International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-021-00604-4

GENERAL

Regular Paper

A domain-specific language to design false data injection tests for air traffic control systems

Alexandre Vernotte[1] · Aymeric Cretin[1] · Bruno Legeard[1,2] · Fabien Peureux[1,2]

Accepted: 4 January 2021

Abstract
The ADS-B—automatic dependent surveillance-broadcast—technology requires aircraft to broadcast their position and velocity periodically. As compared to legacy radar technologies, coupled with alarming cyber security issues (the ADS-B protocol provides no encryption nor identification), the reliance on aircraft to communicate this surveillance information exposes air transport to new cyber security threats, and especially to FDIAs—false data injection attacks—where an attacker modifies, blocks, or emits fake ADS-B messages to dupe controllers and surveillance systems. This paper is part of an ongoing research initiative toward the generation of FDIA test scenarios and focuses on supporting the test design activity, i.e., supporting ATC experts to meticulously craft test cases in order to assess the resilience of surveillance systems against FDIAs. To achieve this goal, we propose a complete and powerful domain-specific language (DSL), close to natural language, that provides a large expressiveness to support ATC business experts in creating FDIA's test scenarios. We demonstrate the design capabilities of this approach and its productivity gain with respect to manually creating the FDIAs test scenarios.

Keywords Domain-specific language · Air traffic control · ADS-B protocol · Cyber security · False data injection attacks · Automated test generation

1 Introduction

The world of air transport is facing new challenges as the traffic load keeps growing steadily.[1] With an increasingly congested airspace, air traffic control (ATC) needs surveillance technologies that can support the increasing constraints in terms of simultaneously handled aircraft as well as positioning accuracy. The automatic dependent surveillance-broadcast (ADS-B) protocol is currently being rolled out in an effort to reduce costs and improve aircraft position accuracy [46]. Communication via ADS-B consists of participants broadcasting their current position and other information periodically (a.k.a. a beacon) in an unencrypted message [32].

This technology embodies the shift from independent and non-cooperative surveillance technologies, historically used for aircraft surveillance, to dependent and cooperative technologies. In this new context, ground stations need aircraft to cooperate and are dependent on aircraft's global navigation satellite system (GNSS) capabilities to determine their position. These fundamental technological changes have rendered the ATC community unable to foresee the new emerging threats related to cyber security. The ADS-B protocol was not designed with security in mind since securing ADS-B communication was not a high priority during its specification. As a consequence, anyone with the right equipment can listen and emit freely. For instance, there is a market for equipping private aircraft with ADS-B transponders using

[1] http://www.boeing.com/commercial/market/current-market-outlook-2017/.

✉ Aymeric Cretin
aymeric.cretin@femto-st.fr

Alexandre Vernotte
alexandre.vernotte@femto-st.fr

Bruno Legeard
bruno.legeard@femto-st.fr; bruno.legeard@smartesting.com

Fabien Peureux
fabien.peureux@femto-st.fr;
fabien.peureux@smartesting.com

[1] FEMTO-ST inst., Univ. Bourgogne Franche-Comté, CNRS, 16 Route de Gray, 25030 Besançon, cedex, France

[2] Smartesting Solutions & Services, Besançon, France

Published online: 22 February 2021

a smartphone and a dongle.[2] The complete freedom of ADS-B both in emission and reception makes it vulnerable to spoofing, and more precisely to a class of attack called FDIA—false data injection attack—whose purpose is to covertly emit meticulously crafted fake surveillance messages in order to dupe ATC controllers into thinking, for instance, that some aircraft is dangerously approaching a building, while in reality it is flying normally.

Although it is not the only means for aircraft tracking—other protocols like controller-pilot data link communications (CPDLC) or aircraft communication addressing and reporting system (ACARS) are also used in conjunction of radar technologies—ADS-B plays a central role in the current shift regarding how aircraft positions are obtained (initially from radar systems now relying on GNSS [9]). So central in fact that it has become a mandatory brick of air traffic surveillance, and any observed problem will ground all aircraft in the area.[3] Hence, there is a strong need to improve its overall security. Nevertheless, because of the inherent properties of the protocol, current solutions for securing ADS-B communications are only partial or involve an unbearable cost [48]. ATC should be made more secure by strengthening its logic instead, but the ability to differentiate attacks from real situations still remains a challenge that is being tackled by the ATC community. Indeed, multiple integrity checks or detection approaches are under study or being rolled out [17]. Those solutions are new and need to be deeply tested, and testing approaches for this purpose are yet to be created.

The contribution presented in this paper is part of an ongoing research initiative about FDIA testing that ultimately led to the creation of a testing framework called *FDI-T* [13] (false data injection testing framework). This framework allows ATC experts to define FDIAs by creating, modifying and deleting recorded legitimate ADS-B messages in a fruitful, scalable and productive manner. The generated test scenarios can be executed on ATC systems in order to evaluate their resilience against potential security and safety anomalies related to FDIAs. The objective of this tooled approach is thus twofold: first it aims to assess the current cyber security of ATC systems, and second it makes it possible to periodically measure expected cyber security improvements.

The present paper proposes a new FDIA scenario design approach, on which the FDI-T framework is based, that relies on a Domain Specific Language (DSL). The DSL is close to natural language and makes it quite intuitive for ATC experts to express scenarios without programming skills required. Scenarios written using the DSL are generic, meaning they can be applied to any recording. Moreover, the DSL has a large expressiveness allowing for the design of all types of scenarios, easily capturing their complexity and their necessary precision. All these features are demonstrated in a dedicated section with the purpose of validating this new FDIA scenario design approach.

Layout of the paper. Section 2 briefly provides basis for common concepts and current practices regarding air traffic surveillance as well as the key aspects to test such systems, especially regarding test scenarios based on FDIAs. It also introduces the FDI-T testing approach to perform FDIAs on ADS-B messages and describes the automated process supporting the method. Section 3 introduces related work and gives the theoretical foundation of DSL usage and design. Sections 4 to 8 detail, step by step, the dedicated DSL defined within the approach to ease the design of FDIA scenarios. These sections, respectively, describe the expected added value, the domain, the design, the implementation and the validation of the proposed DSL. Finally, Sect. 9 recaps the major contributions of the paper and suggests directions for future work.

2 Business and technical background

This section presents background on air traffic control (ATC) and false data injection attacks (FDIA). Based on these overall preliminaries, the need to test ATC systems against FDIA scenarios and the current state of the practice are also introduced. Finally, this background enables to define the business objectives and theoretical challenges that are addressed by the contributions presented in the rest of the paper.

2.1 Air traffic surveillance

Surveillance is a distinctly complex and critical process whose goal is to detect, localize and identify all active aircraft. Surveillance supports air traffic control (ATC), whose task is to ensure that all aircraft are safely separated (e.g., 3 nautical miles (NM) separation when approaching an airport, 50 NM in oceanic en-routes without surveillance means). Surveillance systems detect aircraft and send detailed information to ATC systems, hence allowing air traffic controllers (ATCo) to safely guide aircraft. While in low-density areas aircraft separation can be fully ensured via manual position reporting (voice or text-based) between the pilot and the ATCo, in highly dense areas, which are increasing in number, the control of air traffic would not be feasible if it was not automated by surveillance systems.

ATC can be divided into three divisions depending roughly on the aircraft's proximity with the airport of departure/landing. Also, there is a division of surveillance into three distinct parts (see Fig. 1):

[2] https://www.uavionix.com/products/skybeacon/.

[3] https://hackaday.com/2019/06/09/gps-and-ads-b-problems-cause-cancelled-flights/.

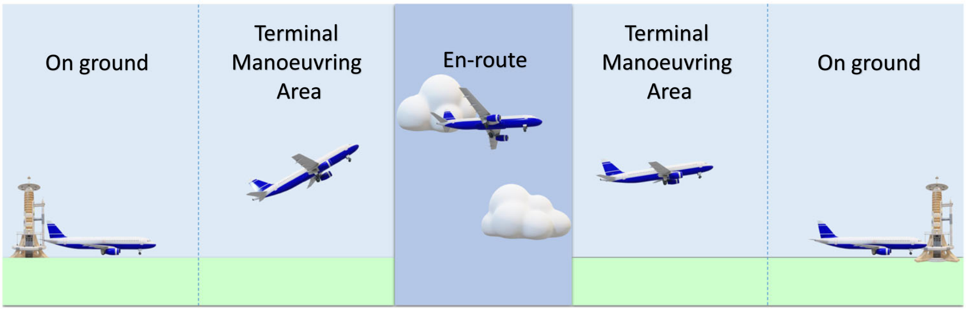


Fig. 1 ATC Surveillance Divisions

- On Ground surveillance: aircraft that are moving on ground at the airport in various areas such as taxiways, inactive runways, and holding areas. This division is handled by ATCos in the control tower, for which airport surface surveillance technologies provide a precise situation picture of the airport.
- Terminal maneuvering area (TMA) surveillance: aircraft that are about to takeoff or land. This division is handled by ATCos in the terminal/approach control center. TMA surveillance technologies assist air traffic controllers in their decisions.
- En-route surveillance: aircraft that are flying at a medium or high altitude, hence relatively far away from airports. This division is handled by ATCos in the area control center. En-route surveillance aims controllers to have a precise recognized air picture of the airspace sector they are in charge of.

Surveillance systems rely on several technologies, historically radar and recently others, in order to track aircraft in all three surveillance divisions. We discuss below the four main technologies.

Primary Surveillance Radar (PSR) This is the first generation of radars, developed in the 1940s, still in use today for approach and sometimes en-route surveillance areas. PSR sensors rely on a signal-based detection approach: a rotating antenna radiates an electromagnetic pulse on a low GHz band within a large portion of space that is reflected by targets within that space, like an echo, thereafter received back by the radar sensor. Target position and velocity are continuously determined by measuring the bearing (the direction the reflection comes from) and round trip time of each pulse [45]. PSR is a non-cooperative independent surveillance technology since distant aircraft do not aid to their surveillance (besides existing physically).

Non-cooperative independent surveillance has the advantage of detecting all aircraft, a quite interesting capability in times of war for instance. For this reason, this type of surveillance will continue to be required in many dense areas for obvious reasons of security as well as completeness of surveillance information (i.e., ensure a full surveillance picture). It also yields a high data integrity level, as it is quite hard for miscreants to temper with pulses covertly. That being said, there are multiple drawbacks to using this technology. While it will detect all aircraft regardless of their equipment (i.e., their will for cooperation), a PSR sensor will also detect all buildings, clouds, mountains, and so on, which hence generate a sensitive amount of noise. Received echos must be easily distinguishable, which involves very high radiation power. Moreover, PSR does not identify aircraft nor provide aircraft altitude (only 2D position and velocity), it has a limited range and a low update rate.

Secondary Surveillance Radar (SSR) This is the second generation of radars, used for approach and en-route surveillance. Contrary to PSR, SSR is a cooperative technology: in the form of digital messages, ground stations broadcast interrogations of aircraft transponders (on the 1030MHz frequency) and the transponders reply with the requested information (on the 1090 MHz frequency) [52]. Similarly to PSR, SSR is an independent surveillance technology as the position and velocity of aircraft is determined using the antenna's bearing and the digital message round time trip. Historically, there were two interrogation modes, called mode A and mode C for identification and altitude, respectively. These are being substituted with mode S (for selective) that allows for selective interrogations of single aircraft. It thus relieves the saturated 1090-MHz channel by avoiding multiple "all-call" interrogations when there are multiple radars in high traffic areas: these situations were indeed prone to mutual interference (garbling) [4]. For this, each aircraft was given its own unique worldwide transponder ID (ICAO 24-bit aircraft address) for selective interrogation, which replaces the old four digits identification "squawk" codes used in mode A interrogations which were too prone for identity collision (note that squawk codes are still in use today as some aircraft still operates under mode A/C. Squawk codes can also be used to communicate abnormal situations such as aircraft hijacking (7500) or global alert (7700)). Mode S also enables more information to be exchanged such as aircraft intent (e.g., selected altitude) or autopilot modes.

As compared to PSR, SSR is less sensitive to interference and covers a larger range. Nonetheless, it still has high latency and still has a low update rate (1 message every 6 to 12 seconds depending on the antenna's range, i.e., its rotating speed). Its data integrity level is also not as high as PSR, since it relies on information exchange. SSR provides aircraft position accuracy of 1 to 2 NM that, given the previously stated update rate, leads to a 3 NM separation between aircraft [55].

ADS-B Communication via ADS-B consists of aircraft using a global navigation satellite system (GNSS) to determine their position and broadcasting it periodically without solicitation (a.k.a beacons or squitters), along with other information obtained from on-board systems such as altitude, ground speed, aircraft identity, and heading. Ground stations pick up on the squitters, process them and send the information out to the ATC system. The ADS-B data link is generally carried on the 1090MHz Extended Squitter (1090ES), the same frequency used by mode S, although there is a new data link standard (UAT—universal access transceiver) dedicated to protocols such as ADS-B, but it requires new hardware and is not very common at the moment. ADS-B is therefore a cooperative (aircraft need a transponder) and dependent (on aircraft data) surveillance technology, which constitutes a fundamental change in ATC. It means for instance that not only ground stations with antennas positioned at the right angle and direction can receive position information. Aircraft can now receive squitters from other aircraft, which notably improves cockpit situational awareness as well as collision avoidance. For instance, the second generation of the traffic alert and collision avoidance system (TCAS-II) is based on ADS-B data.

Its introduction also provides controllers with improved situational awareness of aircraft positions in en-route and TMA airspaces, and especially in NRAs (non-radar areas). It theoretically gives the possibility of applying much smaller separation minima than what is presently used with current procedures (procedural separation) [1]. Indeed, ADS-B offers position accuracy of 0.05 NM and velocity accuracy of 19.4 NM/h, with updates once to twice second. Concretely, ADS-B performance requirements were designed to allow an aircraft lateral separation from 90 to 20 NM and longitudinal separation from 80 to 5 NM in NRAs, and 5 to 3 NM in covered areas [55]. ADS-B has the advantage of being a much cheaper technology as it has minimal infrastructure requirements. For instance, an ADS-B receiver can easily be bought online for a few hundreds euros.[4] As mentioned in the previous paragraph, ADS-B has a much greater accuracy and update rate, with a smaller latency. The major drawback of the technology lies in its lack of encryption and authentication, which is discussed in Sect. 2.2.

Multilateration (MLAT) MLAT is a cooperative independent surveillance technique used for airport surface surveillance, approach surveillance, as well as en-route surveillance with the so-called wide area multilateration (WAM). Multilateration is different from other technologies in the sense that it is not a separate protocol but rather a type of mathematical techniques that use signals from other technologies (SSR, ADS-B), received by multiple receivers, in order to measure their time differences of arrival (also called hyperbolic positioning [11]), or the time sum of arrival (also called elliptic positioning [42]). With enough measurements from separate sources (at least 4), it is possible to determine the origin of the signal and consequently the aircraft's position [43]. Compared to PSR and SSR, the technique has a much higher update rate, reliability, flexibility, stability and accuracy. It bears a low cost when it comes to ground equipment as well as aircraft equipment. However, it is quite complex to manage as it is a distributed system with high synchronization constraints, which can become costly if deployed for large regions.

Regardless of the technology, the goal of surveillance sensors is to capture air traffic and send reports to air traffic services (ATS), typically used by ATCos, so that they can ensure that aircraft are within their designated airspace. Before reaching ATSs, however, surveillance reports are first sent to a surveillance data processing and distribution (SDPD) system. The primary function of an SDPD system is to analyze the surveillance reports and fuse them into a system target track and serve such tracks to subscribed Users (i.e., ATS). The major and most recent SDPD system is EUROCONTROL's ATM Surveillance Tracker and Server (ARTAS), which is being rolled out at the moment in Europe.[5]

A simplified overview of the EUROCONTROL surveillance chain is depicted in Fig. 2. SDPD system receives surveillance reports from multiple sources and fuse them to produce system track data that is fed to Air Traffic Services. The data exchange between all the entities of this chain (sensor data reports, SDPD system, various ATSs) is made possible thanks to ASTERIX (All Purpose Structured EUROCONTROL Surveillance Information Exchange), an extensible application protocol created by EUROCONTROL with the goal of facilitating the exchange of harmonized surveillance information among surveillance and automation systems, within and between countries.[6]

[4] https://flywithscout.com.

[5] https://www.eurocontrol.int/news/artas-surveillance-tracker-programme-goes-further.

[6] https://www.eurocontrol.int/services/asterix.

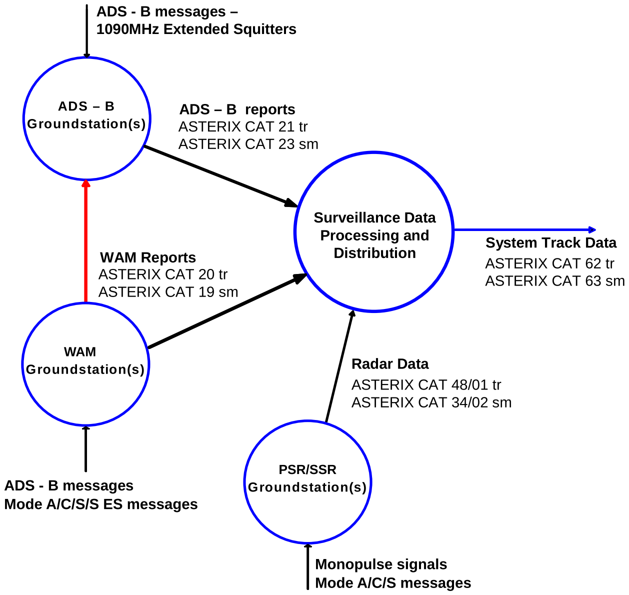


Fig. 2 EUROCONTROL Surveillance Chain Overview

2.2 False data injection attacks

Extensive research can be found in the literature discussing the cyber security of surveillance communications [44,48,55,57]. The progressive shift from independent and non-cooperative technologies (PSR) to dependent and cooperative technologies (ADS-B) has created a strong reliance on external entities (aircraft, GNSS) to estimate aircraft state. This reliance, along with the introduction of air-to-ground data links via modes A/C/S and the broadcast nature of ADS-B, has brought alarming cyber security issues.

None of the aforementioned data link-based surveillance technologies (SSR's mode A/C/S, ADS-B and MLAT) features any kind of encryption or authentication. Which means anyone with the right equipment can emit false data, for instance, to create a fake aircraft. This was not much of an issue with SSR, as the highly directional SSR beam pattern makes it difficult for the attacker to inject a fake aircraft with an arbitrary bearing or altitude. With ADS-B, however, omnidirectional antennas afford attackers flexibility in orientation and proximity [55]. This is because aircraft position is determined on-board, i.e., without independent system validation. The introduction of ADS-B has therefore enabled a class of attack referred to as *false data injection attacks* (FDIAs).

FDIAs were initially introduced in the domain of wireless sensor networks [28]. A wireless sensor network is composed of a set of nodes (i.e., sensors) that send data report to one or several ground stations. Ground stations process the reports to reach a consensus about the current state of the monitored system. A typical scenario consists of an attacker who first penetrates the sensor network, usually by compromising one or several nodes, and thereafter injects false data reports to be delivered to the base stations. This can lead to the production of false alarms, the waste of valuable network resources, or even physical damage. Active research regarding FDIAs has been conducted in the power sector, mainly against smart grid state estimators [15,27]. It shows that these attacks may lead to power blackouts but can also disrupt electricity markets [56], despite several integrity checks.

FDIAs also exist in the domain of air traffic surveillance. There is indeed an analogy between wireless sensor networks (as in, e.g., smart grids) and surveillance systems. Because air traffic surveillance relies on the information provided by aircraft's transponders to ground stations, aircraft transponders are equivalent to nodes from a wireless network, and ground stations are equivalent to base stations. A second analogy, albeit at a larger scale, can also be made: ground stations as nodes and SDPD systems as base stations. This paper proposes an FDIA testing solution covering both contexts.

Performing FDIAs on surveillance communications is no simple task: it requires a deep understanding of the system, its protocol(s) and its logic, to covertly alter (by injecting falsified squitters and deleting genuine ones) the consensus reached by the ground station and SDPD system regarding the recognized air picture. These attacks are much more complex to achieve than, e.g., jamming, because the logic of the communication flow must be preserved and the falsified data must go unnoticed.

The means of the attacker to conduct FDIAs against ADS-B communications have already been detailed in previous work [30,47]. Considering the attacker has the necessary equipment, he can perform three malicious basic operations:

(i) *Message injection* which consists of emitting non-legitimate but well-formed ADS-B messages.
(ii) *Message deletion* which consists of physically deleting targeted legitimate messages using destructive or constructive interference. It should be noted that message deletion may not be mistaken for jamming, as jamming blocks all communications, whereas message deletion drops selected messages only.
(iii) *Message modification* which consists of modifying targeted legitimate messages using overshadowing, bit-flipping or combinations of message deletion and message injection.

The above three techniques allow for the execution of several attack scenarios [44] that can be categorized in a taxonomy:

- **Ghost Aircraft Injection.** The goal is to create a non-existing aircraft by broadcasting fake ADS-B messages on the communication channel.
- **Ghost Aircraft Flooding.** This attack is similar to the first one but consists of injecting multiple aircraft simultaneously with the goal of saturating the recognized air

picture and thus generates a denial of service of the controller's surveillance system.

- **Virtual Trajectory Modification.** Using either message modification or a combination of message injection and deletion, the goal of this attack is to modify the trajectory of an aircraft.
- **False Alarm Attack.** Based on the same techniques as the previous attack, the goal is to modify the messages of an aircraft in order to indicate a fake alarm. A typical example would be modifying the squawk code to 7500, indicating the aircraft has been hijacked.
- **Aircraft Disappearance.** Deleting all messages emitted by an aircraft can lead to the failure of collision avoidance systems and ground sensors confusion. It could also force the aircraft under attack to land for safety check.
- **Aircraft Spoofing.** This scenario consists of spoofing the ICAO number of an aircraft through message deletion and injection. This could allow an enemy aircraft to pass for a friendly one and reduce causes for alarm when picked up by PSR. Designing, implementing and executing test scenarios to assess the behavior of ATC Systems in case of ADS-B messages alterations is the primary goal of our work.

It is of the utmost importance that none of the scenarios represent a real threat to such a critical infrastructure with human lives on the line. However, because of the inherent properties of the ADS-B protocol, current solutions for securing ADS-B communications are only partial or involve an unbearable cost [48].

In light of these features, ATC authorities such as EUROCONTROL opted to add a security layer on top of ADS-B which was developed as part of project SecAR [17]. It consists of continuous data integrity checks of ADS-B squitters, often relying on other sources of data (SSR/MLAT). The main integrity checks are listed and explained below:

- ADS-B message validation via WAM integration: information contained within ADS-B messages is compared to the corresponding value in a WAM reference report with closest time of applicability. If the difference is below a certain threshold, test result is set to *Valid*; otherwise, it is set to *Not Valid*.
- Behavioral analysis of targets: aircraft properties that are reported in ADS-B messages such as velocity, acceleration, altitude, heading are compared with corresponding values calculated from successive horizontal position updates. If the difference is below a certain threshold, test result is set to *Consistent*, otherwise it is set to *Inconsistent*.
- Time difference of arrival test: a network of ADS-B ground stations exchanges ADS-B information on aircraft detected in the common average volume. All ground stations received ADS-B messages with a time of arrival value and send the obtained value to the other stations along with the time of arrival. Then validation is performed and the aircraft's position is calculated similarly to multilateration. If the difference between calculated and reported values is below a certain threshold, the message is set to *Valid*, otherwise it is set to *Not Valid*. If the comparison cannot be done (e.g., ground stations are too far apart), then the message is set to *Not Validated*.
- Power measurement versus range correlation: the closer a signal is emitted from an antenna, the stronger it is. Based on this observation, it is possible to estimate the distance between aircraft and antenna based on how powerful the signal is at reception. Therefore, it should be possible to detect FDIAs where the attacker is close to the receiving antenna.
- Angle of arrival measurement: it is possible to estimate the angle at which ADS-B messages have been received, which gives a rough estimation of the position of emitting aircraft. Measuring this angle may help detecting FDIAs that put aircraft at a completely different angle than what measures suggest.
- Multi-sensor data fusion consistency check: ADS-B messages flagged as *Not Validated* and *Not Valid* are processed by the SDPD system and can result in the message being specifically flagged to the ATCo or in the dismiss of the message. Each integrity check has an appropriate weighing dependent on how reliant it is.

2.3 The false data injection testing framework

It is critical to make sure that SecAR's integrity checks are properly and thoroughly tested. Such a testing campaign needs large sets of test data in the form of air traffic recordings, where each recording is a test scenario, i.e., it presents a situation that should trigger one, several, all or none of the integrity checks.

The contribution presented in this paper is part of an ongoing research initiative about FDIA testing. This work ultimately led to the creation of a testing framework called *FDI-T* [13] (false data injection testing framework) which is developed in partnership with two companies: Smartesting[7] and Kereval.[8] The framework allows ATC experts to design FDIA scenarios, with the objective of altering existing recordings of air traffic surveillance communications by creating, modifying and deleting recorded legitimate ADS-B messages in a fruitful, scalable and productive manner. The altered recordings are then played back (with respect to time requirements) onto real ATC systems, to simulate an attacker tampering with the surveillance communica-

[7] https://www.smartesting.com.

[8] https://www.kereval.com/.

tion flow. The approach allows for the evaluation the ATC systems' resilience against potential security and safety anomalies, and more precisely to FDIAs. Moreover, it makes it possible to periodically measure expected cyber security improvements.

The architecture of the FDI-T framework, depicted in Fig. 3, is composed of five modules:

① **Data acquisition**. The objective is to collect legitimate mode S messages in Beast[9] or SBS[10] formats, obtained either from the Internet or a mode S receiver. Data take the form of a recording, i.e., a sequence of surveillance messages ordered by reception time.

② **Scenario Design**. The ATC expert defines alteration scenarios to be applied on a recording obtained via the data acquisition module. Alteration scenarios have various parameters, such as a time window, a list of targeted aircraft, triggering conditions, an others parameters related to the alteration's type. Once designed, alteration scenarios are parsed and translated into a set of alteration directives, which is the output of the module (an alteration directive is a small modification of the initial recording, usually doable by hand. This scenario design module is textual-based. It relies on a Domain Specific Language, and is precisely the object of the article.

③ **Radar sensors network Simulator**. As an FDIA is the compromising of one or several sensors of the network (see Sect. 2.2), FDI-T allows users to model a network of radar sensors to simulate a multi-sensor attack. A sensor has a location and different parameters related to its type (i.e., SBS, Beast, SSR, PSR, mode S and SDPD), such as its range, bearing, altitude. Each sensor is associated solely with the surveillance messages from the initial recording that were sent within its range. Therefore, the simulator outputs a sub-recording per modeled sensor. Users can apply alteration scenarios to one, several, or all sensors. Only the sensors onto which an alteration scenario is applied have their sub-recording altered, so that some sensors receive unaltered data, while others receive altered data. Such simulations are particularly relevant to test the behavior of an SDPD as its purpose is to aggregate multiple surveillance sources.

④ **Alteration Engine**. This module takes as input a set of original sub-recordings, a set of alteration directives, and a correspondence matrix that defines which alteration scenario should be applied on a given sub-recording. It then produces altered sub-recordings in the ATC system input format. It is possible to output altered recordings in six formats: SBS-3, BEAST, ATX20 (ASTERIX WAM reports), ATX21 (ASTERIX ADS-B reports), ATX48 (ASTERIX PSR/SSR reports), and finally ATX63 (ASTERIX System track toward ATC controllers). This module is described in a precise way in [14], toward which we refer interested readers.

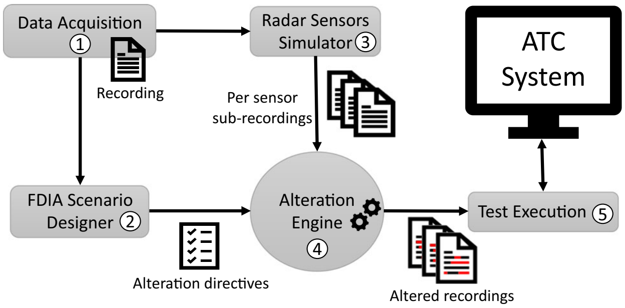


Fig. 3 FDI-T framework architecture

⑤ **Execution Engine**. The obtained altered air traffic sub-recordings are fed to the ATC system *as if* it was receiving live surveillance messages.

The present paper focuses on the scenario design module of the FDI-T framework by proposing a novel design approach that relies on a DSL. Note that the details of the FDI-T framework regarding other activities (i.e., data acquisition, radar sensors network simulation, test execution) are out of the scope of this paper, as they have been presented previously [13,14].

2.4 Research objective and questions

The design of alteration scenarios in the ATC domain can be very complex. This requires at the same time altering or creating false data (e.g., creating false aircraft) while ensuring the consistency of all data. It is also necessary to ensure the widest possible coverage of the types of FDIA attacks known in the literature (and presented above). The creation of a DSL adapted to this work of designing alteration scenarios thus aims to considerably increase the feasibility of in-depth tests of the resistance of ATC systems against FDIA attacks. We have defined the following research objective based on these observations:

Create a DSL-based alteration scenarios design approach that further automates data alteration of genuine ATC surveillance recordings to test FDIAs on ATC systems.

We believe that a DSL close to natural language would allow experts from the ATC domain to easily describe alteration scenarios, they might have intuition for, without programming skills needed. While being intuitive, this language shall give the necessary tools to design complex and precise scenarios, to reproduce all the existing scenarios from the literature, as well as ad hoc scenarios created on the spot. From the above research objective, we have identified 2 research questions that are described below (Sects. 8.1 and 8.2 refer to these questions to evaluate the approach).

[9] http://wiki.modesbeast.com/Mode-S_Beast:Contents.

[10] http://woodair.net/sbs/Article/Barebones42_Socket_Data.htm.

RQ1 **To what extent is it possible to design alteration scenarios in order to cover the taxonomy of attack?** Although they are all based on the same weakness—i.e., the injection of false data—each scenario of the taxonomy has its subtleties. For instance, it should be possible to easily alter aircraft's trajectory while ensuring realism of the modified trajectory throughout time. In turn, if realism (including computation of each latitude/longitude coordinates) is abstracted from the user in exchange of DSL primitives (e.g., by defining waypoints and time of passage), then realism and the computation of a new trajectory shall be done algorithmically. The DSL must integrate what makes the protocol and its domain of application specific, in the form of grammar primitives that assist users in designing scenarios from the taxonomy.

RQ2 **To what extent the use of a DSL can facilitate the creation of FDIA's test scenarios and reduce design effort?** The alteration engine of the FDI-T framework[11] is able to process automatically a list of alteration directives (in an XML file) and apply the corresponding changes on a recording supplied as input [14]. For simple and straightforward alteration scenarios, it would be possible to create alteration directives manually. Actually, the current technique to test and validate machine learning models that detect anomalous ADS-B messages is to perform the alterations by hand or using ad-hoc scripts [2,20]. But for complex scenarios involving multiple targets and strong attack constraints (e.g., attack only when no KLM flights are flying above Strasbourg, and as soon as targeted aircraft are flying under a certain altitude), it is simply not feasible. The DSL comes in handy as it shall automate the thousands of micro-alterations that some attack scenarios require.

3 Related work and DSL approach foundation

To the best of our knowledge, there is no direct previously published work that addresses the topic of assessing the resistance of ATC systems against FDIA attacks. That is why, first we report on related work in the ATC domain, and second we discuss DSL-based approaches for test case definition in other domains of application. Finally, we introduce the foundation of DSL development, which drives the rest of the paper.

3.1 ATC-related testing and simulation approaches

A framework proposed by Barreto et. al. [6] allows for the simulation of an entire air traffic environment (aircraft, radio relay, network infrastructure, etc.). This is achieved by combining two COTS simulation software, MÄK VR-Forces[12] for physical simulation and EXata Cyber[13] for network communication simulation. Cyber attacks are performed either by relying on EXata Cyber's built-in attack library (but it contains generic attacks such as jamming/DoS), or by implementing a custom attack generator. The goal is to evaluate the attack impact on each component of the simulated environment. The approach can provide substantial information on how components react to an FDIA. Still, implementing all network behaviors of a scenario requires a lot of effort and the approach does not allow for the concretization of the simulated attacks on actual ATC software. In addition, there is no information regarding the attack generator or how complex FDIA attacks such as trajectory modification could be performed, as the only experimented attacks were denial-of-service.

In a similar fashion, Manesh et. al. study the effects false message injection on air traffic by developing aircraft traffic encounter using Hardware in the Loop testing platform [31]. An aircraft traffic encounter is a scenario in which minimum separation distance between aircraft may be violated, therefore requiring intervention to maintain or regain proper separation. The simulation testbed involved a real autopilot system mounted on an unmanned aerial system and a portable ground controller station. The goal is to observe the behavior of the pilot's traffic display when they fake an aircraft traffic encounter using ADS-B message injection. Results show that upon injection, traffic display indicates the intruder in red color, meaning that separation is violated and immediate maneuvering is required by the pilot to regain proper separation. The authors do not explain how fake messages are created to make for good simulation scenarios, nor if the resulting data are exploitable for FDIA testing.

Paielli presents a testing method to address air traffic conflict-resolution software (e.g., TSAFE [38]) that automatically generates simulated air traffic encounters. Conflict-resolution algorithms generate resolution maneuvers (typically, altitude and heading changes) when a loss of separation is predicted to occur within a certain time frame (e.g., 2 min for TSAFE). Test generation relies on a trajectory scripting language that generate a simulated reference trajectories and conflicts. Users specify multiple aircraft and, for each, define a trajectory composed of a series of segments (straight segments, turns, climbs, etc.). It also allows the user to generate an encounter or conflict by specifying additional parameters such as the path crossing angle and the minimum separation. The method is able to test for a variety of conflict types with expected altitude/heading maneuvers. It is how-

[11] The source code of the alteration engine is available on GitHub: https://github.com/aymeric-cr/sbs-generation.

[12] https://www.mak.com/products/simulate/vr-forces.

[13] https://www.scalable-networks.com/exatacyber.

ever unable to generate ADS-B traffic and therefore cannot be utilized to evaluate ATC (tracker) systems against FDIA.

3.2 DSL-based test case definition

There is a long history of DSLs being used to define test cases. Cucumber for instance is a testing tool using a DSL (Gherkin[14]) to define test cases using *Given-When-Then* rules. Silva et al. proposed the test specification language (TSL) [29] which produces test cases written in Gherkin when interpreted. This process leads on a transformation from TSL to another DSL (e.g., Gherkin) with a different level of abstraction. Following this method, an abstract test specification written in TSL is concretized into multiple test cases. Indeed, a test specification features entities containing abstract valued parameters and therefore shall be made concrete before it can be converted into test cases. Concretization consists of converting discrete values and enumeration values from a predefined value set into continuous parameter values, when applicable.

A similar approach is presented in the domain of the development of autonomous driving systems (ADS). Menzel et al. define a method to create test cases from scenarios with different levels of abstraction [34]. The highest level of abstraction is close to plain English and aims to be used by domain experts to define informal *hazardous scenarios*. The second level of abstraction formalizes the terms used in the first level such as the width of road's lanes, radius of road's curve, and vehicles' length. At this level a DSL can be used to write scenarios. However, similarly to abstract test specifications from Silva et. al.'s work, a scenario at this level of abstraction does not formalizes the terms with discrete values but rather with continuous ranges value or set of values. The third level of abstraction corresponds to the concretization of second level scenarios. Third-level scenarios are usually written in a machine readable format, e.g., XML or JSON. Relying on Menzele et al.'s work, Queiroz et al. propose a DSL, *GeoScenario* [41], based on an existing language from OpenStreetMap, similar to XML. This DSL allows for the creation of formal definitions of Menzel et al. scenarios, the goal of this DSL is to be processed by many ADS simulators in order to generate testing data corresponding to the specified scenarios.

In the domain of network performance testing, a DSL named coNCePTuaL [39] has been developed. This DSL allows for the design of network performance test cases that the author describes as patterns, but matching with the definition of a scenario. The result of a coNCePTuaL program is a set of abstract machines sending/receiving messages from/to each other. In both examples, a scenario (or pattern) describes the behaviors of domain-specific entities; then, these behaviors are generated at the data level by simulators able to interpret such patterns.

Several articles use, as a case study, DSLs in the ATC domain. These DSLs are able to define scenarios from landing to take-off [22], or scenarios of voice communication between pilots and controllers [50].

3.3 Motivating the use of a DSL for alteration scenario design

In a general manner, DSLs are high-level languages that provide abstractions, notations and constructs tailored to a particular application domain [54]. They offer substantial gains in expressiveness and usability (with corresponding gains in productivity and reduced maintenance costs) as opposed to general-purpose, multi-domain languages (e.g., for programming languages, like Java or C, and for modeling languages, like UML) [35]. Another benefit of DSLs is that they enable to connect experts of the application domain and programmers through a unified terminology and semantics. There are many instances of successful development of DSLs in a variety of domains, such as DBMS management with SQL, web page formatting with HTML and CSS, software build automation with Make, circuit hardware description with CHDL.

DSLs also come with several drawbacks; the main one is the high development cost (implementation, creation of training material, language support, maintenance, etc.), especially for external DSLs, i.e., languages that have no reliance whatsoever with an existing general purpose language. Moreover, it is rarely obvious that a DSL is the appropriate solution to a problem, nor that this particular syntax and semantics are adequate. To help DSL developers in these crucial decisions, DSL development has been formalized as a set of 7 sequential activities [37]. These 7 activities of DSL development are *decision*, *domain analysis*, *design*, *implementation*, *testing*, *deployment* and *maintenance*. The first step, the decision, is about identifying whether the creation of a DSL constitutes an adequate solution to a given problem, and that this solution is worth the cost of development. During the next step, domain analysis, the goal is to model the application domain, i.e., defining the representation vocabulary to represent the objects and concepts of the domain, and defining the body of knowledge related to the vocabulary (the relationships and dependencies between the various entities). Based on the application domain model, the design activity consists in creating the language constructs and semantics. The implementation activity is about choosing the most suitable approach: should DSL constructs be interpreted, compiled into machine code, or translated into an existing base language? During the testing phase, the DSL is evaluated, e.g., in terms of productivity gains. Finally, the deployment activ-

[14] https://cucumber.io/docs/gherkin/.

ity defines the actual usage of the DSL, while maintenance is about updating the DSL to reflect new requirements.

The next sections of the paper, respectively, detail decision (Sect. 4), domain analysis (Sect. 5), design activities conducted to create the proposed DSL (Sect. 6), implementation (Sect. 7) and finally testing activities (Sect. 8). It should be noted that the deployment and maintenance activities are not discussed since they are deemed out of the scope of this project.

4 DSL decision

It can be quite complex to decide in favor of a new DSL, since it is hardly evident that a DSL might be fruitful, and even if it is, that it would be worth the high development and maintenance cost. Decision patterns have been identified to aid in the decision process [35], such as facilitating system configuration, eliminating repetitive tasks. In the subsequent paragraphs, we first define the problem we are trying to solve, and then we rely on the decision patterns to motive our choice of going for a new DSL.

An *FDIA scenario*, as viewed within the FDI-T framework, is a test case that consists of:

- a *recording* onto which alterations are performed;
- one or more *alteration scenarios*;
- one or more *radar sensors*;
- for each alteration scenario, the targeted sensors.

A *recording* is defined as a set of aircraft squitters, each being timestamped and filled with information about aircraft's position, velocity, status or identity. Recordings can be obtained from online providers (e.g., the OpenSky Network[15]), or simply by eavesdropping using an antenna (e.g., an SBS-3 antenna).
An *alteration scenario* consists of the definition of one or more *alteration directives*.
An *alteration directive* is a low-level simple modification instruction to be applied on a recording. It is given by:

- a time window;
- an operation type (creation, deletion, modification);
- an aircraft identifier (the target);
- a set of key-value pairs related to the operation type (e.g., for a modification, a pair is composed of an aircraft property such as altitude and its new value).

For simple and straightforward alteration scenarios, e.g., a false alarm attack that is not triggered by any particular event and that has no particular aircraft targeting conditions, creating the corresponding list of alteration directives by hand or using a script would be acceptable. The alteration engine of the FDI-T framework is able to process automatically a list of alteration directives (in an XML file) and apply the corresponding changes on a recording supplied as input [14]. However, most scenarios are not as straightforward; they involve potentially applying several types of alterations, each having complex triggering events, and fine aircraft targeting which conditions are based on the very nature of the recording.

This can translate into hundreds, thousands of alteration directives, each meticulously crafted with specific alteration properties. Moreover, testing campaigns often involve executing the same test cases with different test input data to attain a certain coverage criteria (i.e., similarly to traditional software testing techniques). It is therefore necessary to automatically derive alteration directives from an alteration scenario, and there is a need for an appropriate, user-friendly albeit formal way to design alteration scenarios.

ATC experts reason in terms of recognized air picture (RAP) statuses and aircraft-related events. They consider aircraft based on their properties (flight altitude, ground speed, route, etc.), and they envision potential security threats as high-level surveillance data alterations. FDIA scenarios are defined informally and written in natural language.

Based on the above observation, it appears that a DSL would be a great candidate as a design means for alteration scenarios. The following decision patterns further strengthen this idea:

- *User-friendly notation*: instead of having experts describe relevant FDIA scenarios in natural language for the software development team to translate into test cases, a DSL would make it possible to generate the scenarios automatically. Using a syntax close to what experts are used to, natural language. Using an ATC-related terminology and definition rules close to natural language would allow ATC experts, who usually have little to no programming skills, to easily express the scenarios they have intuition for, and have them automatically applied to a recording.
- *Task automation*: manual definition of alteration directives is repetitive and time consuming, and may require an in-depth pre-analysis of the recording in order to determine which aircraft to alter, and when. A DSL would allow for the automated production of hundreds of alteration directives, without any pre-analysis needed, as aircraft selection and alteration triggering would be part of the DSL.
- *System front-end*: a DSL would make a good front-end for ATC experts who want to translate informal FDIA scenarios into machine-readable scenarios.

[15] https://opensky-network.org/.

```
scen
alter all_planes satisfying ”filt”
at 10 seconds triggered_by”trigg”
with_values SQUAWK = 7700

filt
F(ALTITUDE > 33000 and G(ALTITUDE > 23500)

trigg
eval when (ALTITUDE > 32500)
```

Fig. 4 DSL example of false alarm FDIA scenario with aircraft filtering and alteration triggering

Let us illustrate what ATC experts would be capable of with such a DSL. For instance, to define a false alarm attack scenario, an expert would be writing the DSL-based scenario such as the one shown in Fig. 4.

The scenario ***scen*** specifies to alter the ADS-B messages of aircraft that satisfy filter ***filt***, starting ten seconds after the recording's first message, and according to alteration trigger ***trigg***. Filter ***filt*** targets aircraft that flew at least once at an altitude higher than 33000 feet and never descended under 23500 feet. Trigger ***trigg*** marks aircraft for alteration when their reported altitude is at 32500 feet and above. Finally, the alteration consists of changing aircraft squawk value to 7700.

Of course, this is a straightforward FDIA scenario. The next section reports on the domain analysis that allowed us to capture in the DSL all the relevant entities, datatypes, etc., thus providing us strong confidence that we would result in a grammatically correct, featureful, fully functional DSL.

5 DSL domain analysis

The goal of domain analysis is to properly identify the application domain and to define its scope, as well as gather appropriate domain knowledge, i.e., the various entities, datatypes, and their relationships, to be integrated into a coherent domain model. Various sources of information may be used for that purpose, such as consultation of domain experts, customer surveys, and technical documents.

Many methodologies for domain analysis have been developed over the last three decades, with various information extraction techniques and degrees of formality. Examples are DARE (domain analysis and reuse environment) [18], DSSA (domain-specific software architectures) [51], FODA (feature-oriented domain analysis) [23], and FAST (family-oriented abstractions, specification, and translation) [12]. These methodologies have shown to result in good language design, but they also have proven to be very complex and time consuming to implement. Moreover, clear guidelines on how to exploit the gathered information for the design phase are rarely provided. There are no mature and up-to-date tools to assist in the process, forcing DSL developers to do it manually, largely augmenting the probability to result in an incomplete and/or erroneous analysis due to the complexity of interrelated activities and work products [26]. As a consequence, the use of these methodologies is still very limited, and the domain analysis is often performed informally. In fact, for the vast majority of the DSLs found in the literature where the domain analysis is explicitly detailed, this activity is done informally [24]. This has the benefit of avoiding the high cost of respecting the canvas of the aforementioned methodologies, although the probability for incomplete/erroneous design remains consequent. Another methodology has gained attraction recently as a means for the domain analysis activity, which is ontology design [10,50].

The most common definition of an ontology is "a formal, explicit specification of a shared conceptualization" [49]. Formal because it is machine-readable, and explicit because the concepts and properties of the domain, as well as their relationships, are explicitly defined. A conceptualization is an overall view of the target domain, albeit abstract and simplified. It is shared because it captures consensual knowledge, understood and agreed upon by a group of experts of the target domain. Ontologies are used in a variety of domains, such as in artificial intelligence, but their prime usage is as a brick of the semantic web [8].

Previous work has demonstrated that ontology-based domain analysis (OBDA) is at least as efficient as more traditional domain analysis techniques [10]. The main benefit of OBDA is that ontology design is a well-established practice with standardized languages (RDF [25], OWL [33]) and numerous open-source and commercial tools to aid in the design process (Protégé,[16] Stardog Studio,[17] Topbraid Composer[18]). Thanks to the formal notations and supporting tools, ontology design has reasoning capabilities (e.g., with reasoners such as FaCT++ [53] or HermiT [19]), as well as querying capabilities. It is therefore possible to validate ontologies, which significantly contributes to reduce or prevent errors in DSL development. Finally, there is existing work on providing transformation rules to convert Ontology axioms into grammar rules [40], as well as tools capable of automating the transformation.

Therefore, the domain analysis of this DSL-based alteration scenario design approach has been captured in an ontology designed in OWL, as it is the most commonly used ontology language, and aided by Protégé, the reference open-source tool for ontology design.[19] Practically speaking, the ontology is a hierarchy of classes, a set of relationships

[16] https://protege.stanford.edu.

[17] https://www.stardog.com/studio.

[18] https://www.topquadrant.com/products/topbraid-composer/.

[19] The OWL file is available on GitHub at: https://github.com/aymeric-cr/dsl-scenario/blob/master/fdit-dsl-ontology.owl.

between classes, and a set of relationship between classes and data types. These are described in this paper using description logic [5].

As mentioned during the decision activity, the DSL should be a design means for ATC experts to formally define alteration scenarios that can be automatically transformed into a set of alteration directives. Below are the requirements for the DSL that were established based on experts consultations.

- **REQ-1.** The DSL should allow for the design of scenarios that cover the taxonomy of attacks. This includes the capability to set new values for aircraft properties as well as define trajectories by means of way-points and time of passage (as in alteration directives). Moreover, all alteration capabilities encompassed in alteration directives shall be also doable via the DSL. This includes the ability to combine different types of attack, such as aircraft disappearance and ghost aircraft creation for instance, to allow for elaborate alteration scenarios.
- **REQ-2.** The DSL should have the ability to define aircraft filtering criteria, based on aircraft dynamic and static properties, in order to precisely target a subset of aircraft onto which alter the messages.
- **REQ-3.** The DSL should have the ability to specify aircraft property-based events, related to one or multiple aircraft, to trigger and stop the alteration process, for each targeted aircraft individually.
- **REQ-4.** The DSL should be generic, in the sense that scenarios should be applicable to any recording.
- **REQ-5.** The DSL should have combinatorial capabilities. It involves the ability to define list of values assigned to variables, and to make references to the variables instead of hard-coded values. It allows to obtain one scenario per value in case of a single variable reference, and one scenario per combination of value in case of multiple variable references.

The next subsections present the classes that are modeled in the ontology, and, for each class, its various relationships with other classes. In addition, the complete class inheritance tree is depicted in "Appendix 1." Note that these classes are translated later into grammar production rules, rule alternatives, and terminals. The domain analysis activity has been conducted with the above requirements in mind, i.e., with the objective of fulfilling them. References to requirements are made throughout the subsections to indicate the motivation for the creation of classes and relationships. An exception occurs for REQ-4: there are no specific classes or relationship that can be tied to fulfill this requirement. Instead, it served as a guideline for the domain analysis process.

5.1 Scenario

We define (alteration) *Scenario* individuals as a composition of *Declaration* individuals (i.e., variables that contain lists and ranges of values) and (alteration) *Schema* individuals through object properties *hasDeclaration* and *hasSchemas*, respectively. Scenarios do not necessarily include declarations, i.e., *hasDeclaration* is optional. Conversely, members of class *Scenario* shall include at least one *Schema*. Moreover, scenarios are related to exactly one *Recording* through the *hasRecording* property. This information is expressed in description logic with the following axiom:

$$\begin{aligned}
Scenario \quad \sqsubseteq \quad & \forall\ hasSchemas.Schemas && (1)\\
& \sqcap \forall\ hasRecording.Recording && \\
& \sqcap \forall\ hasDeclaration.Declaration && \\
& \sqcap \exists\ hasSchemas.Schemas && (2)\\
& \sqcap\ \geq 1\ hasRecording.Recording && \\
& \sqcap\ \leq 1\ hasRecording.Recording &&
\end{aligned}$$

Sub-axiom *(1)* is a closure axiom consisting of a universal restriction that acts along the property *hasSchemas*, specifying that it can only be filled by instances of the *Schema* class. The same restrictions are defined for the *hasRecording* and *hasDeclarations* properties. These restrictions are necessary in OWL to enforce typing because of its open-world property: an individual is a member of all classes, unless explicitly stated. Formally speaking, we specify that the set of all individuals from class *Scenario* is included in the set of individuals who have their *hasSchemas* property either unset or solely filled with *Schema* individuals. The same reasoning applies for the $hasRecording$ property.

Sub-axiom *(2)* adds cardinality restrictions to the properties. Formally speaking, the set of all individuals from class *Scenario* is included in the set of individuals that have at least one schema and exactly one recording, through their respective properties. Therefore, part *(1)* restricts the "range" (i.e., type of the ownee) of the properties for when their "domain" (i.e., type of the owner) is a member of class *Scenario*, and part *(2)* adds cardinality restrictions.

Closure axioms are implied for the rest of the domain analysis, as they have been added systematically to all properties of all classes. The focus will be on cardinality restrictions only. Moreover, we consider that all classes from the same depth are disjoint, unless specified otherwise (e.g., class *TimeWindow* and its subclasses).

5.2 Schema

Alteration schemas are the backbone of the approach as they bridge the gap between high-level alteration objectives (scenarios) and low-level alteration instructions (directives). They differ from alteration directives as they are multi-targets and offer the ability to pick out aircraft based on their properties, can be triggered by events, and have combinatorial capabilities. We believe that schemas make for a well-suited intermediary entity between scenarios and directives.

A schema is a specification of one type of alteration (e.g., aircraft creation, deletion, trajectory modification, etc.), i.e., based on the taxonomy of attack, and mapping the different types of alteration directives from the alteration generation module [14], and according to requirement REQ-1. The various types of alteration are represented in the OWL model as subclasses of *Schema*:

- *AlterationSchema* consists of changing the properties of aircraft based on user-supplied values.
- *CreationSchema* creates a fake track from scratch, implying that fake messages are created and inserted into the target recording.
- *DeletionSchema* deletes targeted aircraft messages for a certain period of time.
- *TrajModSchema* modifies the initial trajectory of targeted aircraft given a time window and a sequence of waypoints.
- *SaturationSchema* generates chaos by duplicating aircraft multiple times and making each ghost aircraft slightly diverge from the original (in terms of trajectory), therefore making it appear that the aircraft is dividing itself into many different tracks.
- *ReplaySchema*, given a source, a target recording and a time window, extracts targeted aircraft from the source recording and adds them (i.e., their ADS-B squitters) into the target recording.

Because schemas necessarily belong to an alteration type, we consider the parent class *Schema* as abstract. This can be specified in description logic using a covering axiom, such as:

$$\begin{aligned} \textit{Schema} \quad \equiv \quad & AlterationSchema \sqcup CreationSchema \\ & \sqcup DeletionSchema \sqcup \; ReplaySchema \\ & \sqcup TrajModSchema \\ & \sqcup SaturationSchema \end{aligned}$$

The axiom specifies that the set of all members from class *Schema* is equivalent to the set of all members from either of *Schema*'s subclasses, therefore preventing any individual from being a member of class *Schema* without being a members of one of its subclasses as well. In the rest of the paper, it is implied that an axiom defined to a class also covers its subclasses.

5.3 Target

A target indicates whether the schema should be applied on a single aircraft or on all aircraft (which meet selection criteria if any, see next item). Therefore, the abstract class *Target* has been created, which is covered by its two subclasses, *AnyPlaneTarget* and *AllPlaneTarget*.

Not all types of schemas have a target: aircraft creation schemas do not need a target, as their purpose is only to create a single aircraft. Therefore, we create abstract class *TargetedSchema*, make it a subclass of class *Schema*, and make all Schemas subclasses, except *CreationSchema*, subclasses of *TargetedSchema*:

$$\begin{aligned} \textit{TargetedSchema} \quad \sqsubseteq \quad & Schema \\ & \sqcap \geq 1 \; hasTarget.Target \\ & \sqcap \leq 1 \; hasTarget.Target \end{aligned}$$

5.4 Time window

A time window consists of hard-coded start and/or end time values. Its purpose is to start and end alteration times for all targeted aircraft, based on the recording duration, as in alteration directives. *TimeWindow* members may have one or two positive integer properties to represent start and end times.

All types of schema may have a time window, but the relationship is optional. If not present, the specified alterations are performed throughout the recording. Nonetheless, schemas cannot be related to more than one time window:

$$\textit{Schema} \quad \sqsubseteq \quad \neg(> 1 \; hasTW.TimeWindow)$$

5.5 Waypoint

For schemas involving aircraft creation or trajectory modification, users should provide a trajectory as a sequence of waypoints. A waypoint is composed of coordinates, an altitude value and a time of passage. Hence, the following axioms apply to members of class *Coordinates*:

$$\begin{aligned} \textit{Coordinates} \quad \sqsubseteq \quad & \geq 1 \; hasLatitude.\mathbb{R} \\ & \sqcap \leq 1 \; hasLatitude.\mathbb{R} \\ & \sqcap \geq 1 \; hasLongitude.\mathbb{R} \\ & \sqcap \leq 1 \; hasLongitude.\mathbb{R} \end{aligned}$$

Similarly, the following axioms apply to members of class *Waypoint*:

$$\begin{aligned} \textit{Waypoint} \quad \sqsubseteq \quad & \sqcap \geq 1 \; hasCoordinates.Coordinates \\ & \sqcap \leq 1 \; hasCoordinates.Coordinates \\ & \sqcap \geq 1 \; hasAltitude.\mathbb{N}+ \\ & \sqcap \leq 1 \; hasAltitude.\mathbb{N}+ \\ & \sqcap \geq 1 \; hasTime.\mathbb{N}+ \\ & \sqcap \leq 1 \; hasTime.\mathbb{N}+ \end{aligned}$$

Two types of schema require users to provide a sequence of waypoints: trajectory modification schemas and creation schemas. Trajectory modification schemas require at least one waypoint:

$$\textit{TrajModSchema} \quad \sqsubseteq \quad \geq 1 \; hasWayPoint.Waypoint$$

A single waypoint is sufficient to alter an existing trajectory considering that a schema has a timewindow. Therefore, the aircraft position at the start and end time of the time window constitute two more waypoints, and a new trajectory can be drawn using the resulting three waypoints. For creation schemas, conversely, there is no existing waypoint; therefore, it is mandatory that users provide at least 2 waypoints to draw a trajectory:

$$CreationSchema \sqsubseteq > 2\ hasWayPoint.Waypoint$$

5.6 Assertion

Assertions allow for the assignment of test verdict. They define how the ATC system should react to the alteration, e.g., the type of alert or the message's content that the ATC system is supposed to output following the attack. A class named *Assertion* has been created along with the property *hasAssertion* which domain is *Schema* and range is *Assertion*.

There are two types of assertion, which makes for two subclasses: *StringAssertion* and *FileAssertion*. A string-based assertion specifies a string that should be matched in the response of the ATC system under test in order to validate the assertion. A file-based assertion specifies a path to a file containing executable code (typically, groovy files), and the output of the file execution constitutes the test verdict. This allows for more precise and complex test verdicts, as well as for the externalization of the test verdict assignment process.

5.7 Aircraft properties

Aircraft properties, as their name suggest, are the various properties belonging to a certain aircraft at a certain moment in time. Dynamic properties evolve over time, such as altitude and ground speed, while static properties remain unchanged throughout time, such as the ICAO and callsign. Aircraft properties are central to alteration schema design. They appear inside expressions (see next section), where their value is retrieved to be part of some calculus. They are also part of property evaluations (see Sect. 5.10), to do comparisons between aircraft property values and expression results. They also appear inside schema parameters (see Sect. 5.11) in order to affect them new values (e.g., increasing the aircraft's altitude by a 1000 feet).

There are three types of scope for aircraft properties. It aims to design expressions not only relating to a single aircraft, but also relating to a group of aircraft. This directly addresses requirement REQ-3 by triggering events based on properties of multiple aircraft. Below are the three property scopes:

- *Aircraft-based*: properties of a single (targeted) aircraft, e.g., its altitude at a certain time, or its ICAO.
- *Global*: properties relate to the whole RAP, i.e., multiple aircraft. The nature of this scope's properties is slightly different from aircraft-based properties, since properties should correspond to a single value. Thus, minimum maximum and mean values of aircraft properties can be referred to instead (e.g., mean altitude of all aircraft, or minimum ground speed, etc.).
- *Filter-Based*: this context is a refinement of the global context since only aircraft satisfying the conditions expressed by a filter are involved for the computation of the property's value.

Depending on the properties' usage context (e.g., inside an arithmetic expression, or on the left side of a schema parameter), the global and filter-based scopes are not applicable. Typically, when assigning new values to aircraft properties, only the aircraft-based scope can be used.

Class *AircraftProperty* is an abstract class, covered by its two non-disjoint subclasses, *AircraftPropertyType* and *AircraftPropertyScope*.

AircraftProperty's subclasses are not disjoint because they represent unrelated aspects of an aircraft property. Therefore, individuals can be members of both subclasses, therefore representing an aircraft property of a certain type and with a certain scope.

Regarding the actual properties, such as the altitude, longitude, callsign, and so on, it was decided to represent them as individuals rather than subclasses in the ontology. This contributes to make the ontology lighter as well as more flexible, less "modeled in stone." It also indicates that actual properties are to be kept out of the resulting grammar.

5.8 Expressions

Expressions designate common arithmetic operations (sum, product, etc.). They are used in property evaluations (see Sect. 5.10), as well as in schema parameters (see Sect. 5.11). In property evaluations, an aircraft property value is compared to the result of an expression. In schema parameters, a property is assigned with the result of an expression.

Expressions rely on values that are either directly supplied by users (e.g., integers, strings, etc.), that are references to declarations (see Sect. 5.14) or values that originate from aircraft properties.

Expressions are defined by an abstract class in the ontology, covered by its three subclasses, one that represents expressions with an operator and two operands, one for negated expression, and one for all atomic expressions, including primitive values (e.g., integers, strings) but also aircraft property value references.

5.9 Zone

Oftentimes, a filter or trigger can involve some kind of spatial-based conditions, e.g., identify aircraft that stay in a particular zone or eventually exit another. It would be extremely cumbersome to design such conditions with arithmetic comparisons between aircraft properties (latitude, longitude and altitude) and values. To make it straightforward for users to define spatial-based conditions, the concept of zones is introduced into the DSL. A zone in this context, geometrically speaking, is a prism. It is defined as a set of 3+ coordinates and a mandatory altitude range. A *Zone* class is therefore added to the ontology along with the following axiom:

$$\begin{aligned} Zone \quad \sqsubseteq \quad & \sqcap \geq 2\ hasCoordinates.Coordinates \\ & \sqcap \geq 1\ haslowerAltitude.\mathbb{N}+ \\ & \sqcap \leq 1\ haslowerAltitude.\mathbb{N}+ \\ & \sqcap \geq 1\ hasupperAltitude.\mathbb{N}+ \\ & \sqcap \leq 1\ hasupperAltitude.\mathbb{N}+ \end{aligned}$$

Similarly to expressions, zones are used in property evaluations to ultimately be a part of a filter or trigger.

5.10 Property evaluation

Property evaluations are used in filters and triggers. The objective is to compare an aircraft property value (altitude, etc.) with the result of an expression (e.g., $1200 * 5$). In addition, it is possible to pair property evaluations with conjunctions and disjunctions, as well as negate them. Concretely, *PropertyEvaluation* is an abstract class covered by its four subclasses:

- *AndOrPropertyEvaluation* is for conjunctions and disjunctions of property evaluations and is therefore a nesting class. Members of this class must be linked to two members of *PropertyEvaluation*, for both operands of the conjunction/disjunction.
- *NotPropertyEvaluation* is for negating property evaluations. Members of this class are linked to exactly one member of *PropertyEvaluation* through object property *hasSubPropEval*.
- *InZonePropertyEvaluation* is for evaluating whether aircraft are inside a certain zone. This class has a mandatory object property which range is class *Zone*.
- *StraightPropertyEvaluation* models the actual comparison between aircraft properties and expressions. It has two object properties, *hasProperty* which range is class *AircraftProperty*, and *hasPropExpression* which range is class *Expression*.

5.11 Schema parameters

Schema parameters specify the objective of schemas, i.e., aircraft property values (e.g., altitude, longitude and latitude, squawk code, etc.) to be changed for new values in case of an alteration schema, the number of fake aircraft to create in case of a saturation schema, etc. They share similarities with property evaluations in the sense that they consist of a property, an expression and an operator. Nevertheless, since the objective is value assignment, there are no conjunctions, disjunctions, and negations. As well, there are no comparison operators but an assignment operator. Each type of schema has its own set of parameters.

SchemaParameter is therefore an abstract class covered by its two subclasses:

- *AlterationParameter* enables to change aircraft properties. This means setting the value of aircraft properties (both static and dynamic) with the result of an expression. It has two object properties, one with range *AircraftProperty* and another with range *Expression*.
- *CreationParameter* makes it possible to add a fake aircraft to the RAP. Besides supplying sequence of waypoints, this means setting the value of aircraft static properties such as ICAO and callsign. Class *CreationParameter* has similar properties than *AlterationParameter*, but the range of its *hasAircraftProperty* is restricted to *StaticAircraftProperty*.

There are several schema types which parameters did not need to be represented by a class. It is the case for replay schemas: their main parameter is a source recording from which a given aircraft is extracted. This is represented by a mandatory object property *hasRecording* with domain *ReplaySchema* and range *Recording*. Similarly, saturation schemas involve making a large amount of copies of a given aircraft and having the copies slowly diverge in terms of trajectory. While the divergence itself is computed automatically by the generation module, users must supply the number of aircraft copies that should be made. It is represented by a mandatory datatype property *hasNumberOfCopies* with domain *SaturationSchema* and range *positive integer*.

5.12 Filters

It is very common for an FDIA to only target a subset of aircraft present in a recording. Schemas thus offer the possibility to define selection criteria by means of filters, as specified by REQ-2. Aircraft filtered out by the selection criteria will not be affected by the schema's alteration. The filtering may be trivial, e.g., *target only aircraft from Lufthansa*. In other cases, it may be more complex and precise, e.g., *target only aircraft of which the altitude has never exceeded 30000ft and* *has been flying at a given speed while flying in a given 3D area*.

Filters allow users to express conditions related to the properties of aircraft, such as altitude, ground speed, callsign and combine conditions with logical conjunctions and disjunctions. They integrate the concept of time and allow for a deterministic evaluation of filter expressions by introducing two temporal operators from the linear temporal logic (LTL) [7]:

- ALWAYS: expressions subjected to this operator must always be true throughout the recording.
- EVENTUALLY: expressions subjected to this operator must be true eventually during the recording.

With the introduction of these two temporal operators, expressed conditions relate to the entire recording. The objective of filter is to generate the list of all aircraft that satisfy the expressed conditions.

Only subclasses of *TargetedSchema* may be related to *Filter* individuals, although there is no cardinality restriction on the relationship.

Filters are referred to use an identifier. Class *FilterExpression* has been created to represent the expression part of the filter, while class *Filter* acts as the identifier.

FilterExpression is an abstract class that is covered by its three subclasses. They are distinct since each one represents a specific feature of filters:

- *AndOrFilterExpression* is an abstract class that models conjunctions and disjunctions of filter expressions (each is a subclass). Members of this class are linked to two members of class *FilterExpression*, representing both operands of the expression. This makes it possible to create cascades of disjunctions/conjunctions.
- *NotFilterExpression* models negation of a filter expression. Members of this class are linked to exactly one member of class *FilterExpression*. This also enables to negate conjunctions and disjunctions.
- *TemporalFilterExpression* is an abstract class representing the two temporal operators, ALWAYS and EVENTUALLY, as well as the absence of temporal operator, called *None*. Each one is modeled as a subclass because static properties do not fluctuate and therefore have no temporality. Members of this class are linked to a *PropertyEvaluation* member, to model that a temporal operator applies on a property evaluation, e.g., *the aircraft's latitude is* always *under 9.3476 throughout the recording*.

5.13 Triggers

Aircraft involved in a given schema are selected by means of filters. The next (optional) step in designing a schema is to define *when* the alterations shall be performed. As opposed to time windows, triggers express conditions in which targeted aircraft should be affected, reflecting requirement REQ-3. Triggers can be based on event such as *as soon as this event happens* and/or *until this event happens* (for instance, *perform the action as soon as all Air France aircraft are flying above 10000 feet*). In this case it affects all targeted aircraft similarly. Triggers can also be related to aircraft intrinsic properties (e.g., *perform the action when aircraft's velocity is under 300 NM/h*). In this case, it affects each targeted aircraft singularly.

Unlike filters, triggers do not include LTL logic operators. Instead, they have specific operators to express triggering conditions. Moreover, instead of returning a list of aircraft, the goal of a trigger is, for each aircraft, to determine the time intervals for when a given property evaluation is evaluated positively.

Three trigger operators are introduced into the language to define which action (alter, do not alter) is associated with a positive property evaluation:

- WHEN: alterations are only performed when the enclosed property evaluation is true.
- AS SOON AS: alterations are performed as soon as the enclosed property evaluation is true and until the end of the recording. Thus, even if the property evaluation is false after being true in the first place, the alterations will be performed.
- UNTIL: alterations are performed from the start of the recording and until *expr* is true. Thus, even if the enclosed expression is evaluated to false after being evaluated to true in the first place, the alterations will not be performed any longer.

Triggers are modeled in the ontology in a similar fashion than filters. There are three subclasses that cover class *Trigger*, one for conjunction and disjunction, one for negation, and one for trigger operators. Only *TargetedSchema* members may be related to *Trigger* individuals.

5.14 Declarations

Declarations allows for the definition of lists and ranges of values, referred to by an identifier, i.e., a variable name. Users first define a declaration (e.g., of list of integers), and thereafter make a reference to that declaration in an expression.

Declaration is an abstract class with an identifier (*hasName* property), covered by its two subclasses:

- *ListDeclaration* is an abstract class, and has a subclass for each primitive type (integers, floats, etc.) and for filters and triggers. Each subclass has a mandatory object of datatype property, without maximum cardinality but

with corresponding range (e.g., *stringListDeclaration* is defined by a *hasStringValues* with range *String*).
- *RangeDeclaration* is an abstract class as well, with two subclasses, one for integers and one for floats. Both have two datatype properties to represent the lowerbound and upperbound of the range declaration.

A schema that contains a reference to a declaration is considered abstract, as there would be multiple values for, e.g., a schema parameter. Schema concretization consists of creating one concrete schema per declaration value. In case of a schema with several references to declarations, a concrete schema shall be created per combination of declaration values. The rationale behind declarations and their relationship with schemas therefore directly addresses REQ-4.

6 DSL design

The design activity consists in creating the language's syntax and semantics based on the ontology defined during domain analysis. Depending on the result of domain analysis, several design patterns may be employed to construct the language's grammar [36]. A prerequisite is to study whether there exist languages and grammars that already define similar constructs and that can be reused. To address that issue, a design pattern called *piggybacking* can considerably reduce design and implementation efforts, and may also be easier for users to familiarize with the language. Similar to *piggybacking* are the *specialization* and *extension* patterns. The former restricts an existing language (e.g., by removing production rules or alternatives inside production rules), while the latter extends an existing language with production rules, rule alternatives, or terminals. A last design pattern, called *invention*, can be used when the DSL bears no relationship to any existing language, and its grammar must be created from scratch.

There are several aspects of the proposed DSL that are piggybacks/specializations/extensions of existing languages and constructs, namely:

- arithmetic expressions with associativity, commutativity and distributivity;
- Boolean comparisons for property evaluations;
- disjunctions and conjunctions for Boolean comparison, filters and triggers;
- LTL logic operators for filters;
- variable assignments for declarations.

Conversely, the actual definition of schemas is fully invented.

As mentioned in the previous section, there are approaches for automatically transforming OWL ontologies into context-free grammars [40]. However, for this work, the transformation was conducted manually. Indeed, it appeared clear that a sensible part of the DSL could be borrowed from existing notations while the remaining aspects, i.e., the definition of schemas, were straightforward to translate into production rules as it is sequential and flat. Nonetheless, conducting a formal domain analysis through ontology design was of tremendous help to identify piggybackable features, extractable standalone sub-languages, and reusable components (used across several standalone sub-languages).

The resulting ontology showed that two classes had the potential to be standalone sub-languages, filters and triggers. One of the DSL's requirements (REQ-4) is about genericity, and having filters and triggers as standalone sub-languages certainly contributes to the fulfillment of this requirement: users can create filters and triggers separately from schemas, and use them as part of multiple schemas. Moreover, it would overcome potential usability issues and spread out schemas' design effort over several simpler tasks. Therefore, it was decided to divide the design process into the following three stages:

1. Definition of aircraft filters
2. Definition of alteration triggers
3. Definition of alteration schemas

The ontology also showed that several entities would be required in several standalone languages, such as classes *Expression*, *AircraftProperty*, and *PropertyEvaluation*. Thus, instead of duplicating their corresponding production rules into the various standalone languages' grammar, a dedicated "abstract" grammar was created for expressions (including aircraft properties) and property evaluations. These grammars do not translate into stand-alone sub-languages, but the three standalone languages integrate and extend the abstract grammars in a factorized effort.

The next subsections describe the two abstract grammars for expressions and property evaluations, as well as the design process for the three standalone languages dedicated to filters, triggers and schemas.

6.1 Base expression grammar

The base expression grammar (BEG) is a simple language for arithmetic operations, directly originating from the *Expression* class and subclasses of the ontology. The BEG is presented in extended Backus–Naur form (EBNF) in Fig. 5. Expressions are only described in the ontology in a listing fashion. It would be cumbersome to draw grammar rules from the modeled classes and relationships, especially for including arithmetic properties such as associativity, commutativity and distributivity. Moreover, since arithmetic operations are

```
          add    ::=   mult (( + | - ) mult)*
         mult    ::=   unary (( * | / | mod ) unary)*
        unary    ::=   -? atomic
       atomic    ::=   ( add )
                  |    value
                  |    aircraft_prop
aircraft_prop    ::=   (ac_scope .)? ac_charac
        value    ::=   <integer> | <float> | <string>
    ac_charac    ::=   ( [A − Z] | '_' )+
     ac_scope    ::=   RAP |  " <identifier> "
```

Fig. 5 EBNF expression grammar

expressed using a widely known notation, the BEG therefore largely piggybacks existing arithmetic grammars.[20]

The BEG also provides an extension for allowing users to include aircraft's properties inside expressions. The ontology specifies that aircraft properties have a scope and a type. Since only three property scopes have been identified, and that no additional scope is in the foreseeable future, scopes translate into an optional terminal rule in the grammar, either *RAP* (i.e., Global) or an identifier that should point to an existing filter. The absence of scope implies that the property has the scope *aircraft*.

Conversely, property types are fixed for each property and it is not an aspect that users can decide upon. Therefore, property types are not part of the grammar and are not handled at this stage of DSL development. They are left to the semantic analysis.

An example of expression using this notation would be:

(**RAP**.**MEAN_ALTITUDE** − **ALTITUDE**) / 2

The expression's context is a targeted aircraft, in order to replace aircraft-related properties with their corresponding value. This expression results in half the difference of the RAP's mean altitude (the mean altitude of all existing aircraft), and the altitude of the targeted aircraft.

It should be noted that expressions can only be evaluated on a "snapshot" of the RAP, i.e., based on the RAP's state at a given time. Indeed, aircraft dynamic properties change over time, which means Boolean comparisons would be non-deterministic if applied to a time window instead of an instant.

6.2 Property evaluation grammar

The property evaluation grammar (PEG) is a common Boolean comparison grammar and, as such, it piggybacks existing grammars for the comparison aspects. For arithmetic expressions, it fully includes the BEG presented in the previous section. A Boolean expression opposes two values with a comparison operator; the result is *True* if the comparison is verified and *False* if it is not. The PEG slightly specializes existing grammars to force the presence of an aircraft property on the left side of the comparison (so called "property evaluation"). As presented in Sect. 5.10, a property evaluation compares an aircraft property (e.g., altitude) with the result of an expression. The PEG also extends existing grammars to enable zone-based evaluations, using 3+ coordinates and an altitude range, as defined in the ontology. The PEG is presented in Fig. 6.

```
prop_eval   ::=   negation ( (and | or ) negation)*
 negation   ::=   not ( relation )
             |    relation
 relation   ::=   bel.ac_charac (=|! =|<|>|<=|>=) bel.expr
             |    in zone
             |    ( prop_eval )
     zone   ::=   prism with_vertices vertices and
                   altitude from bel.expr to bel.expr
 vertices   ::=   coords , coords ( , coords )+
   coords   ::=   ( bel.expr , bel.expr )
```

Fig. 6 EBNF property evaluation grammar

Below is an example of a conjunction of Boolean comparisons involving aircraft properties:

ALTITUDE > 350 **and SPI** == true

This expression checks, for a given aircraft, if it is flying above 350 feet (**ALTITUDE** > 350) while having its landing gear deployed (**SPI** == true), a situation considered as abnormal.

Another example using zones:

outside prism with_vertices (43.02,51.09), (42.87,47.26), (40.66,53.11) and altitude from 5600 to 8100

This expression evaluates whether aircraft is outside a zone, expressed as a set of three coordinates (43.02,51.09), (42.87,47.26), (40.66, 53.11) and an altitude range (**altitude from 5600 to 8100**). As envisioned in the ontology, zone-based expressions are straightforward to formulate using this construct, while it would be extremely cumbersome to craft a classical query (i.e., with arithmetic relations between aircraft properties and values) that yields the same results.

Similarly to the BEG, property evaluations need an aircraft as context, and should be evaluated on an RAP's snapshot.

6.3 Filter language

The filter language, as described in the ontology in Sect. 5.12, brings two temporal operators from the Linear Temporal Logic (LTL) [7]. It makes it possible to select aircraft on the basis of their properties throughout the recording, and temporal operators makes it possible to conduct property evaluations on a time window rather than on a frozen snapshot. The grammar of the filter DSL is shown in Fig. 7.

The filter language extends the PEG with temporal operators Always and Eventually, formally referred to as **G** and **F**, respectively. To do so, an entry rule *filter* was added. It calls PEG's entry rule *prop_eval*, which has been overridden to call a new production rule called *temporal* instead

[20] http://users.monash.edu/~lloyd/tildeProgLang/Grammar/Arith-Exp/.

```
   filter     ::=   eval prop_eval
prop_eval     ::=   temporal ((and|or) temporal)*
 temporal     ::=   G ( relation )
                |   F ( relation )
                |   not ( relation )
                |   relation
 relation     ::=   bel.ac_charac (=|! =|<|>|<=|>=) bel.expr
                |   in zone
                |   ( prop_eval )
     zone     ::=   ......
```

Fig. 7 PEG grammar extended with temporal operators. Extensions of the BEG appear in bold red

of *negation*. Rule *temporal* contains the temporal operators, as well as the negation operator. This makes it possible to negate temporal operators as well as property evaluations. Note that temporal operators are not applicable to expressions that include static properties, such as aircraft's ICAO, for example, and therefore temporal operators are made optional. Since properties' type is outside the scope of the grammar, so is the restriction that temporal operators are solely enclosing dynamic property evaluations.

Below is an example of a typical filter:

eval not (**G**(**ALTITUDE** > 30000)) **and**
F(**ALTITUDE** > 34000 **and LONGITUDE** < 2.30)

This expression is composed of two sub-expressions. The first one, **not** (**G**(**ALTITUDE** > 30000)), means that aircraft should not have an altitude constantly higher than 30000 feet. It could have been presented in the alternative form **F**(**ALTITUDE** < 30000). The second sub-expression **F**(**ALTITUDE** > 34000 **and LONGITUDE** < 2.30) means that aircraft must have flown at least once at an altitude above 34,000 feet while being at a longitude inferior to 2.30 (in other terms, aircraft should have flown south of Paris).

The example above illustrates how the filter language makes it possible to express whether two Boolean expressions with the same temporality (**F** or **G**) shall be checked concurrently or independently. In the example, both expressions are inside the same temporal operator, grammatically speaking, which indicates that both expressions should be checked concurrently, i.e., both expressions must be true at least once and *at the same time*. In natural language, such an expression can be written as *aircraft has flown at least once above 34,000 feet in the south of Paris*. If each expression had been enclosed in separate *eventually* operators, such as: **F**(**ALTITUDE** > 34000) **and F**(**LONGITUDE** < 2.30), this would have indicated that the two expressions should be checked independently, i.e., each expression must be true at least once, although not necessarily at the same time. In natural language, this could be written as *aircraft has flown at least once south of Paris, and has flown at least once above 34,000 feet*. This feature greatly increases the expressiveness of filters. Note that there are two cases where this subtlety exists: *eventually* operators and logical conjunctions, and *always* operators and logical (inclusive) disjunctions.

```
    trigger   ::=   eval trigg_eval
 trigg_eval   ::=   time_window ((and|or) time_window)*
time_window   ::=   asap ( prop_eval )
                |   when ( prop_eval )
                |   not_when ( prop_eval )
                |   until ( prop_eval )
                |   ( trigg_eval )
  prop_eval   ::=   negation ( (and | or ) negation)*
   negation   ::=   ...
```

Fig. 8 PEG grammar extended with triggers

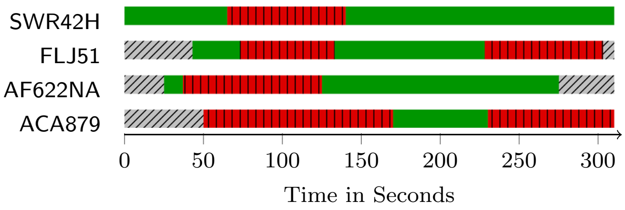


Fig. 9 Evaluation result of a trigger on 4 aircraft

6.4 Trigger language

The objective of triggers is to define *when* the alterations will be performed, as mentioned in Sect. 5.13. Although there are no existing grammar to piggyback, the trigger language is invented by copying the filter language's grammar. Indeed, both languages are similar in essence: they bring operators that enclose property evaluations. Its grammar is presented in Fig. 8.

However, one key difference between operators of the two languages is that trigger operators cannot be negated. Therefore, another strategy was employed to extend the PEG. While, for the filter language, extension involved overwriting grammar rules, grammar rules are added for the trigger language. Rules *trigg_eval* and *time_window* specify trigger expressions' syntax, directly referenced by rule *trigger*, the entry rule of the grammar. Rule *prop_eval*, PEG's entry rule, is now enclosed in trigger operators.

Instead of returning a Boolean like the filter language, the evaluation of a trigger on an aircraft returns a time line that indicates when the Boolean expression is evaluated positively and when it is evaluated negatively. An example of a trigger time line for four aircraft is depicted in Fig. 8. Green bars represent time intervals during which the trigger is true, red dashed bars represent time intervals during which the trigger is false, and striped grey bars indicate that there is no surveillance information about an aircraft.

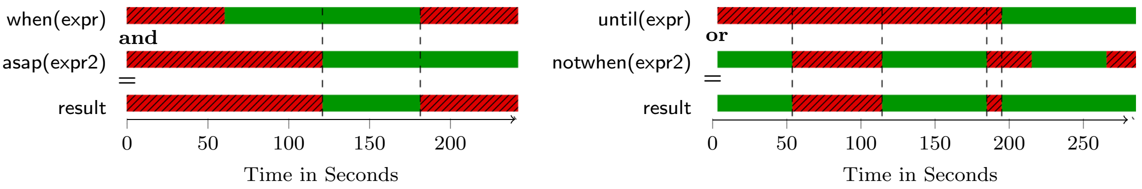
Here is a simple example of a trigger:

eval when(**LONGITUDE** > 5.25)

This expression means that aircraft will be altered (depending on the scenario) only when they are at a longitude less than 5.25.

Here is another example, with an RAP scope:

eval as_soon_as(**RAP.ALTITUDE** <= 35000)



(a) An conjunction of a*when* and an *as soon as* trigger

(b) A disjunction of a *not when* and an *until* trigger

Fig. 10 Conjunction and disjunction of triggers. Plain green (resp. red dashed) sections represent positive (resp. negative) intervals (color figure online)

This expression means that aircraft will be altered (depending on the scenario) as soon as the RAP reaches a state where there is no aircraft flying above 35000 feet.

As mentioned earlier, it is possible to combine triggers using logical conjunctions and inclusive disjunctions. For instance:

eval when(AIRCRAFT.LONGITUDE > 5.25)
and **as_soon_as**(RAP.ALTITUDE <= 35000)

This expression means that affected aircraft are altered only when they are at a longitude less than 5.25, but not before the RAP reaches a state where there is no aircraft flying above 35000 feet. The way conjunctions and disjunctions are evaluated as depicted in Fig. 10. For a conjunction to be true at *time* $= t$, both expressions should be positively evaluated at that time. For a disjunction to be positively evaluated at *time* $= t$, at least one trigger should be true at that time.

6.5 Schema language

All the languages previously presented are extensions or specializations of existing ones, well-established notations. The schema language, conversely, is pure language invention and is an almost direct transposition of the ontology: a subset of classes are converted into production rules, while others (oftentimes subclasses of abstract classes, see rules *schema*, *target*, *timeWindow*) are converted into rule alternatives. Object and datatype properties are converted into non-terminals and potentially with an operator, depending on the properties' cardinality. In addition, the schema language fully integrates the expression language. Conversely, it does not integrate the filter and trigger languages, for usability issues, but rather allows users to reference triggers and filters by their identifier. The resulting grammar of the schema language is shown in Fig. 11.

Let us illustrate the schema language with two examples. The language is further illustrated in Sect. 8.1, where we demonstrate the language's ability to cover the taxonomy of attacks.

First, a very basic alteration is defined below:

alter all_planes at 126 seconds
with_values GROUNDSPEED = 229.6

```
 scenario   ::=  decl* schema+
     decl   ::=  let var = (list | range)
   schema   ::=  ( hide | saturate ) target filter
                   t_scope trigger assertion?
              |  create t_scope trigger params
                   waypoints? assertion?
              |  alter target filter t_scope trigger
                   (params|waypoints) assertion?
   target   ::=  any_plane | all_planes
   filter   ::=  satisfying <string>
  trigger   ::=  within <string>
  t_scope   ::=  at time
              |  from time until time
     time   ::=  (number) seconds
   params   ::=  with_values param (and param)*
    param   ::=  param_type = value
waypoints   ::=  with_waypoints [ waypoint ( , waypoint)* ]
 waypoint   ::=  coord with_altitude number at time
   coords   ::=  ( value , value )
assertion   ::=  assert <string>
     list   ::=  ( value (, value)* )
    range   ::=  <integer> .. <integer>
    value   ::=  <string> | number | var
   number   ::=  <integer> | <float>
```

Fig. 11 Syntax of the alteration schema language

The schema applies on all aircraft (no filter), starting 126 seconds after the first message of the recording (no trigger), and changes aircraft' ground speed to a constant 229.9 K throughout the recording.

Another example, slightly more advanced:

create plane from 12 seconds until 251 seconds with_values
ICAO = "*39AC47*" **and with_waypoints**
[
(24.85,53.23) **with_altitude** 4000 **at 12 seconds** ,
(24.62,53.94) **with_altitude** 4250 **at 251 seconds**
]

This schema creates a fake aircraft between seconds 12 and 251 of the recording. The aircraft has the ICAO *39AC47*, and a flight trajectory defined by two way-points.

To generate multiple test cases (i.e., multiple altered recording variants) from a single scenario, it is possible to define list and ranges of values. For instance:

let *$varname* = [1,5]

This is a very standard variable definition, using the keyword **let** as in many functional programming languages. It defines a variable called *varname*, which is set to a value range from 1 to 5. This declaration is equivalent to the one below:

let *$varname* = {1,2,3,4,5}

The variable *varname* here is set to a list of 5 integers ranging from 1 to 5. Besides integers, lists can contain string values, e.g., a list of ICAO 24 bits addresses.

The advantage of using lists and ranges of values is to increase the coverage of test cases and the expressiveness of the DSL through the use of combinatorial between the variables with many values. This principle is further explained in Sect. 7.4.

7 DSL implementation

A java-based prototype of the presented design approach was created, available on GitHub.[21] All languages were developed with Xtext, a framework proposed by Eclipse for the development of DSLs,[22] while their corresponding graphical editors as well as a common graphical user interface were developed with JavaFX.[23]

The filter and the trigger languages are interpreted. In opposition to compiled languages, it means the result of the program execution is not a lower level language, but data structures that are detailed later in this section. The scenario language can be considered as a hybrid between an interpreted and a compiled language because its execution produces an XML file corresponding to a lower level language; however, the approach is similar to the approach used to interpret the trigger and the filter languages, following a fetch-decode-execute cycle [36].

Before it can be interpreted, each language must be analyzed by a semantic analyzer to determine its semantic correctness. This analysis checks the validity of the values associated to aircraft properties (e.g., an altitude cannot be associated with a string), the validity of specified values regarding to the associated recording (e.g., a time window cannot exceed the duration of a recording), or the validity of the properties' scope depending on the context (e.g., a global property cannot be assigned a new value as part of a schema parameter).

This section first details how the surveillance data are represented to enable the DSLs' interpretation. Two subsections are dedicated, respectively, to the implementation of the filter and trigger languages through presentations of their interpretation algorithms. Then, a last subsection details the process from the alteration scenario to multiple sets of alteration directives to apply to a certain recording.

7.1 On representing air surveillance data

An ADS-B recording is a collection of ADS-B squitters, each containing some information (position, velocity, or identity) about an aircraft's state at the time of sending. The ADS-B protocol is designed for the transmission of a maximum average of five messages per seconds, i.e., two for position, two for velocity and one for status. ADS-B does not provide any guarantee that the messages stream will be uninterrupted. Data dropouts can occur because of terrain variability or electromagnetic disturbances, for example. These dropouts can represent up to tens of seconds of missing data between two valid messages.

We propose to consider a recording as a set of aircraft rather than a set of surveillance messages during the scenario design phase. In that sense, the ATC expert designs a scenario that targets aircraft with certain properties at a certain time, instead of targeting messages holding certain values. It proves to be more natural from a conception point of view. However, the discontinuity in surveillance data is an issue when designing aircraft-based scenarios, especially with aircraft dynamic properties involved (ground speed, altitude, etc.). It is only possible to have the value of a property at the moment of its last transmission, and therefore it is necessary to find a means to obtain property values at any given time, which would allow for a much more precise (and correct) filtering and triggering system.

Dynamic properties of aircraft should be formalized as continuous functions of property values related to time, which mimic real-life aircraft physical behavior as closely as possible. A good candidate for this is interpolation because, as opposed to regression that takes a set of value and calculates a form of relationship between the values, interpolation only "fills the gap" between each pair of consecutive values, therefore ensuring data preservation. Interpolation is a form of approximation and as such there is a certain share of uncertainty in the calculated property values (interpolated between two squitters), i.e., interpolation divergence. While this level of uncertainty would not be acceptable for critical applications such as aircraft trajectory prediction to help issue tracks to ATCos, it is certainly acceptable in a research context.

We opted for the Akima interpolation [3] as interpolation technique. It is local interpretation technique, i.e., it only uses the neighboring points for its calculation, as opposed to global interpolation techniques that use all known points to compute an approximation. Global interpolation is subjected to Runge's phenomenon, i.e., a problem of oscillation at the edges of an interval that occurs over a set of equally spaced interpolation points [16], which would typically make it impossible to correctly model aircraft flying in a straight line. Moreover, using just the neighboring points for approximating values naturally leads to faster computation, and since

[21] https://github.com/aymeric-cr/dsl-scenario.

[22] https://www.eclipse.org/Xtext/.

[23] https://openjfx.io/.

recordings can be substantial in volume (a 30-min recording from one sensor may contain around 150000 squitters), a fast interpolation method certainly contributes to the approach's scalability.

The proposed approach relies on interpolation functions for the completion of several tasks. First, they are used as part of the interpretation of filters and triggers. Second, they are also used for the generation of aircraft trajectory (in case of a creation schema), as well as trajectory modification. Indeed, given a set of way-points fed to an interpolation function, it can approximate the aircraft's state at all time between the supplied way-points. Again, in an industrial context, this approach would fall short when it comes to realism.

7.2 Aircraft filtering

An interpretation algorithm has been created to evaluate aircraft filtering expressions. It takes advantage of the aircraft interpolation functions, as detailed in Sect. 7.1, to evaluate expressions throughout the recording. The algorithm, of $O(n)$ complexity, behaves as most language interpretation algorithms using the visitor or interpreter pattern [21]: it navigates the abstract syntax tree (AST) generated by the language's parser that corresponds to the written filter, and perform one or more operation when entering and exiting nodes.

Algorithm 1: Filter Expression Evaluation Algorithm

```
Input: $a\_id$ \ aircraft identifier
       $rel$ \ Boolean relation operator
       $prop$ \ aircraft property type
       $c\_val$\ compared value
       $temp$ \ temporal operator
       $itvl$ \ time interval
       $t\_start$ \ date of first squitter
       $t\_end$ \ date of last squitter
Result: true/false
1  $t\_count \leftarrow t\_start$
2  while $t\_count \leq t\_end$ do
3      $prop\_val \leftarrow interpolateProp(a\_id, prop, t\_count)$
4      $result \leftarrow evaluateRelation(prop\_val, rel, c\_val)$
5      if $(temp = F) \wedge result$ then
6          return $true$
7      else if $(temp = G) \wedge \neg result$ then
8          return $false$
9      else
10         $t\_count \leftarrow t\_count + itvl$
11         if $t\_count > t\_end \wedge t\_count < (t\_end + itvl)$ then
12             $t\_count \leftarrow t\_end$
13         end
14     end
15 end
16 return $temp = G$
```

Once an atomic relation node (i.e., a relation operator, an aircraft property type and a compared value) is reached, the node's content is sent to an evaluation method, as presented in Algorithm 1, along with the identifier of the aircraft under evaluation, a time interval, the temporal operator that contains the relation, and the dates of first and last squitters of the aircraft. The objective is to use the aircraft's interpolation functions to obtain the aircraft's property values over time, given a certain time interval. Starting from the aircraft's first squitter's date, the first step is to obtain the interpolated value (line 3), then the relation is evaluated using the interpolated value (line 4). There are two cases where the evaluation method can return a result immediately: either the enclosing temporal operator is *F*, in which case the relation only needs to be evaluated once negatively in order to return *true* (lines 5–6), either the enclosing temporal operator is *G*, in which case the method will returns *false* as soon as the relation is evaluated positively (line 7–8). If neither of these conditions are met, then the timer is advanced (line 10) depending on the chosen time interval, and the relation is reevaluated. If the timer was advanced too much that it points to a date later than the last squitter's date, then it is reset to the last squitter's date (lines 11–13). This ensures that the very last aircraft's state is taken into account in the evaluation. Lastly, if the algorithm iterated until reaching the last squitter's date (line 16), it shall return false if the enclosing temporal is *F* (the relation was never evaluated positively), and it shall return true if the enclosing temporal is *G* (the relation was always evaluated positively).

A slight modification of the algorithm is necessary to correctly evaluates conjunctions enclosed in a *F* operator and disjunctions enclosed in a *G* operator. Instead of returning a Boolean, the algorithm stores dates where the relation is evaluated positively. The obtained dates list is then compared to the dates list of the other relation from the conjunction/disjunction. For a conjunction in an *F* operator, relations should have identical dates list to guarantee that both relation are true *at the same time*. For a disjunction in a *G* operator, the union of the dates list should cover the entire time between the first and last squitter's date, given the chosen time interval.

Choosing the right value for *itvl* really depends on the nature of the filter, the length of the recording, the degree of precision required, and the computation time available. The bigger the time interval, the lower the precision but the faster the computation.

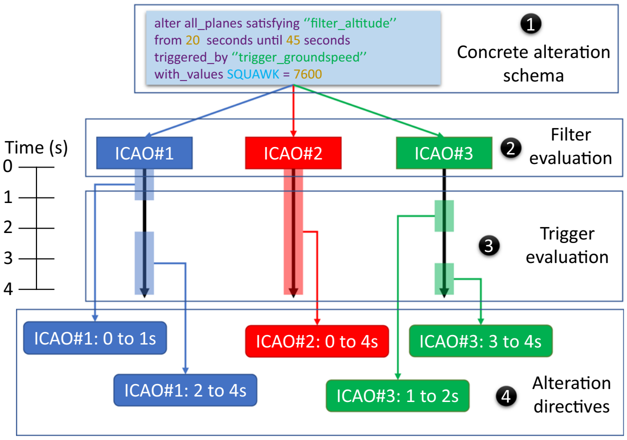
7.3 Triggering events

As for the filter language, an interpretation algorithm of $O(n^2)$ complexity has been created to evaluate trigger expressions, which functions in a similar fashion: once an atomic node is reached it is sent to an evaluation method. Algorithm 2 presents how Boolean relations are evaluated when the affiliated context is either the whole RAP or a group of aircraft obtained from evaluating a filter. The evaluation of a Boolean relation with this type of context affects all aircraft of the recording. Indeed, this method is uncorrelated from individual aircraft trigger calculation, it is therefore executed only once per Boolean relation, prior trigger calculation of the first aircraft.

Algorithm 2: Trigger Expression Evaluation Algorithm for Multiple Aircraft

```
   Input: rel \ Boolean relation operator
          prop \ aircraft property type
          c_val\ compared value
          targets \ list of aircraft
          itvl \ time interval
          t_start \ start date of recording
          t_end \ end date of recording
   Result: true/false
 1 $timeline \leftarrow list()$
 2 for $t\_count = t\_start$ to $t\_end$ by $itvl$ do
 3     $holds \leftarrow true$
 4     $a\_count \leftarrow 0$
 5     while $(holds \wedge a\_count \leq size(targets))$ do
 6         $ac \leftarrow targets(a\_count)$
 7         if $exists(ac, t\_count)$ then
 8             $prop\_val \leftarrow interpolateProp(a\_id, prop, t\_count)$
 9             $holds \leftarrow evaluateRelation(prop\_val, rel, c\_val)$
10         end
11         $a\_count++$
12     end
13     $addToList(timeline, holds)$
14     if $t\_count > t\_end \wedge t\_count < (t\_end + itvl)$ then
15         $t\_count \leftarrow t\_end$
16     end
17 end
18 return $timeline$
```

It is considered that the context has already been parsed and evaluated, resulting in the list of aircraft named *targets*. The objective of this algorithm is, for each time interval throughout the recording, to evaluate whether selected aircraft all satisfy a given Boolean relation (line 2). For a given time interval *t_count*, the method iterates the aircraft list as long as the Boolean relation holds (line 5), i.e., that it is being positively evaluated after each iteration. In an iteration, it is first necessary to check that the aircraft is present in the recording at the current time interval (line 7). If that is the case, then its property value is interpolated (line 8), and the Boolean relation is evaluated (line 8). After the aircraft iteration ends, the content of the variable *holds* (which contains true if all aircraft have been iterating, or contains falls if the Boolean relation has stopped holding) is added to the timeline list (line 11). Finally, once the whole recording has been evaluated, the algorithm returns *timeline*, a list of Boolean values. The real time line can be obtained by multiplying each Boolean value's position in the list with the time interval *itvl*, and adding the result to the starting date of the recording.



```
alter all_planes satisfying ''filter_altitude''
from 20  seconds until 45 seconds
triggered_by ''trigger_groundspeed''
with_values SQUAWK = 7600
```

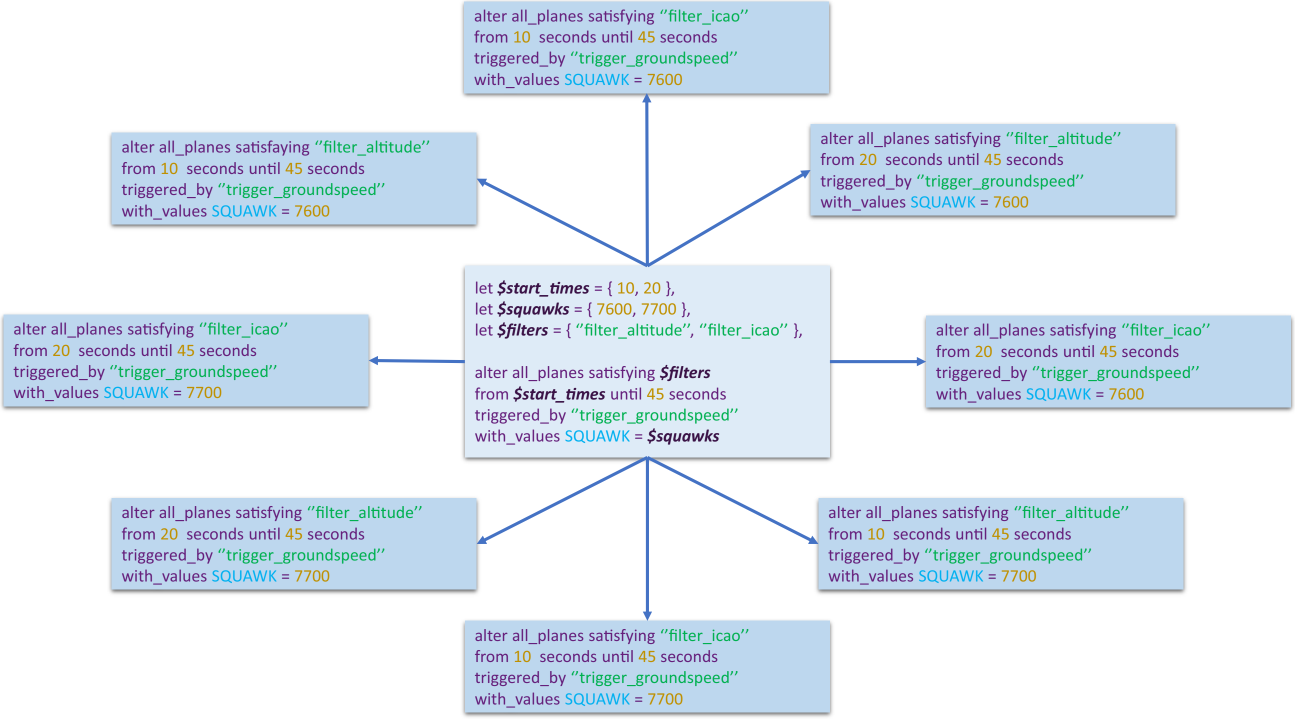
Fig. 12 Obtaining alteration directives from a concrete alteration schema

7.4 Generation of alteration directives from schemas

This section details the process of generating alteration directives from an abstract alteration schema with filters, triggers and combinatorial properties.

The computation of alteration directives can be split into three activities:

1. **Concretization of alteration schema.** As shown in Sect. 6.5 it is possible to define lists or ranges of values as variables in a alteration schema. The rationale is as follows: if a schema contains one or more references to a variable that contains a list or range of value, that schema has an abstract nature. A form of concretization is therefore necessary, which consists of creating concrete instances of this schema, where each concrete schema is valued with an entity from the abstract schema's list or range. If two or more lists/ranges are defined in an abstract schema, this implies the creation of a concrete schema for each unique combination of lists values. Therefore, if three lists, one of size 2, one of size 4 and one of size 5, are referenced in an abstract schema, then $2 * 4 * 5 = 40$ instances of the alteration are created. An example of combination of parameters is shown in Fig. 13. The concretization activity thus allows for the potential generation of large sets of concrete alteration schemas, translating into a large set of test cases.
2. **Target Selection.**
   As depicted in Fig. 12, target selection (i.e., aircraft targeted by the scenario) is performed for each concrete schema ①. A list of aircraft identifiers ② is obtained by the interpretation of filters presented in Sect. 6.3. The



```
alter all_planes satisfying ”filter_icao”
from 10  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7600
```

```
alter all_planes satisfaying ”filter_altitude”
from 10  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7600
```

```
alter all_planes satisfying ”filter_altitude”
from 20  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7600
```

```
let $start_times = { 10, 20 },
let $squawks = { 7600, 7700 },
let $filters = { ”filter_altitude”, ”filter_icao” },

alter all_planes satisfying $filters
from $start_times until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = $squawks
```

```
alter all_planes satisfying ”filter_icao”
from 20  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7700
```

```
alter all_planes satisfying ”filter_icao”
from 20  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7600
```

```
alter all_planes satisfying ”filter_altitude”
from 20  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7700
```

```
alter all_planes satisfying ”filter_altitude”
from 10  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7700
```

```
alter all_planes satisfying ”filter_icao”
from 10  seconds until 45 seconds
triggered_by ”trigger_groundspeed”
with_values SQUAWK = 7700
```

Fig. 13 From an abstract schema to concrete schemas

resulting list is used next for the computation of time intervals for each targeted aircraft.

3. **Computation of time intervals.** As explained in Sect. 2.3, in addition of a targeted aircraft, an alteration directive requires a time window during when the alteration shall be performed. This is done by evaluating alteration triggers, if any, the process of which is described in Sect. 6.4. The result is, for each targeted aircraft, a list of time windows ③. A time window, once associated with its targeted aircraft (and parameters to modify if relevant), corresponds to an alteration directive ④.

To summarize, variables and triggers make it possible to define complex alteration scenarios with ease. On the one hand, variables factorize the definition of alteration, while, on the other hand, triggers automatically define high numbers of alteration time windows based on constraints. Both these design accelerators shall be processed in order to convert the alteration scenario given as input into a list of alteration directives.

8 DSL testing

ATC constitutes one of the most critical infrastructures on the planet and, as such, it is not possible to demonstrate the use of FDI-T in real situation by feeding falsified data to a real ATC system and reporting results in a publicly available journal. This could highlight the fact that some systems, even if thought to be deprecated and not in use anymore, are vulnerable to FDIAs and it is unclear what such type of information could say about current systems. Therefore, we opted instead for a demonstration of design capabilities, i.e., the ability to design scenarios according to the taxonomy, and a demonstration of productivity, i.e., how cumbersome it would be to modify recordings by hand or using ad-hoc scripts and regular expressions. The goal is thus to answer the two research questions that were defined in Sect. 2.4. We first demonstrate the design capabilities of the DSL as pointed out in *RQ1*, and thereafter its potential of productivity compared to manual and script-based alterations, as pointed out in *RQ2*.

8.1 Demonstration of design capabilities

We propose to validate the expression power of the scenario DSL by demonstrating its ability to generate test scenarios according to the taxonomy stated in Sect. 2.2, and well acknowledged in the literature, in an effort to answer *RQ1*.

An example of *Aircraft Flooding Attack* is depicted in Fig. 14. The instruction **saturate** (line 1) constitutes the *Alteration* part which, in this example, is a multiple message creation.

A saturation is defined by two parameters:

```
1  saturate all_planes satisfying ”filter”
2  triggered_by ”trigger”
3  from 20 seconds until 25 seconds
4  with_values random ICAO and
5  AIRCRAFT_NUMBER = 20
6  assert ”Multiple Fake tracks detected”
```

Fig. 14 Example of aircraft flooding

```
1  alter all_planes satisfying ”filt1”
2  from 120 seconds until 140 seconds
3  with_values SQUAWK = 7700
4  assert ”False Alarm detected”
```

Fig. 15 Example of false alarm

```
1  hide all_planes triggered_by ”trigg”
2  from 45 seconds until 60 seconds
3  assert ”Suspicious aircraft disappearance:”
```

Fig. 16 Example of aircraft disappearance

- **AIRCRAFT_NUMBER**: This parameter determines the number of ghost aircraft generated for each targeted aircraft.
- **ICAO**: This parameter determines the ICAOs of generated ghost aircraft. It can be a simple value, a list or randomly generated.

A saturation results in a certain number of ghost aircraft generated around each (real) targeted aircraft, the number of generated aircraft is defined by the parameter **AIRCRAFT_NUMBER**. The properties of ghost aircraft are the same than the real targeted aircraft, except the latitude and longitude that are automatically generated in a way to spread ghost aircraft around the real aircraft, and their ICAO is defined by the user according to the parameter **ICAO**.

Both the *Selection* and *Trigger* parts (line 1) are delegated to a filter named "*filter*" and a trigger named "*trigger*". Saturation should take place from 20 **seconds** to 25 **seconds** according to the start of the input recording. The *parameters* part (line 4) specifies that fake aircraft should have a randomly generated ICAO. Finally, the *Assertion* part verifies whether the ATC system return a message that multiple fake aircraft have been detected.

Figure 15 illustrates how to create a *False Alarm* scenario using FDI-T. It relies on an **alter** alteration that solely targets aircraft selected with the filter "*filt1*" (line 1), which occurs between 120 and 140 seconds after the start of the recording. During that time interval, the squawk of targeted aircraft is set to 7700, meaning that these aircraft are in emergency mode. A built-in function of the alteration engine consists of setting the *ALERT* bit to one in the first altered message of each aircraft, as this notifies the ground station that the aircraft's squawk code has just been changed.

```
1   create plane from 20 seconds until 320 seconds
2   with_values ICAO = ”39AC47” and
3   CALLSIGN = ”AFR3984” and
4   GROUNDSPEED = 102.2 and
5   LATITUDE = 2.57684 and
6   LONGITUDE = 47.49875 and
7   ALTITUDE = 26598
8   with_waypoints [
9   (2.476,47.69) with_altitude 29843 at 72 seconds ,
10  (2.39,47.93) with_altitude 33879 at 215 seconds ,
11  (2.319,48.01) with_altitude 38743 at 320 seconds ]
12  assert ”Fake aircraft detected”
```

Fig. 17 Example of a ghost aircraft injection

```
1  alter plane satisfying ”filt2”
2  from 100 seconds until 200 seconds
3  with_waypoints [
4  (29.11,14.21) with_altitude 33902 at 110 seconds ,
5  (29.235,14.3) with_altitude 33979 at 130 seconds ,
6  (29.29,14.35) with_altitude 33954 at 150 seconds,
7  (29.29,14.35) with_altitude 33954 at 170 seconds ]
8  assert ”Abnormal trajectory detected”
```

Fig. 18 Example of a trajectory modification

An aircraft disappearance attack can be achieved with scenarios as the one presented in Fig. 16. In this example, all aircraft will be hidden during time window 45 to 60 seconds and according to trigger *trigg*. The ATC system should conclude of a suspicious aircraft disappearance and look for another source of surveillance data while alerting the ATCo that ADS-B data is being tempered with.

Figure 17 shows an example of a *ghost aircraft injection* scenario. The instruction **create** (line 1) constitutes the *Alteration* part which, in this example, is a message creation. The *time window* part (line 1) is hard-coded: a plane should be created from 20 **seconds** to 320 **seconds** according to the start of the input recording. The *parameters* part contains six parameters to set (lines 2-7), namely ICAO, ground speed, call sign, latitude, longitude and altitude. The next part (lines 8–11) concerns the trajectory that the fake aircraft should adopt, with the specification of three waypoints, giving coordinates and altitudes at 72, 215, and 320 seconds. Finally, the *Assertion* part is string based. It means that, for the test case to pass, the ATC system should return a message claiming it detected a fake aircraft.

An example of a *trajectory modification* scenario is illustrated in Fig. 18. It relies on an **alter** alteration that targets one aircraft selected with the filter "*filt1*" (line 1), which occurs between 100 and 200 seconds after the start of the recording (line 2). The modified trajectory consists of four waypoints (lines 3–7) providing coordinates and altitudes at 110, 130, 150, and 170 seconds. In this case, starting at second 100, the new trajectory will diverge from the initial one in order to rally the first waypoint by second 110. Similarly, between seconds 170 and 200, the generated trajectory will attempt to rally the initial trajectory in order to put back the aircraft in its normal state by second 200. Finally, the *Assertion* part is string based. It means that, for the test case to pass, the ATC system should return a message claiming it detected an abnormal aircraft trajectory.

As a reminder, the first research question that we initially defined in Sect. 2.4 is as follows:

RQ1 **To what extent is it possible to design alteration scenarios in order to cover the taxonomy of attack scenarios?**

Based of the above demonstration, it is possible to confidently answer positively to *RQ1*. All the scenarios from the taxonomy can be effectively designed using the DSL, making it a suitable approach for evaluating the resilience of ATC systems against FDIAs. What remains to be demonstrated is whether the DSL accelerates the design of alteration scenarios.

8.2 Facilitating FDIA scenarios creation and reducing test design effort: evaluation

In order to provide a meaningful answer to *RQ2*, this section shows how cumbersome it can be to modify recordings without the use of the proposed DSL, i.e., directly crafting an XML file with alteration directives to be fed to the generation module.

In other words, the goal of this experimentation is to demonstrate how filters, triggers, and the combinatorial capabilities of the DSL translate in terms of alteration directives.

The objective is to demonstrate the DSL's capabilities:

1. to generate directives based on aircraft properties;
2. to generate large amounts of directives automatically;
3. to generate a whole test suite in one go by taking advantage of list of values in alteration schemas.

We detail below the FDIA Scenario components that demonstrates the productivity capabilities of the DSL.

Initial Recording

The demonstration relies on a 17 minutes and 45 seconds recording of April 19, 2019, between 15:30 and 15:47, which was received by a powerful radar sensor in Luxembourg and obtained through the OpenSky Network. The recording contains 76353 squitters and features 218 aircraft.

Filters

There are three filters for this scenario:

filt_lowalt: **eval** F(GROUNDSPEED > 350) **and** G(ALTITUDE < 34000)

This filter specifies that aircraft should have a ground speed superior to 350 nautical miles at least once, while their altitude should always be lower than 34000 feet. 64 aircraft satisfy this filter.

filt_lat: **eval** G(LATITUDE < 49.5)

```
let $start = {30,200,254,1065}
let $squawks = {7500,7600,7700}
alter all_planes satisfying "filt_lat"
at $start seconds triggered_by "trig_vertrate"
with_values SQUAWK = $squawks
assert "False alarm detected"
```

Fig. 19 First schema: false alarm attack

This filter helps identify aircraft always flying at a latitude that is under 49.5. 85 aircraft satisfy this filter.

filt_prism: **eval** F(**inside prism with_vertices** (48,7.7), (50.31,6.35), (49.18,4.22) **and altitude from** 1000 **to** 35000)

This filter specifies that aircraft should fly at least once inside a triangle above Luxembourg while being at an altitude between 1000 and 35000 feet. 30 aircraft satisfy this filter.

Triggers

There are also three triggers needed for this scenario:

trig_vertrate: **eval when**(AIRCRAFT.VERT_RATE >= 0.15)

This trigger marks aircraft to be altered when their vertical rate is equal or higher to 0.15. It results in 820 time intervals in total, with some aircraft without any time interval (their vertical rate is never within the specified range), and for instance aircraft *SWR97X* with 26 time intervals throughout its existence.

trig_altgs: **eval when**(RAP.ALTITUDE < 41001) **and until**(AIRCRAFT.GROUNDSPEED > 350)

This trigger marks aircraft to be altered when the whole RAP is flying under 41001 feet and until their ground speed increases over 350 nautical miles. It results in 254 intervals in total, although all aircraft have almost identical intervals.

trig_prism: **eval as_soon_as**(**inside prism with_vertices** (48,7.7), (50.31,6.35), (49.18,4.22) **and altitude from** 1000 **to** 35000)

This trigger marks aircraft to be altered as soon as they enter the same triangle as defined in filter *filt_prism*. It results in 30 intervals, with 0 or 1 interval per aircraft.

Schemas

There are two alteration schemas.

The first schema, as presented in Fig. 19, represents a false alarm attack. It has two variables, a four-integer list for time windows and a three-integer list for squawk codes. This makes for twelve combinations of values, and therefore, twelve concretized schemas will be created.

The second schema simulates an aircraft disappearance attack (the goal is to hide certain aircraft at certain time), as shown in Fig. 20. It has two variables, a list of two filters and a list of two triggers, making for four concretized schemas.

Scenario

The scenario is a combination between the both previous schemas for the generation of test suites. The false alarm schema leads to twelve concretized schemas, while the aircraft disappearance schema leads to four concretized

```
let $filters = {"filt_lowalt","filt_prism"}
let $triggers = {"trig_altgs","trig_prism"}
hide all_planes satisfying $filters
at 0 seconds triggered_by $triggers
assert "Aircraft Disappearance detected"
```

Fig. 20 Second schema: aircraft disappearance attack

schemas, therefore the resulting test suite will be composed of $12 * 4 =$ **48** test cases.

Results
Generating one test case meant applying between *9 to 346* alteration directives to the recording, for an average of *233.25* alteration directives per test case, 211.75 of which are modifications and 54.75 are deletions. Directives had a time window of alteration that ranged from *160 milliseconds to 17 minutes*. All in all, the generation of the whole test suite involved the performing of *11196 alteration directives*, including *9720 modifications* and *1476 deletions*. This is clearly not feasible if one had to manually specify each alteration directive.

The goal of this subsection is to answer the following research question:

RQ2 **To what extent the use of a DSL can facilitate the creation of FDIA's test scenarios and reduce the effort to create such test scenarios?**

Given the number of alteration directives, one would have to design in order to create the above test scenario, it is safe to say that it would not be possible without the proposed DSL. Therefore, we can also answer positively to *RQ2*, as not only test productivity is noticeably augmented using the DSL, but this approach is the only alternative for certain types of scenarios.

9 Conclusion and future work

In the context of improving cyber security in air surveillance communications, and especially with regard to the ADS-B protocol, this paper describes a novel test design approach to be part of an existing FDIA testing framework dedicated to ATC systems, called FDI-T. More precisely, the main contribution concerns the test design activity, in the form of a DSL of which the grammar, specificities and capabilities are extensively presented in this paper. The two objectives of the proposed approach are, first, to cover the taxonomy of attacks as described in the literature on multiple occasions, and second, to sensitively improve test design productivity as compared to hand-made alterations. We show how both objectives have been reached by successfully designing all classes of scenarios from the taxonomy using the DSL and by relying on a running example to demonstrates productivity gains when compared to manual and script-aided alterations.

The most imminent future work is to study the genericity potential that brings a DSL to the FDI-T framework, with the objective of porting it to other domains of application. A logical first portage would be toward the Maritime domain. The AIS protocol is extremely similar to the ADS-B protocol in many aspects (lack of security included), vessel surveillance resembles air traffic surveillance, and extending the approach to the Maritime domain makes sense.

Another future work concerns the ability to automate the design of meaningful and relevant alteration scenarios. This could be achieved using machine learning techniques, of which the best candidate to date based on preliminary research is generative adversarial network (GAN). A discriminator network would be trained to detect FDIAs generated by a generative network, which itself would be trained to generate FDIAs that are stealthy enough to not be spotted by the discriminant model. Efforts have been concentrated so far on representing the surveillance data into features, while the first round of experimentation shall start in the near future.

Acknowledgements This work is part of an ongoing research initiative toward the generation of FDIA test scenarios partially supported by the GeLeaD ANR ASTRID Project & the EIPHI Graduate school (Contract "ANR-17-EURE-0002").

Appendix A: Ontology inheritance tree

All the entities that were identified and modeled into the DSL's ontology during the domain analysis activity (see Sect. 5) are depicted in Figs. 21, 22 and 23. More importantly, the figures show the inheritance relationships between entities. Note that as there is a unique *Thing* entity from which all other entities directly or indirectly inherit from (similarly to the Object class in Java), the tree was originally depicted in a single figure. But for obvious space reasons, that figure was eventually split into three.

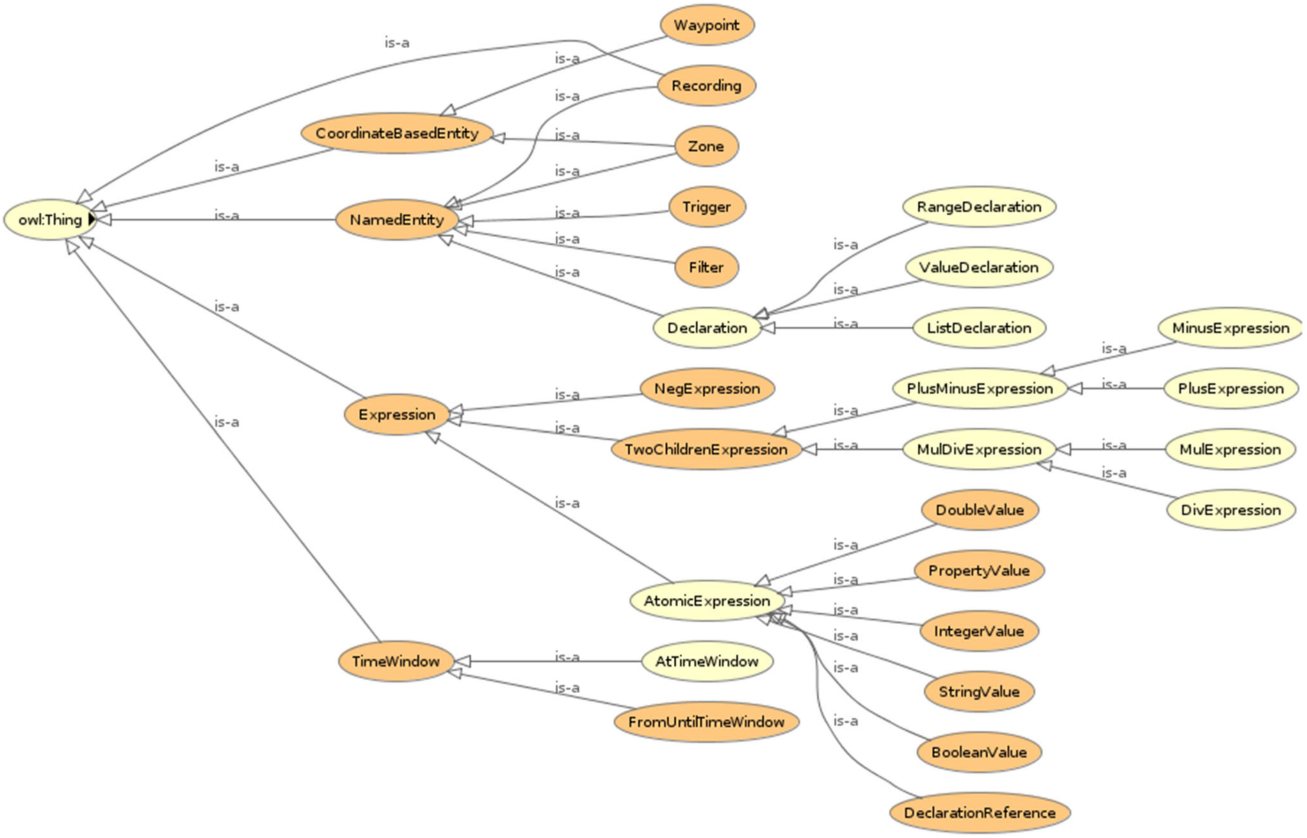


Fig. 21 Ontology inheritance tree 1/3

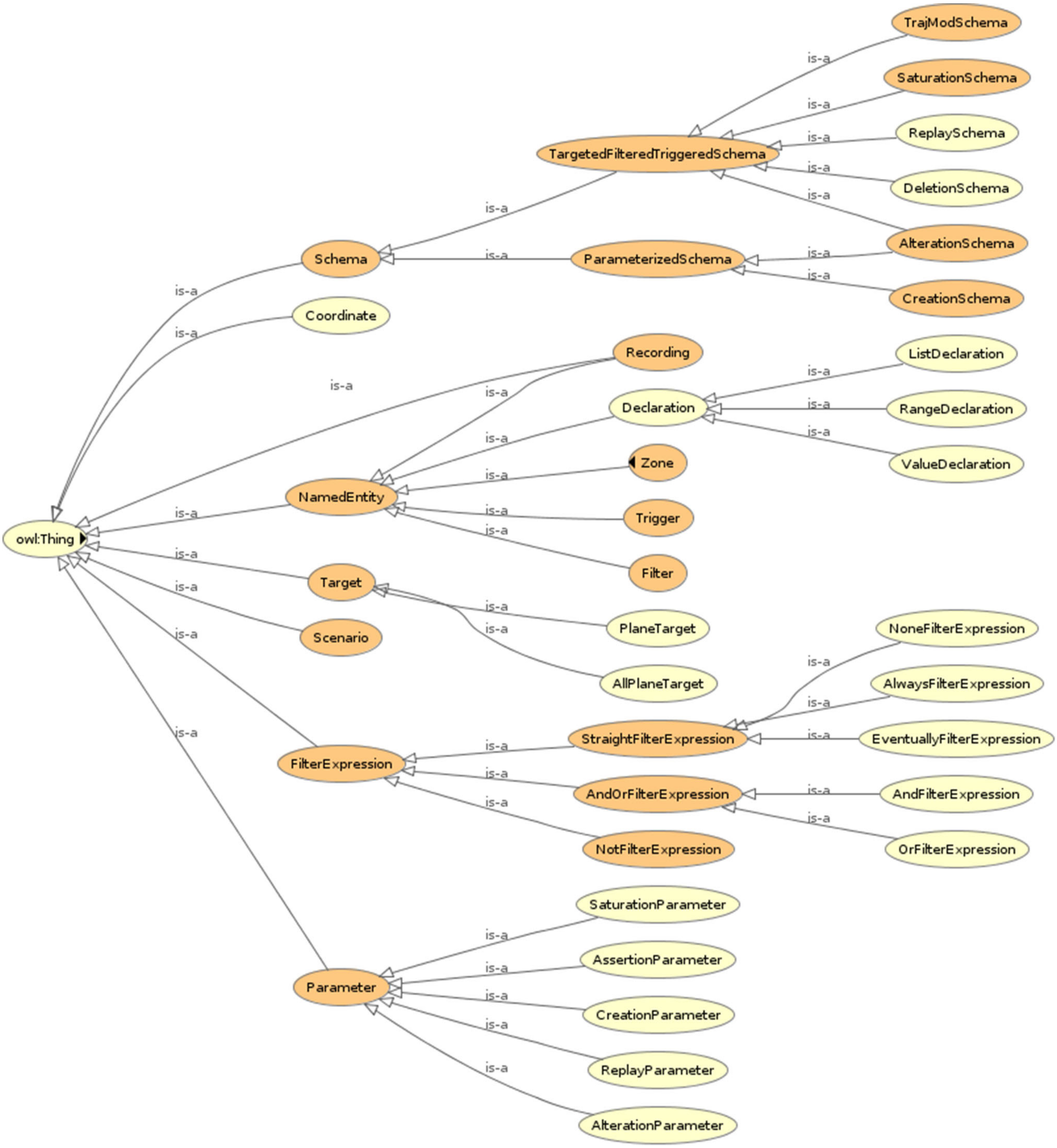


Fig. 22 Ontology inheritance tree 2/3

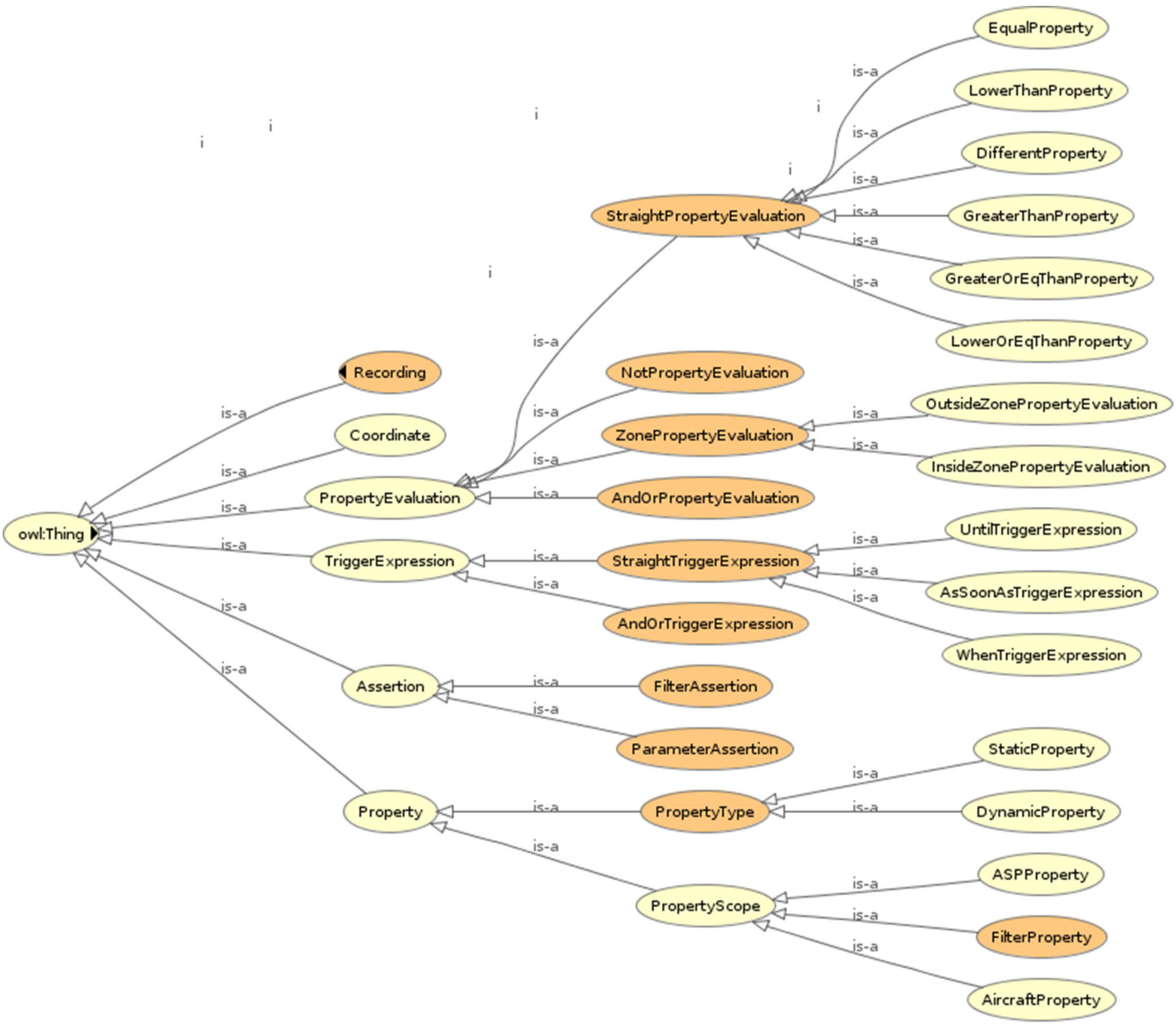


Fig. 23 Ontology inheritance tree 3/3

References

1. 51, E.W.G.: Safety, performance and interoperability requirements document for ADS-B/NRA application. Tech. rep., The European Organisation for Civil Aviation Equipment (2005). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.6059&rep=rep1&type=pdf
2. Akerman, S., Habler, E., Shabtai, A.: VizADS-B: Analyzing sequences of ADS-B images using explainable convolutional LSTM encoder-decoder to detect cyber attacks (2019). arXiv preprint arXiv:1906.07921
3. Akima, H.: A new method of interpolation and smooth curve fitting based on local procedures. J. ACM **17**, 589–602 (1970). https://doi.org/10.1145/321607.321609
4. Asia, I.C.A.O., (ICAO), P.O.: Guidance material on issues to be considered in atc multi-sensor fusion processing including the integration of ADS-B data. Tech. rep., APANPIRG/19 (2008). https://www.icao.int/APAC/Documents/edocs/cns/grpt_atcmulti_adsbdata.pdf
5. Baader, F., Horrocks, I., Sattler, U.: Description logics. In: Handbook on Ontologies, pp. 3–28. Springer (2004)
6. Barreto, A.B., Hieb, M., Yano, E.: Developing a complex simulation environment for evaluating cyber attacks. In: Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), vol. 12248, pp. 1–9 (2012)
7. Belta, C., Yordanov, B., Aydin Gol, E.: Temporal Logics and Automata, pp. 27–38. Springer International Publishing, Cham (2017)
8. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Sci. Am. **284**(5), 34–43 (2001)
9. Brooker, P.: Sesar and nextgen: investing in new paradigms. J. Navig. **61**(2), 195–208 (2008)
10. Ceh, I., Crepinšek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Comput. Sci. Inf. Syst. **8**(2), 317–342 (2011)
11. Chan, Y.T., Ho, K.: A simple and efficient estimator for hyperbolic location. IEEE Trans. Signal Process. **42**(8), 1905–1915 (1994)

12. Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. IEEE Softw. **15**(6), 37–45 (1998)
13. Cretin, A., Legeard, B., Peureux, F., Vernotte, A.: Increasing the resilience of ATC systems against false data injection attacks using DSL-based testing. In: Proceedings of the 8th International Conference on Research in Air Transportation (ICRAT'18), Doctoral Symposium, pp. 1–4. Barcelona, Spain (2018)
14. Cretin, A., Vernotte, A., Chevrot, A., Peureux, F., Legeard, B.: Test data generation for false data injection attack testing in air traffic surveillance. In: 4th International Workshop on Testing Extra-Functional Properties and Quality Characteristics of Software Systems (ITEQS 2020). Porto, Portugal (2020)
15. Dan, G., Sandberg, H.: Stealth attacks and protection schemes for state estimators in power systems. In: Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on, pp. 214–219. IEEE (2010)
16. Epperson, J.F.: On the runge example. Am. Math. Mon. **94**(4), 329–341 (1987)
17. EUROCONTROL: D23—security assessment for ADS-B ground system—3rd iteration 00.01.02. Tech. rep., Sesar Joint Undertaking (SJU) (2014)
18. Frakes, W., Prieto, R., Fox, C., et al.: Dare: Domain analysis and reuse environment. Ann. Softw. Eng. **5**(1), 125–141 (1998)
19. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: an owl 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014)
20. Habler, E., Shabtai, A.: Using lstm encoder-decoder algorithm for detecting anomalous ADS-B messages. Comput. Secur. **78**, 155–173 (2018)
21. Hills, M., Klint, P., van der Storm, T., Vinju, J.: A case of visitor versus interpreter pattern. In: Bishop, J., Vallecillo, A. (eds.) Objects, Models, Components, Patterns, pp. 228–243. Springer, Berlin (2011)
22. Jafer, S., Chhaya, B., Durak, U.: Owl ontology to ecore metamodel transformation for designing a domain specific language to develop aviation scenarios. In: Proceedings of the Symposium on Model-Driven Approaches for Simulation Engineering, pp. 1–11 (2017)
23. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. rep. (1990)
24. Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: a systematic mapping study. Inf. Softw. Technol. **71**, 77–91 (2016). https://doi.org/10.1016/j.infsof.2015.11.001
25. Lassila, O., Swick, R.R., et al.: Resource description framework (rdf) model and syntax specification (1998)
26. Lisboa, L.B., Garcia, V.C., Lucrédio, D., de Almeida, E.S., de Lemos Meira, S.R., de Mattos Fortes, R.P.: A systematic review of domain analysis tools. Inf. Softw. Technol. **52**(1), 1–13 (2010)
27. Liu, Y., Ning, P., Reiter, M.K.: False data injection attacks against state estimation in electric power grids. ACM Trans. Inf. Syst. Secur. (TISSEC) **14**(1), 13 (2011)
28. Ma, M.: Resilience against false data injection attack in wireless sensor networks. In: Handbook of Research on Wireless Security, pp. 628–635. IGI Global (2008)
29. Maciel, D., Paiva, A.C., da Silva, A.R.: From requirements to automated acceptance tests of interactive apps: an integrated model-based testing approach. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 265–272. SCITEPRESS-Science and Technology Publications, Lda (2019)
30. Manesh, M.R., Kaabouch, N.: Analysis of vulnerabilities, attacks, countermeasures and overall risk of the automatic dependent surveillance-broadcast (ADS-B) system. Int. J. Crit. Infrastruct. Prot. **19**, 16–31 (2017). https://doi.org/10.1016/j.ijcip.2017.10.002
31. Manesh, M.R., Mullins, M., Foerster, K., Kaabouch, N.: A preliminary effort toward investigating the impacts of ADS-B message injection attack. In: 2018 IEEE Aerospace Conference, pp. 1–6. IEEE (2018)
32. Martinovic, I., Strohmeier, M.: Security of ADS-B: State of the art and beyond. DCS (2013)
33. McGuinness, D.L., Van Harmelen, F., et al.: Owl web ontology language overview. W3C Recommend. **10**(10): 2004 (2004)
34. Menzel, T., Bagschik, G., Maurer, M.: Scenarios for development, test and validation of automated vehicles. In: 2018 IEEE Intelligent Vehicles Symposium (IV), pp. 1821–1827. IEEE (2018)
35. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005). https://doi.org/10.1145/1118890.1118892
36. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (CSUR) **37**(4), 316–344 (2005)
37. Mernik, M., Hrnčič, D., Bryant, B.R., Javed, F.: Applications of grammatical inference in software engineering: domain specific language development, pp. 421–457. Imperial College Press (2010). https://doi.org/10.1142/9781848165458_0008
38. Paielli, R.A.: Automated generation of air traffic encounters for testing conflict-resolution software. J. Aerosp. Inf. Syst. **10**(5), 209–217 (2013)
39. Pakin, S.: The design and implementation of a domain-specific language for network performance testing. IEEE Trans. Parallel Distrib. Syst. **18**(10), 1436–1449 (2007)
40. Pereira, M.J.A.V., Fonseca, J.A., Henriques, P.R.: Ontological approach for dsl development. Comput. Lang. Syst. Struct. **45**(C), 35–52 (2016). https://doi.org/10.1016/j.cl.2015.12.004
41. Queiroz, R., Berger, T., Czarnecki, K.: Geoscenario: an open dsl for autonomous driving scenario representation. In: 2019 IEEE Intelligent Vehicles Symposium (IV), pp. 287–294. IEEE (2019)
42. Rui, L., Ho, K.: Elliptic localization: performance study and optimum receiver placement. IEEE Trans. Signal Process. **62**(18), 4673–4688 (2014)
43. Savvides, A., Park, H., Srivastava, M.B.: The bits and flops of the n-hop multilateration primitive for node localization problems. In: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications, pp. 112–121. ACM (2002)
44. Schäfer, M., Lenders, V., Martinovic, I.: Experimental analysis of attacks on next generation air traffic communication. In: International Conference on Applied Cryptography and Network Security, pp. 253–271. Springer (2013)
45. Skolnik, M.I.: Radar Handbook. 3rd edn (2008)
46. Smith, A., Cassell, R., Breen, T., Hulstrom, R., Evers, C.: Methods to provide system-wide ADS-B back-up, validation and security. In: 25th Digital Avionics Systems Conference, pp. 1–7. IEEE (2006)
47. Strohmeier, M.: Security in next generation air traffic communication networks. Ph.D. thesis, Oxford University (2016)
48. Strohmeier, M., Schäfer, M., Pinheiro, R., Lenders, V., Martinovic, I.: On perception and reality in wireless air traffic communications security. IEEE Trans. Intell. Transp. Syst. **18**(6), 1338–1357 (2017). https://doi.org/10.1109/TITS.2016.2612584
49. Studer, R., Benjamins, V.R., Fensel, D.: Knowledge engineering: principles and methods. Data Knowl. Eng. **25**(1–2), 161–197 (1998)
50. Tairas, R., Mernik, M., Gray, J.: Using ontologies in the domain analysis of domain-specific languages. In: International Conference on Model Driven Engineering Languages and Systems, pp. 332–342. Springer (2008)
51. Taylor, R.N., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures: Cdrl a011-a curriculum module in the sei style. ACM SIGSOFT Softw. Eng. Notes **20**(5), 27–38 (1995)
52. Trim, R.: Mode s: an introduction and overview (secondary surveillance radar). Electron. Commun. Eng. J. **2**(2), 53–59 (1990)

53. Tsarkov, D., Horrocks, I.: Fact++ description logic reasoner: system description. In: International Joint Conference on Automated Reasoning, pp. 292–297. Springer (2006)
54. Van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. J. Comput. Inf. Technol. **10**(1), 1–17 (2002)
55. Wesson, K.D., Humphreys, T.E., Evans, B.L.: Can cryptography secure next generation air traffic surveillance? IEEE Secur. Privacy Mag. (2014)
56. Xie, L., Mo, Y., Sinopoli, B.: False data injection attacks in electricity markets. In: Smart Grid Communications (SmartGridComm), First International Conference on, pp. 226–231. IEEE (2010)
57. Zhang, R., Liu, G., Liu, J., Nees, J.P.: Analysis of message attacks in aviation datalink communication. IEEE Access (2017)