UDC 510.64:004.052

S.B. OSTROUMOV[1], L.V. LAIBINIS[2], E.A. TROUBITSYNA[2]

[1]National aerospace university named after N.E. Zhukovsky "KhAI", Ukraine
[2]Åbo Academy University, Finland

EVENT-B PATTERNS FOR DEVELOPING FPGA-BASED HARDWARE

The paper describes the first step of methodology for designing dependable hardware which is based on field programmable gate array technology. This step means the development of patterns using Event-B language useful thanks to mathematical proofs of a model. The report shows and describes the patterns developed according to synchronism technique because a great number of systems are synchronous. The patterns describe different component interconnections which are often used in hardware design. These patterns are the necessary condition to convert correctly developed model into hardware description language (e.g. VHDL).

Key words: *Formal development, Event-B, invariant, patterns, hardware, design, FPGA, Rodin*

Introduction

Nowadays many companies increasingly use FPGA chips because of their flexibility and simplicity as well as impressive computational power. Furthermore, there are various kinds of chips with different parameters of radiation, stress, power consumption, size etc., which allow to use them in diverse critical areas such as aviation, space and nuclear power [1]. Moreover, modern FPGA chips enable to use hard logic along with the soft-processor core with flexible software. The core has RISC processor architecture and is used when a great performance or calculation is required [2]. Therefore, it is reasonable to use such a modern technology as FPGA for developing critical control systems of high complexity.

The main problem with developing critical systems using FPGA and hardware description languages (such as VHDL) is hazards connected with design faults. Classical approaches to safety and fault tolerance help to defend a system from physical faults using different forms of redundancy (doubling, tripling etc.) including internal redundancy (e.g. doubling inside a chip) and external one (e.g. two or more chips are used), and their combinations [3]. However, these approaches are unable to guarantee protection against design errors.

That is why a new technique is required to struggle with such a problem. The solution can be found in using formalism to create mathematically proved specification [4-6] which, in its turn, will be converted into a hardware description language. This approach uses invariants – constant properties which are to be true at all stages of system development and exploitation.

There have been several attempts to implement such an approach, but nobody has managed to develop this approach for using it without restrictions. For instance, Boulanger describes the BHDL tool [7] which is supposed to be helpful for developing hardware based on VHDL. However, this tool uses VHDL code as a source for model checking approach and cannot be used for initiative development. Another example [8] describes an attempt to develop a technique for translating classical B model into VHDL code; nevertheless, this approach has no practical results.

Therefore, the purpose of the article is to show the first stage of the approach for developing FPGA based hardware systems. The main idea of the stage is the creation of patterns using Event-B method in accordance with synchronization.

1. Event-B specification and hardware description

Event-B language is the next stage of the B-method development. It was created by Jean-Raymond Abrial [4]. The main idea of this language is the use of Abstract Machine Notation (AMN), which describes the behavior of a system. AMN is a set of events in a system performed. The major problem is that a developer can define the events in different ways. That is why the first step of design hardware using Event-B is the development of patterns. They characterize a system as a component at the abstract level and interconnections between hardware components which compose the system during refinement.

A great number of control systems are synchronous because we have to know when and what is executed at the moment; thus, we have created patterns according to synchronization. We have developed an abstract model in accordance with the refinement approach to show the use of different patterns at refinement steps (Fig. 1). All the patterns can be used as many times as needed.

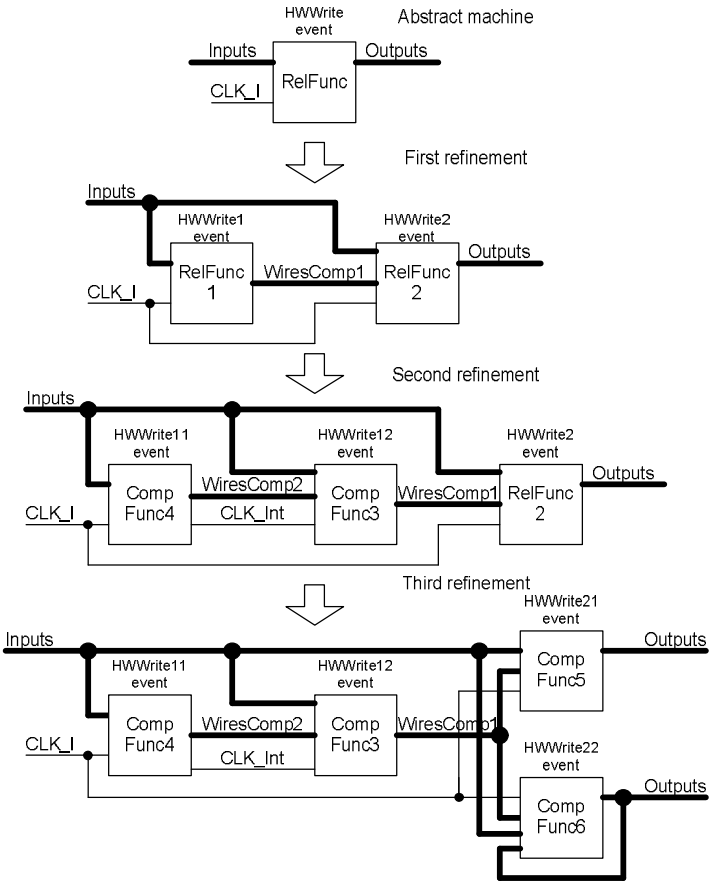

Fig. 1. The example of HW components refinement

First of all, at the abstract level we define inputs and outputs of a system and a special input called global clock signal (CLK_I). This signal controls the behavior of the system according to ticks going from clock generator, which changes this signal from logical zero to logical one (rising edge). Then, during refinement steps we apply patterns taking into consideration that we pack all the operations into one global tick. Also, we define the dependency outputs on inputs, but using common rules instead of precise ones because we will add all the necessary functions and concrete principles of outputs assignment during the refinement steps.

To define all the signals, we have to define a set of their states. The set is called STD_LOGIC according to FPGA terminology and contains the following elements: U – uninitialized, X – forcing Unknown, LOGIC_0 – forcing 0, LOGIC_1 – forcing 1, Z – high impedance, W – weak unknown, L – weak 0, H – weak 1, DC – don't care [9]. All the signals have X state from the very beginning which means that they are initialized by X state in a machine. As the machine starts working, it changes the states of signals because the clock signal tells the machine when to read and change the value of inputs.

Secondly, we have to declare a set of global clock to perform a sequence of events. This set is called CLKFLAGSET and has two states (Fig. 2). It helps to interconnect events describing synchronous scheme, which has the following progression. First, CLK_I changes from LOGIC_0 to LOGIC_1 (rising edge), then a writing operation is performed and then CLK_I changes from LOGIC_1 to LOGIC_0 (falling edge). When CLK_I signal has rising edge, all the inputs are read (CLKRisEdge event); so, CLK_Flag variable has the state which is called Read. After that writing operation produces outputs according to inputs read and the flag changes to Written state (HWWrite event) and then CLK_I has falling edge (CLKFalEdge) (Fig. 3).

```
context CComp1HW
constants Read Written U X LOGIC_0
LOGIC_1 Z W L H DC

sets CLKFLAGSET STD_LOGIC

axioms
   @axm0
   partition(STD_LOGIC,{U},{X},{LOGIC_0},
   {LOGIC_1},{Z},{W},{L},{H},{DC})

   @axm2
   partition(CLKFLAGSET,{Read},{Written})
end
```

Fig. 2. The definitions of the sets used in abstract machine

```
machine Comp1HW sees CComp1HW

variables CLK_I Inputs Outputs CLK_Flag

invariants
  @inv1 CLK_I ∈ STD_LOGIC
  @inv2 CLK_Flag ∈ CLKFLAGSET
  @inv3 Inputs ∈ STD_LOGIC
  @inv4 Outputs ∈ STD_LOGIC

events
  event INITIALISATION
    then
      @act1 CLK_I ≔ LOGIC_0
      @act2 CLK_Flag ≔ Written
      @act3 Inputs ≔ X
      @act4 Outputs ≔ X
  end

  event CLKRisEdge
    where
      @grd1 CLK_I = LOGIC_0
    then
      @act1 CLK_I ≔ LOGIC_1
      @act2 CLK_Flag ≔ Read
      @act3 Inputs :| Inputs' ∈ STD_LOGIC ∧
Inputs' ≠ DC ∧ Inputs' ≠ U ∧ Inputs' ≠ X
∧ Inputs' ≠ W ∧ Inputs' ≠ Z
  end

  event HWWrite
    where
      @grd1 CLK_I = LOGIC_1
      @grd2 CLK_Flag = Read
      @grd3 (Inputs = LOGIC_1) ∨ (Inputs =
LOGIC_0) ∨ (Inputs = H) ∨ (Inputs = L) //
Here is I define dependency outputs on in-
puts */
    then
      @act1 CLK_Flag :∈ CLKFLAGSET
      @act2 Outputs :| Outputs' ∈ STD_LOGIC
// This is just a simplest operation, but can
be a function
  end

  event CLKFalEdge
    where
      @grd1 CLK_I = LOGIC_1
      @grd2 CLK_Flag = Written
    then
      @act1 CLK_I ≔ LOGIC_0
  end
end
```

Fig. 3. The events describing abstract behavior

In this example we restrict states of input signals to a few values generally used in programmable logic. They are chosen by a non-deterministic assignment, limited with LOGIC_0, LOGIC_1, H and L (CLKRisEdge event). The machine reads the current state of input signals and then process them in the further events.

The dependency between inputs and outputs has to be written in the following way. In the guards section we write inputs which influence outputs and in the actions section we write all the outputs which have to be changed accordingly to the state of inputs written in guards section (grd3, act2 in HWWrite event). Thus, we can have not only one Write event, but also many events. They will describe all possible states of declared outputs in accordance with the states of inputs because, basically, a system has many inputs and outputs. Besides, we can also define invariants describing possible combinations of inputs and outputs, both separately and together, because sometimes inputs as well as outputs of a system have forbidden combinations and we have to check whether events have them or not.

Finally, we can also write invariants showing the main limitations and properties of a system to prove its correctness correspondently to refinement approach.

After writing down all the signals and invariants of a system and receiving an abstract model with mathematical proofs of its correctness, we begin refining the model correspondingly to one of the following patterns depending on a purpose we want to achieve.

2. Event-B patterns

2.1. A sequence of components with global clock

Quite often components have to be interconnected into a sequence where outputs of the first component influence inputs of the second one and both of them have a global clock signal (Fig. 1, the first refinement). Although these two components are clocked by one rising edge of clock signal at the moment, there is a difference between them in time. Firstly, the second component produces outputs correspondently to the state of the first component outputs. Then, the first component generates its outputs using inputs altered during CLKRisEdge event execution.

To describe such a sequence we defined a set called REFCOMPSSTEP2, which includes two states Comp1 and Comp2 (Fig. 4).

```
context CComps2 extends CComp1HW

constants Comp1 Comp2
sets REFCOMPSSTEP2
axioms
  @axm1 partition(REFCOMPSSTEP2,
            {Comp1},{Comp2})
end
```

Fig. 4. The context of the sequence pattern

During the execution of events a variable of the set type (RefCompsStep2) contains a state of a component

being executed at the moment (the current component) (Fig. 5). After being performed, the first component modifies the value of the variable to make it possible to execute the next component (@act3 in HWWrite2 event).

```
machine Comps2_global_CLK refines Comp1HW sees CComps2

variables CLK_I CLK_Flag Inputs Outputs WiresComp1 RefCompsStep2

invariants
  @inv1 WiresComp1 ∈ STD_LOGIC
  @inv2 RefCompsStep2 ∈ REFCOMPSSTEP2

events
  event INITIALISATION extends INITIALISATION
    then
      @act5 WiresComp1 ≔ X
      @act6 RefCompsStep2 ≔ Comp2
  end

  event HWWrite2 refines HWWrite
    where
      @grd1 CLK_I = LOGIC_1
      @grd2 CLK_Flag = Read
      @grd3 (Inputs = LOGIC_1) ∨ (Inputs = LOGIC_0) ∨ (Inputs = H) ∨ (Inputs = L)
      @grd4 (WiresComp1 = X) ∨ (WiresComp1 = Z) ∨ (WiresComp1 = LOGIC_1) ∨ (WiresComp1 = LOGIC_0) ∨ (WiresComp1 = H) ∨ (WiresComp1 = L)
      @grd5 RefCompsStep2 = Comp2
    then
      @act2 Outputs :| Outputs' ∈ STD_LOGIC ∧ Outputs' ≠ DC ∧ Outputs' ≠ U ∧ Outputs' ≠ X ∧ Outputs' ≠ W
      @act3 RefCompsStep2 ≔ Comp1
  end

  event HWWrite1 refines HWWrite
    where
      @grd1 CLK_I = LOGIC_1
      @grd2 CLK_Flag = Read
      @grd3 (Inputs = LOGIC_1) ∨ (Inputs = LOGIC_0) ∨ (Inputs = H) ∨ (Inputs = L)
      @grd4 RefCompsStep2 = Comp1
    then
      @act1 CLK_Flag :| CLK_Flag' ∈ CLKFLAGSET
      @act3 RefCompsStep2 :| RefCompsStep2' ∈ REFCOMPSSTEP2 ∧ RefCompsStep2' = Comp2
      @act4 WiresComp1 :| WiresComp1' ∈ STD_LOGIC
  end
end
```

Fig. 5. The components sequence with global clock

The same operation is done by the second component in its turn (@act3 in HWWrite1 event). To allow an execution accordingly to the described sequence scheme, we have to add the check of the variable in guards section of each event (@grd5 and @grd4 correspondently).

Therefore, to use a sequence of two components which have global clock we have to define a set of states which describe components. Then we use a variable that helps to specify components execution. In addition, we have to remove the action which modifies clock flag (@act1, Fig. 3) from the actions section of the second component because if the second component finished working, the first one will work before falling edge. Besides, we have to remove the action which produces outputs (@act2, Fig. 3) from the action list of the first component and add the action which assign values of internal wires (@act4, Fig. 5) because the first component must not produce outputs but internal signals.

2.2. A sequence of components with internal clock

The next combination of components is a sequence where the first component influences the second one not only by internal signals, but also by internal clock signal (Fig. 1, the second refinement). The internal clock signal is produced by a function InternalCLKFunc, which generally takes some or all inputs and generates a state in compliance with STD_LOGIC set (Fig. 6).

```
context CComps3 extends CComps2
constants InternalCLKFunc
axioms
  @axm1 InternalCLKFunc ∈ INPUTS → STD_LOGIC
End
```

Fig. 6. The definition of internal clock function

This pattern contains two cases. The first one presupposes that the second component is not executed, so a falling edge comes after the first component has done its operations. Alternatively, the first component modifies internal clock signal from logical zero to logical one, which makes the second component perform; so, we have straight sequence and a falling edge comes after both components have been executed.

To implement this sequence scheme we have to use two variables (CLK_Int and CLK_IntFlag) (Fig. 7). The first variable depends on a rule which makes a predicate dependening on inputs; the last one shows which component has to work at the moment. If CLK_Int is modified from logical zero to logical one, CLK_IntFlag variable will allow the second component to work and then to perform a falling edge.

```
machine Comps3_internal_CLK refines Comps2_global_CLK sees CComps3

variables ... CLK_Int CLK_IntFlag WiresComp2

invariants
  @inv1 CLK_Int ∈ STD_LOGIC
  @inv2 CLK_IntFlag ∈ CLKFLAGSET
  @inv3 WiresComp2 ∈ STD_LOGIC

events
  event INITIALISATION extends INITIALISATION
    then
      @act7 CLK_Int ≔ LOGIC_0
      @act8 CLK_IntFlag ≔ Written
      @act9 WiresComp2 ≔ X
  end
  ...
  event HWWrite11 refines HWWrite1
    where
      ...
      @grd4 RefCompsStep2 = Comp1
      @grd5 CLK_IntFlag = Written
    then
      @act1 CLK_Flag :| (Inputs ≠ LOGIC_1 ⇒ CLK_Flag' ≠ Read) ∧(Inputs = LOGIC_1 ⇒ CLK_Flag' = CLK_Flag)
      @act3 RefCompsStep2 :| (Inputs ≠ LOGIC_1 ⇒ RefCompsStep2' = Comp2) ∧(Inputs = LOGIC_1 ⇒ RefCompsStep2' = RefCompsStep2)
      @act4 WiresComp2 :| WiresComp2' ∈ STD_LOGIC
      @act5 CLK_Int :| CLK_Int' ∈ STD_LOGIC ∧ CLK_Int' = Inputs
      @act6 CLK_IntFlag :| (Inputs = LOGIC_1 ⇒CLK_IntFlag' = Read) ∧ (Inputs ≠ LOGIC_1 ⇒CLK_IntFlag' = Written)
  end

  event HWWrite12 refines HWWrite1
    where
      ...
      @grd4 RefCompsStep2 = Comp1
      @grd5 CLK_Int = LOGIC_1
      @grd6 CLK_IntFlag = Read
      @grd7 WiresComp2 ∈ STD_LOGIC
    then
      @act1 CLK_Flag :| CLK_Flag' ≠ Read
      @act3 RefCompsStep2 ≔ Comp2
      @act4 WiresComp1 :| WiresComp1' ∈ STD_LOGIC ∧ WiresComp1' ≠ DC ∧ WiresComp1' ≠ U ∧ WiresComp1' ≠ X ∧ WiresComp1' ≠ W
      @act5 CLK_IntFlag ≔ Written
  end
  ...
end
```

Fig. 7. The components sequence with internal clock

Therefore, if we want to use a sequence of components with internal clock signal, we have to define a function which has the rule of internal clock creation. We also have to use two variables: one of them permits the second component to be executed, the other one shows the operation executed.

2.3. Concurrent components execution

Another possible combination is concurrent components execution (Fig. 1, the third refinement). This scheme works in the following way. We do not know which of the components works first. That is why we have an AbsComp state, which shows that neither of the components has been executed (Fig. 8). We also have two states showing which of the components has been performed (in our case CompStep41 and CompStep42 states).

```
context CComps4 extends CComps3
constants AbsComp CompStep41 CompStep42
sets CLKPARALLELSET
axioms
  @axm1 partition (CLKPARALLELSET, {AbsComp},{CompStep41},{CompStep42})
end
```

Fig. 8. The definition of a set for defining parallel components

What we do next is to check whether components have worked or not in guards section of each component (Fig. 9). Consequently, we have two events available at the same time. Depending on the state of CLK_ParFlag variable each event assigns different values to it. For instance, if the first component finished its function, the second one will assign the AbsComp state to permit a fulfillment at the next iteration cycle of system execution (Fig. 9, @act4 in HWWrite22 event). The first component will do the same thing if the second one finished operations (Fig. 9, @act4 in HWWrite21 event). But if neither of them has worked yet, the first to be executed will assign the value of its state to the CLK_ParFlag variable. It shows the other component that the former has been performed. It allows to exclude the executed component and enables the remaining one.

Thus, to use the concurrent scheme of components performance, we have to define a set of states that includes each component and an abstract one to permit events to be available at the same time. Besides, we have to add the check of a variable of defined set into guards section of each component and add actions which change the value of the variable correspondingly to the execution rule.

```
machine Comps4_parallel refines Comps3_internal_CLK sees CComps4

variables ... CLK_ParFlag

invariants
  @inv1 CLK_ParFlag ∈ CLKPARALLELSET

events
  event INITIALISATION extends INITIALISATION
    then
      @act10 CLK_ParFlag ≔ AbsComp
  end
  ...
  event HWWrite21 refines HWWrite2
    where
      ...
      @grd5 RefCompsStep2 = Comp2
      @grd6 CLK_ParFlag = AbsComp ∨ CLK_ParFlag ≠ CompStep41
    then
      @act2 Outputs :| Outputs' ∈ STD_LOGIC ∧ Outputs' ≠ DC ∧ Outputs' ≠ U ∧ Outputs' ≠ X ∧ Outputs' ≠ W
      @act3 RefCompsStep2 :| (CLK_ParFlag = CompStep42 ⇒ (RefCompsStep2' = Comp1)) ∧ (CLK_ParFlag = AbsComp ⇒ (RefCompsStep2' = RefCompsStep2))
      @act4 CLK_ParFlag :| ((CLK_ParFlag = AbsComp ⇒ CLK_ParFlag' = CompStep41) ∧ (CLK_ParFlag ≠ AbsComp ⇒ CLK_ParFlag' = AbsComp))
  end

  event HWWrite22 refines HWWrite2
    where
      ...
      @grd5 RefCompsStep2 = Comp2
      @grd6 CLK_ParFlag = AbsComp ∨ CLK_ParFlag ≠ CompStep42
    then
      @act2 Outputs :| Outputs' ∈ STD_LOGIC ∧ Outputs' ≠ DC ∧ Outputs' ≠ U ∧ Outputs' ≠ X ∧ Outputs' ≠ W
      @act3 RefCompsStep2 :| (CLK_ParFlag = CompStep41 ⇒ (RefCompsStep2' = Comp1)) ∧ (CLK_ParFlag = AbsComp ⇒ (RefCompsStep2' = RefCompsStep2))
      @act4 CLK_ParFlag :| ((CLK_ParFlag = AbsComp ⇒ CLK_ParFlag' = CompStep42) ∧ (CLK_ParFlag ≠ AbsComp ⇒ CLK_ParFlag' = AbsComp))
  end
  ...
end
```

Fig. 9. Parallel components execution

Conclusion

In the article we have shown an example of different interconnections of hardware components and their implementation as patterns using Event-B modeling language. We have also displayed the use of the patterns during refinement process. The developed approach makes it possible to design FPGA based hardware using mathematical proofs for ensuring correctness of system and diminishing common mode failure caused by design faults. The next step of the methodology is the translation from completed specification to hardware description language [10]. Patterns shown can be converted into HDL precisely because they use a strict definition in accordance with hardware design and describe any possible interconnection between components. Moreover, there is a possibility to use library elements of computer-aided design (CAD) system that improve the performance of a system and reduces the use of different resources of a chip.

The future research will be connected with the development of a mechanism for injecting library elements of CAD into formal specification and creation of translation algorithms and their implementation for Rodin platform.

References

1. Mikrin E. On-board spacecraft control complexes and software development for them / E. Mikrin. – M.: MSTU, 2003.

2. Soft microprocessor [electronic resource] –access mode http://en.wikipedia.org/wiki/Soft_microprocessor.

3. Prokhorova Y. Dependable SoPC-based On-board Ice Protection System: from Research Project to Implementation / Y. Prokhorova, V. Kharchenko, S. Ostroumov, S. Yatsenko, M. Sidorenko, B. Ostroumov // DepCoS-RELCOMEX. – 2008. – P. 135-142.

4. Abrial J.-R. The B-Book: Assigning Programs to Meanings / J.-R. Abrial. - Cambridge: Cambridge University Press, 1996.

5. Schneider S. The B-Method: An Introduction / S. Schneider. – UK: Palgrave, Cornerstones of Computing series, 2001.

6. Event-B and the Rodin Platform [electronic resource] – Access mode http://www.event-b.org/

7. Boulanger J.L. Formalization of digital circuits using the B method / J.L. Boulanger, A. Aljer and G. Mariano // Computers in Railways. – 2002. – Vol. VIII. – P. 691-700.

8. Seceleanu T. Systematic Design of Synchronous Digital Circuits / T. Seceleanu // TUCS Dissertations, Turku Centre of Computer Science. – 2001. – Vol. 32.

9. IEEE STD_LOGIC_1164. Package STD_LOGIC_1164

10. Prokhorova Y. An application of Event-B for developing systems on programmable logic / Y. Prokhorova, S. Ostroumov, E. Troubitsyna, L. Laibinis. // Radioelectronic and computer systems. – 2009. – Vol. 6(40). – P. 230-235.

Поступила в редакцию 15.01.2010

Рецензент: д-р техн. наук, проф., зав. кафедрой компьютерных систем и сетей В.С. Харченко, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Харьков.

EVENT-B ШАБЛОНЫ ДЛЯ РАЗРАБОТКИ ОТКАЗОУСТОЙЧИВЫХ АППАРАТНЫХ СРЕДСТВ

С.Б. Остроумов, Л.В. Лайбинис, Е.А. Трубицина

В статье рассмотрен первый шаг методологии по разработке гарантоспособных аппаратных средств на основе технологии программируемых логических интегральных схем. Сущность этого шага заключается в разработке шаблонов с использованием языка формальной спецификации и верификации Event-B, мощь которого состоит в использовании математических доказательств правильности модели. Статья рассматривает и описывает шаблоны, разработанные согласно синхронизму, т.к. большое количество систем является синхронными. Шаблоны описывают различные варианты соединений компонент, которые часто используются при проектировании аппаратуры. Разработанные шаблоны являются необходимым условием для корректной трансляции разработанной модели (спецификации) в язык описания аппаратуры (например, VHDL).

Ключевые слова: формальная разработка, Event-B, инвариант, шаблоны, аппаратура, проектирование, FPGA, Rodin.

EVENT-B ШАБЛОНИ ДЛЯ РОЗРОБКИ ВІДМОВОСТІЙКИХ АПАРАТНИХ СРЕДСТВ

С.Б. Остроумов, Л.В. Лайбініс, О.А. Трубіцина

У статті розглянуто перший шаг методології розробки гарантоздатних апаратних засобів на основі технології програмованих логічних інтегральних схем. Суть цього шагу є розробка шаблонів з використанням мови формальної специфікації та верифікації Event-B, що використовує математичні доводи коректності моделі. Стаття розглядає та описує шаблони, розроблені згідно з технікою синхронізму оскільки більшість систем синхронні. Шаблони описують різні варіанти з'єднання компонент, що часто використовуються при проектуванні апаратури. Розроблені шаблони є необхідною умовою для коректної трансляції розробленої моделі (специфікації) у мову опису апаратури (наприклад, VHDL).

Ключові слова: формальна розробка, Event-B, інваріант, шаблони, апаратура, проектування, FPGA, Rodin.

Остроумов Сергей Борисович – аспирант кафедры компьютерных систем и сетей Национального аэрокосмического университета им. Н.Е. Жуковского «ХАИ», Харьков, Украина, e-mail: S.Ostroumov@csac.khai.edu.

Лайбинис Линас Владавич – канд. техн. наук, ст. научный сотрудник кафедры информационных технологий Åbo Akademi University, Турку, Финляндия, e-mail: Linas.Laibinis@abo.fi.

Трубицына Елена Анатольевна – канд. техн. наук, доцент кафедры информационных технологий Åbo Akademi University, Турку, Финляндия, e-mail: Elena.Troubitsyna@abo.fi.