A Composition Mechanism for Refinement-Based Methods

Thai Son Hoang, Dana Dghaym, Colin Snook, Michael Butler
Electronics and Computer Science
University of Southampton, Southampton, UK
Email: {t.s.hoang, dd4g12, cfs, mjb}@ecs.soton.ac.uk

***Abstract*—Event-B developments are mostly structured around the refinement relationship. This top-down development architecture enables system details to be gradually introduced into the formal model. However, this results in large models with monolithic structures. We develop a composition mechanism allowing to develop models bottom-up. In particular, our proposed mechanism works seamlessly with the existing refinement technique in Event-B. As a result we have built a formal development method that can take advantage of both top-down and bottom-up approaches. We prove the correctness of machine inclusion with refinement using the supporting Rodin platform.**

***Keywords*—Machine Inclusion, Composition, Refinement, Event-B**

I. Introduction

Developing dependable large software systems is a challeging task and formal methods are being seen as one of the solutions for improving system quality. Event-B [1] is a formal method for system development based on discrete-transition systems (called machines). In particular, to cope with system complexity, Event-B developments are mostly structured around *refinement* and *decomposition* relationships [2]. Refinement enables system details to be gradually introduced into the formal models consistently. Decomposition allows a model to be separated in several (smaller) sub-models which can be further refined independently. A disadvantage of this top-down development style is that this often results in large models with monolitic structures.

Our motivation is to incorporate the bottom-up development approach by reusing existing models. Our aim is the seamless integration of composition technique with the current existing top-down development process. This will facilitate the reuse of existing models, result in the development of models in separate work streams, supporting teamwork development. While various composition techniques have been proposed [3]–[6], most of them rely on translation and are not smoothly integrated with the refinement development process. (More information regarding related work can be found in Section VII).

In this paper, we introduce the notion of *machine inclusion* into Event-B. This allows to construct an Event-B machine by composing one or more machines. The included machine is reused in a "correct-by-construction" fashion that allows us to utilise its properties without reproving them. Furthermore, we illustrate that the new mechanism can be used together with refinement-based development process with minimal effort. Our approach enables the possibility of a top-down and bottom-up combined development process.

The rest of the paper is organised as follows. Section II presents some background on the Event-B modelling method including its proof obligations. In Section III, we describe our proposal for machine inclusion mechanism and its integration with the refinement process. Section IV illustrates the usage of the machine inclusion mechanism to develop a system for controlling cars on a bridge. We discuss our implementation supporting machine inclusion in Section V. We prove the correctness of the proposed proof obligations for machine inclusion in Section VI. Finally, we draw some conclusions, discuss the related work and future research direction in Section VII.

II. Background

Event-B [1] is a formal method for system development. Main features of Event-B include the use of *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* that constrain the carrier sets and constants. A machine `M` contains *variables* `v`, *invariants* `I(v)` that constrain the variables, and *events*. An event comprises a guard denoting its enabling-condition and an action describing how the variables are modified when the event is executed. In general, an event `e` has the following form, where `t` are the event parameters, `G(t, v)` is the guard of the event, and `v := E(t, v)` is the action of the event[1].

```
e == any t where G(t,v) then v := E(t,v) end
```

A special unguarded event (called the `INITIALISATION`) is used for initialising the variables. In this paper, we do not present a separate treatment for the `INITIALISATION`, it is a special case of *normal* events.

An Event-B machine `M` corresponds to a transition system where *variables* `v` represent the states and `M`'s *events* specify the transitions. Event-B defines proof obligations, which must be discharged to ensure that the formal model fulfills its specified properties. These obligations are expressed in terms of *sequents* of the form $H \vdash G$ meaning that the *goal* G

[1] Actions in Event-B are, in the most general cases, non-deterministic [7].

holds under the set of *hypotheses* H. For example, the proof obligation for invariant `I(v)` to be preserved by event `e` is as follows,

$$G(t, v), I(v) \vdash I(E(t, v)) \ . \qquad \text{(INV)}$$

I.e., under the assumption that the guard `G(t,v)` and the invariant `I(v)` hold, the (modified) invariant `I(E(t,v))` is re-established. For convenience, we say that machine `M` is *consistent* if all of its events maintain its invariants.

Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. Machine `M` can be *refined* by machine `N` (we call `M` the abstract machine and `N` the concrete machine). The state of `M` and `N` are related by a gluing invariant `J(v, w)` where `v`, `w` are variables of `M` and `N`, respectively. Intuitively, any "behaviour" exhibited by `N` can be simulated by `M`, with respect to the gluing invariant `J(v,w)`. Refinement in Event-B is reasoned event-wise. Consider an abstract event `e` and the corresponding concrete event `f`. Somewhat simplifying, we say that `e` is refined by `f` if `f`'s guard is stronger than that of `e` and `f`'s action is simulated by `e`'s action, taking into account the gluing invariant `J`. More precisely, given event `e` above and event `f` as follows[2].

```
f == any t where H(t,w) then w := F(t,w) end
```

Event `f` is a refinement of `e` with respect to the gluing invariant `J(v,w)` if the following proof obligations hold.

$$I(v), J(v, w), H(t, w) \vdash G(t, v) \qquad \text{(GRD)}$$

$$I(v), J(v, w), H(t, w) \vdash J(E(t, v), F(t, w)) \qquad \text{(INV\_REF)}$$

The *guard strengthening* obligation (GRD) states that the concrete guard `H(t,w)` is stronger than the abstract guard `G(t,v)`. The *(refinement) invariant preservation* (INV_REF) states that the invariant `J` is maintained by the abstract event `e` and the concrete event `f`. For convenience, we say that concrete machine `N` is *consistent* if all its events satisfy (GRD) and (INV_REF) proof obligations.

More information about Event-B can be found in [7]. Event-B is supported by the *Rodin platform* (Rodin) [8], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

III. A Composition Mechanism for Refinement-based Development

In this section, we first present the machine inclusion mechanism in Section III-A, then consider the relationship between machine inclusion and refinement in Section III-B.

A. Machine Inclusion

We propose the *machine inclusion* mechanism for Event-B as follows. Consider the following machine `B0` with variables `v`, invariants `I0(v)` and event `e`.

[2]In general the event's parameters can also be refined.

```
machine B0
variables v
invariants
  @I0: "I0(v)"
events
  e
  any t where
    @grd1: "G0(t, v)"
  then
    @act1: "v := E0(t, v)"
  end
end
```

A machine `A0` that includes machine `B0` will inherit `B0`'s variables `v` and invariants `I0(v)`. Machine `A0` can have its own variables `x`. As a result, invariant `J0` of `A0` can refer to both `v` and `x`, i.e., `J0(v,x)`. Machine `A0` cannot directly assign to `v`. In order to modify `v`, events of `A0`, such as `f`, have to *synchronise* with events of the included machine `B0`, e.g., `e`.

```
machine A0
includes B0
variables x
invariants
  @J0: "J0(v,x)"
events
  f
  synchronises e
  any u where
    @grd2: "H0(u, x)"
    @grd3: "K0(t, v, u, x)"
  then
    @act2: "x := F0(u, x)"
  end
end
```

Guards of event `f` can refer to parameter `u` and variable `x` declared explicitly in `A0`, e.g., `H0(u,x)`. Moreover, via event synchronisation, a guard of `f` can also refer additionally to parameter `t` and variable `v` of the included machine `B0`, e.g., `K0(t,v,u,x)`. Essentially, the guard `K0` act as an explicit synchronisation link between the including and included machines.

The semantics of machine inclusion and event synchronisation are captured by the *flattened* machine `(flattened)A0` as follows. Variables `v` and invariants `I0(v)` explicitly become variables and invariants of `(flattened)A0`.

```
machine (flattened)A0
variables v x
invariants
  @I0: "I0(v)"
  @J0: "J0(v,x)"
events
  (flattened)f
  any t u where
    @grd1: "G0(t, v)"
    @grd2: "H0(u, x)"
    @grd3: "K0(t, v, u, x)"
  then
    @act1: "v := E0(t, v)"
    @act2: "x := F0(u, x)"
  end
end
```

Since the meaning of A0 is essentially represented by machine (flattened)A0, consistency of A0 is the same as that of (flattened)A0. In particular, since A0 includes B0, reasoning about the consistency of A0 can be separated accordingly, as illustrated by the following theorem.

Theorem 1 (Inclusion Invariant Preservation)**.** *Given machine B0 and A0 where A0 includes B0 as above, if machine B0 is consistent and the following proof obligation holds for every event f of A0*

$$\begin{array}{l} \quad I0(v), J0(v,x), G0(t,v), H0(u,x), K0(t,v,u,x) \\ \vdash \\ \quad J0(E0(t,v), F0(u,x)) \end{array} \tag{INC\_INV}$$

then A0 is also consistent.

Proof (Sketch). The fact that every event maintains (implicit) invariant I0(v) is guaranteed by consistency of B0 and event synchronisation. Proof obligation (INC_INV) guarantees that invariant J0(v,x) is maintained by all events. As a result, A0 is consistent. □

Theorem 1 allows us to reuse the consistency of the included machine B0 (without reproving) to reason about the consistency of the including machine A0.

Multiple instances of the same machine can be included using prefixing. For example, the following syntax allows machine A0 to include two instances of B0: one with prefix First, one with prefix Second.

```
machine A0
includes B0 as First Second
```

Events, variables, and parameters of the included machine are prefixed accordingly. For example, we use First.e to refer to event e of the First instance of B0, and First_v, First_t to refer to the corresponding variables and parameters of the same instance of B0. We chose this syntax (with keyword **as**) since we consider the option of using underscore, e.g., "First_B0", is ambigous, given that name of the machine can also contain underscores. An alternative is to use the dot notation, e.g., "First.B0". However, this notation is used for standard "qualified name" which we reserve for refering to machines from a different project [9].

B. Machine Inclusion and Refinement

Consider the refinement B1 of B0 with the gluing invariants I1(v,w) linking the abstract variables v and concrete variables w.

```
machine B1
refines B0
variables w
invariants
  @I1: "I1(v, w)"
events
  e
  refines e
  any t where
    @grd1: "G1(t, w)"
  then
    @act1: "w := E1(t, w)"
  end
end
```

Consider the machine A1 which includes B1 and refines A0 as follows.

```
machine A1
includes B1
refines A0
variables
  x
events
  f
  synchronises e
  refines f
  any u where
    @grd2: "H0(u, x)"
    @grd3: "K1(t, w, u, x)"
  then
    @act2: "x := F0(u, x)"
  end
end
```

Here, we assume that the variables x from A0 are retained in A1. We also consider the situation where minimal changes need to be made in A1 to include B1. Comparing the abstract event f in A0 and its corresponding event in A1, the only necessary change is that the guard K0(t,v,u,x) is replaced by K1(t,w,u,x). This is due to the data-refinement of v by w in B1. Here, we avoid data refinement of A1 in order to focus on the relationship between machine inclusion and refinement. In general, it is possible to data-refine x at the same time.

Consistency of A1 can rely on the consistency of B1 as stated in the following theorem.

Theorem 2 (Inclusion Guard Strengthening)**.** *Given machine B1 (refining B0) and machine A1 (including B1) as above, if B1 is consistent and the following proof obligation holds for all events f of A1*

$$\begin{array}{l} \quad I0(v), J0(v,x), I1(v,w), H0(u,x), K1(t,w,u,x), G1(t,w) \\ \vdash \\ \quad K0(t,v,u,x) \end{array} \tag{INC\_GRD}$$

then A1 is also consistent.

Proof (Sketch). Comparing the abstract event f in A0 and the concrete event f in A1, the action assigning to x and guard H0 are retained. As a result, we only need to consider guard strengthening for abstract guard K0, which is guaranteed by proof obligation (INC_GRD). □

More often B1 is a *superposition* refinement of B0, i.e. variables v are subset of variables w. In this case, K1 can be the same as K0 and the proof obligation (INC_GRD) becomes trivial. This is applicable in our example in Section IV.

Later, in Section VI, we prove the correctness of Theorems 1 and 2 using Rodin. Essentially, these theorems define

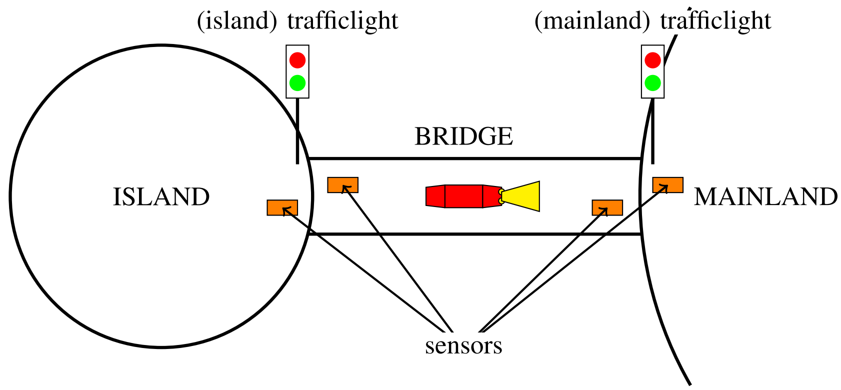

Fig. 1. Cars on a bridge

proof obligations associated with machine having the inclusion clause.

IV. Example. Controlling Cars on a Bridge

In this section, we illustrate the machine inclusion mechanism and refinement using the "cars on a bridge" example from [1, Chapter 2]. We first present the description of the example. Our formal models are available online at http://users.ecs.soton.ac.uk/tsh2n14/developments/ICECCS2017/.

A. Description

The system is controlling cars on a one-way bridge connecting the mainland to an island. The overall system can be seen in Fig. 1. The system is equipped with two traffic lights at both entrances to the bridge. At any time, the number of cars on the island is limited. In order to track the number of cars on the island, the system is equipped with four sensors that detect cars entering and leaving the bridge at both ends. The following set of requirements are extracted from [1, Chapter 2].

REQ 1 The bridge is one-way.

REQ 2 The system is equipped with two traffic lights at both entrances of the bridge.

REQ 3 A traffic light is either green or red.

REQ 4 Cars are not supposed to pass on a red traffic light.

REQ 5 The system is equipped with four sensors at both entraces detecting cars entering and leaving the bridge.

REQ 6 A sensor is either on or off.

REQ 7 An "on" sensor means that a car is willing to enter or leaving the bridge.

REQ 8 The number of cars on the island is limited.

In the following, we present the model of the sensors (Section IV-B), the model of the traffic lights (Section IV-C), and finally the model of the system using the machine inclusion mechanism (Section IV-D).

B. Model of A Sensor

Sensors are devices capable of detecting the presence of cars. Our model of a sensor makes a clear separation between physical equipment and the software controller associated with the sensor. The model of the sensor is developed using the following refinement strategy.

- Sensor_m0: Model of the physical sensor.
- Sensor_m1: Counting the number of (physical) cars departed from the sensor.
- Sensor_m2: Model of the signals from the sensor to the controller.
- Sensor_m3: Model of the sensor controller.

1) Sensor_m0. We model the state of the sensor using a Boolean variable SNSR, i.e., TRUE for "on" and FALSE for "off" (REQ 6). Two events SNSR_on and SNSR_off model the the situation when the sensor is going to "on" or "off" respectively.
2) Sensor_m1. A variable DEP (a natural number) is introduced to count the number of departed cars from the sensor. An action is added to SNSR_off to increase DEP accordingly.
3) Sensor_m2. Two new variables Snsr_01 and Snsr_10 are introduced into the model to represent the signals from the sensor to the controller when the sensor changes from "off" to "on" and from "on" to "off" respectively. The new invariants are as follows.

```
@inv1_1: "Snsr_01 = TRUE ⇒ SNSR = TRUE"
@inv1_2: "Snsr_10 = TRUE ⇒ SNSR = FALSE"
@inv1_3: "Snsr_01 = FALSE ∨ Snsr_10 = FALSE
    "
```

Invariants inv1_1 and inv1_2 link the signal to the actual state of the sensor. Invariant inv1_3 state that at most one signal can be active at the one point. The original events SNSR_on and SNSR_off are *extended* with additional guards and actions accordingly. Two new events ctrl_Senses_Snsr_01 and ctrl_Senses_Snsr_10 are introduced to model the action of the controller receiving the signals.

4) Sensor_m3. In this refinement, we introduce the variables of the controller, i.e., ctrl_snsr, ctrl_dep, ctrl_snsr_01, ctrl_snsr_10, corresponding to the sensor status, the number of cars departed, and the signals' status as stored by the controller. Note that they are the controller's version of the physical entities and do not always correspond exactly to the physical version. For example, the invariants relating ctrl_dep and DEP are as follows.

```
@inv2_3: "Snsr_10 = FALSE ∧ ctrl_snsr_10 =
    FALSE ⇒ ctrl_dep = DEP"
@inv2_4: "Snsr_10 = TRUE ∨ ctrl_snsr_10 =
    TRUE ⇒ ctrl_dep = DEP − 1"
```

The invariants state that ctrl_dep and DEP are the same only if there are no pending signals indicating that the sensor is going from "on" to "off" to process. Two new events ctrl_on and ctrl_off are introduced in this refinement for the controller to process the signals accordingly.

C. Model of A Traffic Light

Similar to the model of the sensor, we also separate the physical traffic light and the software controller. The refinement strategy for modelling a traffic light is as follows.

- TrafficLight_m0: Model of the physical traffic light.
- TrafficLight_m1: Model the actuator from the controller to the traffic light.
- TrafficLight_m2: Model of the traffic light controller.
- TrafficLight_m3: Model of the sensor from the traffic light to the controller.

This refinement strategy for a control system follows the guideline provided in [10].

1) TrafficLight_m0. This first model of the traffic light contains a variable LIGHT which is either RED or GREEN (REQ 3). Two events GREEN_2_RED and RED_2_GREEN change the status of the traffic light from GREEN to RED and RED to GREEN, respectively.
2) TrafficLight_m1. In this refinement, we introduce the actuators, namely, Act_RED and Act_GREEN, commanding the traffic light to RED or GREEN, respectively. The original events GREEN_2_RED and RED_2_GREEN are refined using the actuators information. Two new events, namely, ctrl_Acts_RED and ctrl_Acts_GREEN are added to set the value of the actuator
3) TrafficLight_m2. In this refinement, we introduce the controller side of the traffic light. This includes variables ctrl_light to keep controller status of the light (which might be different from the actual status of the light, i.e., LIGHT). Another (Boolean) variable, namely ctrl_act, is introduced to indicate that the controller needs to send a command to change the traffic light status.
4) In this model, we complete the control-loop for the traffic light with the sensors Snsr_RED and Snsr_GREEN. They are set when the physical traffic light changes status, i.e., in events GREEN_2_RED and RED_2_GREEN accordingly. Two new events ctrl_Senses_RED and ctrl_Senses_GREEN are introduced to model the controller processing these sensors.

D. Controlling Cars System Model using Machine Inclusion

Our refinement strategy for developing the system for controlling cars on a bridge is as follows.

- Car_m0: Model the cars on the bridge and on the island.
- Car_m1: Introduce the 2 physical sensors for detecting cars entering the bridge (from both ends) by *including two instances of* Sensor_m0.
- Car_m2: Introduce the 2 physical traffic lights by *including two instances of* TrafficLight_m0.
- Car_m3: Refine the number of cars on the bridges using the 4 physical sensors by *including four instances of* Sensor_m1.
- Car_m4: Introduce the controller for the 4 sensors by *including four instances of* Sensor_m3.
- Car_m5: Introduce the number of cars on the bridge and on the island as kept by the controller. This is linked with the the 4 sensors controller introduced previously.
- Car_m6: Introduce the traffic light controller by *including two instances of* TrafficLight_m3.
- Car_m7: Refine the controller processes for the two traffic lights properly.

The refinement and inclusion relationships (including multiplicity) between the different machines can be seen in Fig. 2. In the following, we present some important modelling aspects, focusing on the use of the machine inclusion mechanism.

1) Car_m0 – Cars on the Bridge and the Island: This machine is the same as the one presented in [1, Chapter 2]. We have three variables A, B, and C representing the (actual) number of cars on the bridge (going into the island), the number of cars on the island, and the number of cars on the bridge (going into the mainland). Important invariants are as follows.

```
@inv0_4: "A = 0 ∨ C = 0"
@inv0_5: "A + B + C ≤ D"
@thm0_1: "B ≤ D" theorem
```

They are stating that the bridge is one-way (REQ 1) and that the number of cars on the island is limited (REQ 8). There are 4 events, namely, ML_in, ML_out, IL_in, IL_out, to model the situation where a car is entering/leaving the mainland (ML) or the island (IL) respectively. For example, the event related to the island is as follows.

```
IL_in
when
  @grd1: "A ≠ 0"
```

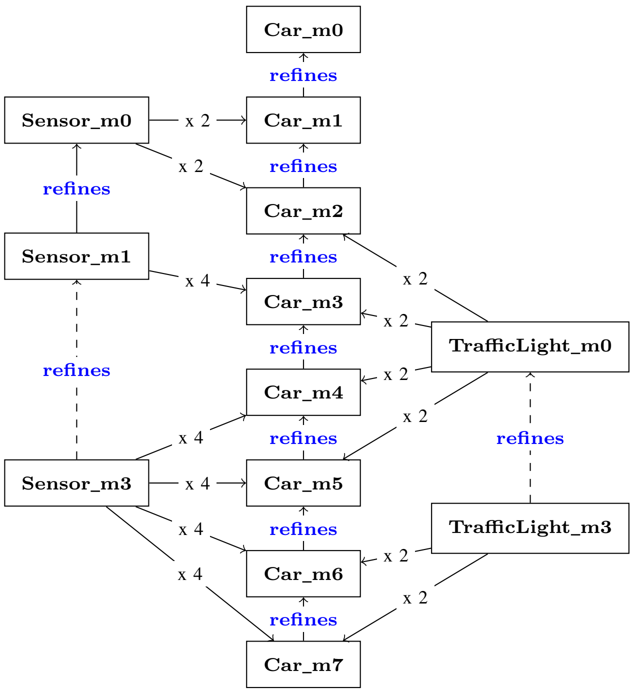

Fig. 2. Development of the controller for cars on a bridge

```
  then
    @act1: "A := A − 1"
    @act2: "B := B + 1"
  end

  IL_out
  when
    @grd1: "B ≠ 0"
    @grd2: "A = 0"
  then
    @act1: "B := B − 1"
    @act2: "C := C + 1"
  end
```

2) `Car_m1` – Sensors for Cars Entering the Bridge: In this model, we introduce the sensors detecting cars entering the bridge on both ends by including (twice) `Sensor_m0`.

```
machine Car_m1
includes Sensor_m0 as ML_out IL_out
invariants
  @inv1_1: "IL_out_SNSR = TRUE ⇒ B ≠ 0"
```

Invariant `inv1_1` links the status of the `IL_out` sensor with the number of cars on the island: if there is a car willing to leave the island, then the number of car on the island must not be 0. Event `IL_out` is refined by event synchronisation as follows.

```
  IL_out
  synchronises IL_out.SNSR_off
  refines IL_out
  when
    @grd2: "A = 0"
  then
    @act1: "B := B − 1"
    @act2: "C := C + 1"
  end
```

Note that the abstract `grd1` of `IL_out` event is removed as a consequence of invariant `inv1_1`. The meaning of the event synchronisation is that when a car leaves the island, the sensor `IL_out` is going "off". Here, consider the consistency of `Car_m1`, we can apply Theorem 1, i.e. to prove that `inv1_1` is maintained by all events of `Car_m1`. The proofs are trivial and are omitted here.

A new event `IL_out_ARR` models the situation where a car arrives on the `IL_out` sensor. As a result, this event synchronises with `IL_out.SNSR_on`. The guard `grd2` for this event is to ensure that invariant `inv1_1` is maintained.

```
  IL_out_ARR
  synchronises IL_out.SNSR_on
  when
    @grd2: "B ≠ 0"
  end
```

3) `Car_m2` – Traffic lights: In this machine, we introduce the traffic lights by including two instances of `TrafficLight_m0` (REQ 2). Note that this machine still includes two instances of `Sensor_m0` as before. From now on, for simplification, we only show the machine inclusion clauses when they are changed.

```
machine Car_m2
includes TrafficLight_m0 as ML IL
includes Sensor_m0 as ML_out IL_out
invariants
  @inv2_3: "IL_LIGHT = GREEN ⇒ A = 0"
  @inv2_4: "ML_LIGHT = RED ∨ IL_LIGHT = RED"
```

Important invariants in this machine include `inv2_3` stating that if the island traffic light (`IL_LIGHT`) is `GREEN` then there are no cars on the bridge going into the island, and `inv2_4` stating that at most one of the traffic lights is `GREEN` at any time.

Event `IL_out` is refined as follows to take into account REQ 4. Notice that it still synchronises with the `SNSR_off` event from the `IL_out` machine as before.

```
  IL_out
  synchronises IL_out.SNSR_off
  refines IL_out
  when
    @grd2: "IL_LIGHT = GREEN"
  then
    @act1: "B := B − 1"
    @act2: "C := C + 1"
  end
```

Events changing the traffic lights are added as new events in this machine. For example, events for changing the island traffic light are as follows.

```
  IL_light_GREEN
  synchronises IL.RED_2_GREEN
  when
    @grd2: "ML_LIGHT = RED"
    @grd3: "A = 0"
```

```
end

IL_light_RED
synchronises IL.GREEN_2_RED
end
```

4) `Car_m3` – Counting the Cars on the Bridge: In this machine, we perform data-refinement of the number of cars, i.e., `A`, `B`, `C` using the sensors. For this, we include `Sensor_m1` four times representing the four sensors (REQ 5, REQ 7). The gluing invariants for removing variables `A`, `B`, and `C` are as follows.

```
machine Car_m3
includes Sensor_m1 as ML_out IL_out ML_in IL_in
invariants
  @inv3_1: "A = ML_out_DEP − IL_in_DEP"
  @inv3_2: "B = IL_in_DEP − IL_out_DEP"
  @inv3_3: "C = IL_out_DEP − ML_in_DEP"
```

The invariants link the number of cars on the bridge and the island with the number of cars departed from different sensors. For example, invariant `inv3_1` states that the number of cars on the bridge going into the island is the difference between the number of cars departed the `ML_out` sensor (i.e., going out of the mainland) and the number of cars departed the `IL_in` sensor (i.e., going into the island).

References to `A`, `B`, `C` in guards and actions are removed and replaced accordingly. For instance, refinements of `IL_out` and `IL_light_GREEN` are as follows.

```
IL_out
synchronises IL_out.SNSR_off
refines IL_out
when
  @grd2: "IL_LIGHT = GREEN"
end

IL_light_GREEN
synchronises IL.RED_2_GREEN
refines IL_light_GREEN
when
  @grd2: "ML_LIGHT = RED"
  @grd3: "IL_in_DEP = ML_out_DEP"
end
```

5) `Car_m4` – Sensors controller: In this machine, we incorporate the controller sensors into the model by replacing the inclusion (4 times) of `Sensor_m1` with `Sensor_m3`.

```
machine Car_m3
includes Sensor_m3 as ML_out IL_out ML_in IL_in
```

Since the abstract variables in `Sensor_m1` are retained in `Sensor_m3`, we do not need to refine the guards of any event. Moreover new events in `Sensor_m3` (compared to `Sensor_m1` are added as new events in this machine (using event synchronisation). For example, events for controller to process the `IL_out` sensor are as follows.

```
IL_out_ctrl_on
synchronises IL_out.ctrl_on
end

IL_out_ctrl_off
synchronises IL_out.ctrl_off
end
```

6) `Car_m5` – Sensors Controller: Given the introduction of the sensors controller in the previous machine `Car_m4`, we now start designing our controller part for counting the number of cars. Three new variables `car_a`, `car_c`, and `car_n` representing the number of cars on the bridge going to the island, the number of cars on the bridge going to the mainland, and the total number of cars on the bridge and the island, respectively (as calculated by the controller). The invariants related to the new variables are as follows.

```
@inv5_1: "ctrl_a = ML_out_ctrl_dep −
    IL_in_ctrl_dep"
@inv5_2: "ctrl_c = IL_out_ctrl_dep −
    ML_in_ctrl_dep"
@inv5_3: "ctrl_n = ML_out_ctrl_dep −
    IL_out_ctrl_dep"
```

The invariants show how the controller calculate the number of cars using the sensors. Events are **`extended`** accordingly. For example, event `IL_out_ctrl_off` is extended with actions changing `ctrl_c` and `ctrl_n` as follows.

```
IL_out_ctrl_off extended
refines IL_out_ctrl_off
begin
  @act1: "ctrl_c := ctrl_c + 1"
  @act2: "ctrl_n := ctrl_n − 1"
end
```

7) `Car_m6` – Traffic Light Controller: In this machine, we introduce the traffic light controller by replacing the inclusion (twice) of `TrafficLight_m0` with `TrafficLight_m3`.

```
machine Car_m3
includes TrafficLight_m3 as ML IL
```

Similar to Section IV-D5, new events in `TrafficLight_m3` are *promoted* accordingly. For example, events for controlling the `IL` traffic light are as follows.

```
IL_ctrl_RED_2_GREEN
synchronises IL.ctrl_RED_2_GREEN
end

IL_ctrl_GREEN_2_RED
synchronises IL.ctrl_GREEN_2_RED
end
```

8) `Car_m7` – Refine Traffic Light Controller: In this machine we refine the controlling of traffic lights using information from counting the number of cars. In particular, changing the traffic light from `RED` to `GREEN` requires attention as this could violate system safety. Consider the refinement of `IL_ctrl_RED_2_GREEN` below.

```
IL_ctrl_RED_2_GREEN
synchronises IL.ctrl_RED_2_GREEN
```

```
refines IL_ctrl_RED_2_GREEN
when
  @grd1: "ML_ctrl_snsr_RED = TRUE"
  @grd2: "ctrl_a = 0"
  @grd3: "ML_out_Snsr_10 = FALSE"
  @grd4: "ML_out_ctrl_snsr_10 = FALSE"
  @grd5: "IL_in_Snsr_10 = FALSE"
  @grd6: "IL_in_ctrl_snsr_10 = FALSE"
end
```

Guards `grd1` and `grd2` assert that, according to the controller, the `ML` traffic light is `RED` and there are no cars on the bridge going to the island. Guards `grd3`–`grd6` ensure that all the releveant signals have been processed accordingly by the controller. This is to guarantee that the controller has the correct up-to-date information about the `ML` traffic light and the number of cars on the bridge. We omit the presentation of the relevant invariants and refinement of other events here.

E. Summary

In order to estimate the effect of the inclusion mechanism, we compare the number of proof obligations for developments with and without machine inclusion, assuming that we follow the same refinement strategy (see Table I). In this example, all proof obligations are discharged automatically. As one can see, using machine inclusion, we reduce the number of proof obligations to about one third of this development. Taking into account the proof obligations for the model of the sensor (50 POs) and for the model of the traffic light (92 POs), we reduce the number of proof obligations by using machine inclusion by 234 POs (44% of the total number of POs without maching inclusion). Note that this number (234 POs) roughly corresponds to 3 times the POs for the sensor plus the POs for the traffic light, which is what we expected to save by using machine inclusion.

TABLE I
PROOF STATISTICS

Machine	With inclusion	Without inclusion
Car_m0	20	20
Car_m1	6	6
Car_m2	26	26
Car_m3	9	33
Car_m4	0	192
Car_m5	12	12
Car_m6	0	160
Car_m7	88	88
Total	161	537

V. IMPLEMENTATION

Implementation of the inclusion feature is based on our EMF framework for Event-B [11]. This framework has been developed in order to leverage the extensive range of *Eclipse Modeling Framework* (EMF) [12] utilities that are available from the Eclipse foundation. It also provides a framework for extending the Event-B language with additional features (e.g. inclusion). Extensions are translated into "pure" Event-B and therefore are not required to be processed by the Rodin tools. The framework provides the following features:

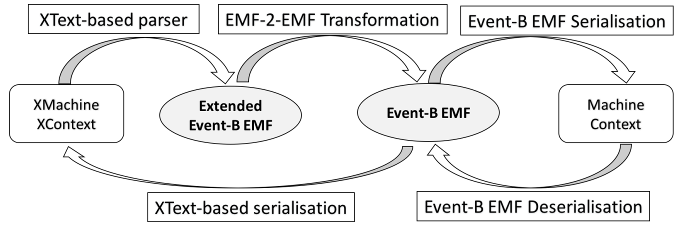

Fig. 3. XEvent-B Extended Architecture

- an EMF meta-model for Event-B,
- EMF Event-B model repository code (generated by the meta-model),
- extension mechanisms for extending the Event-B meta-model and model code,
- persistence (synchronisation) of EMF models and their extensions using the Rodin Database,
- a facility to extend the Rodin navigator with EMF-based extension elements,
- a generic (clone) contribution to the Rodin *refine* operation with provision to configure how references are handled,
- a generic translator facility to support the implementation of translations (either to "pure" Event-B or to other target languages).

The inclusion metamodel is an extension to the Event-B metamodel, where machines are allowed to include other machines with the possibility of prefixing, and events can synchronise with other events and can also apply prefixing. After extending the Event-B meta-model, we define a model-to-model transformation from the inclusion meta-model to the Event-B meta-model. This transformation will generate a flattened Event-B machine that can be serialised by Rodin.

We use an Xtext-based editor as a front-end for machine inclusion. Xtext [13] is an eclipse-based open source framework for the development of domain-specific languages. Using Xtext, we define the grammar of machine inclusion which also generates a parser, a serialiser and a smart editor. When an Xtext machine file is saved, the Xtext generator will call the inclusion translator, which in turn will generate the flattened Event-B machine. The flattened machine is a normal Event-B machine, hence the Event-B verification can be applied. Fig. 3 summarises the architecture of the Xtext-based Event-B (XEvent-B), and its relation to the Event-B EMF.

VI. CORRECTNESS

We have used Rodin [8] to prove the theorems in Section III related to machine inclusion. The approach that we use is from [14] (for proving consistency of Event-B extensions) containing the following steps.

1) Encode the generic input model.
2) Encode the generic output model.
3) Gather the consistency conditions of the input model.
4) Prove the consistency of the output model using the consistency of the input model.

We illustrate our verification for Theorem 1 as follows.

1) The generic input model is the machine B0 as shown in Section III-A. In particular, to define the various generic formulae, i.e., I0, G0, E0, we use the Theory plug-in [15]. The theory associated with B0 is as follows where I0, G0, E0 are defined as operators with appropriate types.

```
theory B0
types V T
operators
   I0(v: "V")
   G0(t: "T", v: "V")
   E0(t: "T", v: "V"): "V"
```

2) The generic output model is the machine (flatten)A0 which is generated from A0 using our implementation in Section V. We also define the generic formulae in A0 using the Theory plug-in as follows.

```
theory A0
imports B0
types U X
operators
   J0(v: "V", x: "X")
   H0(u: "U", x: "X")
   K0(t: "T", v: "V", u: "U", x: "X")
   F0(u: "U", x: "X"): "X"
```

3) The consistency condition for B0 corresponding to the proof obligation (INV) is encoded as an axiom of the theory B0.

```
axioms
   @B0/e/I0/INV: "∀ t,v· t ∈ T ∧ v ∈ V ∧ I0(
       v) ∧ G0(t,v) ⇒ I0(E0(t,v))"
```

Similarly, the additional proof obligation (INC_INV) in Theorem 1 is encoded as an axiom of the theory A0.

```
axioms
   @A0/f/J0/INV: "∀ t,v,u,x· I0(v) ∧ J0(v,x)
       ∧ G0(t,v) ∧ H0(u,x) ∧ K0(t,v,u,x) ⇒
       J0(E0(t,v), F0(u,x))"
```

Corresponding proof rules are defined according to these axioms.

4) All the proof obligations associated with (flatten)A0 are automatically discharged by Rodin. In particular, the fact that event f maintains invariant I0 (relying on axiom B0/e/I0/INV) and maintains invariant J0 (relying on axiom A0/f/J0/INV) is proved as expected.

The verification for Theorem 2 is similar and is omitted here. The models are available online at http://users.ecs.soton.ac.uk/tsh2n14/developments/ICECCS2017/.

VII. Conclusion

In this paper, we present a machine inclusion mechanism for Event-B. The proposed mechanism allows us to construct a model 'bottom-up' by combining existing models. Moreover, we illustrate that the new mechanism integrates seamlessly with the existing refinement development process of Event-B. By including multiple instances of a machine, we also reuse the modelling and proving effort in developing the formal model. We have extended Rodin to support machine inclusion using EMF and Xtext. Using the developed plug-in tool, we verify the correctness of the proof obligations related to machine inclusion by constructing generic models and reason about them with the Theory plug-in.

A. Related Work

Various composition approaches have been proposed before for Event-B [3]–[6]. The initiative of our work can be found in [16]. In [3], the authors introduced a *modularisation* approach for including a "module" via operation calls. However, modules are a new construct introduced by the modularisation approach and need to be treated differently from machines, including different proof obligations. In [5], the authors defined an architecture for incorporating a refinement-chain (called a pattern) into a development. While reusing a refinement-chain is similar to our approach, the pattern needs to be matched with a part of the current development. In our approach, we can directly reuse the pattern using machine inclusion. In [4], the author presented a notion of event *fusion* for Event-B and proved that event fusion preserved refinement. Event fusion allows combining events of models with shared variables, whereas in our approach, included machines contribute different sets of variables to the including machine. Moreover, composition of refinement patterns in [4] gave a quite rigid modular arrangement. For example, each refinement step in the pattern results in a corresponding refinement step in the main development. As shown in the example in Section IV, our development architecture is quite flexible in terms of where or when to include the refinement of the patterns. In [6] the authors used shared-event composition to construct a composed-machine from existing models. However the composed-machine itself does not have any variables and it is more restricted than the machine inclusion mechanism.

Our machine inclusion mechanism is influenced by the similarly named mechanism in classical B [17], including machine renaming and restrictions on modifying variables of the included machine. In classical B, operations of included machines are *called* from the including machine, whereas we use event synchronisation. Furthermore, machine inclusion in classical B only supports including a specification; i.e., the top-level abstraction of a refinement-chain. The reuse of refinement-chains in our approach is basically applying some refinement pattern as specified by the included refinement-chain. The same idea has been developed for classical B into a tool for *automatic refinement* [18]. The difference between BART and our tool is that BART is a model transformation tool (according to some user-defined rules) and still requires proofs in order to make sure that the proposed refinement is correct.

B. Future Work

In order to include a machine, both the including and the included machines need to have the same context and this does not hold priori. In order to realise the full potential of reusing existing models, we need to apply *generic instantiation* to instantiate the context of the included machine accordingly. We can benefit from the experience of existing approaches [6], [19] to ensure the consistency of instantiation. Currently our implementation of the supporting tool generates a flattened model corresponding to the machine with its inclusion clauses. This is to utilise the existing support for static checking and proof-obligation generating capability of Rodin. However, this also means that obligations which have already been proven in the included machine are regenerated again in the including machine. Our immediate task is to ensure that only necessary proof obligations as specified in Theorems 1 and 2 are generated. At the same time, we need to evaluate our approach on more case studies, including those from the Enable-S3 project [20], for example the RailGround case study [21]. Our inclusion mechanism enables the possiblity of reusing formal models. As a result, we would like to develop a library of reusable models, such as the model of sensors, that are useful for many different systems.

ACKNOWLEDGMENT

This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Union's HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

REFERENCES

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

[2] J. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to Event-B," *Fundam. Inform.*, vol. 77, no. 1-2, pp. 1–28, 2007. [Online]. Available: http://content.iospress.com/articles/fundamenta-informaticae/fi77-1-2-02

[3] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Supporting reuse in event B development: Modularisation approach," in *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, ser. Lecture Notes in Computer Science, M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, Eds., vol. 5977. Springer, 2010, pp. 174–188. [Online]. Available: https://doi.org/10.1007/978-3-642-11811-1_14

[4] M. Poppleton, "The composition of Event-B models," in *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, ser. Lecture Notes in Computer Science, E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., vol. 5238. Springer, 2008, pp. 209–222. [Online]. Available: https://doi.org/10.1007/978-3-540-87603-8_17

[5] T. S. Hoang, A. Fürst, and J. Abrial, "Event-B patterns and their tool support," *Software and System Modeling*, vol. 12, no. 2, pp. 229–244, 2013. [Online]. Available: https://doi.org/10.1007/s10270-010-0183-7

[6] R. Silva and M. J. Butler, "Supporting reuse of Event-B developments through generic instantiation," in *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, ser. Lecture Notes in Computer Science, K. K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 466–484. [Online]. Available: https://doi.org/10.1007/978-3-642-10373-5_24

[7] T. S. Hoang, "An introduction to the Event-B modelling method," in *Industrial Deployment of System Engineering Methods*. Springer-Verlag, 2013, pp. 211–236.

[8] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in Event-B," *Software Tools for Technology Transfer*, vol. 12, no. 6, pp. 447–466, Nov. 2010.

[9] H. Kuruma and T. S. Hoang, "Crossed-project reference for managing model variations," in *Proceedings of the Rodin Workshop 2016*, M. Butler, S. Hallerstede, T. S. Hoang, M. Leuschel, and L. Voisin, Eds. University of Southampton, May 2016.

[10] S. Hudon and T. S. Hoang, "Development of control systems guided by models of their environment," *Electr. Notes Theor. Comput. Sci.*, vol. 280, pp. 57–68, 2011. [Online]. Available: https://doi.org/10.1016/j.entcs.2011.11.018

[11] C. Snook, F. Fritz, and A. Iliasov, "Event-B and Rodin Documentation Wiki: EMF Framework for Event-B," http://wiki.event-b.org/index.php/EMF_framework_for_Event-B, 2009, accessed July 2017.

[12] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Eclipse Modeling Framework*, 2nd ed., ser. The Eclipse Series. Addison-Wesley Professional, December 2008.

[13] M. Eysholdt and H. Behrens, "Xtext: Implement Your Language Faster Than the Quick and Dirty Way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 307–309. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869625

[14] S. Hallerstede and T. S. Hoang, "Refinement of decomposed models by interface instantiation," *Sci. Comput. Program.*, vol. 94, pp. 144–163, 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2014.05.005

[15] M. J. Butler and I. Maamria, "Practical theory extension in Event-B," in *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, Z. Liu, J. Woodcock, and H. Zhu, Eds., vol. 8051. Springer, 2013, pp. 67–81. [Online]. Available: https://doi.org/10.1007/978-3-642-39698-4_5

[16] M. Butler, S. Hallerstede, T. S. Hoang, M. Leuschel, and L. Voisin, Eds., *Proceedings of the Rodin Workshop 2016*. University of Southampton, May 2016.

[17] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[18] A. Requet, "BART: A tool for automatic refinement," in *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, ser. Lecture Notes in Computer Science, E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., vol. 5238. Springer, 2008, p. 345. [Online]. Available: https://doi.org/10.1007/978-3-540-87603-8_33

[19] A. Fürst, T. S. Hoang, D. A. Basin, N. Sato, and K. Miyazaki, "Large-scale system development using abstract data types and refinement," *Sci. Comput. Program.*, vol. 131, pp. 59–75, 2016. [Online]. Available: https://doi.org/10.1016/j.scico.2016.04.010

[20] The Enable-S3 Consortium, "Enable-S3 European project," 2016, www.enable-s3.eu.

[21] K. Reichl, "RailGround model on github," 2016, https://github.com/klar42/railground/ (Accessed 20/04/2017).

[22] E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5238. Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-87603-8