The original publication is available at http://doi.org/10.1109/ISSRE.2017.40
In The 28th International Symposium on Software Reliability Engineering (ISSRE 2017) © IEEE 2017.

Formal Development of Policing Functions for Intelligent Systems

C. Bogdiukiewicz†, M. Butler*, T.S. Hoang*, M. Paxton†, J. Snook*, X. Waldron† and T. Wilkinson*
*Electronics and Computer Science
University of Southampton
Southampton, UK
Email: {mjb, t.s.hoang, jhs1m15, stw08r}@ecs.soton.ac.uk
†TEKEVER Ltd.
Southampton, UK
Email: {martin.paxton, xanthippe.waldron, chris.bogdiukiewicz}@tekever.com

***Abstract*—We present an approach for ensuring safety properties of autonomous systems. Our contribution is a system architecture where a policing function validating system safety properties at runtime is separated from the system's intelligent planning function. The policing function is developed formally by a correct-by-construction method. The separation of concerns enables the possibility of replacing and adapting the intelligent planning function without changing the validation approach. We validate our approach on the example of a multi-UAV system managing route generation. Our prototype runtime validator has been integrated and evaluated with an industrial UAV synthetic environment.**

***Index Terms*—Formal Methods; Policing Function; UAVs; Event-B; correct-by-construction**

I. Introduction

a) Motivation: Autonomous systems offer many potential advantages over their more traditional human-controlled counterparts, however they also raise several significant challenges. One of the key challenges is demonstrating that an autonomous system can reliably perform the tasks assigned to it, and that it can do so without endangering the safety of persons or property.

b) Challenge: The difficulty in ensuring the safety of an autonomous system arises from (1) the complexity of autonomous decision making mechanisms, and (2) vague and ambiguous safety properties. On the one hand, autonomous systems often employ complex planning functions to produce their output. These intelligent functions usually deploy several heuristic strategies and are hence often unpredictable. Furthermore, they are subjected to frequent improvements and adaptations. On the other hand, whereas safety properties for these autonomous systems are often clear at an intuitive level, e.g., collision avoidance, it is difficult for the system designers to demonstrate that these safety requirements are met.

c) Approach: The approach that we take is to formally develop a runtime policing function to validate the system safety. The policing function is separated from the system's intelligent planning function and checks the system output against safety constraints. This separation of concerns allows us to decouple the validation of the system safety properties from the complexity of the planning function. It would allow the intelligent system to be modified without having to re-verify the policing function. To address the second challenge and to demonstrate that the policing function correctly validates the safety constraints, we use a correct-by-construction approach: the policing function is formally constructed from its abstract specification, through several consistent refinement steps to derive its implementation.

d) Contribution: Our contribution therefore is a system architecture where a *formally developed policing function* validating system safety properties is separated from the system's intelligent planning function (Figure 1). The intelligent system

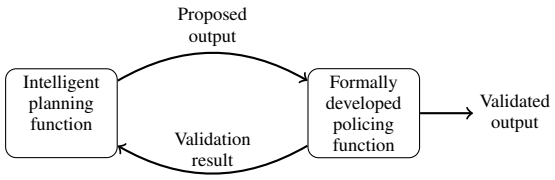

Fig. 1. System architecture for formally developed policing function

can be adapted or replaced without changing the validation approach. Validation of additional safety constraints can be added to the policing function independently of the intelligent function. By formally developing the policing function, we ensure the correctness of the policing function in validating the system safety properties. We illustrate our approach on the example of a multi-*Unmanned Aerial Vehicle* (UAV) system managing route generation. The formalisation of our policing function is developed using the Event-B [1] modelling method.

e) Structure: The rest of the paper is structured as follows. Section II gives an overview of the multi-UAV system under consideration and of the Event-B modelling method. Section III illustrates our approach by formally developing a `C` implementation of a policing function for the multi-UAV system. The `C` implementation of the policing function is integrated with the synthetic environment for the multi-UAV system and its performance is evaluated in Section IV. We summarise and give some conclusions in Section V.

II. Background

In this section, we first give an overview of the multi-UAV system under consideration in Section II-A and give some background information on the Event-B modelling method in Section II-B.

A. System Overview

The system under consideration is a *Ground Control Station* (GCS) which is able to produce a set of routes for a number of UAVs. The system overview can be seen in Figure 2. The *Operators* are users who interact with the human machine interface of the system. They give commands to the components of the system. The *Human Machine Interface* allows operators to interact with the system by displaying relevant information to them and taking inputs and commands. The *Route Planner* is an intelligent function which generates route plans according to goals and constraints specified by the operators and other components of the system. The *Network Comms Interface* allows for communication between the control station and the aircraft. The *Aircraft* take commands and routes from the control station. They can be tasked individually.

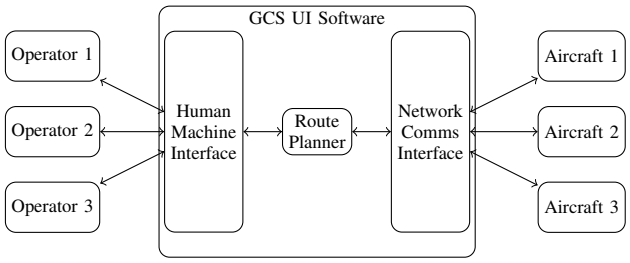

Fig. 2. The system overview

Several use cases of the system are considered. The system is suitable for use as a "static" tool, meaning that it can be used before any flight begins to check the scheduled routes. It can also be used in a "dynamic" scenario, where one or more aircraft are already in flight and the operator wishes to change the current state. The system also differentiates between two types of requests: "command" and "planning". Commands are used to issue orders to aircraft, such that if the validation is successful the new route will be sent to the aircraft as an immediate update to their current plan. Planning requests allow an operator to query the validation system with a suggested plan, without tasking any aircraft with that plan. The challenge in the system is to ensure that the aircraft fly safely. For example, their current routes are not conflicted and they are not flying into any restricted airspace.

B. Event-B

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* (similar to types), *constants*, and *axioms* that constrain the carrier sets and constants. Machines contain *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].

$$e \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(t,v) \textbf{ then } v := E(t,v) \textbf{ end}$$

A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions. Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. A machine M can be *refined* by a 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. 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 (i.e., if f is enabled then e is also enabled) and f's action can be simulated by e's action, taking into account the gluing invariant J. More information about Event-B can be found in [2]. Event-B is supported by *Rodin platform* (Rodin) [3], 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.

1) Theory Extension: The Theory plug-in [4] enables modellers to extend the mathematical modelling language of Event-B with *theories* containing new (polymorphic) data types and operators upon those data types. These additional modelling concepts might be defined directly (including inductive definitions) or axiomatically.

Operators can be defined *directly*, *inductively* (on inductive data types) or *axiomatically*. An operator defined without any definition will be defined axiomatically. In this paper, we focus on the use of operators with direct definition. For example, assume that we have declared a type of real numbers $\mathbb{R}$ and Euclidean space operator euclidean_space(n) where n is the number of dimensions. We can define a route as a list of waypoints as follows.

```
operators
  TIME ≙ ℝ
  POSITION ≙ euclidean_space(3)
  WAYPOINT ≙ POSITION × TIME
  ROUTE ≙ {n, wp · n ∈ ℕ₁ ∧ wp ∈ 0‥ n− 1 → WAYPOINT | wp}
```

Each waypoint is a pair of position and time, where position is a point in a 3-dimensional Euclidean space, and time is a real number. In the above definitions, S → T denotes the set of total function from S to T. The summary of the Event-B mathematical language can be found in [5].

[1] Actions in Event-B are, in the most general cases, non-deterministic [2].

Finally, theories can be constructed in hierarchical manner: a theory can *import* one ore more other theories to define and declare more data types, operators, and axioms.

III. Development

In Section III-A, we illustrate the policing function (called Route Validator) for our GCS and its functional requirements. In Section III-B, we describe the formal development of the Route Validator using Event-B.

A. The Route Validator Requirements

The functional architecture of the Route Validator can be seen in Figure 3. The Route Validator checks each route

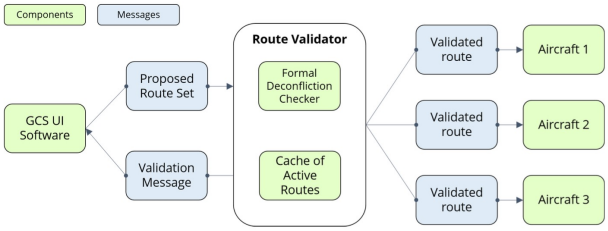

Fig. 3. Functional Architecture of the Route Validator

against already active routes flown by each aircraft in the system and deems a route "valid" if there are no conflicts. We mean to plan routes that avoid air-to-air collisions primarily between our set of UAVs, but also consider the more broad case between all aircrafs, and between our UAVs and terrain/airspace constraints. In this paper, we focus on air-to-air collision avoidance between aircraft (deconfliction property). The Route Validator can be used as a planning tool, and is also used as a checking function when a new route is sent to an aircraft. It encompasses the *Formal Deconfliction Checker*.

The requirements for the Route Validator is as follows. The validator must maintain an updatable store of existing aircraft information (including active and inactive aircraft) and routes. In order to ensure the safety of the flying (active) aircraft, the validator must also maintain an awareness of which aircraft are flying which routes. This is implemented by the *Cache of Active Routes*.

- [REQ 1] The validator must maintain an updateable store of aircraft information and routes.
- [REQ 2] The validator maintains a cache of active aircraft and their flying routes.

The GCS sends new route commands to one or more aircraft to update their flying routes. It is important that the Route Validator only allows new route commands to be issued to aircraft if there are no conflicts detected between routes (including the current and the new flying routes). Upon receiving a route command, an aircraft updates its flying route and sends a confirmation message back to the Route Validator. When the Route Validator receives a confirmation message from an aircraft, it updates its cache to reflect the current flying route of that aircraft.

- [REQ 3] The validator only allows consistent route commands to be sent to the aircraft.
- [REQ 4] The aircraft must confirm to the Route Validator when receiving a route command.
- [REQ 5] The Route Validator updates its cache accordingly when receiving a confirmation message.

Note that due to the communication delay between Route Validator and the aircraft, the cache information of the Route Validator might not reflect the actual flying routes of the active aircraft. We also do not consider communication failures at the moment.

The most important requirement for the system is that there must not be any air-to-air collision between aircraft. For this purpose, the system must ensure that any active aircraft maintains a minimum separation distance from the other aircraft.

- [REQ 6] Any active aircraft must maintain a minimum separation distance from other aircraft.

B. Formal development

In order to formalise the Route Validator, we first build a set of theories for physical modelling.

1) Theories for Physical Modelling: The routes of the aircraft are represented as continuous paths in 3-dimensional space parameterised by time, i.e. continuous functions from time to 3-dimensional space. This abstract representation of a route is independent of the representation any implementation would use to either generate routes, or communicate them to the UAVs. To do this we need to extend the mathematical theories of Event-B as follows.

- We need a representation of real numbers, operators for real numbers, and basic support proof rules.
- We develop a theory of n-dimensional Euclidean spaces and continuous paths.

Based on these theories, we develop a theory for deconfliction which is the basic for our specification of the route validator.

a) Theory of Real Numbers: This theory underpins all our work. This theory is developed as a collection of theories that build upon each other. The hierarchy of the theories for real numbers can be seen in Figure 4. Here, an arrow signifies a dependency relationship which is transitive. The most primitive theory is the theory _FieldAxioms which defines constants (real numbers) zero and one, and the basic operations of addition and multiplication. Then through a collection of intermediate theories, the Real theory is extended up to the point where the square root function can be defined. This is required for the subsequent definition of the Euclidean metric in space. To define square root we require the Least Upper Bound axiom of the real numbers, and this is defined in the _CompletenessAxioms theory. Without this our theory would not have a value for $\sqrt{2}$ as this is not a rational number. A similar theory of real numbers is also developed by Babin et. al. [6].

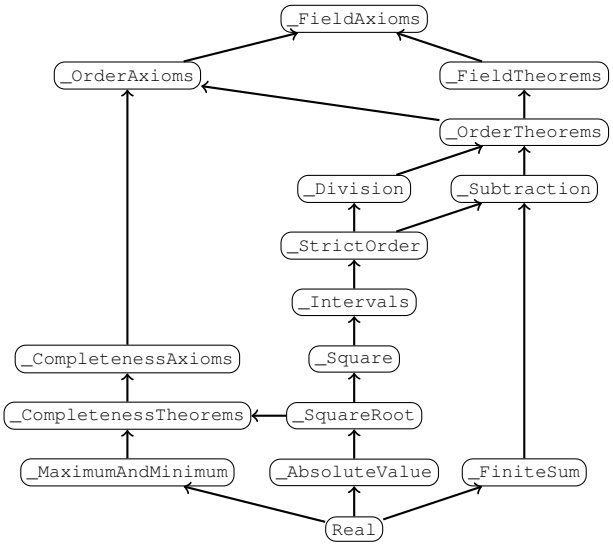

Fig. 4. The hierarchy of theories for real numbers

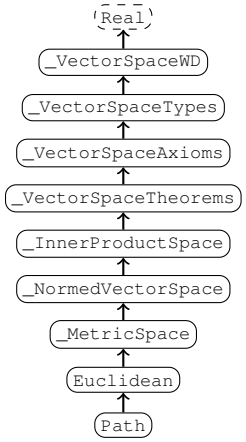

Fig. 5. The hierarchy of theories for Euclidean space

b) Theories of Euclidean Space and Paths: Based on the Real theory, we construct a theory of Euclidean space. We do this in a general way and construct a theory of n-dimensional Euclidean space. Once again the theory is constructed incrementally (Figure 5). First we introduce vectors of real numbers and their addition and scalar multiplication, and then build upon that to add an inner product, the Euclidean norm (length of a vector), and finally the Euclidean metric that gives the distance between two points in n-dimensional space. The UAV routes are modelled as continuous paths in Euclidean space, and so the Path theory builds upon the Euclidean theory.

c) Theory for deconfliction: Recall the definition of a route as a list of waypoints from Section II-B1 as follows.

```
ROUTE ≙ {n, wp · n ∈ ℕ₁ ∧ wp ∈ 0.. n− 1 → WAYPOINT | wp}
```

Since we consider the deconfliction of a set of routes associated with aircraft, we define ROUTE_T as the Cartesian product of AIRCRAFT and ROUTE. We call a pair of aircraft and route an aircraft route. For convenient, operators route_id and route returns the aircraft and route component from an aircraft route, respectively.

```
ROUTE_T ≙ AIRCRAFT × ROUTE
route_id(rt) ≙ prj₁ (rt)
route(rt) ≙ prj₂ (rt)
```

We define the route_pair_conflicts between two aircraft routes rt1 and rt2, given a minimum separation distance ms as follows.

```
route_pair_conflicts(rt1, rt2, ms) ≙
  {m, n, p1, p2 · m ∈ 0 .. len(rt1)− 2 ∧ n ∈ 0 .. len(rt2) − 2 ∧
   s1 = route_id(rt1) ↦ (route(rt1)(m) ↦ route(rt1)(m + 1)) ∧
   s2 = route_id(rt2) ↦ (route(rt2)(n) ↦ route(rt2)(n + 1)) ∧
   route_id(rt1) ≠ route_id(rt2) ∧
  ¬ segment_pair_safe(s1, s2, ms)
  | s1 ↦ s2}
```

Here route_pair_conflicts(rt1, rt2, ms) is a set of segment pairs of s1 ↦ s2, where s1 is a segment (the line connecting two consecutive waypoints) of rt1, s2 is a segment of rt2, and s1 and s2 are not safe with respect to the minimum separation distance ms. (The set of segments is defined as SEGMENT_T = AIRCRAFT × (WAYPOINT × WAYPOINT).)

The definition of segment_pair_safe is as follows based on the notion of deconfliction between paths.

```
segment_pair_safe(s1, s2, ms) ≙
    deconfliction(segment_path(s1), segment_path(s2), ms)
```

Here, the segment_path(s) denotes the path (a function of time to 3-dimensional space) corresponding to the segment s (we omit its definition here).

The definition of deconfliction is as follows.

```
deconfliction(pth1, pth2, ms) ≙
  ∀ t· t ∈ TIME ⇒ ms ≤ dist(3, pth1(t), pth2(t))
```

Two paths pth1 and pth2 are de-conflicted (with respected to a minimum separation distance ms) if for all time t, the distance between pth1 and pth2 at time t is at least ms. Operator dist returns the Euclidean distance between two points in Euclidean space. In the the definition of deconfliction, and we use dist for 3-dimensional space distances.

We can now use the theory for deconfliction to specify our system and develop the deconfliction checking program. Our formal development of the Route Validator contains two stages. At the system-level stage (Section III-B2), we model the overall system including the cache and the communications between the Route Validator and the aircraft. In the second stage (Section III-B3), we extract the core functionality of the Formal Deconfliction Checker and develop this further towards the implementation.

2) System-Level Modelling: The formal system model[2] is developed using an initial model capturing the safety property

[2]The model is available from the University of Southampton repository at http://doi.org/10.5258/SOTON/D0217.

of the system. We subsequently refine this model by introducing the cache and finally the communications between GCS and the aircraft.

a) The initial model: At the system level, there are two types of entity in the system, namely *aircraft* and *routes*. They are modelled by carrier sets in Event-B. Axiom @axm1 states that the minimum separation distance ms is non-negative.

```
context c0
sets AIRCRAFT ROUTE
constants ms
axioms
  @axm1: "zero ≤ ms"
end
```

For convenient, we define operator consistent for a set of aircraft routes S as follows.

```
consistent(S) ≙ ∀ rt1, rt2 · route_pair_conflicts(rt1, rt2, ms) = ∅
```

The definition states that S is consistent if there are no conflicts between any pair of aircraft routes in S with respect to the minimum separation distance ms.

The dynamic model of the system contains variables aircraft and routes representing the existing aircraft and routes in the system (REQ 1). A relationship active denotes the current active route associated with the aircraft. It is a partial function (see @inv3) with its domain representing the set of active aircraft. Invariant @inv4 capture the main safety requirement of the system (REQ 6) stating that the set of active aircraft are always "consistent".

```
machine m0
sees c0
variables aircraft routes active
invariants
  @inv1: "aircraft ⊆ AIRCRAFT"
  @inv2: "routes ⊆ ROUTE"
  @inv3: "active ∈ aircraft ⇸ routes"
  @inv4: "consistent(active)"
events
  INITIALISATION
  begin
    @act1: "aircraft := ∅ "
    @act2: "routes := ∅ "
    @act3: "active := ∅ "
  end
```

Initially, all variables are assigned the empty set (no existing aircraft or routes).

We have events CreateAircraft, RemoveAircraft, CreateRoute, RemoveRoute for adding or removing aircraft and routes accordingly (REQ 1). For example, events CreateAircraft and RemoveAircraft is specified as follows.

```
  CreateAircraft
  any ac where
    @grd1: "ac ∉ aircraft"
  then
    @act1: "aircraft := aircraft ∪ {ac}"
  end

  RemoveAircraft
  any ac where
    @grd1: "ac ∈ aircraft"
    @grd2: "ac ∉ dom(active)"
  then
    @act1: "aircraft := aircraft \ {ac}"
  end
```

Notice that the guard @grd2 of event RemoveAircraft is to ensure that it maintains invariant @inv3: an aircraft ac can only be removed if it is no longer active.

An event namely DeactivateAircraft is used to model the deactivation of an active aircraft ac, e.g., when it finishes flying its route. Here S ⩤ f removes all the mappings in f originating from an element in S.

```
  DeactivateAircraft
  any ac where
    @grd1: "ac ∈ dom(active)"
  then
    @act1: "active := {ac} ⩤ active"
  end
```

Finally, we have an event UpdateRoute to update the route of an aircraft ac to fly a new route rt. The new route information is represented by the parameter rts. Here f ⩥ g denote relational overriding of relation f by g. Guard @grd3 ensures that the active aircraft remains consistent after updating the route of ac. Later on, this abstract guard will be refined using the route command sent from the GCS.

```
  UpdateRoute
  any ac rt where
    @grd1: "ac ∈ aircraft"
    @grd2: "rt ∈ routes"
    @grd3: "consistent(active ⩥ {ac ↦ rt})"
  then
    @act1: "active(ac) := rt"
  end
```

b) The First Refinement: In this first refinement, we introduce the model of the cache of the aircraft routes. Moreover, due to the communication delays, we also model the set of pending commands that the GCS sent to the aircraft for updating but have not yet received any confirmation. We focus here on @inv6 stating that the active routes must be contained within the cache and pending routes. Intuitively, an active aircraft at any point flies the original route, i.e., stored in the cache or already updated to the new route as specified in the pending commands. Invariant @inv8 ensures that the set of cache and pending routes must be consistent.

```
variables cache pending
invariants
  @inv4: "cache ∈ aircraft ⇸ routes"
  @inv5: "pending ∈ aircraft ⇸ routes"
  @inv6: "active ⊆ cache ∪ pending"
  @inv7: "pending = ∅ ⇒ active = cache"
  @inv8: "consistent(cache ∪ pending)"
  @inv9: "dom(pending) ⩤ cache ⊆ active"
```

We refine event UpdateRoute as follows, i.e., the aircraft ac update its route to rt if there is a corresponding pending command.

```
  UpdateRoute
  refines UpdateRoute
  any ac rt where
    @grd1: "ac ↦ rt ∈ pending"
  then
```

```
    @act1: "active(ac) := rt"
  end
```

The correctness of the refinement of event UpdateRoute is relied on invariants @inv6, @inv8 and the property of consistent operator below, stating that consistency is preserved with the subset, i.e., ⊆ , relation. The property of consistent is proved from its definition based on route_pair_conflicts.

```
"∀ s,c · consistent(c) ∧ s ⊆ c ⇒ consistent(s)"
```

We add two new events, namely SetPending and RemovePending to model how the pending commands are modified. In SetPending, a new set of routes commands rts is sent to the aircraft. In RemovePending, the pending command ac ↦ rt is removed and the cache is updated, provided that the aircraft ac has already flies the new route rt.

```
SetPending
any rts where
  @grd1: "pending = ∅ "
  @grd2: "rts ∈ aircraft ⇸ routes"
  @grd3: "cache ∪ rts ∈ consistent"
then
  @act1: "pending := rts"
end

RemovePending
any ac rt where
  @grd1: "ac ↦ rt ∈ pending"
  @grd2: "ac ↦ rt ∈ active"
then
  @act1: "pending := pending \ {ac ↦ rt}"
  @act2: "cache(ac) := rt"
end
```

We omit the details about refinement of the other events here.

c) The Second Refinement: In this refinement, we introduce the communications between the GCS and the aircraft. Two variables messages and confirms are added to the model denoting the set of commands that have been sent to the aircraft but not yet received and the set of confirmations message from the aircraft to the GCS. Invariants @inv12 states that if an aircraft ac confirms to the GCS then it must already fly the new route as specified by the pending commands. Invariant @inv13 states that an aircraft is either already confirm to fly the new route or has not yet received the corresponding command.

```
variables
  messages
  confirms
invariants
  @inv10: "messages ⊆ pending"
  @inv11: "confirms ⊆ dom(pending)"
  @inv12: "∀ ac · ac ∈ confirms ⇒ ac ↦ pending(ac) ∈ active"
  @inv13: "confirms ∩ dom(messages) = ∅ "
```

We refine event SetPending by event SendCommands to model situation where the GCS sends consistent route commands to a set of aircraft (REQ 3). An additional (with respect to the abstract SetPending event) action @act2 to update the set of messages is added to event SendCommands.

```
SendCommands refines SetPending
any rts where
  @grd1: "pending = ∅ "
  @grd2: "rts ∈ aircraft ⇸ routes"
  @grd3: "cache ∪ rts ∈ consistent"
then
  @act1: "pending := rts"
  @act2: "messages := rts"
end
```

We refine event UpdateRoute further as follows. When an aircraft ac receives a command to fly a new route rt, it update its fly-path, sends a confirmation to the GCS (REQ 4).

```
UpdateRoute
any ac rt where
  @grd1: "ac ↦ rt ∈ messages"
then
  @act1: "active(ac) := rt"
  @act2: "confirms := confirms ∪ {ac}"
  @act3: "messages := messages \ {ac ↦ rt}"
end
```

We refine event RemovePending by event Confirms to model the situation where the GCS receives a confirmation (REQ 5).

```
Confirms refines RemovePending
any ac rt where
  @grd1: "ac ∈ confirms"
  @grd2: "rt = pending(ac)"
then
  @act1: "pending := pending \ {ac ↦ rt}"
  @act2: "cache(ac) := rt"
  @act3: "confirms := confirms \ {ac}"
end
```

The correctness of event Confirms relies on @inv12. We omit the detailed reasoning here.

3) Formal Deconfliction Checker: The most important functionality of the route validator described in the previous section is to check if the set of aircraft routes are consistent. In particular, in the case where the aircraft routes are inconsistent, the route validator must return the set of conflicts segments. In this section, we outline our formalisation of the route validator implementation in C. The implementation contains several C functions. We first present our approach then highlight the formalisation of some selective functions.

a) Approach: Our approach is to capture the functionality of each C function in a separate Event-B development. The specification of the function is captured abstractly using a state-less machine with a single event corresponding to the function. The parameters of the function are declared as constants, with the precondition represented as a set of axioms. The output(s) of the function is modelled using parameter(s) of the event with the events guard representing the function's post-condition. Assuming that we have a following C function.

```
/*
 * Pre-condition: pre(p1, p2, ..., pn)
 * Post-condition: post(out, p1, p2, ..., pn)
 */
T func(T1 p1, T2 p2, ..., Tn pn);
```

The abstract specification of `func` in Event-B is as follows.

```
constants p1 p2 ... pn
axioms "pre(p1, p2, ..., pn)"
```

```
events
  func
  any out where
    "post(out, p1, p2, ..., pn)"
  then
    skip
  end
```

The implementation of the function is introduced using one or more refinements, where the C local variables are introduced as Event-B variables of the refinement. To simplify our model, we focus on the functionality of the C function and omit details such as memory handling, etc. Furthermore, we compress the effect of sequential statements into parallel assignments in Event-B. To further modularise our model, we develop nested loops using different Event-B refinement chains. We extract the abstract specification of the inner loop and create a separate Event-B refinement chain (essentially, it is the same as creating a new C function corresponding to the inner loop).

b) Function `get_conflicts`*:* We start our formalisation with the specification of the main function `get_conflicts`. The signature of the function is as follows.

```

/*
 * Get the conflicts of a set of routes.
 *
 * r : an array of routes
 * len : the size of r (number of routes)
 * min_sep : the required minimum separation distance
 *
 * Pre-condition:
 * - there must be at least two routes
 * - the minimum separation distance is positive
 *
 * Post-condition:
 * The return value is a structure containing an array
 * of conflicting route segments.
 */
separation_conflicts_t *get_conflicts(route_t *r,
           int len,
           distance_t min_sep) {
  assert(r != NULL);
  assert(2 <= len);
  assert(0.0 < min_sep);
  ...
}
```

The Event-B abstract specification corresponding to get_conflicts function is as follows.

```
constants r len min_sep
axioms
  "2 ≤ len"
  "r ∈ 0 .. len − 1 → ROUTE_T"
  "zero ≤ min_sep"
events
  get_conflicts
  any cflts where
    "cflts = (⋃ m,n · m ∈ 0 .. len − 1 ∧
                                n ∈ m+1 .. len − 1
      | route_pair_conflicts(r(m), r(n), min_sep))"
  then
    skip
  end
```

The Event-B specification uses the operator route_pair_conflicts declared earlier. The output cflts is the (generalised) union (⋃) of all pairwise conflicts (with respect the the minimum separation distance min_sep) within the input array of aircraft routes r. Note that we model the type of result as a set of conflict segments, rather than as an array. This is a valid abstraction which make the formal model much simpler to reason about.

The implementation of the `get_conflicts` is a nested loop as follows.

```
separation_conflicts_t *get_conflicts(route_t *r,
           int len,
           distance_t min_sep) {
  ....
  separation_conflicts_t *conflicts = sc_create();

  for (int i = 0; i < len; ++i)
  {
    for (int j = i + 1; j < len; ++j)
    {
      if (r[i].id == r[j].id)
      {
        /* An aircraft is never in conflict with itself. */
        continue;
      }

      route_pair_conflicts(conflicts,
        r[i],
        r[j],
        min_sep);

    }
  }
}
```

We first develop the outer loop as a refinement of the above specification, abstracting from the inner loop.

```
variables i conflicts
invariants
  "i ∈ 0 .. len"
  "conflicts ∈ ℙ (WAYPOINT × WAYPOINT)"
  "conflicts = (⋃ m,n · m ∈ 0 .. i − 1 ∧
                                n ∈ m+1 .. len − 1
    | route_pair_conflicts(r(m),r(n),min_sep))"
events
  INITIALISATION
  begin
    "i := 0"
    "conflicts := ∅ "
  end

  get_conflicts
  refines get_conflicts
  any cflts where
    "¬ (i < len)"
    "cflts = conflicts"
  end

  progress_i
  when
    "i < len"
  then
    "i := i + 1"
    "conflicts := conflicts ∪
            (⋃ n · n ∈ i+1 .. len − 1
      | route_pair_conflicts(r(i), r(n), min_sep))"
  end

end
```

The correctness of the refinement of event get_conflicts is straightforward, relying on the invariants and the guard of the concrete event.

The above events representing a looping program

```
INITIALISATION
```

```
do progress_i od;
get_conflicts
```

Here, do ... od keeps iterating as long as the guard of progress_i holds. Correctness of refinement means that the abstract event get_conflicts is refined by the intermediate looping program above. In the next stage, progress_i is refined in a similar way leading to a program with nested loops.

Event progress_i is the abstraction of the inner loop (with counter j). We extract this specification, in particular, the assignment to conflicts and start a new Event-B refinement chain. In this new refinement chain, i and the current value of conflicts becomes a constant. The context of the new refinement chain is as follows.

```
constants i init_conflicts
axioms
  "i ∈ 0 .. len"
  "i < len"
  "init_conflicts ∈ ℙ (SEPARATION_CONFLICTS_T)"
```

The specification of the inner loop progress_i becomes as follows.

```
  progress_i
  any cflts where
    @grd1: "cflts = init_conflicts ∪
                    (⋃ n · n ∈ i+1 .. len − 1
                  | route_pair_conflicts(r(i),r(n),min_sep))"
  end
```

The refinement of the inner loop specification for progress_i is as follows.

```
variables j conflicts
invariants
  "j ∈ i+1 .. len"
  "conflicts ∈ ℙ (WAYPOINT × WAYPOINT)"
  "conflicts=init_conflicts∪ (⋃ n· n∈ 0 .. j − 1∧ i<n | route_pair_conflicts(r(i)
        ↦ r(n) ↦ min_sep))"
events
  INITIALISATION
  begin
    "j := i + 1"
    "conflicts := init_conflicts"
  end

  continue
  when
    "j < len"
    "route_id(r(i)) = route_id(r(j))"
  then
    "j := j + 1"
  end

  route_pair_conflicts
  any rp_conflicts where
    "j < len"
    "route_id(r(i)) ≠ route_id(r(j))"
    "rp_conflicts = route_pair_conflicts(r(i) ↦ r(j) ↦ min_sep)"
  then
    "conflicts := conflicts ∪ rp_conflicts"
    "j := j + 1"
  end

  progress_i
  refines progress_i
  when
    "¬ (j < len)"
  with
    @cflts: "cflts = conflicts"
  end
```

```
end
```

c) Function `route_pair_conflicts`*:* The C implementation of `route_pair_conflicts` is as follows.

```
/*
 * Finds all conflicts between line segments in route r1
 * and line segments of route r2.
 *
 * Pre-condition:
 * - a route is required to have at least 2 waypoints.
 * - min_sep is positive
 *
 * The conflicts are returned in result. This is passed
     into the function
 * so we don't need to keep recreating the
     separation_conflicts_t object.
 */

void route_pair_conflicts(
      separation_conflicts_t *result,
      route_t r1,
      route_t r2,
      distance_t min_sep)
{
  assert(2 <= r1.len);
  assert(2 <= r2.len);
  assert(r1.wp != NULL);
  assert(r2.wp != NULL);
  assert(0.0 < min_sep);

  for (int i = 0; i < r1.len - 1; ++i)
  {
    waypoint_pair_t p1 =
          {r1.id, r1.wp[i], r1.wp[i + 1]};

    for (int j = 0; j < r2.len - 1; ++j)
    {
      waypoint_pair_t p2 =
          {r2.id, r2.wp[j], r2.wp[j + 1]};

      if (!line_seg_pair_safe(p1, p2, min_sep) ) {
          sc_append_conflict_pair(
            result, (conflict_pair_t){ p1, p2 });
      }
    }
  }
}
```

Here, `sc_append_conflict_pair` appends the conflict pair $(p1, p2)$ to the array `result`. The formalisation in Event-B of `route_pair_conflicts` is similar to `get_conflicts` (given that they are both implemented using nested loops) and is omitted here.

d) Function `segment_pair_safe`*:* The function `segment_pair_safe` checks if two segments are safe, i.e., not conflicted. Here, the segments are conflicted if there is a point in time that the distance between the two segments are smaller than the allowed minimum separation distance. The Event-B formalisation and the C implementation of the function based on the following case analysis.

- **C1** The line segments do not overlap in time, the two segments are trivially safe.
- **C2** The line segments overlap at a single point in time t. In this case we only need to consider the distance between the two segments at time t.
- **C3** The line segments are parallel and travelled at the same speed. In this case the distance between the agents remains constant and we only need to consider the case at the start of the time overlap.

C4 The two segments are always moving away from each other. In this case the time of closest approach is at the start of the time overlap.

C5 The two agents are segments moving towards each other. In this case the time of closet approach is at the end of the time overlap.

C6 The two segments initially move towards each other, pass through a point of closest approach, and then move away from each other. In this case the time of closet approach is at the (necessarily) unique time t such that the derivative (with respect to time) of the distance between the two segments t is 0.

We omit the details of the C implementation and its Event-B formalisation here.

IV. Integration and Evaluation

As mentioned earlier in Section II-A, the Route Validator is used whenever the GCS send *commands* to the aircraft. Furthermore, it is also used to validate suggested plans from operators and reports any conflict that it found. As a result, an important performance requirement for the route validator is that the component should have a small computational footprint.

In order to evaluate the performance of the Route Validator, we integrated the C implementation of the Formal Deconfliction Checker into the TEKEVER's synthetic environment. The implementation architecture of the Route Validator within the synthetic environment can be seen in Figure 6. In particular, the Route Validator is deployed as one of the standalone application communicating with other system component via the Command and Control Layer. Figure 7 and Figure 8 show

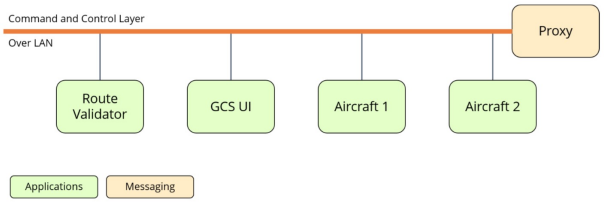

Fig. 6. Implementation Architecture Diagram

the Route Validator console application, with the control and viewing options available to the user (Figure 7) and the log of validation requests (Figure 8). We assessed the Route Validator in large part through running simulated missions and perform computational load tests.

A. Simulation

Several predefined scenarios have been simulated on the GCS system with the integrated Route Validator. The use cases include static planning of routes, commanding aircraft to start flying new routes, dynamic planning of routes, commanding a route change. In particular, in the case where an operator specifies the routes manually, the operator was confidently able to plan routes for a set of UAVs even though routes overlapped spatially because the validator could assure them

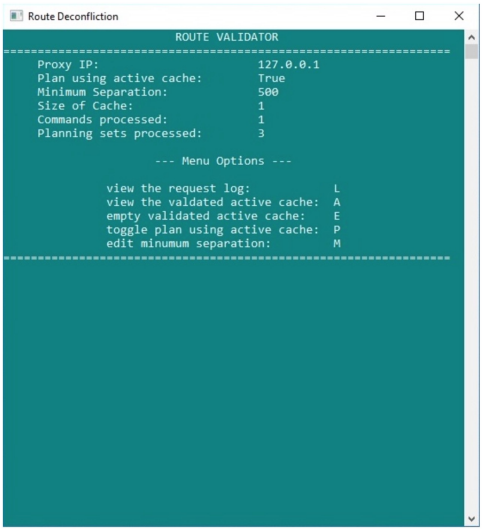

Fig. 7. Route Validator Control

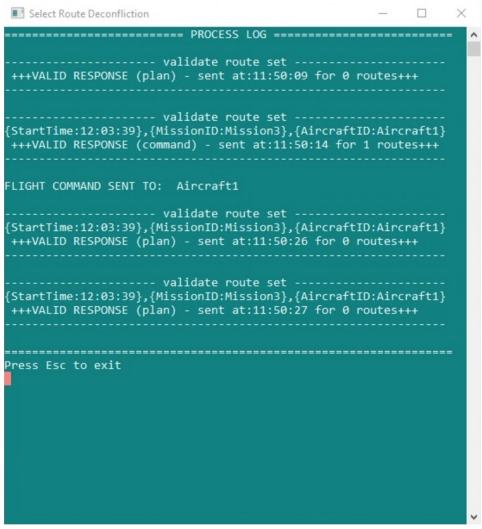

Fig. 8. Route Validator Log of validation requests

the routes were temporally de-conflicted. This is something that was not easy to discover visually. Moreover, we were able to swap between an operator specifying each waypoint and task scheduling algorithms creating the plan without changing the validation approach.

B. Computational Load Tests

To understand the computational performance of the route validator, some performance analysis was carried out. We randomly generated routes for the validator. The data was recorded using a PC with 8GB RAM and a dual 2.70 GHz Intel Core i5-6400 processor, using a program written in C# which timed a single run of the algorithm for various route inputs.

Firstly, the algorithm performance was measured with the quantity of input routes as an independent variable. The chosen length for each of these routes was 25 waypoints, representing

a route manually entered by a UAV operator. The algorithm was then executed with 50 to 1000 input routes in steps of 50. The result of this experiment can be seen in Figure 9. The

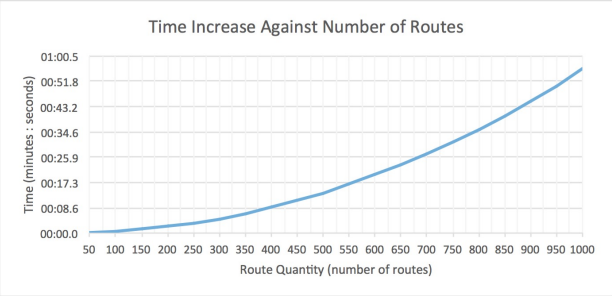

Fig. 9. Time taken by the algorithm against number of routes (of length 25)

algorithm performed well with increase in number of routes. Inputs of 400 routes were validated in less than 10 seconds.

Secondly, we measured the algorithm performance with the route length as an independent variable. This time we have 25 routes and for each generated route, the length was altered from 50 to 1000 waypoints in steps of 50. The result of this experiment can be seen in Figure 10. Discussions

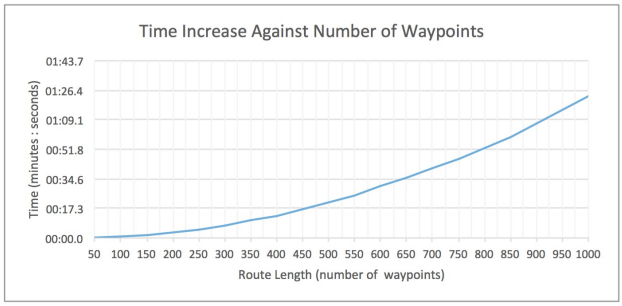

Fig. 10. Time taken by the algorithm against route lengths (for 25 routes)

with TEKEVER's flight team indicate that 25 waypoints is a reasonable route length to consider for a typical flight. Additionally, we consider 250 routes to be a conservative upper bound: we would anticipate that a single operator would have control of up to four aircraft, and when collecting ADS-B data for use in background air traffic over a radius of approximately 75km, there would typically be less than fifteen aircraft in the area at any time. The validation process was tested on a desktop computer with processing power comparable to that of one of TEKEVER's GCS computers, indicating that performance will be similar when operating during a mission. Because the validation process can run in a matter of seconds, rather than minutes or hours, this indicates that it can be used both for planning pre-mission, and for online validation with new information. Additionally, there are optimisations that can be made (such as simplifying route structures, and parallelisation), which further improve performance. Consequently, it was felt that this demonstrated that the route validator can be operated in representative scenarios without hindering operator workflows or delaying processes.

V. Conclusion

In this paper, we demonstrated the use of formal methods in Verification and Validation of an autonomous system in the UAVs domain on an industrial synthetic environment. We focused on a multi-UAV system for managing route generation and allocation. The UAV routes are defined by a combination of human operator and an intelligent system. Our approach was to apply formal methods from high abstraction level to the development of a *safety policing function* implemented in C. The policing function was integrated with TEKEVER's synthetic environment for multi-UAV coordination and validation and performance testing was performed. The advantage of developing a policing function is that verifying the correctness of this function is less complex than verifying the intelligent system. Moreover, this also enables the possibility to improve or adapt the intelligent system without changing the validation approach.

We have developed background formal theories for domain-specific physical concepts and used these concepts to model the core algorithms of our system. Formal modelling was also used to clarify the interaction between TEKEVER's synthetic environment and the policing function. Our case study illustrates that the safety of UAV movement can be addressed in a formal way.

Our approach of formally developing a separate safety policing function is general and also applicable to other forms of autonomous vehicles. Here safety properties are often about collision avoidance and space restrictions. In particular, we ensure that our formal models are generic enough and organised in a hierarchical manner, so that they can be adapted to a new problem with minimal effort.

The challenge in verifying aircraft software has been identified by Rushby earlier [7]. In particular, he stated that "methods (...) that are capable of analyzing high-level requirements, or architectures that monitor safety properties at runtime, are worthy of consideration." In [8], Caseley proposed generic architectures, claims and limitations for combinations of automatic and autonomous functions for manned and unmanned systems. Our policing function is an example of the proposed automatic supervising safety function. More specifically, Caseley identified the need for formal development towards implementation, e.g., "for higher risk functions it may require the confidence of formal assurance of the determinism (e.g. proof) supported by refinement of the proof to the implemented function ...". In [9], Singh et. al. presented an approach for formal modelling and verification of self-adaptive autonomous systems based on Event-B/Rodin. They were also intereseted in collision avoidance and illustrate their approach on an industrial case study (called TwIRTee) involving a set of autonomous rovers. At the architecture level, their system also has a supervision station. The difference with our system is that collision detection is checked during the operations of rovers and if conflict is detected, some resolution actions will

be applied. In our system, collision dectection is carried out in the route planning phase before conflict-free routes are sent to the UAVs. Moreover, their formal model focused solely at the system-level, whereas we also develop the model of the implementation in C.

There are several future challenges for our work. In general, for a careful design approach to handling uncertainty in the world in online situations a validator is only part of the solution. Our extending work is in looking at the interaction of third party aircraft and the online monitoring / intervention required when conflicts are detected. Firstly, the system has to cope with uncertainties which is often omitted in the formal models. We also aim to extend our model to consider *communication failures*, *time drift*, etc. Another important aspect of the system that will need to be examine is the *security of communication*, in particular in safety-critical missions. At the technical level, we have identified the need to increase *automation in formal verification*, including integration of existing tools on the market, and support for automation of *code generation* from concrete Event-B models. Similar to [6], our formalisation of real numbers required considerable proving effort. We are investigating other frameworks [10] for the purpose of real analysis.

In the near future, we plan to engage with organisations such as NATS, *Civil Aviation Authority* (CAA), etc. to discuss the role of formal methods in building trustworthy complex systems, using our case study as an illustrative example. In the longer term, we want to investigate how our approach (in particular the policing function) could support certification standards such as DO-178C [11]. Our experience from this project is that deploying formal methods in a way that focuses on safety policing is feasible and makes the most of their strength. This suggests a promising future direction both for deployment of existing formal methods and treatment of current limitations.

Acknowledgment

The work in this paper is supported by the ASUR Programme project 1014 C6 PH1 104.

a) Disclaimer.: This document is an overview of MOD sponsored research and is released to inform projects that include safety-critical or safety-related software. The information contained in this document should not be interpreted as representing the views of the MOD, nor should it be assumed that it reflects any current or future MOD policy. The information cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

References

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

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

[3] 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.

[4] 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: http://dx.doi.org/10.1007/978-3-642-39698-4_5

[5] K. Robinson, "Concise summary of the Event-B mathematical toolkit," 2014, http://wiki.event-b.org/images/EventB-Summary.pdf.

[6] G. Babin, Y. A. Ameur, N. K. Singh, and M. Pantel, "Handling continuous functions in hybrid systems reconfigurations: A formal Event-B development," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, ser. Lecture Notes in Computer Science, M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró, Eds., vol. 9675. Springer, 2016, pp. 290–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-33600-8_23

[7] J. M. Rushby, "New challenges in certification for aircraft software," in *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds. ACM, 2011, pp. 211–218. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038675

[8] P. Caseley, "Claims and architectures to rationate on automatic and autonomous functions," in *11th International Conference on System Safety and Cyber-Security (SSCS 2016)*, 2016.

[9] N. K. Singh, Y. A. Ameur, M. Pantel, A. Dieumegard, and E. Jenn, "Stepwise formal modeling and verification of self-adaptive systems with Event-B. the automatic rover protection case study," in *21st International Conference on Engineering of Complex Computer Systems, ICECCS 2016, Dubai, United Arab Emirates, November 6-8, 2016*, H. Wang and M. Mokhtari, Eds. IEEE Computer Society, 2016, pp. 43–52. [Online]. Available: https://doi.org/10.1109/ICECCS.2016.015

[10] S. Boldo, C. Lelay, and G. Melquiond, "Formalization of real analysis: a survey of proof assistants and libraries," *Mathematical Structures in Computer Science*, vol. 26, no. 7, pp. 1196–1233, 2016.

[11] Requirements and Technical Concepts for Aviation (RTCA), *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, Dec. 2011.