# Calculated Moves: Generating Air Combat Behaviour

Toubman, A.

# Appendices

# Appendix A

# The Lightweight Air Combat Simulator

In this appendix we present the Lightweight Air Combat Simulator (LWACS). The appendix is organised as follows. First, we provide a general description of LWACS (Appendix A.1). Next, we describe the CGFs in LWACS (Appendix A.2), and then briefly introduce the scripting language by which behaviour models for the CGFs are created (Appendix A.3). Furthermore, we present the air combat scenarios that we have developed for use in LWACS (Appendix A.4).

## A.1 Description

LWACS simulates a section of airspace. The simulated section of airspace is inhabited by air combat CGFs (see Appendix A.2) that engage each other in predefined scenarios (see Appendix A.4). LWACS was designed to require few computational resources to run, so that it can comfortably run many simulations in parallel on a modern desktop computer. The term "lightweight" in the name of the simulator refers to the low system requirements of the simulator.

The LWACS program is written in the Java programming language. It can be run in two modes: (1) with a graphical user interface (GUI) that allows inspection of the simulated airspace, and (2) with a command-line interface (CLI) (a.k.a. headless). In the headless mode, LWACS is able to run automated simulations in two ways: both (1) in faster-than-real-time and (2) in parallel. LWACS currently does not support human-in-the-loop simulations.

LWACS was developed at NLR in the context of the Smart Bandits (SB) project (see Appendix D). In the SB project, LWACS served as a platform for (1) testing behaviour models and (2) to explore the use of machine learning within air combat simulations. Because LWACS was completely developed at NLR, we had access to the entire source code. This allowed us to adapt LWACS to our research purposes (i.e., the automated simulations in Chapters 3, 4, and 5) as needed.

## A.2 Computer generated forces

LWACS supports one kind of CGF, which represents a generic fighter jet. In LWACS, each CGF carries three types of devices: (1) a radar, (2) air-to-air missiles, and (3) a radar warning receiver. We describe the devices below.

**Radar.** The radar is a sensor that detects other aircraft. In the simulation, it produces a forward-looking cone, emanating from the front of the aircraft carrying the radar. Any aircraft inside this cone are detected by the radar. The radar has two modes: (1) the search mode, which produces a wide cone (120°), and (2) the tracking mode, which produces a narrow cone (30°). The search mode is used to find opposing aircraft. The tracking mode is used to track the movement of one specific opposing aircraft. The radar has a detect range of 100 km. Because of the limited range of the radar, deliberate manoeuvring is required to find and track opponents.

**Air-to-air missiles.** Each aircraft carries four air-to-air missiles. These missiles can only be fired at opponents. A prerequisite for firing a missile at an opponent is that the opponent should be tracked by the radar of the aircraft firing the missile. Hitting an opponent with a missile disables their aircraft and removes it from the simulation. Each aircraft only carries four missiles. Furthermore, missiles do not hit in a deterministic manner. Upon impact, each missile calculates its so-called probability-of-kill ($P_k$, see Chapter 4). The $P_k$ of a missile decreases as the distance flown towards the target of the missile increases. The precise decrease of the $P_k$ is defined by means of a predefined curve. The curve used in LWACS is shown in Figure 4.2. The $P_k$ of missiles in LWACS starts at 1, and stays 1 until the missile has flown 50 km. From 50 km onward, the $P_k$ of missiles declines until it reaches 0 after 80 km. After having flown 80 km, missiles are removed from the simulation.

**Radar warning receiver (RWR).** The RWR is a device that detects whether the aircraft is inside the radar cone of another aircraft. The pilot may use this information to assume that a missile will be fired or has already been fired at him, and then take action accordingly. The RWR has a detection range of 200 km.

A screenshot of LWACS is shown in Figure A.1. The CGFs in LWACS are graphically displayed as F-16 fighter jets. All CGFs belong to one of two teams: either (1) the blue team or (2) the red team. The graphical models of the CGFs are coloured to indicate team membership.

Three different indicators can be shown next to each CGF. The indicators appear as small coloured triangles. They serve to visualise the status of each CGF. The three indicators are as follows. First, a light blue indicator shows whether the CGF has detected another CGF by means of its radar. Second, a red indicator shows whether the CGF has detected another CGF by means of its RWR. Third, white indicators show the number of missiles remaining in the inventory of each CGF.

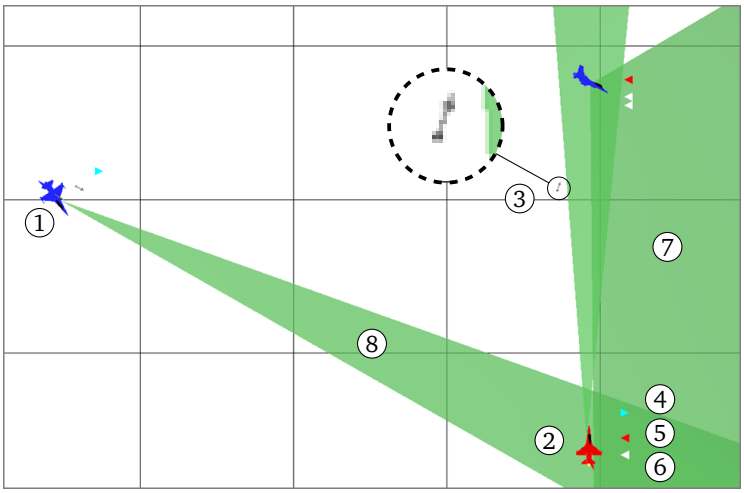


**Figure A.1** A screenshot of LWACS, showing: (1) a blue CGF, (2) a red CGF, (3) a missile (with magnification), (4) a light blue radar indicator, (5) a red radar warning receiver indicator, (6) white missile inventory indicators, (7) a wide radar cone (search mode), (8) a narrow radar cone (tracking mode).

In LWACS, the CGFs have two restrictions on their movement. The first restriction is the use of a basic flight model. The flight model allows the CGFs to either (1) move at a constant velocity, (2) accelerate, and (3) decelerate, all as if the CGFs were in a vacuum. In other words, the flight model has no notion of aerodynamics or gravity. The second restriction is that the CGFs are only allowed to move in the horizontal plane. The reason for the second restriction is that the radars of the CGFs were unable to operate in a three-dimensional environment at the time of our research. Regardless, vertical movement in LWACS is meaningless because the flight model provides no speed penalties for ascending or speed gains for descending. The two restrictions on CGF movement make LWACS a simulator with a relatively low fidelity. However, the CGFs in LWACS still represent the basic functions that are required in air combat (e.g., manoeuvring, the use of radar and RWR, firing and evading missiles). Many combinations of these functions are possible in LWACS, making it difficult to design good behaviour models manually. Therefore, we consider LWACS an adequate simulator for our investigation into the use of machine learning for the automatic generation of behaviour models.

## A.3 Scripting language

In LWACS, CGF behaviour is defined by scripts. The scripts are written in a custom scripting language. The grammar of this language and the available functions are described in Appendix B.

The scripts are collections of rules. Each CGF is assigned a script. At a rate of 50 Hz, the

simulation checks the scripts of all CGFs to determine if any rules should fire. If a rule fires, the actions in the consequence of that rule are executed.

It may occur that multiple rules fire at the same time, e.g., if the rules have overlapping observations in their conditions. In order to provide some control over rule execution in such occurrences, we assign a priority value to each rule. The priority value allows the creators of the rules to specify which rules provide the most urgent or important behaviour. Now, when multiple rules fire, only the rule with the highest priority value is allowed to execute its consequence. In the rare case that multiple rules fire with the same priority value, only the rule that first appears in the script is allowed to execute its consequence.

The CGFs are allowed to maintain a *state* variable. The state is expressed as an alphanumeric identifier. The scripting language can read and write the state variable, so that (1) certain rules can only fire if the CGF is in a particular state, and (2) rules can change the state of the CGF when certain observations are made. The state variable enables sequential scripting, i.e., firing one rule first, and then a second rule, and so on. For instance, we used sequential scripting to create a script that described a patrol route for a CGF. We assigned the value 'patrolling' to the state variable of the CGF. We defined two points in the simulation airspace that the CGF had to patrol, which we call point A and point B. In the script, we included two rules that defined the patrol between points A and B. The first rule was written as "if I am near point A, and my state is 'patrolling', fly towards point B". Vice versa, the second rule was written as "if I am near point B, and my state is 'patrolling', fly towards point A". Additionally, we included a rule that said "if my state is patrolling, and I detect an opponent CGF, change my state to 'engaging'". The result of these rules was that the CGF would fly its patrol between points A and B as long as it did not detect an opponent CGF. Once it detected an opponent CGF, the CGF would no longer consider firing the rules that were written for its patrol, since its state variable did no longer satisfy these rules (i.e., it was not longer set to 'patrolling').

## A.4 Scenarios

We developed four two-versus-one scenarios, and four two-versus-two scenarios. In the two-versus-one scenarios, two red CGFs encounter a single blue CGF. In the two-versus-two scenarios, the two reds encounter two blue CGFs. The scripts that governs the behaviour of the blue CGFs in each scenario are presented in Appendix C. Below, we describe the two-versus-one scenarios (Appendix A.4.1) and the two-versus-two scenarios (Appendix A.4.2) in detail.

### A.4.1 Two-versus-one scenarios

We developed four two-versus-one scenarios for use in LWACS: (1) the basic scenario, (2) the close range scenario, (3) the evasive scenario, and (4) the mixed scenario. We describe the four scenarios below.

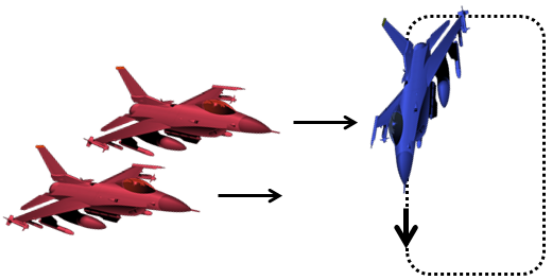

**Figure A.2** The initial positions of the CGFs in the four LWACS scenarios. Two red CGFs (left) approach the blue CGF (right), who is flying a CAP.

**The basic scenario.** One blue CGF flies a combat air patrol (CAP) from north to south (see Figure A.2). When the blue CGF detects an opponent by means of its radar or RWR, it engages that opponent. The blue CGF does not attempt to evade missiles that are fired at it. Two red CGFs fly from east to west and then try to engage the blue CGF.

**The close range scenario.** The close range scenario is equal to the basic scenario, with one change. The blue CGF only fires missiles from a shorter range. This makes missiles more dangerous, as they need to travel less distance to reach their target, and therefore have a higher $P_k$ on impact.

**The evasive scenario.** The evasive scenario is equal to the basic scenario, with one change. The blue CGF performs evasive actions when it detects that it is being fired upon.

**The mixed scenario.** The mixed scenario is a special scenario, because it is a combination of the other three two-versus-one scenarios. Rather than defining new behaviour for the blue, the mixed scenario enables the blue to switch behaviours between encounters. We refer to the mixed scenario as one of the scenarios for convenience. However, in some cases it is helpful to distinguish the mixed scenario from the other three scenarios. In these cases, we refer to the other three two-versus-one scenarios as the *individual* scenarios.

The mixed scenario works as follows. At the start of a run of encounters, one of the three individual scenarios is selected at random, and then used for the first encounter between red and blue. At the end of each encounter, the following individual scenario is selected based on the winner of the encounter. If blue wins the encounter, the same scenario is used in the next encounter. However, if red wins the encounter, the next individual scenario is selected at random. The mixed scenario is inspired by the *consecutive tactic* by Spronck et al. (2006, p. 230).

Compared to only using one of the three individual scenarios, the mixed scenario presents the red team with a *moving target learning problem* (cf. Buşoniu, Babuška and De Schutter, 2010). In other words, red is pressured to come up with behaviour that is effective against an unpredictable opponent. We designed that the mixed scenario to form a more difficult challenge for the red team than only having to find effective behaviour in the three individual scenarios.

The four scenarios have two properties in common. First, the initial positions of the CGFs are the same in all four scenarios. These initial positions are shown in Figure A.2. Second, the four scenarios share their termination criteria. All of the scenarios terminate when either (1) one CGF on either team is hit by a missile, or (2) ten minutes of simulated time has passed.

### A.4.2 Two-versus-two scenarios

We developed four two-versus-two scenarios for use in LWACS. The first three of these scenarios are based on the three individual two-versus-one scenarios. In these scenarios, we supplied the single blue CGF with a wingman blue CGF. This made the first blue CGF the lead of the blue two-ship. In all three of the scenarios, the wingman flies the same CAP as the lead, lagging half a pattern behind the lead. The remaining behaviour of the wingman is governed by the same rules as the lead. Thus, during the encounters with the reds, the wingman uses the same tactics as the lead: either (1) attacking the reds without evading (basic scenario), (2) attacking the reds from close range (close range scenario), or (3) attacking the reds while also evading incoming missiles (evasive scenario).

As the fourth scenario, we developed a novel two-versus-two scenario. This scenario is called the lead-trail scenario. We describe this scenario below.

**The lead-trail scenario.** This scenario is based on the lead-trail tactic that is commonly used by two-ship formations. In this scenario, the blue lead flies head-on towards the red two-ship. The blue wingman flies straight after the blue lead as they approach the reds. When the reds detect the blue lead, the blue lead turns away, with the intention of keeping the attention (viz. a radar lock) of the reds. Then, the blue wingman is able to stay undetected, and then create an opportunity to fire at the reds.

The two-versus-two scenarios have the same two termination criteria as the two-versus-one scenarios.

# Appendix B

# The LWACS scripting language

In this appendix, we present the LWACS scripting language. The language is used to write the scripts that define the behaviour of the CGFs in LWACS. Below, we first describe the grammar of the scripting language (Appendix B.1). Next, we describe the functions that are available for use in the scripts (Appendix B.2).

## B.1 Grammar

Listing B.1 is a formal description of the grammar of the LWACS scripting language in Extended Backus-Naur form. The boolean functions, numerical functions, and action functions are further explained in Section B.2.

**Listing B.1** Grammar of the LWACS scripting language.

⟨*script*⟩ ::= ⟨*list-of-rules*⟩

⟨*list-of-rules*⟩ ::= ⟨*rule*⟩ end-of-line ⟨*list-of-rules*⟩
| '#' comment-string end-of-line ⟨*list-of-rules*⟩
| end-of-line ⟨*list-of-rules*⟩
| ⟨*empty*⟩

⟨*rule*⟩ ::= ⟨*name*⟩ [⟨*weight*⟩] ⟨*priority*⟩ ⟨*condition*⟩ '→' ⟨*consequence*⟩ ';'

⟨*name*⟩ ::= '[' identifier ']'

⟨*weight*⟩ ::= '[' integer ']'

⟨*priority*⟩ :: = '[' integer ']'

⟨*condition*⟩ :: = ⟨*boolean-expression*⟩

⟨*boolean-expression*⟩ ::= ⟨*boolean*⟩
| ⟨*boolean-function*⟩
| 'not' ⟨*boolean-expression*⟩
| '(' ⟨*boolean-expression*⟩ ')'
| ⟨*boolean-expression*⟩ ('and' | 'or' | '==') ⟨*boolean-expression*⟩
| ⟨*numerical-expression*⟩ ('>' | '<' | '==') ⟨*numerical-expression*⟩
| 'state == ' ⟨*state*⟩

⟨*boolean*⟩ ::= 'true' | 'false'

⟨*boolean-function*⟩ ::= 'isAlive(' ⟨*cgf*⟩ ')'
| 'isRadarMode(' ⟨*radar-mode*⟩ ')'
| 'messageReceived(' ⟨*message*⟩ ')'
| 'missileFlyingAt(' ⟨*cgf*⟩ ')'
| 'missilesLeft'
| 'onEvent(' ⟨*event*⟩ ')'

⟨*cgf*⟩ ::= 'ownship'
| 'wingman'
| 'nearestRadarObservation'
| 'nearestRadarWarningReceiverObservation'
| 'entity(target)'

⟨*event*⟩ ::= 'newRadarObservation'
| 'newRadarWarningReceiverObservation'
| 'newMissileFlyingAtMe'

⟨*team*⟩ ::= 'enemy'

⟨*message*⟩ ::= identifier

⟨*state*⟩ ::= identifier

⟨*radar-mode*⟩ ::= 'searching' | 'track'

⟨*numerical-expression*⟩ ::= ⟨*number*⟩
| ⟨*numerical-function*⟩
| ⟨*numerical-expression*⟩ ('+' | '−' | '*' | '/') ⟨*numerical-expression*⟩

⟨*number*⟩ ::= integer | float

⟨*numerical-function*⟩ ::= '`countRadarObservations(`' ⟨*team*⟩ '`)`'
| '`countRWRObservations(`' ⟨*team*⟩ '`)`'
| '`distanceToPoint(`' ⟨*cgf*⟩ '`,`' ⟨*numerical-expression*⟩ '`,`'
⟨*numerical-expression*⟩ '`,`' ⟨*numerical-expression*⟩ '`)`'
| '`heading(`' ⟨*cgf*⟩ '`)`'
| '`random(`' ⟨*number*⟩ '`,`' ⟨*number*⟩ '`)`'
| '`relativeBearing(`' ⟨*cgf*⟩ '`,`' ⟨*cgf*⟩ '`)`'

⟨*consequence*⟩ ::= ⟨*action-list*⟩

⟨*action-list*⟩ ::= ⟨*action-function*⟩
| ⟨*action-function*⟩ ⟨*action-list*⟩

⟨*action-function*⟩ ::= '`changeHeading(`' ⟨*numerical-expression*⟩ '`);`'
| '`changeState(`' ⟨*state*⟩ '`);`'
| '`fireMissile(`' ⟨*cgf*⟩ '`);`'
| '`flyTo(`' ⟨*number*⟩ '`,`' ⟨*number*⟩ '`,`' ⟨*number*⟩ '`);`'
| '`radarTrackTarget(`' ⟨*cgf*⟩ '`);`'
| '`radarSearchTarget;`'
| '`sendMessage(`' ⟨*message*⟩ '`,`' ⟨*cgf*⟩ '`);`'
| '`skip;`'
| '`turn(`' ⟨*numerical-expression*⟩ '`);`'

Three comments regarding the scripting language and its grammar:

- `identifier` represents any alphanumeric word. Dashes are allowed in identifiers, but not as leading or trailing characters.
- `comment-string` represents any comment in natural language.
- `entity(target)` represents the point in space that is the center of the blue CGF's CAP. It can be used in place of a ⟨*cgf*⟩ parameter. In some scripts and rulebases, it is used to explicitly steer the red team towards the blue team at the beginning of a simulation.

## B.2 Function descriptions

Below, we describe the boolean functions (Subsection B.2.1), the numerical functions (Subsection B.2.2), and the action functions (Subsection B.2.3).

### B.2.1 Boolean functions

**isAlive**(*target*: ⟨*cgf*⟩)

Returns `true` if *target* is alive, returns `false` otherwise.

**isRadarMode**(*mode*: ⟨*radar-mode*⟩)

Returns `true` if the CGF's radar is set to *mode*, returns `false` otherwise.

**messageReceived**(*message*: ⟨*message*⟩)

Returns `true` if the CGF's has received *message*, returns `false` otherwise. The message is consumed.

**missileFlyingAt**(*target*: ⟨*cgf*⟩)

Returns `true` if a missile is flying at *target*, returns `false` otherwise.

**missilesLeft**

Returns `true` if there are missiles left in the CGF's inventory, returns `false` otherwise.

**onEvent**(*event*: ⟨*event*⟩)

Returns `true` if the CGF has been notified of *event*, returns `false` otherwise. The event is consumed.

### B.2.2 Numerical functions

**countRadarObservations**(*team*: ⟨*team*⟩)

Returns the number of CGFs belonging to *team* that are detected by the CGF's radar.

**countRadarWarningReceiverObservations**(*team*: ⟨*team*⟩)

Returns the number of CGFs belonging to *team* that are detected by the CGF's RWR.

**distanceToPoint**(*target*: ⟨*cgf*⟩, *x*: ⟨*numerical-expression*⟩, *y*: ⟨*numerical-expression*⟩, *z*: ⟨*numerical-expression*⟩)

Returns the distance (in kilometers) of *target* to the point $(x, y, z)$.

**missilesLeft**

Returns `true` if there are missiles left in the CGF's inventory, returns `false` otherwise.

### B.2.3 Action functions

**changeHeading**(*heading*: ⟨*numerical-expression*⟩)

Steer the CGF towards heading *heading* (in degrees).

**changeState**(*state*: ⟨*state*⟩)

Set the CGF's state to *state*.

**fireMissile**(*target*: ⟨*cgf*⟩)

Fire a missile at *target*.

**flyTo**(*x*: ⟨*numerical-expression*⟩, *y*): ⟨*numerical-expression*⟩, *z*: ⟨*numerical-expression*⟩)

Steer the CGF towards point (*x*, *y*, *z*).

**sendMessage**(*message*: ⟨*message*⟩, *target*: ⟨*cgf*⟩)

Send *message* to *target*.

**skip**(*mode*: ⟨*radar-mode*⟩)

Do nothing (viz. continue flying on the current heading with the current speed).

**radarTrackTarget**(*target*: ⟨*cgf*⟩)

Set the radar to tracking mode and direct it to track *target*.

**radarSearchTarget**

Set the radar to search mode.

**turn**(*offset*: ⟨*numerical-expression*⟩)

Change the CGF's heading by *offset* (in degrees).

## Appendix C

# Rulebases and scripts

Appendix C contains a listing of the rulebases and scripts that are used in this thesis. The rulebases and scripts are available for download at `http://www.armontoubman.com/phd`. The rules in the rulebases and scripts are formatted as follows:

```
[name] [weight] [priority] condition → consequence
```

Below, we provide an index of the rulebases and scripts that were used in this thesis. Each starred item has been archived as a separate file. For reasons of confidentiality, we are unable to distribute the rulebases and scripts that were used in Chapter 7.

**Red team**

- Team coordination (Chapter 3)
  - CENT method
    - Red lead (rulebase)
    - Red wingman (script)
  - TACIT method
    - Red lead (rulebase)
    - Red wingman (rulebase)
  - DECENT method
    - Red lead (rulebase)
    - Red wingman (rulebase)
- The AA-REWARD reward function (Chapter 4)
  - Red lead (rulebase)
  - Red wingman (rulebase)
- Transfer of behaviour models between scenarios (Chapter 5)

  - Red lead (rulebase)
  - Red wingman (rulebase)

**Blue team**

- Two-versus-one scenarios
  - Basic scenario
    * Blue lead (script)
  - Close range scenario
    * Blue lead (script)
  - Evasive scenario
    * Blue lead (script)
  - Mixed scenario
    * Note: in the mixed scenario, the blue CGF used the scripts from the other three scenarios. Therefore, no specific scripts were made for the blue in the mixed scenario.
- Two-versus-two scenarios
  - Basic scenario
    * Blue lead (the same script as in the two-versus-one basic scenario)
    * Blue wingman (script)
  - Close range scenario
    * Blue lead (the same script as in the two-versus-one close range scenario)
    * Blue wingman (script)
  - Evasive scenario
    * Blue lead (the same script as in the two-versus-one evasive scenario)
    * Blue wingman (script)
  - Lead-trail scenario
    * Blue lead (script)
    * Blue wingman (script)

# Appendix D

# The Fighter 4-Ship simulator

Appendix D describes the Fighter 4-Ship simulator. The Fighter 4-Ship is a research fighter aircraft simulator used by the Netherlands Aerospace Centre NLR (Netherlands Aerospace Centre, 2017b). The simulator allows four human fighter pilots to participate simultaneously in a simulated air combat mission. The purpose of the Fighter 4-Ship is to enable concept development and experimentation in the area of training simulations.

The Fighter 4-Ship consists of (a) four cockpit mock-ups (referred to as the *ships*), and (b) a station for the *instructor*, i.e., the person who controls the operation of the Fighter 4-Ship. This station is accordingly called the instructor operating station (IOS). Below, we describe the hardware of the ships (Appendix D.1), the IOS (Appendix D.2), and the software packages (Appendix D.3) that make up the Fighter 4-Ship.

## D.1 The ships

The four ships of the Fighter 4-Ship are modelled after the cockpit of the F-16 fighter jet. Figure D.1 shows a schematic top view of a ship, and Figure D.2 shows a photograph of one of the ships in operation. Each of the four ships is comprised of the following nine items. The items are marked in Figure D.1.

1. A seat.
2. A touchscreen monitor in front of the seat.
3. A physical side stick controller to the right of the seat.
4. A physical throttle to the left of the seat.
5. Physical rudder pedals.

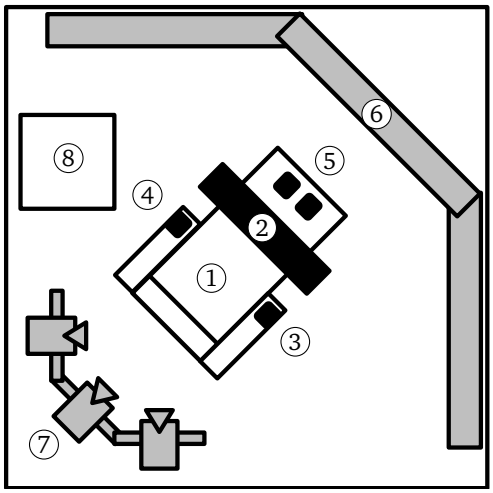


**Figure D.1** Schematic top view of a ship.

6. Projector screens that present the participant with (1) the outside view, and (2) an overlayed head-up display (HUD). Two of the ships are equipped with three screens: a left screen, a right screen, and a centre screen. The remaining two ships are equipped with a centre screen only.

7. A projector for each projector screen.

8. Computers that (1) generate the outside visuals which are projected onto the projector screens, (2) control the touchscreen, and (3) handle the voice communications.

9. A headset (not depicted in Figure D.1) for (1) sound effects (e.g., engine noise, warning signals) and (2) voice communication with the instructor and the participants in the other ships over simulated radio channels.

The seat, stick, throttle, and pedals are replicas of the equipment in a real-world F-16 cockpit. The touchscreen monitor shows representations of the F-16 consoles and controls. The representations include two MFDs, viz. square screen which provide overviews of the aircraft's situation, and allow the pilot quick access to various functions by means of hierarchical menus.

The ships are *fixed-base*, viz. they do not provide any motion effects. Each of the four ships is located in a separate but adjacent room. The aircraft noise that is to be heard over the headsets prevent any verbal communication between the pilots, other than using the simulated radio channels. Furthermore, because of the physical separation, a participant in one of the ships is unable to see any of the other ships.

**Figure D.2** Photograph of a ship being operated by a participant.

## D.2 The instructor operating station

The simulation is controlled from the instructor operating station (IOS). This station provides the instructor with the capability to (1) control the operation of the individual ships, (2) start and stop scenarios, (3) add and remove CGFs, (4) assign behaviour models to CGFs, (5) communicate with the participants in the ships, (6) view and record the MFDs of the four ships in order to monitor the actions of the participants, and (7) view and record the simulated environment and all entities within it. The IOS is comprised of the following two items.

1. A desktop computer with five monitors.
    i. A monitor that displays EUROSIM, the software that starts/stops/pauses the operation of each of the four ships. The functions of EUROSIM and the other software that is displayed on the monitors are explained in Appendix D.3.
    ii. A monitor that displays STAGE, the software that creates the CGFs.
    iii. A monitor that displays SMART BANDITS, the software that executes the behaviour models of the CGFs.
    iv. A monitor that displays a video stream of the MFDs of the four cockpits.
    v. A monitor that displays the debrief software that records the simulated environment.
2. A headset for voice communication with the participants in the ships. Additionally, this headset monitors all simulated radio channels. So, the instructor at the IOS remains informed of all communication that happens among participants.

The use of five monitors allows the instructor to easily view and access all functions that are relevant to controlling and monitoring the simulations.

## D.3 Software packages

The Fighter 4-Ship runs on four software packages: (1) EUROSIM, (2) STAGE, (3) SMART BANDITS, and (4) PCDS. We describe the four software packages below.

**EUROSIM.** EuroSim (2017) is a simulation framework that provides interfaces for air and space simulations. In the Fighter 4-Ship, EUROSIM executes the models of the flight dynamics the aircraft that the four ships represent. Additionally, EUROSIM simulates the avionics (e.g., the radar) of the ships. EUROSIM provides a GUI by which the operation of individual ships can be controlled and inspected by the instructor.

**STAGE.** STAGE (Presagis, 2012) is a "simulation development framework" that provides an environment for building and executing scenarios. The instructor can select CGFs from a database, and place them into the simulated world. STAGE includes a basic behaviour editor, which is currently not used in the Fighter 4-Ship. Finally, STAGE provides a GUI by which the state of CGFs can be inspected and manipulated during simulations.

**SMART BANDITS.** SMART BANDITS (Netherlands Aerospace Centre, 2017a) is software by which CGF behaviour can be modelled as a FSM. In SMART BANDITS, the states and transitions that make up FSMs can be combined by means of drag-and-drop functionality. This way, it allows professionals such as training specialists to create behaviour models without any explicit programming knowledge. During simulations, the behaviour of CGFs can be inspected and manipulated.

**PCDS.** PCDS (Naval Air Systems Command, 2010) provides record and playback facilities for simulation environments. It is intended to aid debriefing, i.e., the meeting after a simulation session when the instructor reviews the simulation with the participants (e.g., to identify learning opportunities). Video files (e.g., recordings of the MFD streams) can be connected to the playback of simulations, so that the internal situation in each ship can be seen when a simulation is reviewed.

The four software packages communicate with each other and with the four ships by means of the distributed interactive simulation (DIS) standard (IEEE, 2012). This standard defines a common format for the exchange of information regarding, e.g., the location and status of CGFs. This information is used by the four software packages to determine, e.g., how a CGF should be visualised.

## D.4 Dynamic scripting in the Fighter 4-Ship

At the beginning of our research, the Fighter 4-Ship had no machine learning capabilities. Therefore, the integration of a machine learning technique such as dynamic scripting required us to determine the placement of this technique between the Fighter 4-Ship's software packages. In other words, we needed to consider the architecture of the Fighter 4-Ship's software.

In the Fighter 4-Ship, STAGE is the software package that generates the CGFs. STAGE exposes an API by which other software packages can (1) read the state and observations of its CGFs, (2) direct the actions of the CGFs, and (3) start and stop predefined scenarios. SMART BANDITS controls the CGFs in STAGE by means of this API. In SMART BANDITS, the state and observations of a CGF are read from the API, and then serve as input for a behaviour model. The behaviour model then outputs the actions that the CGF should take. The actions are sent back to the CGF in STAGE via the API.

To implement dynamic scripting, we require software with a function similar to how SMART BANDITS executes behaviour models for the CGFs in STAGE. However, compared to SMART BANDITS, we require two additional functions: (1) running the dynamic scripting algorithm to generate new behaviour models (particularly, the rule-based FSM models that are described in Appendix E), and (2) control over the scenarios that are run in the simulator.

For our research, we developed a new program which combines the three functions. We call this program STAGEDS. STAGEDS provides us with two capabilities: (1) to let CGFs learn by means of the dynamic scripting algorithm, and (2) to halt the learning process and only execute the learned behaviour models. We make extensive use of these two capabilities in Chapter 7. Ideally, in the future, these capabilities will be built into the software that is currently used for the design and execution of CGF behaviour models (i.e., SMART BANDITS).

## Appendix E

# Generating finite-state machines

Appendix E discusses the generation of finite-state machines by means of dynamic scripting.

In the automated simulations that were performed in LWACS, the rules in the scripts of CGFs produced behaviour in an *ad-hoc* manner: whenever the condition of a rule was met, the actions in that rule's consequence would immediately be executed. As we have shown in the Chapters 3 to 5, the behaviour which is produced in this manner is effective for CGFs in automated simulations. However, since then, we have been informed that this way of producing behaviour is not completely suitable for human-in-the-loop simulations.

During simulations, scripts are sensitive to small changes in the environment. Some of these small changes should have no effect on the behaviour of a CGF, but may still cause different rules to fire. As an example, assume that a CGF is under attack, and a defensive rule fires which causes the CGF to make a defensive manoeuvre. However, during this manoeuvre, the CGF simultaneously detects another opponent by means of its radar, and an offensive rule is prompted to fire. At this point, the CGF is still in danger from the first opponent, and should continue with its defensive manoeuvre instead of preparing an attack on the second opponent. Moreover, from an external point of view, such sudden jumps between rules make the CGF appear erratic and indecisive, i.e., not precisely the way how a formidable adversary should behave. To remedy such a jump in behaviour, the CGF needs to be able to remember what it is doing and why. In other words, the CGF needs a concept such as a state in which it feels situated, so that the CGF can select and display behaviour that is relevant for that state.

Returning to the topic of representation, in Chapter 2, we mentioned FSMs as one of the executable forms that a behaviour model can take. The management of states and the behaviours therein, e.g., the states and behaviours of a CGF, is the prime function of FSMs. Furthermore, since FSMs are comprised of graph-like structures, they are highly suitable for graphical representation

as demonstrated in, e.g., the SMART BANDITS program. Consequently, we have a preference to generate CGFs with dynamic scripting.

Below, we show how the states and transitions that make up an FSM can be readily expressed in the form of rules (Appendix E.1). By expressing states and transitions in the form of rules, dynamic scripting is able to combine the states and transitions into new FSMs, in the same way that dynamic scripting combines "regular" if-then rules into scripts. However, at this point we foresee that the dynamic scripting algorithm needs to be modified so that it can combine these rules into functioning FSMs. We present the exact modifications that we make to the original dynamic scripting algorithm (Appendix E.2). Thereafter, we conclude the appendix by a summary (Appendix E.3).

## E.1 Expressing finite-state machines as rules

In this section, we explain how FSMs can be expressed as rules. We do so by means of the following example. Consider a CGF that performs a patrol between two points, namely point A and point B. Patrolling between point A and point B should take place until the CGF detects some hostile CGF. At that point, it should engage the hostile CGF. Then, when the hostile CGF is defeated, it should return to its patrol.

The FSM for the example above is shown in Figure E.1. This FSM contains two states: (1) the *Patrol* state, shown at the top, and (2) the *Engage* state, shown at the bottom. The FSM has two transitions between the states. The CGF starts in the *Patrol* state, and flies between point A and point B. When the CGF is in the *Patrol* state and detects a hostile CGF, a transition from the *Patrol* state to the *Engage* state takes place. While in the *Engage* state, the CGF performs some actions with the goal of defeating the hostile CGF. Once the hostile CGF is defeated in some way, our CGF transitions back from the *Engage* state to the *Patrol* state, and then continues its patrol.

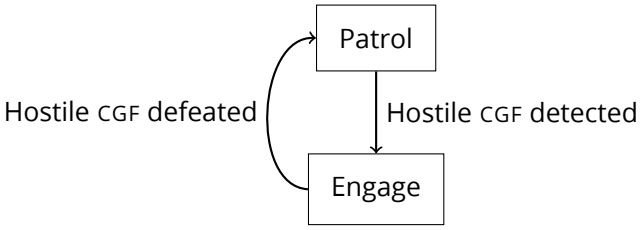


**Figure E.1** A basic example of an FSM as a behaviour model.

We translate the FSM from Figure E.1 into rules as follows. The resulting rules are shown in Listing E.1. In the first rule (shown on line 1–3), we define the behaviour that the CGF should perform when it is in the *Patrol* state. We call this type of rule a *state rule*. When the CGF is in the *Patrol* state, it should fly between point A and point B. We capture this behaviour in the rule by introducing control statements (e.g., *if/then/else* or *while*) into the consequence of the rule.

**Listing E.1** The FSM from Figure E.1 expressed in the form of rules.

```
1 in_state(Patrol) →
2     if near(point_A) then fly_towards(point_B)
3     else if near(point_B) then fly_towards(point_A);
4 in_state(Patrol) and hostile_CGF_detected() →
    change_state_to(Engage);
5 in_state(Engage) → attack(detected_hostile_CGF);
6 in_state(Engage) and is_defeated(detected_hostile_CGF) →
    change_state_to(Patrol);
```

In the second rule (line 4) we define the transition from the *Patrol* state to the *Engage* state. We call this rule a *transition rule*. When the CGF is in the *Patrol* state and it detects a hostile CGF, it transitions to the *Engage* state.

The third rule (line 5) defines the behaviour for the *Engage* state. Here, we tell the CGF to attack the detected hostile CGF. Afterwards, when the hostile CGF is defeated, the fourth rule fires (line 6) and the CGF returns to its patrol between point A and point B.

The resulting rules can now be stored in a rulebase, which serves as the input for dynamic scripting. A script with state rules and transition rules then becomes the implementation of a FSM. By creating variations of the state rules and the transition rules, and storing these variations in the rulebase as well, dynamic scripting is able to explore the space of possible FSMs.

## E.2 The modified dynamic scripting algorithm

In the original description of dynamic scripting (Spronck et al., 2006), rules are selected in a probabilistic manner, under the assumption that all rules are valid choices for inclusion in a script. However, when the rulebase is filled with state rules and transition rules, a problem arises: namely, the rules that are selected for inclusion in a generated script must together form a valid FSM. We define an invalid FSM as one that contains unreachable states.

To help dynamic scripting create valid FSMs, we restrict the generation of FSMs to those that follow a specific template. We define a template to be a collection of (1) states and (2) transitions between two states, without any specification of the behaviour that these states and transitions represent. Henceforth, we use the term *element* to refer to either a state or a transition. For example, the FSM shown in figure 7.2 has four elements: (1) the Patrol state, (2) the Engage state, (3) the transition from Patrol to Engage, and (4) the transition from Engage to Patrol. The use of a template allows us to define the structure that a FSM should follow, thereby providing a guarantee that the FSM is valid. However, templates make it possible to leave the choice of *implementation*

**Listing E.2** Modified script generation algorithm.

```
1  # input:    rulebase, fsm_template
2  # output:   script
3  script = []
4  for element in fsm_template:
5      sum_of_weights = 0
6      candidate_rules = []
7      for rule in rulebase:
8          if rule.is_implementation_of(element):
9              candidate_rules.append(rule)
10             sum_of_weights += rule.weight
11     if sum_of_weights == 0:
12         selected_rule = random.choice(candidate_rules)
13         script.append(selected_rule)
14     else:
15         selected_rule = roulette_wheel(candidate_rules)
16         script.append(selected_rule)
17 return script
```

(i.e., the actual behaviour for states, and the specific conditions on which transitions are made) to dynamic scripting.

We replace the original script generation algorithm (Spronck et al., 2006, p. 224, Algorithm 1) by the new algorithm shown in Listing E.2. The new algorithm takes as input a rulebase and an FSM template. First, an empty script is created (line 3). Next, for each element in the FSM template, an implementation is selected for inclusion in the script (line 4–16). This is done per element by first filtering out the candidate rules in the rulebase that are an implementation of that element (line 8–10). From these candidate rules, an implementation is selected by means of the original weight-proportionate roulette wheel selection (as explained in Subsection 2.3.3).

## E.3 Summary

In this appendix, we enabled the dynamic scripting algorithm to generate FSMs. In some cases, the use of scripts as a behaviour model allows CGFs to jump erratically between behaviours. The use of FSMs provides the CGFs with a sense of state, so that the CGFs will only display behaviour that is relevant to the state they are in.

Two steps were required for generating FSMs by means of dynamic scripting. First, we translated the states and the transitions between the states to the form of rules. This way, the

states and transitions can be treated as rules in the rulebase of dynamic scripting. Second, we altered the mechanism by which the dynamic scripting algorithm selects rules from its rulebase, so that the generated FSMs follow a pre-specified template. The use of this templates ensures that all generated FSMs are valid, i.e., they contain states and the appropriate transitions between the states. By enabling the dynamic scripting algorithm to generate FSMs, we gain the benefits of FSMs (i.e., the concept of state and the possibility of graphical representation) while maintaining the benefits of dynamic scripting (i.e., easily inspectable rules and rulebases, and diverse combinations of behaviour from the rulebase).

## Appendix F

# The Assessment Tool for Air Combat CGFs

In this appendix, we present our implementation of the ATACC questionnaire. We implemented the ATACC as a single-page form. This form was used by expert assessors to assess the behaviour of CGFs in human-in-the-loop simulations (see Chapter 7).

On the form, we included (a) the nine rating items of the ATACC, and (b) four fields for additional information. The nine rating items of the ATACC are discussed in Section 6.4. The four fields are labelled as follows: (1) *Tactical*, (2) *Set code*, (3) *Start time*, and (4) *Operational status*. Below, we explain these four fields.

**Tactical** The tactical is a personal code name that is used for both operational security and convenience. We included the tactical of the assessors to be able to quickly refer to specific forms that were filled in.

**Set code** In the preparation of the recorded human-in-the-loop simulations, we assigned a code consisting of a letter and a number to each specific encounter (a.k.a. a *setup* or *set* by the assessors) that was recorded. Each assessor was provided with a sheet that showed all the set codes in an individually randomised order. The assessors were instructed to (1) take the codes in the order that they were listed on the sheet, then (2) use each code to look up the encounter belonging to that set in the PCDS program that was running on their laptop, after which they were to (3) take an empty form, note down the code, and assess the behaviour of the CGFs in the recording. The set code field on the form acted as a control to ensure that the assessors viewed the recorded encounters in the order that was assigned to them.

**Start time** When an assessor used a set code to look up a recorded encounter in the PCDS program, PCDS displayed a time index for the start of that encounter. The assessors were

instructed to write down that time index in the start time field. The start time field acted as a control that allowed us to determine whether the assessors had correctly selected and viewed the encounter which was indicated by the set code.

**Operational status** The operational status of the assessors indicates their level of experience. The seven options that are provided (i.e., wingman, 2FL, 4FL, MC, IP, WIP, and TIP) refer to the qualifications that can be obtained by the assessors.

The following page shows the exact form that was used in Chapter 7.

**Assessment Tool for Air Combat CGFs**

**Tactical:**

**Set code:**

**Start time:**

**Operational status:** Wingman / 2FL / MC / WIP / 4FL / IP / TIP

| | Never | Rarely | Sometimes | Often | Always |
|---|---|---|---|---|---|
| **Red air forced blue air to change their tactical plan.** | O | O | O | O | O |
| **Red air forced blue air to change their shot doctrine.** | O | O | O | O | O |
| **Red air was within factor range.** | O | O | O | O | O |
| **Blue air was able to fire without threat from red air.** | O | O | O | O | O |
| **Red air acted on blue air's geometry.** | O | O | O | O | O |
| **Red air acted on blue air's weapons engagement zone.** | O | O | O | O | O |
| **Red air flew with kinematic realism.** | O | O | O | O | O |
| **Red air's behaviour was intelligent.** | O | O | O | O | O |

| | Strongly disagree | Disagree | Undecided | Agree | Strongly agree |
|---|---|---|---|---|---|
| **Red air's behaviour tested blue air's tactical air combat skills.** | O | O | O | O | O |