are increased by the amounts sk and sd, whose values are calculated using the matrix S outputs so that every situation can be described into different

cells. The values of sk and sd are equal to zero if the matrix S corresponding cells are also equal to 00. If the matrix S corresponding cells are equal to 01, the values of sk and sd are equal to 1. If the corresponding cells are considered equal to 10, the values of sk and sd are equal to 2. The values of sk and sd are equal to 3 if the corresponding cells are considered equal to 11. The maximal number of cells to be filled in is equal to ((m * (m – 1))/2) * 16 * (T – 1)

The fragment of matrix Z at t = 0 is shown in Figure. 9. The "Properties" tag is shortened to "Prop." in the table. Property p1 is equal to zero when t = 0 and equal to one when t = 1 in the analysed matrix S. This corresponds to the case of t = 0 and j = 0 in the APP calculation procedure that calculates the value of sk equal to 1; accordingly, the property p2 is equal to one when t = 0 and equal to zero when t = 1 in the analysed matrix S. This corresponds to the case of t = 0 and j = 0, i = 1 of the APP calculation procedure that calculates the value of sd equal to 2. Thus, the appearance of the index k = 1 and the index d = 6 of the matrix Z records ones in the appropriate cell. Indices k = 1 and d = 9 are determined by a pair of properties p1 and p3 and indices of k = 6 and d = 9 are determined by a pair of properties p2 and p3. In this way, three ones are recorded in the matrix Z.

t	Prop.	Prop.	p1				p2				p3			
		sk\sd	0	1	2	3	0	1	2	3	0	1	2	3
0	p1	0	*	*	*	*	0	0	0	0	0	0	0	0
		1	*	*	*	*	0	0	1	0	0	1	0	0
		2	*	*	*	*	0	0	0	0	0	0	0	0
		3	*	*	*	*	0	0	0	0	0	0	0	0
	p2	0	*	*	*	*	*	*	*	*	0	0	0	0
		1	*	*	*	*	*	*	*	*	0	0	0	0
		2	*	*	*	*	*	*	*	*	0	1	0	0
		3	*	*	*	*	*	*	*	*	*	*	*	*
	p3	0	*	*	*	*	*	*	*	*	*	*	*	*
		1	*	*	*	*	*	*	*	*	*	*	*	*
		2	*	*	*	*	*	*	*	*	*	*	*	*
		3	*	*	*	*	*	*	*	*	*	*	*	*

Figure 9. *Fragment of matrix Z when t = 0.*

```
for t = 0 (1) T - 1
    for i = 0 (1)  n
        for j = 0 (1) m
            sk = 0;
                if (s[t],[i] == 1): sk = sk + 2;
                if (s[t + 1],[i ] == 1): sk = sk + 1;
             sd = 0
                if (s[t][j + n] == 1): sd = sd + 2;
                if (s[t + 1][j + n] == 1): sd = sd + 1;
            k =  4 * i + sk + t  * 4 * n ;
            d = 4 * j + sd ;
            z[k],[d] = 1;
        end for
    endfor
endfor
```

Figure 10. *The procedure AIPP of making matrix Z of adjacent input condition and output property pairs.*

Examination of pairs of input conditions and output properties can be another way to evaluate the input sequence. The modified filling procedure for matrix Z is shown in Figure 10. In this case, pairs of all input conditions and all output properties are considered. The maximum number of cells to be filled is equal to n * m * 16 * (T – 1).

The fragment of matrix Z at t = 0 is shown in Figure. 11. In this case, the whole matrix may already be filled. The 16 cells are desiccated to a pair of conditions and properties. For instances where t = 0, just one box is filled.

T	Cond.	Prop.	p1				p2				p3			
		sk\sd	0	1	2	3	0	1	2	3	0	1	2	3
0	c1	0	0	1	0	0	0	0	1	0	0	1	0	0
		1	0	0	0	0	0	0	0	0	0	0	0	0
		2	0	0	0	0	0	0	0	0	0	0	0	0
		3	0	0	0	0	0	0	0	0	0	0	0	0
	c2	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	0	1	0	0	0	0	1	0	0	1	0	0
		2	0	0	0	0	0	0	0	0	0	0	0	0
		3	0	0	0	0	0	0	0	0	0	0	0	0
	c3	0	0	1	0	0	0	0	0	0	0	0	0	0
		1	0	0	0	0	0	0	1	0	0	1	0	0
		2	0	0	0	0	0	0	0	0	0	0	0	0
		3	0	0	0	0	0	0	0	0	0	0	0	0
	c4	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	0	1	0	0	0	0	1	0	0	1	0	0
		2	0	0	0	0	0	0	0	0	0	0	0	0
		3	0	0	0	0	0	0	0	0	0	0	0	0

Figure 11. *Fragment of matrix Z filled by procedure AIPP.*

The APP and AIPP procedures form the matrix Z of possible pairs. A system failure can be detected if it causes an impossible pair. Sometimes, the failure can lead to a change of one possible combination of properties to another possible combination of properties. Such a situation will not be noticeable. Every possible option pair may recur several times during the execution of the input sequence. It is

therefore possible to find the maximum number of repetitions. A system failure is indicated if the maximum number of repetitions is exceeded. This allows us to detect failures that are not triggering impossible pairs. The formation of such a matrix is shown in Figure 12.

```
for t = 0 (1) T - 1
    for j = 0 (1)  m -1
        for i = j + 1 (1) m
            sk = 0;
                if (s[t],[i + n] == 1): sk = sk + 2;
                if (s[t + 1],[i + n] == 1): sk = sk + 1;
            sd = 0
                if (s[t],[j + n] == 1): sd = sd + 2;
                if (s[t + 1],[j + n] == 1): sd = sd + 1
            k =  4 * j + sk  ;
            d = 4 * i + sd ;
            z[k],[d] =  z[k],[d]  + 1;
        end for
    endfor
endfor
for i = 0 (1) 4 * m
    for j = 0 (1) 4 * m
        if z[i],[j] > m[i],[j]):
            m[i],[j] = z[i],[j];
    end for
end for
```

Figure 12. *The procedure AMPP of making matrix Z for the maximum number of repetitions of output property pairs.*

Like the APP procedure, the AMPP procedure presented forms a matrix Z with the difference that the property points are summed for all the stimuli of the input sequence. The maximum values of matrix Z are written into the matrix M.

How can all the possible property pairs be found? Matrix Z is sequentially supplemented by the new input sequences during system execution. Filling the matrix Z during a long run can find all possible pairs of properties. However, the experiment cannot guarantee that all possible pairs of properties will be found. When the input sequences are analysed for a longer time, the likelihood that possible property pairs will be found increases. In practice, it may be considered that all of the property pairs have already been identified if the matrix Z is saturated with additions. Zero values of matrix Z indicate impossible property pairs and they can be used to identify failures.

Problem detection at runtime depends on the method of capturing problems and upon the magnitude of the input sequences analysed. Functional failures might be seen if this influences the results. It is important to answer the question of how many problems a capture method can detect. How many impossible pairs of matrix Z will allow the detection of failures? To answer these questions, we carried out an experiment.

6. Experiments

Circuit B14 is a VIPER processor and all combinations of signal values of inputs and outputs are available. Impossible combinations of signal values can occur only when the clock cycles of the input sequence are evaluated. Internal states and order of stimuli in the input sequence determine the possible output values. An input sequence consists of 31 input stimuli. Circuit B14 has 34 inputs, 54 outputs, and 507 lines of VHDL code. A synthesized circuit has 3461 gates, 247 triggers, and 27,592 stuck-at faults.

The maximal number of output values of pairs is equal to ((54 * 53)/2) * 16 = 22896. The maximal number of pairs of input/output values is equal to (34 * 54) * 16 = 29376. Random input sequences were generated for the purpose of having the most complete matrix Z according to the procedures APP and AIPP until the filling in procedure was exhausted. The maximum filling quantities for each clock cycle are shown in Table 5. Matrix parts corresponding to the first clock cycle are at the least filled, and at the same time, they have the greatest number of pairs of values that should be impossible. Changing the internal state, every second clock cycle also has an impact.

The maximum number of possible pairs of output values for all pairs of clock cycles is equal to 686,880 and comprehensive random generation has found 390,038 pairs of output, representing 56.8%. In this case, 296,842 pairs of output values should be impossible and should enable the capture of the system failures. Similarly, the maximum number of possible input/output pairs of values for all clock cycles is equal to 881,280 and comprehensive random generation found 638,439 pairs of input/output, representing 72.4%. In this case, 242841 input/output value pairs should be impossible and should make it possible to capture the system failures.

Table 5. *Maximum number of cells filled in during each clock cycle.*

Clock cycles	Maximum number of output pairs (APP)	Maximum number of input/output pairs (AIPP)	Sum of the maximum values (MPP)
1	1431	3564	6216
2	1431	7074	5349
3	2846	10073	9118
4	6506	15344	8992
5	9517	19631	8762
6	9441	19379	8543
7	16845	27651	8362
8	10687	19378	8054
9	18380	27695	7848

Clock cycles	Maximum number of output pairs (APP)	Maximum number of input/output pairs (AIPP)	Sum of the maximum values (MPP)
10	10687	19382	7608
11	19407	27904	7422
12	10687	19386	7150
13	19504	27560	6969
14	10687	19240	6685
15	19565	27560	6502
16	10687	19240	6107
17	19565	27560	5387
18	10687	19240	4747
19	19565	27560	3917
20	10687	19240	6049
21	19537	27560	5448
22	10687	19454	5211
23	19560	27842	4343
24	10687	19522	4216
25	19564	27560	4074
26	10687	19240	3983
27	19565	27560	3858
28	10687	19240	3707
29	19565	27560	3576
30	10687	19240	3421

A verification sequence of 100006 input stimuli that had been grouped into sequences who have 31 stimuli (a total of 3226 sequences) was randomly generated for the experiments. Several dozen mutations (distortions) of the VHDL description, the introduction of additional features, and the insertion of stuck-at faults were analysed. Distortions imitate the system failure. The results are shown in Table 6.

The second line shows distortions that alter the conditions of VHDL operators. The third line shows distortions that reverse the branches of CASE operators. The results of the insertion of additional circuit functions are presented in the fourth row. The last two lines are intended for insertion of stuck-at faults.

The average number of output value changes after the distortion is shown in the second column. The percentage of detected distortions is indicated in parentheses.

The system disorders are very different. The number of disorders can be enormous. Examination of all disorders is impossible. The number and types of disorders found during the experiment are far from covering all possible system disorders. Therefore, observations made in accordance with the results of the experiment cannot be categorical and generalizations should only be made with some caution.

The numbers listed in Table 6 show how many impossible properties detect distortion (disorder) on average. The percentages in brackets show how many disorders have been detected from all of the examined. The maximum percentage is indicated in the second column because were considered just such distortions, which change the output values. First of all, it should be noted that a large proportion of the disorders are not detected when the impossible combinations of properties are evaluated, although such disorders cause changes in the output values (second column). Therefore, each specific problem always has to have an assessment of the extent to which impossible properties can detect disorders.

The distortions that related to design errors are the most difficult to detected. This is consistent with mutations of the software, when the conditions of the source are changed (the second and third rows of the Table 6). The addition of the extra features is detected best, indicating the possibilities of detection of unauthorized changes to the code (fourth row). Hardware problems (stuck-at faults) are also detected relatively well (the last two lines).

Table 6. *The results of an experimental study.*

Failures	The average number of conflicting output values	The average number of impossible pairs of adjacent output values (APP)
Amendment of comparison conditions	7561 (100%)	0 (0%)
Swapping of branches (CASE)	3606 (100%0	23 (3%)
Insertion of additional functions	12560 (100%)	7442 (83%)
Stuck-at faults ≡ 0	10177 (100%)	2949 (71%)
Stuck-at faults ≡ 1	22121 (100%)	511 (54%)

Table 6. *Continued.*

Failures	The average number of impossible pairs of adjacent input/output values (AIPP)	The average number of adjacent output values in excess of the maximum value (MPP)
Amendment of comparison conditions	0 (0%)	49 (6%)
Swapping of branches (CASE)	0 (0%)	14 (2%)
Insertion of additional functions	8938 (70%)	788 (91%)
Stuck-at faults ≡ 0	141 (9%)	346 (66%)
Stuck-at faults ≡ 1	780 (44%)	132 (39%)

The results of the experiment (Table 6) are scattered, which prevents the formation of unambiguous assertions. It does not indicate that one of the techniques dealing with the detection of disorders is superior to the others because there were such disorders detected by only one technique. In general, input/output pairs (the fourth column) detect fewer disorders, but it must be remembered that the present case has not restricted the combinations of input conditions. With such restrictions, the situation might change. The maximum value method (last column) harder saturated with the formation of matrix Z but captures more different disorders.

7. Conclusions

Runtime verification is based on inspection of the output results. The system failures alter the values of results. Comparison of the results obtained with the expected results allows the detection of failures. The expected results may be determined on the basis of the behavioural model. Otherwise, runtime verification is based on properties. Properties that cannot be fulfilled in any one of the input sequences are referred to as impossible properties. The capture of impossible properties during service shows possible system failures. Only some failures may be detected during the inspections of impossible properties. Examination of combinations of properties increases the capability to detect system failures during service.

During service or during the generation of random input sequences, all possible combinations of properties are recorded until the process becomes saturated. The number of possible combinations of properties can be enormous, and their treatment requires the use of simplified methods. Combinations of properties that do not occur can be considered as impossible combinations of properties and can be used for runtime verification. Determination of impossible combinations of properties during service allows us to highlight such impossible combinations of properties that are clearly not visible from the specification. The experimental results cannot guarantee that combination of properties that should be impossible are indeed impossible. This can be interpreted as a warning. Without a system failure, an impossible combination of properties can be changed to the possible combination.

Three methods of marking the possible combinations of properties are proposed. One method examines the combinations of the output values of adjacent stimuli of the sequence. Another method examines the combinations of input and output values. The third method assesses the maximum number of combinations of output values. The capabilities of the proposed marking methods are assessed by examining circuit B14 of the ITC benchmark suite. VHDL mutations and stuck-at faults have been considered as potential system failures. Many failures that change the output results are not monitored by examining impossible combinations of properties. This shows that the runtime verification in respect of the properties must have information on how many disruptions in the system can be detected.

The scattering of experimental results shows that none of the three marking methods has any obvious advantage. It is appropriate to use all three of the proposed marking techniques together during runtime verification.

This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

References

[1] Colin, S., & Mariani, L. (2005). “18 Run-Time Verification”. In *Model-Based Testing of Reactive Systems* (pp. 525-555). Springer, Berlin, Heidelberg. doi.org/10.1007/11498490_24.

[2] Leucker, M., & Schallhart, C. (2009). “A brief account of runtime verification”. *The Journal of Logic and Algebraic Programming*, *78* (5), 293-303. doi.org/10.1016/j.jlap.2008.08.004.

[3] Colombo, C., Pace, G. J., & Schneider, G. (2008). “Dynamic event-based runtime monitoring of real-time and contextual properties”. In *International Workshop on Formal Methods for Industrial Critical Systems* (pp. 135-149). Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-642-03240-0_13.

[4] Aktug, I., & Naliuka, K. (2008). “ConSpec—a formal language for policy specification”. *Science of Computer Programming*, *74*=(1-2), 2-12. doi.org/10.1016/j.scico.2008.09.004.

[5] Pnueli, A. (1977). ‘The temporal logic of programs”. In *Foundations of Computer Science, 1977., 18th Annual Symposium on Foundations of Computer Science* (pp. 46-57). IEEE. doi.org/10.1109/sfcs.1977.32.

[6] Penczek, W., & Lomuscio, A. (2003). “Verifying epistemic properties of multi-agent systems via bounded model checking”. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (pp. 209-216). doi.org/10.1145/860575.860609.

[7] Bodden, E., Hendren, L., Lam, P., Lhoták, O., & Naeem, N. A. (2007). “Collaborative runtime verification with tracematches”. In *International Workshop on Runtime Verification* (pp. 22-37). Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-540-77395-5_3.

[8] Reger, G., Hallé, S., & Falcone, Y. (2016). “Third International Competition on Runtime Verification“. Lecture Notes in Computer Science, 21–37. doi.org/10.1007/978-3-319-46982-9_3.

[9] Bartocci, E., Bortolussi, L., Nenzi, L., & Sanguinetti, G. (2015). “System design of stochastic models using robustness of temporal properties”. *Theoretical Computer Science*, *587*, 3-25. doi.org/10.1016/j.tcs.2015.02.046.

[10] Colombo, C., Pace, G. J., Camilleri, L., Dimech, C., Farrugia, R., Grech, J. P.,... & Adami, K. Z. (2016). “Runtime verification for stream processing applications”. In *International Symposium on Leveraging Applications of Formal Methods* (pp. 400-406). Springer, Cham. doi.org/10.1007/978-3-319-47169-3_32.