EAI Endorsed Transactions
on AI and Robotics

Research Article EAI.EU

An Event-B Approach to the Development of Fork/Join Parallel Programs

Jie Peng[1,2], Tangliu Wen*[1,3], Yiguo Yang[4], Guoming Huang[1].

[1]Gannan University of Science and Technology, Ganan 341000, China
[2]Ganzhou Cloud Computing and Big Data Research Center, Ganan 341000, China
[3]Ganzhou Key Laboratory of Intelligent Finance, Ganan 341000, China
[4]School of Computer Engineering and Science, Shanghai University, Shanghai 2000444,China

Abstract

Fork/Join is a simple but effective technique for exploiting the parallelism. When developing a parallel program using Fork/Join, one of the main things is how a large task is decomposed into subtasks whose results can be combined as a final result. In this paper we show how to develop Fork/Join parallel programs through refinement and decomposition. We take Fork/Join style task decomposition as a refinement which we call Fork/Join refinement. Proof obligations of refinement can ensure the correctness of decomposition. For practical application, we provide a refinement pattern for the Fork/Join refinement and extend an atomicity decomposition diagram to illustrate it. Our approach provides a good framework for modeling Fork/Join parallel programs and showing proof obligations of correctness for such programs. We illustrate the approach by applying it on a small case.

Received on 23 December 2021; accepted on 01 February 2022; published on 18 February 2022

Keywords: Event-B, Fork/Join, refinement, decomposition

doi:10.4108/airo.v1i.16

1. Introduction

Event-B[1,2] is a formal modelling method that uses abstraction and refinement and falls into the general topic of AI [3-7]. The process of modeling begins with a high level abstraction model as a specification of a system, then refines the model gradually until a desired level abstraction model is gained. If a large model is difficult to be refined, we can decompose it into small sub-models that can be refined easily and independently. The proof obligations of refinement are generated automatically by the Rodin Platform tool. In order to verify the correctness of refinement in Event-B, the proof obligations are given to the Rodin Platform carrying out automatic or interactive proofs.

Fork-join[8,9] is a simple but effective technique for exploiting the parallelism. Fork primitive diverges the program flow into two or more parallel flows, and Join primitive combines multiple parallel flows into a single flow. Fork/Join is often used to develop the parallel version of a divide-and-conquer algorithm. The idea is simple: a large task can be decomposed into small subtasks which can be computed in parallel, then join the results of subtasks to get the result of the original task. When developing a parallel program using Fork/Join, one of the main things to consider is how a large task is decomposed into subtasks whose results can be combined as a final result.

In this paper we focus on developing Fork/Join parallel programs through refinement and decomposition. We take Fork/Join style task decomposition as a refinement(we call it Fork/Join refinement). Proof obligations of refinement can ensure the correctness of decomposition. For practical application, we propose Fork/Join refinement pattern. The pattern starts with a high abstract model containing a single event as a specification of task. Then the model is refined into forks events solving the subtasks and a join event combining the results of subtasks. Forks events as new events

*Email address: pengjie@jxust.edu.cn (Jie Peng),
yangyiguo@139.com (Yiguo Yang),
202039624@qq.com (Guoming Huang),
wqtlglk@163.com, Corresponding author (Tangliu Wen*).

refine skip, join event as main event refines the single event of the initial model. We introduce control variables to realize three synchronization patterns between forks and join events. We give a general function as variant to prove that forks events is convergent. We will decompose the Fork/Join model in such a way that each fork's events and join event form a fork sub-model and a join sub-model, respectively. Then refine the sub-models until the desired level of abstraction is gained.

In order to facilitate practical use of the refinement pattern, we extend atomicity decomposition diagram[10,11] to illustrate it. Our atomicity decomposition diagram illustrates the refinement relationship between the abstract event and Fork/Join refinement events and the synchronization pattern between forks events and join event explicitly.

The rest of the paper is organized as follows. Section 2 gives some background on Event-B and the Rodin tool. Section 3 presents an overview of our approach and gives Fork/Join refinement pattern. Section 4 introduces an example to illustrate our approach. Section 5 gives concluding remarks and discusses related work.

2. Event-B

Event-B is an extension of Abrial's B method for modelling parallel and distributed systems. In Event-B, machines are defined in a context, which has a unique name and is identified by the keyword CONTEXT. It includes the following elements: SETS defines the sets to be used in the model; CONSTANTS declares the constants in the model; and finally, AXIOMS defines some restrictions for the sets and includes typing constraints for the constants in the way of set membership. An Event-B model[1] is described by a machine and multiple contexts. The machine specifies the behavioural properties of a model. The contexts consist of carrier sets, constants and axioms, which describe the static properties of a model. The relationship between machine and contexts is that the machine can refer to contexts. Machine may contain invariants, variables, variants and events. The variables are updated by events. Invariants constraint the variables and must be hold by each event. An event can be defined as "any m when G(m, v) then A(m, v) end" where m is the parameters of an event. The guard G(m, v) specifies when an event is activated. The action A(m, v) describes the state change of a model when the event occurs.

A refinement methodology is used by software architects to incrementally develop a model of a system starting from the initial most abstract specification and following gradually through layers of detail until the model is close to the implementation. Event-B also incorporates a refinement methodology. Assume that the machine M is refined by machine N. M is called a concrete machine and N is called a abstract machine. The gluing invariant establishes a state relationship between the abstract machine and the concrete machine. In refinement, some events of the concrete machine refines the abstract events, some are new events which must be proven to refine skip event. To demonstrate that the machine N is a correct refinement model of the machine M, the proof obligations for refinement must be proven in the Event-B. They include guard strengthening, action simulation and witness feasibility.

Refinement does not solve completely the mastering of the complexity. As a model is more and more refined, the number of its state variables and that of its transitions may augment in such a way that it becomes impossible to manage the whole. At this point, it is necessary to cut our single refined model into several almost independent pieces. Decomposition is precisely the process by which a single model can be split into various component models in a systematic fashion. In doing so, we reduce the complexity of the whole by studying, and thus refining, each part independently of the others. The very definition of such a decomposition implies that independent refinements of the parts could always be put together again to form a single model that is guaranteed to be a refinement of the original one.

Atomicity decomposition diagram has been proposed by Fathabadi et al. The diagram can explicitly illustrate (a) the explicit refinement relationships between abstract and concrete events, and (b) the explicit control flow between events. So it facilitates developers to carry out refinements in explicit way.

Model decomposition[13] is to decompose a large model into several small sub-models which can be refined easily and independently. There are two main methods for the Event-B model decomposition, namely shared event decomposition and shared variable decomposition. Shared event method(shared variable method) is well suited for the sub-models which communicate via message passing (shared variables). We can create and analyse Event-B models by the Rodin tool[14]. The tool can also generate proof obligations and support for automated and interactive theorem proving.

3. Fork/Join refinement pattern

The Fork/Join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached. Fork/Join can be considered a parallel design pattern. Fork/Join parallelism is among the simplest and most effective design techniques for obtaining good parallel performance. Fork/join algorithms are parallel versions of familiar divide-and-conquer algorithms, taking the typical form:

```
Result solve(Problem problem) {
  if (problem is small)
    directly solve problem
  else {
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
  }
}
```

In this section we propose Fork/Join refinement pattern that is used to develop Fork/Join parallel programs. Assume we need to solve a problem using Fork/Join parallel program. First we model the problem abstractly with a initial model. The initial model called specification model which specifies the program at a high abstract level. Secondly, we refine the specification model. In the refinement Fork/Join framework is introduced into the new model. We call the refinement " Fork/Join refinement". We illustrate the refinement pattern by using an extended atomicity decomposition diagrams in Fig.1.

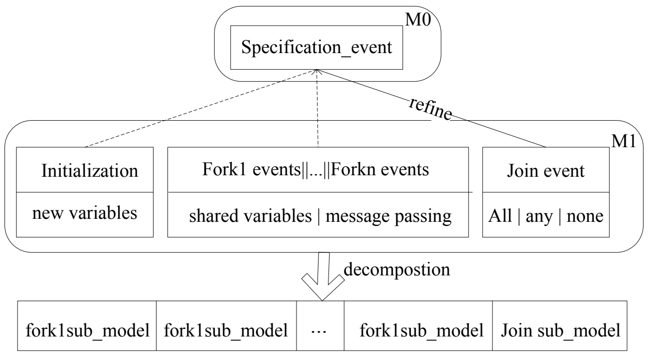

Fig.1. Extended Atomicity Decomposition Diagram for Fork/Join rerfinement

Here, M0 machine denotes the initial model as a specification model. It contains a single Specification_event. Machine M0 is refined by machine M1 called Fork/Join model. In machine M1, the problem is split into multiple subtasks for n parallel threads (fork1..forkn). Each thread contains multiple events to solve the subtask assigned it. These events are called forks events(e.g. fork1 events for fork1's subtask). All forks events run in parallel. Join event solves the problem by joining the results of forks threads. The diagram shows how the Specification_event is split into an initialisation event, forks events and a join event. The initialisation event, forks events are linked by a dashed line which denotes they are new events. Join event are linked by a solid line which means that it refines Specification_event. So the initialisation event, forks events must be proven to refine the skip event.

In forks events section there are two options: shared variables and message passing. They represent communication style among forks threads. They are useful for us to choose the model decomposition methods. If the threads communicate via shared variables(message passing), shared-variable decomposition method(shared-event decomposition) are chosen to decompose the Fork/Join model.

In Join event section there are three options: all, any and none. They denote the synchronization patterns between forks events and join event. The option all is default. It means that join event executes after all of forks events are completed. The option any means that join event executes after any single fork events is completed. The option none means that Join event executes, without waiting for any of forks events to complete. Some control variables can be introduced to realize the synchronization in Event-B model. For example, the all synchronization pattern is illustrated as follows:

```
Fork1_event             Fork2_event             Join_event
status                  status                  when
  convergent              convergent              temination1=true
when                    when                      temination2=true
  temination1=false       temination2=false     then
then                    then                      return result
  temination1=true        temination2=true
end                     end
```

Here, termination1 and termination2 are control variables. Fork1_event and Fork2_event are forks events. The Join_event can be executed only when both Fork1_event and Fork2_event are completed.

In order to guarantee that join event can always be activated, we must prove that forks events are convergent. So the forks events must be classified as convergent events. We give a general function as a variant:H(finish1,. . . ,finishn), where finish1...finshin are control variables. The value of the function is the amount of control variables being true.

In order to further refine forks events independently, we can decompose the model into small sub-models. As mentioned previously, the decomposition approach depends upon the options of forks events. We decompose the model in such a way that each fork's events and join event form fork sub-model and join sub-model, respectively.

Finally, we give a general process of developing Fork/Join parallel programs using the Event-B formal method as follows:

Step 1: Define specification model of the program.

Step 2: Refine the initial model using Fork/Join refinement pattern.

Step 3: Decompose the Fork/Join model into forks sub-models and join sub-model.

Step 4: Refine the sub-models until the desired level of abstraction is gained.

4. Case study:finding a zero of a function

4.1. Problem description and initial model

The zero of a function is a point x such that f(x)=0. Let us assume f is a total function mapping integers into integers. We know the function has at least one zero point. Design a program that finds a zero of the function f. A solution is to split the problem into two sub-problems which can be solved by two threads independently, namely fork1 and fork2. The original problem can be solved by combining the results of two threads. The pseudocode of the program is as follows.

```
fork1
while found1=false and found2=false
do
if f(m)=0  then found1=true
else m++      end
end
```

```
fork2
while found1=false and found2=false
do
 if f(n)=0 then found2=true
else n--  end
end
```

```
main thread
 m=1; n=0;
 fork(fork1)
  fork(fork2)
 join (fork1)
 join (fork2)
result:=(f(m)=0)?m:n
```

Here, fork1 and fork2 threads execute concurrently to look for a zero of the function. When the two threads terminate, the zero is one of the variables m or n. Note that the Boolean variables found1 and found2 are shared by both fork1 and fork2. Here we make the assumption that the writing and reading operations of shared variables are never executed simultaneously. The shared variables are used to exchange information of termination between both threads. When fork1 thread finds a zero of the function, it will terminate with found1=true. Fork2 thread need to read the variable found1 before continuing its search. So fork2 will also stop searching when reading found1=true.

Now we define a initial model for specification of the problem. First, we define the function f by two axioms in the context. The F0_find is only one event of initial model to specify the program at the high abstract level. The zero of the function is to be stored in a variable result which is constrained to be a natural number by the invariant: inv result ∈ N.

```
Constants
  f
axioms:
  axm1:f∈N→N
  axm2: ∃m, m∈N. f(m)=0
```

```
F0_find
 any x where
  x∈N
  f (x)=0
 then
  result=x
 end
```

4.2. Fork/Join refinement

Fork/Join framework was introduced in Fork/Join refinement. Here we split the single event F0_find into Join_event and Fork1 and Fork2 events. The Join_event refines F0_find, Fork1 and Fork2 events are new events. Fork1 events consist of Fork1_found event and Fork1_not_found event(Fork2 events is symmetric). Fork1_found event denotes that fork1 thread has found a positive zero and terminated. Fork1_not_found event denotes that fork1 thread knows that the fork2 thread has found a nonpositive zero of the function and terminated. Join_event and Fork1 events are shown as follows:

```
Fork1_found(convergent)
status
 convergent
any  p   where
 p∈N and p>0
 found1=false
 found2=false
 finish1=false
 f(p)=0
then
 m=p
 found1=true
  finish1=true
end
```

```
Fork1_not_found(convergent)
status
  convergent
when
  found1=false
  found2=true
  finish1=false
then
  finish1=true
end
```

```
Join_event
refines  f0_find
when
  finish1=true
  finish2=true
with
  x=(f(m)=0)?m:n
then
  result=(f(m)=0)?m:n
end
```

The new boolean variable found1 (found2) is introduced to model whether fork1(fork2) has found a zero of the function. The Join_event can not be executed until all Fork1 and Fork2 events have completed. In order to do it, control variables finish1 and finish2 are introduced to model whether fork1 and fork2 events have terminated respectively. The initialization event sets the new boolean variables to false.

Note that forks events are classified as convergent events. In order to prove that forks events are convergent, we need to give a variant which is decreased by each fork event. We define the function H(finish1,finish2) as a variant. As mentioned in Section 3, The value of the function is the amount of finish1 and finish2 being true.

The proofs of action simulation and guard strengthening are required in order to prove F0_find is refined by the Join_event. Note that a witness for the abstract parameter x is given in the Join_event. The variable x is assigned a same expression as result in the witness. Given the witness, it is easy to show the proof of action simulation. In order to prove guard strengthening, some invariants describing relationships of the new variables are introduced as follow:

invariants
inv1 $found1 = false \land found2 = false \Rightarrow finish1 = false$
inv2 $found1 = false \land found2 = false \Rightarrow finish2 = false$
inv3 $found1 = true \Rightarrow f(m) = 0 \land finish1 = true$
inv4 $found2 = true \Rightarrow f(n) = 0 \land finish2 = true$

The invariants inv1 and inv2 state that if two forks events have not found a zero of the function, then they have not terminated. The invariant inv3 states that if fork1 thread has found a zero of the function, then it must have terminated. The invariant inv4 states that fork2 thread has found a zero of the function, then it must have terminated. The proof of action simulation is shown as follows:

Proof.

$$
\begin{aligned}
& \text{inv1} \wedge \text{inv2} \Rightarrow \text{found1} = \text{fasle} \wedge \text{found2} = \text{false} \\
& \Rightarrow \text{finish1} = \text{false} \wedge \text{finish2} = \text{false(predicate logic)} \\
& \Rightarrow \text{finish1} = \text{true} \vee \text{finish2} = \text{true} \\
& \Rightarrow \text{found1} = \text{true} \vee \text{found2} = \text{true} \\
& \qquad\qquad\qquad\qquad \text{(converse negative proposition)} \\
& \Rightarrow \text{finish1} = \text{true} \wedge \text{finish2} = \text{true} \\
& \Rightarrow f(m) = 0 \vee f(n) = 0(\text{inv3} \wedge \text{inv4}) \\
& \Rightarrow \text{finish1} = \text{true} \wedge \text{finish2} = \text{true} \wedge x = (f(m) = 0)?m : n \\
& \Rightarrow f(x) = 0(\text{witness})
\end{aligned}
$$

□

4.3. Model decomposition and further refinement

We want to further refine fork1 and fork2 independently to clearly model how they search for a zero of the function. Here we use the shared-variable approach to decompose the first refinement model. We split the model in such a way that each fork's events and join event form fork sub-model and join sub-model,respectively. Therefore, there are three different sub-models, namely fork1, fork2 and join sub-model. The variable m is shared by fork1 events and join event. Join event does not execute until fork1 events terminate. So, there is no data race between fork1 events and join event. The variables finish1, found1 and found2 are shared by fork1 and fork2 events. Fork1 and fork2 events execute concurrently. Consequently, external events must be added to fork1 sub-model ensure that the behaviour of the shared variable in the non-decomposed model is preserved. According to shared-variable decomposition approach, we define the following external events of fork1 sub-model((fokr2 sub-model is similar):

```
Ext_Fork2_found(convergent)
any p  finish2  n where
p∈N and p<=0
found2=false
found1=false
finish2=false
f(p)=0
then
found2=true
end
```

```
Ext_Fork_2notfound(convergent)
any  finish2   where
  found2=false
  found1=true
  finish2=false
then
  skip
end
```

Now we focus on the further refinement of fork1 sub-model(fork2 sub-model is similar). Note that the zero of the function is found by Fork1_found in a single abstract step. In order to model the process of searching for the positive zero of the function, the fork1_found1 event is split into two sub-events, namely q_next and fork1_found2. Fork1_found2 refines Fork1_found and q_next is a new event. The new model states that fork1 thread starts from 1 using ascending order to search for a zero of the function. A new variable q is introduced to represent the current number which fork1 is checking. A new invariable about q is introduced:
inv7 $\forall x,\ x >= 1 \ \wedge\ x <= q \ \wedge\ f(x) \neq 0 \ \Rightarrow$ found1 = false.

```
Fork1_found2
refine   Fork1_found1
when
  f(q)=0
  found2=false
  found1=false
  finish1=false
with p=q
then
  m=q
  found1=true
  finish1=true
end
```

```
Q_next
when
  found1=false
  finish1=false
     h.  h>=1∧h<=q ⇒ f(h) ≠0
then
   q=q+1
end

initialization
  q=1
end
```

In the refinement, Fork1_not_found keep unchanged. Fork1_not_found, Ext_Fork2_found and Ext_Fork2_not_found do not refer to q, so they satisfy the new invariant. Given the witness p=q, It is easy to prove that Fork1_found2 is a correct refinement of Fork1_found.

5. Related Work and Conclusion

Developing parallel programs using Event-B is a very active field of research[17,21,22,23]. Many researches emphasize how to decompose a large model into parallel sub-models. Butler[15,18,19] proposes a shared-event decomposition in which sub-models interact via shared events. Hoang and Abrial[20] proposes a shared-variable decomposition in which sub-models interact via shared variables. Pontus Bostrom et al.[16] extends the decomposition methods by introducing explicit control flow for the concurrent sub-models. Sritharan et al. [24] present an approach to generate SPARK code from Event-B models. System models in Event-B are translated into SPARK packages including proof annotations. Properties of the Event-B models such as axioms and invariants are also translated and embedded in the resulting models as pre- and post-conditions. Generating proof annotations from Event-B models has been investigated in [25]. Their work explores the mapping between Event-B and Dafny [26] constructs. The methods focus on general parallel programs. In this paper, we concentrate on developing Fork/Join parallel programs using Event-B. Our approach introduces the concept of Fork/Join refinement corresponding to the Fork/Join task decomposition. In the refinement, forks events correspond to subtasks, join event corresponds to joining the results of subtasks. For practical application, Our approach provides a pattern for the refinement and illustrate it by using an extended atomicity decomposition diagram. We will choose a model decomposition method according to communication style among forks events to decompose the Fork/Join model into multiple sub-models. Then refine the sub-models until the desired level of abstraction is gained. Our approach provides a good framework for modeling Fork/Join parallel programs and showing proof obligations of correctness for such programs.

Acknowledgement

We thank anonymous referees for their suggestions and comments on an earlier version of this paper. The work is supported by the Science and Technology Research Project of Jianxi Province Educational Department (Grant No.209411) and the Scientific Research Foundation for Young Doctoral Teachers of College of Applied Science, Jiangxi University of Science and Technology.

References

[1] J. R. Abrial, *Modeling in Event B: System and Software Engineering*, Cambridge University Press, London, 2010

[2] J. R. Abrial, S. Hallerstede, Refinement Decomposition and Instantiation of Discrete Models: Application to Event-B, Fundamenta Informaticae 2 (2007) 1-28

[3] A. T. Khan, X. Cao, Z. Li, and S. Li, “Evolutionary Computation Based Real-time Robot Arm Path-planning Using Beetle Antennae Search”, EAI Endorsed Trans AI Robotics, vol. 1, pp. 1–10, Jan. 2022.

[4] Z. Chen, J. Walters, G. Xiao, and S. Li, “An Enhanced GRU Model With Application to Manipulator Trajectory Tracking”, EAI Endorsed Trans AI Robotics, vol. 1, pp. 1–11, Jan. 2022.

[5] Y. Yang, L. Liao, H. Yang and S. Li, “An Optimal Control Strategy for Multi-UAVs Target Tracking and Cooperative Competition,” IEEE/CAA Journal of Automatica Sinica, vol. 8, no. 12, pp. 1931-1947, 2020.

[6] L. Liao, Y. Yang, “Adaptive radial basis function neural network bi-quadratic functional optimal control for manipulators,” Control Theory & Applications, vol. 37, no. 1, pp. 47-58, 2020.

[7] S. Akkara and J. T, “PI Controller Based Switching Reluctance Motor Drives using Smart Bacterial Foraging Algorithm”, EAI Endorsed Trans AI Robotics, vol. 1, pp. 1–8, Jan. 2022.

[8] D. Lea, A Java fork/join framework, in: Proc. ACM 2000 conference on Java Grande, 2000, pp. 36-43

[9] M. D. Wael, How to achieve scalable fork/join on many-core architectures, in: Proc. Annual conference on Systems, programming, and applications: software for humanity, 2012, pp. 85-86

[10] A. S. Fathabadi, M. Butler, A systematic approach to atomicity decomposition in event-b, in: Proc. Software Engineering and Formal Methods, 2012, pp. 78-93

[11] A. S. Fathabadi, M. Butler, Applying Event-B atomicity decomposition to a multi media protocol, in: Proc. Formal Methods for Components and Objects, 2010, pp. 89-104

[12] S. Hallerstede, M. Leuschel, D. Plagge, Refinement-animation for Event-B-towards a method of validation, in: Proc. Abstract State Machines, Alloy, B and Z, 2010, pp. 287-301

[13] T. S. Hoang, A. Iliasov, R. A. Silva, et al, A survey on Event-B decomposition, Electronic Communications of the EASST, 46 (2011) 1-15

[14] J. R. Abrial, M. Butler, Hallerstede S, et al, Rodin: an open toolset for modelling and reasoning in Event-B, International journal on software tools for technology transfer, 16(2010) 447-466

[15] A. Edmunds, M. Butler, Linking Event-B and concurrent object-oriented programs, Electronic Notes in Theoretical Computer Science, 214 (2008) 159-182.

[16] P. Bostrom, F. Degerlund, K. Sere, et al, Derivation of concurrent programs by stepwise scheduling of Event-B models, Formal Aspects of Computing, 8 (2012) 1-23

[17] D. Mery, N. K. Singh, Automatic code generation from event-B models, in: Proc. the Second Symposium on Information and Communication Technology, 2011, pp. 179-188.

[18] L. Butler, Decomposition structures for Event-B, in: Proc. Integrated Formal Methods, 2009, pp. 20-38

[19] R. Silva, C. Pascal, M. Butler, et al, Decomposition tool for Event-B, Software: Practice and Experience, 41 (2011) 199-208.

[20] T. S. Hoang, J. R. Abrial, *Event-B decomposition for parallel programs*, Springer Berlin, Heidelberg, 2010

[21] G. Dupont, Y. Ameur, M. Pantel, et al. Handling refinement of continuous behaviors: a proof based approach with event-B, 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE). IEEE, 2019: 9-16.

[22] S. Sritharan, T. S. Hoang, Towards generating SPARK from Event-B models, International Conference on Integrated Formal Methods. Springer, Cham, 2020: 103-120.

[23] M. Leuschel, M. Mutz, M. Werth, Modelling and validating an automotive system in classical B and event-B, International Conference on Rigorous State-Based Methods. Springer, Cham, 2020: 335-350.

[24] Sritharan, Sanjeevan, and T. S. Hoang. Towards generating SPARK from Event-B models, International Conference on Integrated Formal Methods. Springer, Cham, 2020.

[25] D. Mohammadsadegh, B. Michael, R. Abdolbaghi, Verifiable code generation from scheduled Event-B models. In 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018.

[26] K. Rustan M. Leino. Developing verified programs with Dafny. In Rajeev Joshi, Peter Muller, and Andreas Podelski, editors, Verified Software: Theories, Tools, Experiments-4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings, volume 7152 of Lecture Notes in Computer Science, page 82. Springer, 2012.