Automatic Refinement for Event-B through Annotated Patterns
Badr Siala, Jean-Paul Bodeveix, M Filali, Tahar Bhiri

To cite this version:
Badr Siala, Jean-Paul Bodeveix, M Filali, Tahar Bhiri. Automatic Refinement for Event-B through Annotated Patterns. 25th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2017), Mar 2017, Saint Petersburg, Russia. pp.287-290, 10.1109/PDP.2017.72. hal-01887899

HAL Id: hal-01887899
https://hal.science/hal-01887899
Submitted on 4 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author’s version published in:
http://oatao.univ-toulouse.fr/n° de post 19076

Official URL: https://ieeexplore.ieee.org/document/7912661/
DOI : http://doi.org/10.1109/PDP.2017.72

To cite this version: Siala, Badr and Bodeveix, Jean-Paul and Filali, Mamoun and Bhiri, Tahar Automatic Refinement for Event-B through Annotated Patterns. (2017) In: 25th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2017), 6 March 2017 - 8 March 2017 (Saint Petersburg, Russian Federation).

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr
Automatic refinement for Event-B through annotated patterns

Badr Siala
Université de Sfax and
IRIT UPS Université de Toulouse
France
Email: siala@irit.fr

Jean-Paul Bodeveix
IRIT UPS
Université de Toulouse
France
Email: bodeveix@irit.fr

Mamoun Filali
IRIT CNRS
Université de Toulouse
France
Email: filali@irit.fr

Mohamed Tahar Bhiri
Université de Sfax
Tunisia
Email: tahar_bhiri@yahoo.fr

Abstract—In this paper, we investigate how patterns could be used in order to generate Event-B refinements automatically through DSL(s) for temporal, timed or distribution patterns. Our ultimate goal is to generate code for a concurrent, or distributed framework, e.g., BIP.

I. INTRODUCTION

In this paper, we are interested by the construction of correct by construction distributed systems. For such a purpose, we propose DSL for expressing temporal, timed and distribution requirements over an input model. Then, from such a model Event-B machines are automatically generated. Ultimately, BIP executable models are produced in order to be deployed over a concurrent or distributed architecture. The rest of the paper is organized as follows. Section 2 and 3 give an overview of the BIP and Event-B languages. Section 4 outlines the coupling between Event-B and BIP. Section 5 illustrated the approach through a case study. We conclude by sketching future issues.

II. BIP

BIP (Behavior, Interaction, Priority) [1] is a semantic model for the component-based construction of systems. It is implemented in a language and a toolset. The BIP language offers primitives and constructs (atomic component, interface (data and port), connector and compound component) for modeling and composing atomic components. The latter are described as state machines, extended with data and functions written in the C language. A compound component is built from subcomponents, connectors and dynamic priorities. A BIP program is described by a tree whose terminal nodes are atomic components, non-terminal nodes are compound components and the root corresponds to the application.

The BIP toolset allows to simulate the execution of a BIP program by generating C++ code executable on a dedicated platform [2]. It also allows a posteriori verification of BIP programs by using model checking techniques [3].

III. EVENT-B

Event-B is a formal method that allows the development of correct by construction systems and software [4]. Event-B is based on the theory of abstract machines. An Event-B abstract machine is semantically a symbolic transition system where the state space is characterized by an invariant which should be preserved by each event. An Event-B development is a chain of machines linked by a refinement relation which entails a weak simulation relation. These two semantics aspects are enforced by proof obligations to be discharged.

Event-B supports natively a top-down formal development process based on a refinement mechanism with mathematical proofs. The refinement-based method in Event-B consists of developing the system incrementally starting from an abstract model which is a specification of the system. More details of the system are added gradually (step-by-step) in a concrete model which has to preserve the functionality and properties of abstract models. As the Event-B model is expressed in a formal language, it is possible to generate proof obligations ensuring the correctness of the development. Models in Event-B are described in terms of the two basic constructs: contexts and machines. The latter contains the dynamic part of a model whereas contexts contain the static part. Indeed, contexts define abstract data types through sets, constants, axioms and theorems while machines define symbolic labelled transition systems through variables (state) and events specifying their evolution while preserving invariant properties. Moreover, an Event-B machine includes a mandatory event called INITIALISATION which defines the initial state.

As a running example, we will consider the controller for pedestrian crossing and traffic lights as a case study (see section V). The context (Listing 1) describes the roles played by the actors of the system as an enumerated set. The set Role models the two roles (Vehicle and Pedestrian) and the fact that Role contains only these two elements.

context roles
sets Role constants Vehicle Pedestrian axioms
@r partition(Role, {Vehicle}, {Pedestrian})
end

Listing 1. Roles context

In order to reason about our system in a simple way, we start by building an abstract model which takes account of only very few constraints. The turns machine (Listing 2) defines one state variable turn in the variables clause. This variable is typed by the invariant, labelled turn_ty, and initialized in

DOI 10.1109/PDP.2017.72
the INITIALISATION event. The switch event allows to give an access right to each role in turn. The next value of the turn variable is obtained by applying the function defined as an enumerated set of source-target ordered pairs on the current value of turn.

```
machine turns sees roles variables turn invariants @turn:ty turn ∈ Role events event INITIALISATION then @t_init turn := Vehicle end event switch then @t turn := (Vehicle→Pedestrian,Pedestrian→Vehicle)(turn) end
```

Listing 2. The turns machine

Refinement is at the core of Event-B modeling [4], as its predecessor B-method [5]. It is a process allowing the construction of a model in a gradual way. Starting from an abstract model, each refinement step adds further details. Thus, a development is an ordered sequence of models where each element is a refinement of the previous one. A refinement step must guarantee that every behavior of the concrete model is also a behavior of the abstract model. The refinement relation between a concrete and an abstract machine ensures that executions of the concrete machine can be simulated on the abstract machine. Event-B supports also weak simulation: the concrete machine can introduce new events that refine skip. However they should not take control indefinitely. A variant must be provided to help proving this property. Finally, it is possible to express that the concrete machine does not introduce deadlocks that were not permitted by the abstract machine.

As an example, a first refinement of our initial model (Listing 2) is the concrete machine Crossing below. In this refinement, we replace the unique variable turn by two boolean variables modeling the right of each participant and we express the safety property. A gluing invariant relates the variable turn with the two new variables.

```
machine crossing refines turns sees roles variables vehicle_path pedestrian_path invariants @vp:ty vehicle_path ∈ B @pp:ty pedestrian_path ∈ B → gluing invariant @vp.1 vehicle_path = TRUE ⇒ turn = Vehicle @pp.1 pedestrian_path = TRUE ⇒ turn = Pedestrian @safety vehicle_path ≠ pedestrian_path events event INITIALISATION then @wp_init vehicle_path := TRUE @wp_init pedestrian_path := FALSE end event switch refines switch then @sw vehicle_path := pedestrian_path @sw pedestrian_path := vehicle_path end
```

Listing 3. The crossing machine

IV. COUPLING EVENT-B AND BIP

The BIP language provides powerful capabilities for describing the architectural aspects of an application, namely components made of subcomponents connected through their ports. Event-B is used for the formal specification and decomposition of a distributed system. The correction criteria defined by Event-B ensures the preservation of safety properties in the incremental development of the model. Event-B is enriched with decomposition patterns which permit the refinement of a unique machine by a product of sub-machines which may be further refined and decomposed until we get executable specifications. We consider the shared event decomposition method [6] and CSP-like composition which allows data exchange through shared event parameters. Then BIP is used for the composition of atomic components designed in Event-B. Its connectors implement Event-B shared event synchronizations and data exchange while preserving their semantics. Figure 1 summarizes the coupling of Event-B and BIP to produce secure and scalable distributed systems.

In order to assist the user in this design process, we provide a support to refine and decompose an Event-B model and lastly to generate BIP models. These steps are driven by domain specific languages which give parameters to the transformations (events or variables to be introduced, temporal properties to be satisfied, data to be remotely accessed, subcomponent names, variable mapping, location of guard or action computation, ...). The proposed temporal or temporised patterns are in the spirit of those proposed by Dwyer et al. [7].

V. CASE STUDY

A. Requirements specification

A pedestrian crossing light is a road signals system which gives priority to vehicles. Pedestrians are allowed to cross only
when the signals halt vehicle traffic on the road. This system consists of a set of traffic lights for drivers, a set of light signals for pedestrians and a push button. The latter is used by pedestrians to change traffic signal to give pedestrians enough time to cross. In this paper, we show how such a system can be built incrementally in Event-B. The centralized Event-B model is decomposed into several subcomponents which are translated into BIP executable models in order to be deployed over a concurrent or distributed architecture. The following subset of requirements is taken into account:

1) The system switches between pedestrian crossing and vehicle traffic.
2) Pedestrian should ask for crossing right before being allowed to cross.
3) Demands done when pedestrian have already the crossing right are ignored.
4) Pedestrian can cross 30 time units after their first valid demand.
5) The system is made of four components: pedestrian lights, vehicle lights, sensors and the control system.

B. Models

In this section, we illustrate our proposed methodology by constructing the case study described above.

1) Event splitting: The first refinement aims at splitting the event switch depending on pedestrian_path value: when false, switch corresponds to giving crossing authorization to pedestrian; otherwise, it corresponds to end_of_authorization. Refinement is guaranteed. Absence of deadlock introduction comes from the fact that conditions are complementary.

As a result, we get the machine given in Listing 4 where the switch event as been split.

```plaintext
Listing 4. Introducing the authorization event

machine Crossing_authorization refines Crossing
variables vehicle_path pedestrian_path
events
  event INITIALIZATION extends INITIALIZATION end
  event authorization refines switch
      where @wp pedestrian_path = FALSE then
          @wp vehicle_path = FALSE
          @wp pedestrian_path = TRUE end
  event end_of_authorization refines switch
      where @wp pedestrian_path = TRUE then
          @wp vehicle_path = TRUE
          @wp pedestrian_path = FALSE end
```

2) Introduction of the request event: This step adds a new event allowing the pedestrian to ask for the authorization. This event is supposed to be uncontrollable and thus not guarded. It should have no effect if pedestrians are already allowed to cross. It is introduced by constructing a refinement from the abstract model described in Listing 4. We get the machine given in Listing 5 containing the request event.

```plaintext
Listing 5. Crossing_request machine

machine Crossing_request refines Crossing_authorization
variables vehicle_path pedestrian_path authorization_req
invariants
  @req_ty authorization_req in @ @auth_path pedestrian_path = TRUE => authorization_req = FALSE
  @end_of_authorization
  @req authorization_req = FALSE

events
  event INITIALISATION extends INITIALIZATION then
      @auth authorization_req := FALSE end
  event request //precedes authorization
      then @auth authorization_req := TRUE end
  event authorization extends authorization
      where @req authorization_req = TRUE
      event end_of_authorization extends end_of_authorization
      then @req authorization_req := FALSE end
```

3) Adding Timed constraints: This step adds timing constraints between existing events. The authorization should be given 30 time units after the first occurrence of a request in each segment delimited by the end_of_authorization event. We reuse the existing status variable authorization_req and introduce a new status variable is_waiting and a clock variable (waiting) incremented by a new tick event.

The refinement of the previously obtained machine generates a refinement with the two newly introduced variables and the new event tick managing the discrete advance of time. The control variable authorization_req is reused. Existing events are extended so that the refinement property is satisfied by construction.

```plaintext
Listing 6. Crossing_timed machine

machine Crossing_timed refines Crossing_request sees eTiming
variables vehicle_path pedestrian_path authorization_req
is_waiting waiting
invariants
  @w waiting in N
  @w is_waiting in B
  @is_waiting = TRUE => waiting <= WaitingTime

events
  event INITIALISATION extends INITIALIZATION then
      @w waiting := 0
      @w is_waiting := FALSE end
  event request extends request
      then @w waiting := (TRUE => waiting, FALSE => 0)(is_waiting)
      @w is_waiting := (FALSE => TRUE, TRUE => is_waiting)(authorization_req) end
  event authorization extends authorization
      where @c is_waiting = TRUE => waiting = WaitingTime
      then @w is_waiting := FALSE end
  event end_of_authorization extends end_of_authorization
      when @w is_waiting = FALSE
      event tick
          when @w is_waiting = TRUE => waiting < WaitingTime
          then @w waiting := waiting + 1 end
```

Listing 5. Crossing_request machine

Listing 6. Crossing_timed machine
4) Decomposition: This step builds a distributed model from a centralized one. Four subcomponents are introduced: controller, vehicle lights VLights, pedestrian lights PLights and sensors. Then, variables are mapped to components. These declarations are specified as follows:

```
shared event decomposition Crossing_split
refines Crossing_timed
components
    Controller VLights PLights Sensors
mappings
    variables is_waiting authorization_req
        @ local_is_waiting = is_waiting // on Sensors
    variable waiting @ Controller
        @w waiting @ (TRUE->waiting_FALSE->0)(local_is_waiting)
        // on Controller
    variable vehicle_path @ VLights;
    variable pedestrian_path @ PLights;
end
```

From this specification, a refined model is generated [8]. It introduces new variables containing copies of the original ones and events to refresh them. Parameters are also added to existing events when the synchronous access to distant variables is possible. As an example, Listing 7 shows the refined request event with one parameter, local_is_waiting, of which value is given by the guard.

```
event request refines request
any local_is_waiting request
where
    // access to remote variables used in actions
    @wa local_is_waiting = is_waiting // on Sensors
    then
        @auth authorization_req := TRUE // on Sensors
        @w waiting := (TRUE->waiting_FALSE->0)(local_is_waiting)
        // on Controller
        @w is_waiting := (FALSE->TRUE,TRUE->is_waiting)
        (authorization_req) // on Sensors
end
```

Listing 7. The request event

Then, the obtained refined model is projected on the components, as mentioned by the comments associated to guards and actions. Here, the value of the event parameter is provided by the Sensors component. The action updating waiting is performed by the request event of the Controller component. It uses the value of the is_waiting variable provided by the Sensors component through its synchronized request event.

5) Generation of BIP models: We generate for each Event-B component atomic BIP component [8]. Four atomic components are produced by our BIP code generator: ty_Controller, ty_VLights, ty_PLights and ty_Sensors. For example, Listing 8 presents the atomic component ty_VLight which contains three state variables and two exported ports accessible to connectors.

```
atom type ty_VLights()
    /\ state variables /\n    data bool vehicle_path
    /\ temporary variables updated by connectors /\n    data bool waiting
    /\ ports /\n    export port ty_empty_port authorization()
    export port ty_empty_port end_of_authorization()
    place P0
    /\ initial /\n    initial to P0
do { vehicle_path=true ;}
/\ transitions /\n    on authorization from P0 to P0
do { ... }
    on end_of_authorization from P0 to P0
end
```

Listing 8. Atomic component ty_VLight

The composition of BIP components intended to obtain from a set of components (atomic or compound) a compound component that models the application. To achieve this, we define connectors (stateless entities) that enable interactions among a set of components via their interface ports. Listing 9 provides the compound component traffic_light which contains an instantiation of two atomic component (ty_Controller and ty_VLight) and two connectors that interconnect the two instances.

```
compound type traffic_light()
    component ty_Controller Controller()
    component ty_VLights VLights()
    \...
    connector ty_request request(Controller.request, Sensors.request)
    connector ty_authorization authorization(Controller.authorization, VLights.authorization, PLights.authorization, Sensors.authorization)
end
```

Listing 9. Compound component traffic_light

VI. CONCLUSION

In this paper, we have presented an ongoing work which aims at promoting formal methods for the development of distributed systems. We use a refinement-based methodology. Thanks to DSL(s), we generate either Event-B machines which can be refined further or ultimately BIP code. As future work, we envision the (meta) verification of the overall process in order to ensure the correctness by construction of the generated machines.

REFERENCES