Verifying Event-B Hybrid Models Using Cyclone

Hao Wu[1(✉)] and Zheng Cheng[2]

[1] Computer Science Department, Maynooth University, Kildare, Ireland
haowu@cs.nuim.ie
[2] Telecom Nancy, University of Lorraine, Thionville, France
zheng.cheng@inria.fr

Abstract. Modelling hybrid systems using Event-B is challenging and users typically are unsure about whether their Event-B models are over/under-specified. In this short paper, we present a work-in-progress specification language called Cyclone to tackle this challenge. We demonstrate how one can use Cyclone to check an Event-B hybrid model using a car controller example. Our demonstration shows that Cyclone has a great potential to be used to verify Event-B hybrid models.

1 Introduction

Event-B is a widely used specification language that allows users model a system design using set theory [1]. Its platform Rodin has many effective features for stepwise refinement and mathematical proofs [2]. This makes Event-B a quite popular specification language. Recently, there is a trend of using Event-B to model hybrid systems [3–7]. However, the resulting Event-B models are typically very complex and difficult for users to perform analysis or understand. This imposes three immediate challenges on using Event-B: 1) How can users check whether a proposed predicate is a correct invariant for their Event-B models. 2) How can users ensure their design is not under/over specified. 3) How can users identify non-determinism in their models to ensure correct code generation.

In this paper, we present a work-in-progress specification language called Cyclone to tackle these challenges. Cyclone provides users a unique way for describing a complex system using graph-based notations. It allows users to explicitly construct a graph and specify two kinds of properties: graph and computation. The graph-based properties specify a particular set of graph patterns that a path (to be found in a graph) must obey. For example, whether a graph contains non-determinism transitions, Hamiltonian cycle or Euler paths. The computation properties specify a set of computational instructions (e.g. invariants, assertions, conditional transitions) that must be satisfied. For example, finding a path (in a graph) that can make two variables $x \geq 0 \wedge y \leq 0$. By combing both graph-based and computational properties, Cyclone is able to perform powerful checks and analysis for complex models.

U. Glässer et al. (Eds.): ABZ 2023, LNCS 14010, pp. 179–184, 2023.
https://doi.org/10.1007/978-3-031-33163-3_13

2 Current Architecture

Cyclone is mainly written in Java and consists of more than 100k+ lines of code including building scripts, web interface, IDE plug-ins, test cases and configurations. Currently, Cyclone can be compiled on the command-line and can be run on Windows, Linux and MacOS. The current architecture of Cyclone is shown in Fig. 1. The front-end of Cyclone is responsible for parsing, semantic and type checking. The back-end uses a new bounded verification algorithm to generate a set of verification conditions. These conditions can be efficiently solved by an SMT solver[1]. To prove user-defined properties, Cyclone typically either produces a trace if properties can be satisfied or a counter-example to show that properties cannot be satisfied. A trace or counter-example records how system states change within the specified bound.

One can have access to Cyclone using one of the following ways:

- Download link: https://classicwuhao.github.io/cyclone_tutorial/installation.html
- Online playground: https://cyclone4web.cs.nuim.ie/editor/

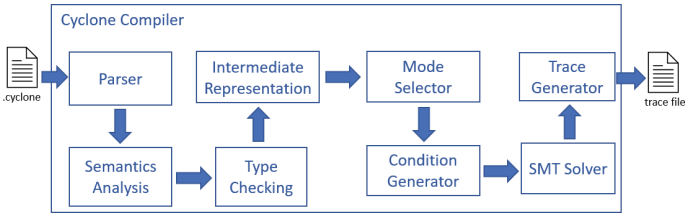

Fig. 1. Architecture of Cyclone.

3 An Illustrative Example

In this section, we use a car controller example to illustrate how one can use Cyclone to check a proposed invariant for an Event-B model [8]. This example models a car controller that must decide when to decelerate the car so it can stop at or near (before) a stop sign at position S.

This controller uses two variables p and v to track a car's position and velocity, respectively. The car face towards the stop sign and its continuous dynamical system is captured by the differential equation $\dot{p} = v, \dot{v} = u$. The controller may change its velocity every δ second by (de)accelerating. To keep this example simple, this is determined by 3 actuation commands: (1) accelerate the car with a rate of A (2) maintain the current velocity by setting acceleration to 0 (3) deaccelerate the car by braking with a rate of $-B$. The safety property is defined as $\forall t \cdot t \in [0, now] \Rightarrow p(t) \leq S \wedge v(t) \geq 0$. This means up until now that the car position should always satisfy $p \leq S \wedge v \geq 0$.

[1] We use Z3 as Cyclone's default solver.

We model this controller using Event-B and the part of our model is shown in Listing 1.1. This model implements a closed-loop design and has two types of events: controller and system. Each controller event decides an actuation command based on different conditions over the system states. The system event (*Progression* in Line 14) specifies how the system behaves (given the actuation command) and for how long. Our Event-B model has a total of three controller events and one system event. The three controller events are: *Acceleration*,*Brake* and *Maintain*. Due to page limitation, we only show *Acceleration* in Listing 1.1 (Line 5–12). This event specifies that it is safe to accelerate the car with a rate of A (Line 10[2] if the current position plus the braking distance of the car is less than position S of the stop sign (Line 7). When the controller events are terminated, the system event *Progression* (Line 14–22) starts. This event updates the position and velocity of the car at time $t+\delta$. When the system event terminates, the controller events start again to act for the next cycle.

```
1   Machine car_controller
2   Variables p v t s u
3   ...
4   Events ...
5       Event Acceleration ≙
6       Where ...
7           grd1 : p_A(t + δ) + v_A(t+δ)^2 / 2B ≤ S
8           ...
9       Then ...
10          act1 : u := A
11          ...
12      End
13      ...
14      Event Progression ≙
15      Where ...
16      ...
17      Then
18          act2 : p := p ⩤ ((t, t + δ] ◁ p_A)
19          act3 : v := v ⩤ ((t, t + δ] ◁ v_A)
20          act4 : t := t + δ
21          ...
22      End
23  End
```

Listing 1.1. The part of the Event-B model for the car controller.The complete Event-B specification is available at: https://classicwuhao.github.io/event_b_spec.pdf

Here, we are interested in checking whether our Event-B model (is initialized at a safe state) could reach to an unsafe state (the safety property does not hold). To do this, we first propose an invariant for our Event-B model. We then use Rodin to generate proof obligations for the invariant. However, proving generated proof obligations of an invariant is challenging and time consuming. Hence, to tackle this challenging task, we take advantage of Cyclone for automated reasoning. We translate our Event-B model into a Cyclone specification and ask

[2] p_u, v_u are the analytical solutions of the differential equations $\dot{p} = v, \dot{v} = u$, where $p_u(t') = p(t) + v(t)(t'-t) + \frac{1}{2}u(t'-t)^2$, and $v_u(t') = v(t) + u(t'-t)$.

Cyclone to certify whether our proposed invariant holds. For our car controller, the proposed invariant ϕ is defined as: $\forall e \cdot e \in [0,t] \Rightarrow p(e) + \frac{v(e)^2}{2B} \leq S \wedge v(e) \geq 0$.

Currently, the translation from an Event-B model to a Cyclone specification is done manually. The aim here is to build a transition system using Cyclone's graph notations. Listing 1.2 shows the translated Cyclone specification from our Event-B model in Listing 1.1. We first map each controller event to a computational node in Cyclone. A computational node (with modifier normal) indicates that the defined instructions inside the node get executed when this node is visited. For example, the *Acceleration* event in Listing 1.1 (Line 30) is mapped to the computational node *Acceleration* in Cyclone. This node contains instructions act_1 (Line 10 in Listing 1.1) which indicates the acceleration of the car is now assigned to A. We also introduce two additional empty nodes: *Init* and *Decide*. We use *Init* node to specify the initial state of the system and *Decide* to indicate the controller makes a decision on which actuation command to be issued.

```
24 option-trace=true; //produce a trace
25 machine car_controller {
26     real p,v,t,u;
27
28     normal start node Init{} //start of the transition system
29     normal node Decide{}
30     normal node Acceleration {act1;}
31     /* end of each decision cycle. */
32     normal final node Progression {act2;act3;act4;}
33     normal node Brake {u=-B;}
34     normal node Maintain {u=0;}
35
36     edge {Init → Decide}
37     edge {Decide → Acceleration where grd1;}
38     edge {Decide → Brake where ... ;}
39     edge {Decide → Maintain where ... ;}
40     edge {Acceleration → Progression}
41     edge {Brake → Progression}
42     edge {Maintain → Progression}
43     edge {Progression → Decide}
44
45     invariant SysInv { p + v^2/2B ≤ S ∧ v ≥ 0; }
46
47     goal{
48       assert (A > 0 ∧ B > 0 ∧ p ≥ 0 ∧ t = 0 ∧
49       p + v^2/2B ≤ S ∧ S ≥ 0 ∧ dt > 0) in (Init);
50
51       check for 3
52     }
53 }
```

Listing 1.2. The Cyclone specification for the Event-B model in Listing 1.1. Here $act_1 \ldots act_4$ and grd_1 are the same as those in Listing1.1.The complete Cyclone specification is available at: https://classicwuhao.github.io/car_abz.cyclone

Next, we build a set of edges (transitions) for our nodes. The guard of an event from our Event-B model is translated to a conditional edge (transition) in Cyclone. For example, the grd_1 in Listing 1.1 (Line 7) is directly mapped

to the conditional edge in Listing 1.2 (Line 37). This means that the transition $Decision \rightarrow Acceleration$ can only happen when the grd_1 is satisfied and this means the controller decides to issue actuation command: acceleration.

We map our proposed invariant ϕ to the invariant (Line 45) in Cyclone. The semantics behind this is that the invariant must hold after each transition. Finally, we need to ensure the controller starts at a safe initial state by setting appropriate conditions in Line 49. Now, we have established a transition graph for the car controller modelled in Event-B. Hence, we can check whether there exits a path to break our proposed invariant (Line 51). We check all transitions that has exact length of 3. This is because each (decision) cycle has a length of 3[3]. For example, a cycle $Init \rightarrow Decide \rightarrow Maintain \rightarrow Progression$ has a length of 3 including node $Init$[4]. In this case, one cycle is enough for Cyclone to discover a counter-example (trace). Figure 2 shows this trace (returned from Cyclone) and it depicts that the controller enters an unsafe state after issuing actuation command:brake. In the real world, after a car brakes and its velocity cannot reach below 0. It is not possible to drive a car backward by braking. Hence, this counter-example shows that our Event-B model for this car controller is under-specified.

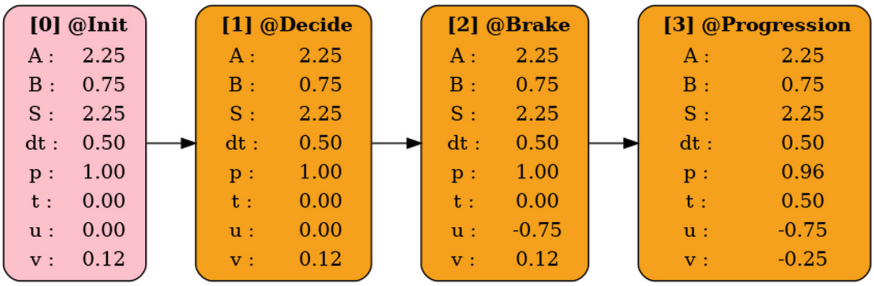

Fig. 2. A trace (a path length=3) generated by Cyclone shows that our proposed invariant does not hold for the car controller. To keep it simple, we set Cyclone to round off to 2 decimal places for each variable.

4 Experience Gained

In this short demo, we have gained two valuable experience. (1) Using Event-B to model hybrid systems is challenging and tools are needed for discharging generated proof obligations, in particular an invariant of a system. (2) Simulating a system with the correct and meaningful values is very useful in helping verification of a hybrid system. However, finding such values is not easy. We successfully applied our new specification language Cyclone on this car controller by demonstrating finding a counter example that breaks an invariant. However, finding

[3] One can check multiple cycles by setting a larger upper bound or multiple bounds.
[4] The length of a path is decided by the number of nodes.

or synthesising correct invariants from an Event-B model remains untackled. It would be ideal to add a new component to the existing Event-B platform to automatically infer an invariant.

5 Future Direction

By now, we have used Cyclone on a few hybrid systems that are modelled using Event-B including a water tank model [3]. Further, we have also collected and designed about 220 sample/test cases from different areas such as program verification, graph searching and model checking for evaluating Cyclone. Cyclone shows a great potential in performance and usability[5] in handling these problems.

For the next milestones, (1) we are now investigating a technique that can automatically translate an Event-B model to a Cyclone specification based on a set of well-defined transformation rules. This technique would allow us to use Cyclone as an oracle to automatically discover an invariant of an Event-B model. (2) we are developing new modules and algorithms for Cyclone so that they can also be used for reasoning non-linear systems in an efficient manner.

Acknowledgments. We thank Dominique Méry and the anonymous ABZ reviewers for their helpful feedback on the paper. This work is supported by the Irish Research Council and the Embassy of France in Ireland under the ULYSSES program, and by the Agence Nationale de la Recherche under the grant ANR-17-CE25-0005.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Su, W., Abrial, J.R., Zhu, H.: Formalizing hybrid systems with Event-B and the Rodin platform. Sci. Comput. Program. **94** (2014)
4. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core hybrid Event-B I: single hybrid Event-B machines. Sci. Comput. Program. **105** (2015)
5. Dupont, G., Ait-Ameur, Y., Singh, N.K., Pantel, M.: Formally verified architectural patterns of hybrid systems using proof and refinement with Event-B. Sci. Comput. Program. **216** (2022)
6. Cheng, Z., Méry, D.: A refinement strategy for hybrid system design with safety constraints. In: Attiogbé, C., Ben Yahia, S. (eds.) MEDI 2021. LNCS, vol. 12732, pp. 3–17. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78428-7_1
7. Mammar, A., Afendi, M., Laleau, R.: Modeling and proving hybrid programs with Event-B: an approach by generalization and instantiation. Sci. Comput. Program. (2022)
8. Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A.: How to model and prove hybrid systems with KeYmaera: a tutorial on safety. Int. J. Softw. Tools Technol. Transfer **18**(1) (2016)

[5] Cyclone is now a part of course at Maynooth University and used by 100+ students.