Correct-ish by Design: From Upfront Verification to Continuous Monitoring of LLM Generated Code

Bernhard K. Aichernig[1,2] and Klaus Havelund[3⋆]

[1] Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria
[2] Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria
[3] Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. As developers increasingly rely on Large Language Models (LLMs) to generate code, the pace of software development is accelerating beyond the capabilities of traditional design-time verification and testing methods. We predict a paradigm shift towards continuous monitoring to complement and eventually supersede upfront verification. By embracing a "correct-ish by design" philosophy, we acknowledge the inevitability of imperfections in LLM-generated code. We anticipate an adaptive approach where real-time monitoring and feedback mechanisms are employed to detect, diagnose, and rectify issues as they emerge in the field. This continuous monitoring strategy not only ensures sustained software reliability and performance, but also provides valuable insights into LLM behavior, facilitating iterative improvements. Specifically, we use an LLM to generate Python code from a formal specification written in the VDM specification language, accessible as a PDF document. The VDM specification formalizes aspects of NASA's SAFER rescue system, which uses small thrusters on a backpack to let astronauts maneuver and return safely to the spacecraft during spacewalks in case they become untethered. We experiment with property-based testing, and by using two Python programs, both generated from the specification by the LLM in two different developments, to monitor each other during runtime.

1 Introduction

Large Language Models (LLMs) are becoming important tools for the software developer. They may even replace the software developer at some point. Especially code generation from natural language prompts is already now practical and used in real development. On the negative side, LLM generated code is (still) known not to be reliably correct, and even if code generation from natural

⋆ The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

language was perfect, the natural language can be imprecise and incomplete. Programmers currently handle this problem by *reviewing* the generated code, and of course *testing* it. However, as LLMs become used more extensively, more code will be automatically generated and full code review will not be practical. At that point, automated program analysis becomes critical.

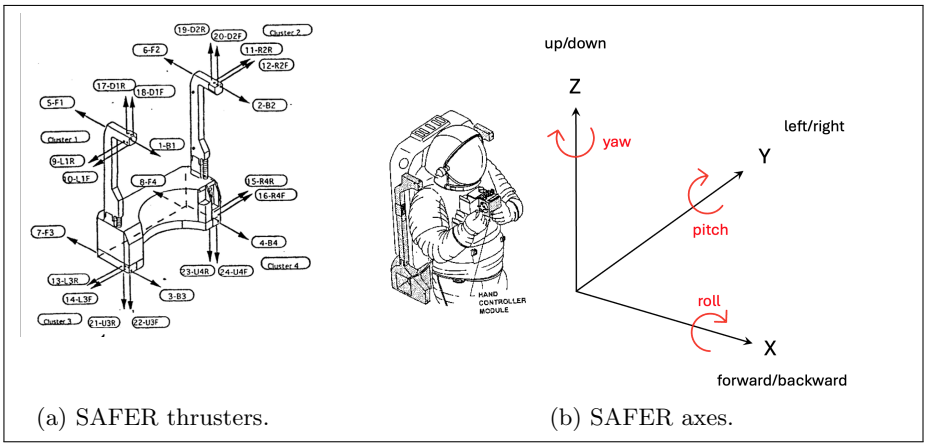
One such program analysis approach is *Runtime Verification* (RV), where the program is monitored during its execution and checked against some other formalization of the expected behavior, either during test, or in deployment. If we assume that software may not be fully correct (only "correct-ish") by design, continuous monitoring may be the only way to improve trust in at least the *execution* of the generated code. In this paper we investigate this idea by letting ChatGPT [9] generate Python code from an existing formal mathematical specification, described in [3, 4], of NASA's SAFER protocol [11], which controls a lightweight backpack propulsion system for an astronaut, who can use it to maneuver safely back to the space vehicle in case of an erroneous separation.

The SAFER protocol is in [3, 4] formally specified in the VDM (Vienna Development Method) specification language [8, 17, 12], in particular the VDM specification language standard VDM-SL [18], from here referred to as VDM. The specification is written in an executable subset of VDM, making translation to Python rather direct. In fact, the executable subset of VDM has many similarities with Python, as we will demonstrate in this paper. In [15] we studied the relationship between VDM and Scala. Many of the observations there can be carried over to Python.

Runtime verification [7, 16] is, as mentioned, the automated verification that an execution of a program (or system) satisfies a formalized specification. The monitored program must be annotated to drive the monitor (online monitoring)[1]. Examples of formal specification formalisms include simple assertions in the code, temporal logics, regular expressions, state machines, production rules, and stream processing formalisms. Differential testing [19, 14], which we apply in this paper, can be considered as a variant of runtime verification where the specification is "another program", the *reference implementation*, with the same expected functionality. The reference implementation can either be an alternative full implementation, as in [19] where different C compilers are compared, or it can be an abstract smaller program, as in [14]. The idea of comparing the execution of an implementation with that of a more abstract implementation, or model, has been formalized in terms of proofs with refinement mappings [1, 2]. In [5] we explored the idea of monitoring such a refinement mapping between an LLM generated implementation in Scala and its abstract model.

In this paper we examine runtime verification with a reference implementation, as well as with assertions. The two authors performed separate *developments*, referred to as A and B, with ChatGPT to generate two Python versions of the VDM specification in the 21 page technical report [4]. We then used these as each other's reference implementation, and in this way found an error introduced

[1] The instrumented code can alternatively generate a log which is then checked against the specification post-mortem.



(a) SAFER thrusters. (b) SAFER axes.

Fig. 1: Thrusters and axes of movement.

in one of them by ChatGPT. It leads to the quite interesting observation that with LLMs one can imagine generating multiple versions of software checking each other, leading to an inexpensive way of applying N-version programming [6]. The paper also demonstrates automated translation from a high-level mathematical specification in a PDF document to code.

The paper is organized as follows. Section 2 introduces the SAFER protocol. Section 3 describes our method. The VDM specification in [4], which we translate to Python using ChatGPT, consists of six modules, which we discuss in sections 4-9. Finally, in Section 10 we discuss lessons learned.

2 The SAFER System

The Simplified Aid for EVA Rescue (SAFER) [11] is a small propulsion backpack designed by NASA to assist astronauts during Extra Vehicular Activities (EVAs) in space. It is primarily meant to be activated only in emergency scenarios when an astronaut accidently becomes separated from the spacecraft. SAFER attaches directly to the astronaut's backpack. It is controlled by the astronaut using a small box sitting on the astronaut's chest, with knobs to control thrusters mounted on the backpack.

The SAFER propulsion system specifically consists of 24 thrusters positioned on the backpack, see Figure 1a, allowing *movement* in all directions, along the X (forward, backward), Y (left, right), and Z (up, down) axes, which are also called the translational axes, see Figure 1b. In each corner of the backpack are six thrusters, three in the front and three on the rear, basically a thruster for each axis X, Y, and Z. The thrusters are numbered 1-24, and are in addition named with letters indicating direction in which they push the astronaut (B=Back, F=Forward, L=Left, R=Right, U=Up, D=Down), numbered in which quadrant

they sit (1=upper left, 2=upper right, 3=lower left, 4=lower right), and finally, if there are two thrusters next to each other, whether they sit on the Rear (R) or Front (F). In addition, it allows *rotation* around these axes: ROLL (around the X axis, tilting from side to side), PITCH (around the Y axis, tilting forward, backward), and YAW (around the Z axis, turning left, right). These six options (moving along three axes and rotating around three axes) are also referred to as the *six degrees of freedom*.

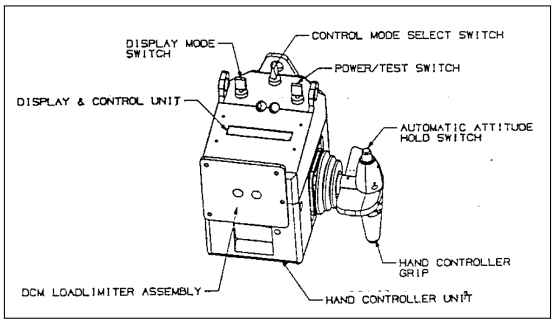


Fig. 2: The hand grip.

Commands to the thrusters are initiated through movements of a joystick, called the *hand grip*, sitting on the right-hand side of a control box, which is placed on the astronaut's chest, see Figure 2. It allows both translational and rotational controls. The joystick can be moved forward and backward (X axis), left-right by pushing it in or pulling it out (Y axis), and can be moved up or down (Z axis). In addition, this joystick can be twisted, always meaning PITCH rotation. A separate knob can switch between translation mode (moving along the X, Y, and Z axes) or rotation mode, rotating around these axes. In translation mode the movements on the X, Y, and Z axes mean exactly that: movement, and twisting means PITCH around the Y-axis. In rotation mode, movement along the X-axis still means exactly that: movement, whereas movement along the Y and Z axes mean respectively YAW and ROLL, and twisting still means PITCH.

In addition to these maneuvering options, the SAFER system can by a button be put into Automatic Attitude Hold (AAH) mode, which stabilizes the astronaut's orientation movements automatically. This means canceling any rotations, and keeping the astronaut stable rotation wise. Uncontrolled rotation can be very disorienting. The button is pushed down and automatically comes back up on release. AAH mode can be turned off again by pressing the button twice within 0.5 seconds. If in AAH mode, and the astronaut issues a rotation command on a specific axis, then that takes priority and cancels the AAH module's control over the rotation around that axis. However, if the astronaut is commanding a rotation while activating the AAH module, then the AAH module takes over full control of that axis and ignores any rotation commands

from the astronaut on that axis until AAH is switched off. Finally, translational commands are prioritized with respect to each other, with X-axis movements being the highest priority, followed by Y, then Z. If rotational and translational commands are simultaneously issued, rotations have higher priority than translational commands.

The VDM-Specification consists of six modules, in this paper presented from bottom to top. The AUX (AUXiliary) module provides auxiliary definitions, such as fundamental types. The HCM (Hand Controller Module) specifies how hand grip operations are translated into logic commands for the thruster selection software. The AAH (Automatic Attitude Hold) module, mentioned above, handles the automated control of near-zero rotation, when activated by the astronaut. The TS (Thruster Selection) module computes the thrusters to be selected for thrusting. The SAFER module is the top-level module executing every cycle, calling functions in the other modules. The TEST module provides functions for testing SAFER.

As the reader may have observed, the logic is not completely trivial. To enhance astronaut safety, the SAFER system enforces specific operational constraints. It limits the number of simultaneously activated thrusters to four, reducing complexity and potential conflicts in thruster firings. Furthermore, the thruster selection logic is designed to avoid opposing thrusters firing simultaneously, which could result in ineffective propulsion and wasted fuel.

3 Methodology

We had three documents available describing the SAFER protocol. These included the original NASA report [11] providing guidelines on the use of formal methods, and using the SAFER protocol as an example, which in the report is explained and formalized using the PVS [20] theorem prover. In addition, we had access to two papers describing its formalization in VDM, including a workshop paper [3] with explanations and some specifications, and a 21 page technical report [4] containing the full VDM specification consisting of 417 lines of VDM (not counting comments and blank lines) with only few explanations. The report [4] ended up being our main source (input to ChatGPT) for the translation.

Each of the two authors independently developed the SAFER system in Python with ChatGPT 4o, referred to as *developments A and B*. A typical interactive workflow involved (1) providing ChatGPT with the PDF [4] of the VDM specification, (2) ask ChatGPT to produce Python code for a particular module, (3) reviewing of the code, (4) ask ChatGPT to correct the code, until satisfactory code was generated. Early attempts to provide ChatGPT with the complete technical report and ask it to produce the complete Python code were not very promising. Hence, we decided to develop the system module by module, starting with the simplest AUX module. In the following sections, we discuss each of the modules AUX, HCM, AAH, TS, SAFER, and TEST, separately. Table 1 shows the number of prompts needed for each module for each of the

developments A and B, as well as the total number of lines of Python code generated (not counting comments and blank lines) from the 417 lines of VDM.

Development	**AUX**	**HCM**	**AAH**	**TS**	**SAFER**	**TEST**	total prompts	Python LOC
A	4	3	20	64	11	8	110	441
B	6	2	18	38	12	29	105	503

Table 1: Number of prompts and Python LOC generated from 417 VDM LOC.

When all the Python was generated, we tested if the pre/post conditions and safety invariants were satisfied. For this an exhaustive test-case generator already present in the VDM specification was used. It basically enumerates all possible combinations of input commands in a manner similar to property-based testing [10]. Finally, when the safety was successfully tested this way, we used the two developments as each other's reference implementation, checking that they were consistent, i.e., that the same thrusters are fired.

Note that in the generated Python code we allow ourselves to occasionally insert newlines to improve layout for reading, also in cases where Python does not allow such newlines.

4 Module AUX

Module AUX is the simplest module since it contains auxiliary definitions without any dependency on other modules. The VDM types of this module specification are shown in Figure 3. Note that it is classic VDM-style to model a problem as a collection of type definitions in this way.

The first three enumerated types (Lines 43-45) define the commands that can be issued along an axis (NEG, ZERO, POS), the possible translation axes (X, Y, Z) and the possible rotations (ROLL, PITCH, and YAW). A translation command, *TranCommand* (Line 46), is a mapping[2] from the X, Y, and Z axes to axis commands. The type of *TranCommand* is in fact defined as the subtype (the keyword *inv*) of such mappings, where the domain equals the set of all translation axes. The value tran-axis-set has elsewhere been defined as the set $\{X, Y, Z\}$. Rotation commands, *RotCommand*, is defined similarly (Line 47). Finally, a hand grip command issued by the astronaut, *SixDofCommand* (Line 48), is a composition (record) consisting of a translation command and a rotation command[3].

We started with the following prompt:

[2] The type $A \xrightarrow{m} B$ denotes in VDM the set of finite maps from A to B. This corresponds to type `Dict[A,B]` of dictionaries in Python.

[3] The construct $id :: id_1 : ty_1, ..., id_n : ty_n$ in VDM defines a record type named id, with fields $id_1, ..., id_n$ of types $ty_1, ..., ty_n$ respectively.

types

43.0 $AxisCommand = \text{NEG} \mid \text{ZERO} \mid \text{POS};$

44.0 $TranAxis = \text{X} \mid \text{Y} \mid \text{Z};$

45.0 $RotAxis = \text{ROLL} \mid \text{PITCH} \mid \text{YAW};$

46.0 $TranCommand = TranAxis \xrightarrow{m} AxisCommand$

.1 inv $cmd \triangleq \mathsf{dom}\ cmd = tran\text{-}axis\text{-}set;$

47.0 $RotCommand = RotAxis \xrightarrow{m} AxisCommand$

.1 inv $cmd \triangleq \mathsf{dom}\ cmd = rot\text{-}axis\text{-}set;$

48.0 $SixDofCommand :: tran : TranCommand$

.1 $\qquad\qquad\qquad\qquad\quad rot : RotCommand$

Fig. 3: The AUX module.

Prompt: I would like to translate the formal VDM spec to Python in steps. First translate the AUX module in Section 6. Translate it in one piece.

ChatGPT read and printed the VDM spec in ASCII format, which showed that it was able to read the PDF perfectly, including VDM's special mathematical symbols such as the map $\xrightarrow{m}$ symbol. It then performed a largely flawless translation to Python that compiled and was executable, although a few prompt iterations were needed to refine the result, as explained in the following. The result of the translation is shown below.

```
class AxisCommand(Enum):
    NEG = "Neg"
    ZERO = "Zero"
    POS = "Pos"

class TranAxis(Enum):
    X = "X"
    Y = "Y"
    Z = "Z"

class RotAxis(Enum):
    ROLL = "Roll"
    PITCH = "Pitch"
    YAW = "Yaw"

TranCommand = Dict[TranAxis, AxisCommand]
def is_valid_tran_command(cmd: TranCommand) -> bool:
    return set(cmd.keys()) == tran_axis_set

RotCommand = Dict[RotAxis, AxisCommand]
def is_valid_rot_command(cmd: RotCommand) -> bool:
```

```
    return set(cmd.keys()) == rot_axis_set

@dataclass
class SixDofCommand:
    tran: TranCommand
    rot: RotCommand

    def __post_init__(self):
        if not is_valid_tran_command(self.tran):
            raise ValueError(f"Invalid TranCommand")
        if not is_valid_rot_command(self.rot):
            raise ValueError(f"Invalid RotCommand")
```

VDM map types are correctly translated to Python dictionaries, and the record with two fields is converted into a Python class with two attributes. It overrides the `__post_init__` method in the class `SixDofCommand`, which is called automatically after the object is initialized, checking the subtype constraints. In development B, ChatGPT left out the definitions of the types `TranCommand` and `RotCommand`, and instead inlined these in the definition of `SixDofCommand`. Although this was correct, we asked it to name these types. In both developments, it forgot to translate the subtype (*inv*) constraints on these types. That is, the functions `is_valid_tran_command`, `is_valid_rot_command`, and `__post_init__` in the class `SixDofCommand` were not in the original translation. Finally, the class `SixDofCommand` was originally in development B defined using Python's classic, but verbose, `__init__` function. The class could be written shorter using Python's dataclass concept (a shorthand for writing `__init__` functions). So in this development we provided the following prompt:

> **Prompt:** First of all, you missed the invariant predicates. Second, you could use a dataclass for type SixDofCommand.

which resulted in the shown solution. We were generally impressed by this encouraging result of converting a VDM specification from a PDF document to Python code. Not a single line of this module implementation was programmed by hand. Note that VDM's enumeration types are more concise and elegant.

5 Module HCM

The Hand Control Module (HCM) models the hand grip controller of the SAFER system and its commands. It imports definitions from the AUX module described in Section 4. The module defines four important types shown in Figure 4, which define how the astronaut operates the handgrip. The type *SwitchPositions* defines the choice between translation mode or rotation mode (*ControlModeSwitch*), and whether the AAH button is pressed down or not (*ControlButton*). Finally, the type *HandGripPosition* defines the astronaut's operation of the hand grip. Note that this represents the physical operation, while the type *SixDofCommand* from the AUX module defines the resulting logical interpretation of that operation. Furthermore, note that *vert* is a move on the Z-axis (up, down), *horiz*

```
31.0  SwitchPositions :: mode : ControlModeSwitch
 .1                       aah : ControlButton;

32.0  ControlModeSwitch = ROT | TRAN;

33.0  ControlButton = UP | DOWN;

34.0  HandGripPosition :: vert : AUX`AxisCommand
 .1                        horiz : AUX`AxisCommand
 .2                        trans : AUX`AxisCommand
 .3                        twist : AUX`AxisCommand
```

Fig. 4: The HCM module, hand grip operation types.

is a move on the X-axis *forward, backward*, *trans* (transverse) is a move on the Y-axis (left, right), and *twist* corresponds to twisting the knob (forward, backward, always representing a PITCH rotation).

ChatGPT translates the two VDM record types to dataclasses and the two union types to Python enums as we have seen in the previous AUX module. However, in both developments ChatGPT did not import the AUX module. In development A we provided Python's error message as prompt:

> **Prompt:** Does not compile: NameError: name 'AxisCommand' is not defined.

and ChatGPT corrected it. In development B, we manually inserted this import, and type name prefixes, and fed the code back to ChatGPT. In that process, we introduced a typo, writing `aux.xisCommand.ZERO` instead of `aux.AxisCommand.ZERO`, which it detected and corrected.

Figure 5 shows the main function *GripCommand* mapping the physical position of the hand grip and the control-mode switch to the corresponding logical six-degree-of-freedom command. The control-mode switch toggles between translation (*TRAN*) and rotation mode (*ROT*). The movements of the hand grip are horizontal (forward, backward), transverse (left, right), and vertical (up, down). In translation mode (*TRAN*) these grip-movements result in movements of the astronaut along the X, Y, and Z axes as shown in Figure 1. In addition, the astronaut can twist the hand grip causing it to pitch. In rotation mode, the movements along the Y axis trigger yawing and along the Z axis trigger rolling. Table 2 summarizes the effects of the grip movements in both modes.

After resolving the issue with importing AUX, ChatGPT correctly translated *GripCommand* to Python in development B. However, in development A it introduced a wrong logic and an additional prompt pointing this out was necessary. Here is the generated code.

```
GripCommand : HandGripPosition × ControlModeSwitch → AUX'SixDofCommand
GripCommand (mk-HandGripPosition (vert, horiz, trans, twist), mode) ≜
  let tran = {X ↦ horiz,
              Y ↦ if mode = TRAN
              then trans
              else ZERO,
              Z ↦ if mode = TRAN
              then vert
              else ZERO},
      rot = {ROLL ↦ if mode = ROT
             then vert
             else ZERO,
             PITCH ↦ twist,
             YAW ↦ if mode = ROT
             then trans
             else ZERO} in
  mk-AUX'SixDofCommand (tran, rot)
```

Fig. 5: The HCM module, function GripCommand.

	horiz	**trans**	**vert**	**twist**
TRAN	X	Y	Z	PITCH
ROT	X	YAW	ROLL	PITCH

Table 2: Hand grip command effects: X = move(forward, backward), Y = move(left, right), Z = move(up, down), YAW = turn(left, right), ROLL = turn(side, side), PITCH = turn(up, down).

```python
def grip_command(hand_grip: HandGripPosition, mode: ControlModeSwitch) ->
     SixDofCommand:
    tran_command = {
        TranAxis.X: hand_grip.horiz,
        TranAxis.Y: hand_grip.trans if mode == ControlModeSwitch.TRAN
            else AxisCommand.ZERO,
        TranAxis.Z: hand_grip.vert if mode == ControlModeSwitch.TRAN
            else AxisCommand.ZERO,
    }
    rot_command = {
        RotAxis.ROLL: hand_grip.vert if mode == ControlModeSwitch.ROT
            else AxisCommand.ZERO,
        RotAxis.PITCH: hand_grip.twist,
        RotAxis.YAW: hand_grip.trans if mode == ControlModeSwitch.ROT
            else AxisCommand.ZERO,
    }
    return SixDofCommand(tran=tran_command, rot=rot_command)
```

Note that in the VDM specification, the first argument to the function is matched against *mk-HandGrip-Position(vert, horiz, trans, twist)* using pattern matching in an argument position. Pattern matching in this location is not supported in Python. In development B ChatGPT did not have great difficulties with this module. It perfectly understood the VDM specification. In development A it hallucinated and introduced a wrong logic which, however, could be easily corrected.

```
ButtonTransition : EngageState × HCM‘ControlButton × AUX‘RotAxis-set × ℕ × ℕ
                   → EngageState
ButtonTransition (estate, button, active, clock, timeout) ≜
  cases mk-(estate, button) :
    mk-(AAH_OFF, UP) → AAH_OFF,
    mk-(AAH_OFF, DOWN) → AAH_STARTED,
    mk-(AAH_STARTED, UP) → AAH_ON,
    mk-(AAH_STARTED, DOWN) → AAH_STARTED,
    mk-(AAH_ON, UP) → if AllAxesOff (active)
                      then AAH_OFF
                      else AAH_ON,
    mk-(AAH_ON, DOWN) → PRESSED_ONCE,
    mk-(PRESSED_ONCE, UP) → AAH_CLOSING,
    mk-(PRESSED_ONCE, DOWN) → PRESSED_ONCE,
    mk-(AAH_CLOSING, UP) → if AllAxesOff (active)
                           then AAH_OFF
                           elseif clock > timeout
                           then AAH_ON
                           else AAH_CLOSING,
    mk-(AAH_CLOSING, DOWN) → PRESSED_TWICE,
    mk-(PRESSED_TWICE, UP) → AAH_OFF,
    mk-(PRESSED_TWICE, DOWN) → PRESSED_TWICE
  end
```

Fig. 6: The AAH module, function ButtonTransition.

6 Module AAH

This module specifies the behavior of the automatic attitude hold (AAH). Its purpose is to stop any uncontrolled rotation, which can seriously disorient an astronaut. It has state-full behavior following a protocol specified as a state-machine, with the current state stored in the variable *toggle*, and defining when it is *on* or *off* or in some transient state in between, based on the operation of the hand grip AAH button. When on, it can be turned off by two subsequent button clicks within 0.5 seconds. Its logic is furthermore complicated due to the priority of AAH compared to manually issued rotation commands. When active, it controls a set of orientation axes stored in the *RotAxis* set *active-axes*. The module defines two main functions: *ButtonTransition* and *Transition*.

The ButtonTransition Function. The *ButtonTransition* function[4], see Figure 6, computes the next state of the AAH state machine, based on the current state, *estate*, the position of the AAH *button* on the hand grip (*DOWN* when pressed and *UP* when released), the set of *active* axes that it currently controls, and a *clock* value (the time) as well as a *timeout* value, which is the clock value at which a timeout will occur during a double button click to turn the AAH

[4] Some VDM fragments are red, as in the original document [4], identifying fragments that were not covered during testing of the VDM specification performed by the authors of [4].

off. For example (see the second and third transition), suppose the state machine is in the state *OFF*, and the astronaut pushes the button, entering the state *AAH_started*, and then releases the button, then we enter the *ON* state. Snippets of the generated Python code for this module are shown below.

```
def button_transition(estate: EngageState, button: hcm.ControlButton,
        active: Set[aux.RotAxis], clock: int, timeout: int) -> EngageState:
    match (estate, button):
        ...
        case (EngageState.AAH_OFF, hcm.ControlButton.DOWN):
            return EngageState.AAH_STARTED
        case (EngageState.AAH_STARTED, hcm.ControlButton.UP):
            return EngageState.AAH_ON
        ...
        case (EngageState.AAH_ON, hcm.ControlButton.UP):
            if clock > timeout:
                return EngageState.AAH_OFF
            else:
                return EngageState.AAH_ON
        ...
        case (EngageState.AAH_CLOSING, hcm.ControlButton.UP):
            return EngageState.AAH_OFF if not active else (
              EngageState.AAH_ON if clock > timeout else EngageState.AAH_CLOSING)
        ...
```

In development B, it first wrongly converted the type *HCM'ControlButton* of the first argument to `aux.ControlButton`. Also, in the same development, the first translation generated `if`-statements, rather than `match-case` statements. An interesting observation here is that there were "missing" explicit cases compared to the VDM specification: it had cleverly detected all the cases where the returned value is the incoming `estate` value, and captured these cases instead with a default '`return estate`' statement at the end. However, we asked ChatGPT also in these cases to generate `case` statements, similar to the VDM specification.

In the same development, the case `(AAH_ON, UP)` was missing. We informed ChatGPT about this and it generated a case returning the `AAH_ON` state unconditionally. However, this is wrong, if `active` is empty, corresponding to the AAH module not controlling any axes, it should transition to the `AAH_OFF` state, which we pointed out. As a result it attempted to correct the mistake as shown in the above final code. However, this is also wrong, and was only detected using differential testing as described in Section 9.3.

The Transition Function. The VDM function *Transition* shown in Figure 7 updates the state *toggle* of the state machine as well as the variables *active-axes* and a variable *ignore-hcm* (to be explained), and a *timeout* in case the button has been pressed once. By default, when turned on, AAH controls all the three rotation axes. However, if during AAH mode the astronaut commands a rotation axis, that command takes over and deactivates the AAH on that axis (it is removed from *active-axes*). This is a truth, with a modification. In case the astronaut is already commanding a rotation axis when turning on the AAH, that and subsequent commands on that rotation axis, will be ignored until the AAH is turned *OFF* (modeling that the astronaut wants to cancel that command by turning AAH on). Such axes are stored in the variable *ignore-hcm*. The function

$$
\begin{array}{l}
Transition : HCM`ControlButton \times AUX`SixDofCommand \times \mathbb{N} \xrightarrow{o} () \\
Transition\,(button\text{-}pos, hcm\text{-}cmd, clock) \triangleq \\
\quad \mathsf{let}\ engage = ButtonTransition\,(toggle, button\text{-}pos, active\text{-}axes, clock, timeout), \\
\qquad starting = (toggle = \mathrm{AAH\_OFF}) \wedge (engage = \mathrm{AAH\_STARTED})\ \mathsf{in} \\
\quad (active\text{-}axes := \{a \mid a \in AUX`rot\text{-}axis\text{-}set \cdot \\
\qquad\qquad\qquad\qquad starting \vee \\
\qquad\qquad\qquad\qquad (engage \neq \mathrm{AAH\_OFF} \wedge a \in active\text{-}axes \wedge \\
\qquad\qquad\qquad\qquad\ (hcm\text{-}cmd.rot\,(a) = \textsc{Zero} \vee a \in ignore\text{-}hcm))\}; \\
\qquad ignore\text{-}hcm := \{a \mid a \in AUX`rot\text{-}axis\text{-}set \cdot \\
\qquad\qquad\qquad\qquad (starting \wedge hcm\text{-}cmd.rot\,(a) \neq \textsc{Zero}) \vee \\
\qquad\qquad\qquad\qquad (\neg\, starting \wedge a \in ignore\text{-}hcm)\}; \\
\qquad timeout := \mathsf{if}\ toggle = \mathrm{AAH\_ON} \wedge engage = \textsc{pressed\_once} \\
\qquad\qquad\qquad\ \mathsf{then}\ clock + click\text{-}timeout \\
\qquad\qquad\qquad\ \mathsf{else}\ timeout; \\
\qquad toggle := engage);
\end{array}
$$

Fig. 7: The AAH module, function Transition.

is by ChatGPT translated as follows, after a few interactions with ChatGPT that we will discuss. The translation is nearly perfect, translating set comprehensions correctly.

```
def transition(button_pos: hcm.ControlButton, hcm_cmd: aux.SixDofCommand, clock: int):
    engage = button_transition(AAH.toggle, button_pos,
                               AAH.active_axes, clock, AAH.timeout)
    starting = AAH.toggle == EngageState.AAH_OFF and
               engage == EngageState.AAH_STARTED
    AAH.active_axes = {
        a for a in aux.rot_axis_set if starting or
            (engage != EngageState.AAH_OFF and
             a in AAH.active_axes and
             (hcm_cmd.rot[a] == aux.AxisCommand.ZERO or a in AAH.ignore_hcm))
    }
    AAH.ignore_hcm = {
        a for a in aux.rot_axis_set if
          (starting and hcm_cmd.rot[a] != aux.AxisCommand.ZERO)
          or
          (not starting and a in AAH.ignore_hcm)
    }
    if AAH.toggle == EngageState.AAH_ON and engage == EngageState.PRESSED_ONCE:
        AAH.timeout = clock + CLICK_TIMEOUT
    AAH.toggle = engage
```

Some minor points in development B was that the global state was passed as an object to the function, and the function then updated this object as a side-effect. This approach is not wrong, but does not quite follow the VDM specification. In this module we observed most of the hallucinations, like in development A, missing arguments. In that development, we had to provide the technical report again, which partly resolved the issues. We also realized that ChatGPT produced code that was different from the VDM version, which made it more difficult to review. For example, it used different parameter and variable names. The lesson we learned was to keep the code as close as possible to the VDM style to ease reviewing. Hence, we had to backtrack to previous versions several times. At one point, we had to make a manual correction. Reviewing of

$$
\begin{array}{l}
RotCmdsPresent : AUX`RotCommand \to \mathbb{B} \\
RotCmdsPresent\,(cmd) \triangleq \\
\quad \exists\, a \in \mathsf{dom}\ cmd \cdot cmd\,(a) \neq \textsc{Zero};
\end{array}
$$

Fig. 8: The TS module function *RotCmdsPresent*.

the translated code was harder compared to the previous modules. The hope was to find any remaining issues during the runtime checking.

7 Module TS

The TS module models the Thruster Selection logic. Hand Controller and AHH commands are merged together in accordance with the various priority rules. The result is that a six-degree-of-freedom command is mapped to a set of thrusters to be fired. Thruster selection tables are used to convert a command to individual actuator commands for opening suitable thruster valves. The development of this module was surprisingly difficult. For example, in development A, we needed 64 prompts to produce the correct 226 lines of Python code. In development A we always let ChatGPT generate a `main` function to quickly check the sanity of the generated code before review (development B relied on reviewing only). In this case ChatGPT dropped the `main` function for testing. Furthermore, it introduced undefined names, arbitrarily renamed existing ones, changed the logical structure, and omitted parts. A simple prompt with these observations did not help and we had to upload the PDF with the VDM specification again. It obviously lost the context. Unfortunately, this did not resolve all issues and we decided to develop this module gradually, function by function. First, we let ChatGPT translate the union type of all 24 thruster names into an `Enum` class.

```
class ThrusterName(Enum):
    B1 = auto()
    B2 = auto()
    B3 = auto()
    ...

def main():
    # Demonstrate usage of ThrusterName enum
    print("List of Thruster Names:")
    for thruster in ThrusterName:
        print(f"{thruster.name}: {thruster.value}")

if __name__ == "__main__":
    main()
```

The RotCmdsPresent Function. Next, we added the Boolean function *RotCmdsPresent* shown in Figure 8. It is an example of how a predicate with an existential quantifier gets translated to Python. First, a different parameter name was generated and we asked ChatGPT to correct this. Furthermore, the type

$$
\begin{array}{l}
PrioritizedTranCmd : AUX`TranCommand \to AUX`TranCommand \\
PrioritizedTranCmd\,(tran) \triangleq \\
\quad \textsf{if } tran\,(\mathrm{X}) \neq \textsc{Zero} \\
\quad \textsf{then } AUX`null\text{-}tran\text{-}command \dagger \{\mathrm{X} \mapsto tran\,(\mathrm{X})\} \\
\quad \textsf{elseif } tran\,(\mathrm{Y}) \neq \textsc{Zero} \\
\quad \textsf{then } AUX`null\text{-}tran\text{-}command \dagger \{\mathrm{Y} \mapsto tran\,(\mathrm{Y})\} \\
\quad \textsf{elseif } tran\,(\mathrm{Z}) \neq \textsc{Zero} \\
\quad \textsf{then } AUX`null\text{-}tran\text{-}command \dagger \{\mathrm{Z} \mapsto tran\,(\mathrm{Z})\} \\
\quad \textsf{else } AUX`null\text{-}tran\text{-}command;
\end{array}
$$

Fig. 9: The TS module, function PrioritizedTranCmd.

`RotCommand` in the type signature was newly created, since ChatGPT did not realize that it is defined in the AUX-module. After two prompts with these corrections, the function was correctly translated.

```
def RotCmdsPresent(cmd: RotCommand) -> bool:
    return any(cmd[axis] != AxisCommand.ZERO for axis in RotAxis)
```

The existential quantifier got translated into Python's `any`-function that takes a predicate and an iterable type and returns true if the predicate holds for any element. Note how ChatGPT optimized the function. Instead of VDM's domain operator *dom* it uses the enum type RotAxis which is the domain type of the map RotCommand. In this case it is correct, since the invariant in the AUX module requires that the mapping is total. Hence, *dom cmd* is equivalent to `RotAxis`. It is really interesting that ChatGPT can do this kind of code optimization during translation.

The PrioritizedTranCmd Function. This function specifies the priorities of the translational commands along the X, Y, and Z axes. The translation was fairly straightforward. In development A, an additional prompt was needed to correct the return type and to use the parameter name `tran`. Here is the resulting code.

```
def PrioritizedTranCmd(tran: TranCommand) -> TranCommand:
    prioritized_tran = null_tran_command.copy()

    if tran[TranAxis.X] != AxisCommand.ZERO:
        prioritized_tran[TranAxis.X] = tran[TranAxis.X]
    elif tran[TranAxis.Y] != AxisCommand.ZERO:
        prioritized_tran[TranAxis.Y] = tran[TranAxis.Y]
    elif tran[TranAxis.Z] != AxisCommand.ZERO:
        prioritized_tran[TranAxis.Z] = tran[TranAxis.Z]

    return prioritized_tran
```

Note that it saved an if-branch by applying an imperative style. In development B, it first came up with a version that spells out the returned map, in fact inlining the values.

$$
\begin{array}{l}
CombinedRotCmds : AUX`RotCommand \times AUX`RotCommand \times AUX`RotAxis\text{-set} \to \\
\qquad AUX`RotCommand \\
CombinedRotCmds\,(hcm\text{-}rot, aah, ignore\text{-}hcm) \triangleq \\
\quad \mathsf{let}\ aah\text{-}axes = ignore\text{-}hcm \cup \\
\qquad \{a \mid a \in AUX`rot\text{-}axis\text{-}set \cdot hcm\text{-}rot\,(a) = \textsc{Zero}\}\ \mathsf{in} \\
\quad \{a \mapsto aah\,(a) \mid a \in aah\text{-}axes\} \cup \\
\quad \{a \mapsto hcm\text{-}rot\,(a) \mid a \in AUX`rot\text{-}axis\text{-}set \setminus aah\text{-}axes\};
\end{array}
$$

Fig. 10: The TS module, function CombinedRotCmds.

```
def prioritized_tran_cmd(tran: aux.TranCommand) -> aux.TranCommand:
    if tran[aux.TranAxis.X] != aux.AxisCommand.ZERO:
        return {aux.TranAxis.X: tran[aux.TranAxis.X],
                aux.TranAxis.Y: aux.AxisCommand.ZERO,
                aux.TranAxis.Z: aux.AxisCommand.ZERO}
    elif tran[aux.TranAxis.Y] != aux.AxisCommand.ZERO:
        return {aux.TranAxis.X: aux.AxisCommand.ZERO,
                aux.TranAxis.Y: tran[aux.TranAxis.Y],
                aux.TranAxis.Z: aux.AxisCommand.ZERO}
    elif tran[aux.TranAxis.Z] != aux.AxisCommand.ZERO:
        return {aux.TranAxis.X: aux.AxisCommand.ZERO,
                aux.TranAxis.Y: aux.AxisCommand.ZERO,
                aux.TranAxis.Z: tran[aux.TranAxis.Z]}
    else:
        return aux.null_tran_command
```

It is functionally correct, but this is not following the VDM specification, which updates the *null-tran-command* variable. Three prompts were needed to obtain satisfactory code, which is almost the same as development A.

The CombinedRotCmds Function. As shown in Figure 10, this function uses sophisticated map operators and ChatGPT had difficulties interpreting them. In development A, we needed six prompts to develop this function. First, the number of arguments was wrong, then parameter names were mixed up and the logic was completely made up. Here is a prompt from development B:

> **Prompt:** Let me explain what happens in that function. aah-axes is ignore_hcm (which is a set) union the set of a in aux.rot-axis-set for which hcm-rot(a) is ZERO. This set is stored in aah-axes. Then what is returned is a map composed of two submaps: (1) a to aah(a) for a in aah-axes and (2) a to hcm-rot(a) for a in aux.rot-axis-set minus aah-axes.

It was a battle to make it follow the VDM specification. The result was as follows. Note that dictionaries are mutable, therefore this slightly obscure notation {`**aah_cmd_map, **hcm_cmd_map`} for creating a new set that is the union of two existing sets.

```
def combined_rot_cmds(hcm_rot: aux.RotCommand, aah: aux.RotCommand,
                      ignore_hcm: Set[aux.RotAxis]) -> aux.RotCommand:
```

$$
\begin{array}{l}
BFThrusters : AUX`AxisCommand \times AUX`AxisCommand \times AUX`AxisCommand \rightarrow \\
\qquad ThrusterSet \times ThrusterSet \\
BFThrusters\,(A, B, C) \triangleq \\
\quad \text{cases mk-}(A, B, C) : \\
\qquad \text{mk-}(\textsc{Neg}, \textsc{Neg}, \textsc{Neg}) \rightarrow \text{mk-}(\{\mathrm{B4}\}, \{\mathrm{B2}, \mathrm{B3}\}), \\
\qquad \text{mk-}(\textsc{Neg}, \textsc{Neg}, \textsc{Zero}) \rightarrow \text{mk-}(\{\mathrm{B3}, \mathrm{B4}\}, \{\}), \\
\qquad \text{mk-}(\textsc{Neg}, \textsc{Neg}, \textsc{Pos}) \rightarrow \text{mk-}(\{\mathrm{B3}\}, \{\mathrm{B1}, \mathrm{B4}\}), \\
\qquad \text{mk-}(\textsc{Neg}, \textsc{Zero}, \textsc{Neg}) \rightarrow \text{mk-}(\{\mathrm{B2}, \mathrm{B4}\}, \{\}), \\
\qquad \text{mk-}(\textsc{Neg}, \textsc{Zero}, \textsc{Zero}) \rightarrow \text{mk-}(\{\mathrm{B1}, \mathrm{B4}\}, \{\mathrm{B2}, \mathrm{B3}\}),
\end{array}
$$

Fig. 11: The TS module, function BfThrusters (only the first 5 of 27 cases shown).

```
    aah_axes = ignore_hcm.union({
                  a for a in aux.rot_axis_set if hcm_rot[a] == aux.AxisCommand.ZERO})
    aah_cmd_map = {a: aah[a] for a in aah_axes}
    hcm_cmd_map = {a: hcm_rot[a] for a in aux.rot_axis_set if a not in aah_axes}
    return {**aah_cmd_map, **hcm_cmd_map}
```

It did come up with an alternative correct translation, which shows "understanding" of the VDM specification, in particular set theory.

```
def combined_rot_cmds(hcm_rot: aux.RotCommand, aah: aux.RotCommand,
                      ignore_hcm: Set[aux.RotAxis]) -> aux.RotCommand:
    aah_axes = ignore_hcm.union({
                  a for a in aux.rot_axis_set if hcm_rot[a] == aux.AxisCommand.ZERO})
    combined_cmd = {
        a: aah[a] if a in aah_axes else hcm_rot[a]
        for a in aux.rot_axis_set
    }
    return combined_cmd
```

The BfThrusters Function. As shown in Figure 11, this function maps commands to the set of thrusters to be fired. More precisely, a pair of sets is produced, the first being the set of mandatory thrusters, the second set contains optional thrusters. This function was surprisingly difficult to develop. A simple precise mapping appeared to be the greatest challenge for ChatGPT. It repeatedly mixed up the thruster names. It even changed already correct ones back to incorrect ones, leading to the prompt:

> **Prompt:** oh gosh, you deleted the correct ones, please go back to the previous version and only correct the ones provided above explicitly as wrong clauses.

In development A we created a table explicitly via a prompt, making up a domain-specific language for this purpose:

Prompt: Here is the functionality for the POS combinations:
Pos, Neg, Neg → F1 and F2,F3
Pos, Neg, Zero → F1,F2 and empty
Pos, Neg, Pos → F2 and F1, F4
...

ChatGPT understands from the context that each line describes a mapping into two sets including our own syntax for empty sets. This is a strength of LLMs. If needed, we can program in a more flexible notation than in classical programming languages. Actually, we can invent syntax as we go as long as it guesses the correct semantics.

8 Module SAFER

This is the top-level module, which models the system's transition that occurs during one cycle (*"once around the main control loop"* [4]) of the controller. The translation of this module did not go smoothly either. ChatGPT seemed confused about what to translate, and we (in both developments) had to upload the technical report again. In development B ChatGPT first went object-oriented, defining the entire module as a class, in contrast to previous modules which were defined in a procedural manner similar to the VDM specification. We will describe the handling of its two functions *ControlCycle* and *ThrusterConsistency*.

The ControlCycle Function. The *ControlCycle* function is shown in Figure 12. It takes three arguments (see Section 5 for the types). First, the switch positions on the hand controller, indicating whether we are in translation or rotation mode, and whether the AAH button is up or down. Second, the physical hand grip moves on the X, Y, and Z axes and twisting of the knob. The third argument represents the movements performed by the automated AAH module itself, assumed measured by sensors. Recall that the AAH module, when active, sends commands to the rotation axes.

The function computes and returns the set of thrusters to be activated. It first computes the *logical* hand grip command (of type *SixDofCommand*) from the *physical* move (*raw_grip*) made by the astronaut, depending on whether we are in translation or rotation mode. Then the thrusters are computed (see Section 7). Finally, the AAH performs its job (see Section 6). The two assertions at the end check that at most four thrusters are now active and that they are consistent, see below.

The translation of the function is shown below. It illustrates how ChatGPT "understands" the meaning of VDM's '**let** *...* **in** *(...)*' construct, simply translating it to assignment statements. It also illustrates how it translates a post-condition to `assert`-statements. Both developments encountered problems with getting names right, for example generating the right types for arguments.

$$
\begin{array}{l}
\mathit{ControlCycle} : \mathit{HCM}`\mathit{SwitchPositions} \times \mathit{HCM}`\mathit{HandGripPosition} \times \mathit{AUX}`\mathit{RotCommand} \xrightarrow{o} \\
\qquad \mathit{TS}`\mathit{ThrusterSet} \\
\mathit{ControlCycle}\ (\mathsf{mk}\text{-}\mathit{HCM}`\mathit{SwitchPositions}\ (\mathit{mode}, \mathit{aah}), \mathit{raw\text{-}grip}, \mathit{aah\text{-}cmd}) \triangleq \\
\quad \mathsf{let}\ \mathit{grip\text{-}cmd} = \mathit{HCM}`\mathit{GripCommand}\ (\mathit{raw\text{-}grip}, \mathit{mode}), \\
\qquad \mathit{thrusters} = \mathit{TS}`\mathit{SelectedThrusters}\ (\mathit{grip\text{-}cmd}, \mathit{aah\text{-}cmd}, \mathit{AAH}`\mathit{ActiveAxes}\ (), \mathit{AAH}`\mathit{IgnoreHcm}\ ())\ \mathsf{in} \\
\quad (\mathit{AAH}`\mathit{Transition}(\mathit{aah}, \mathit{grip\text{-}cmd}, \mathit{clock})\ ; \\
\quad\ \mathit{clock} := \mathit{clock} + 1; \\
\quad\ \mathsf{return}\ \mathit{thrusters}\ ) \\
\mathsf{post\ card}\ \mathit{RESULT} \leq 4 \wedge \\
\qquad \mathit{ThrusterConsistency}\ (\mathit{RESULT})
\end{array}
$$

Fig. 12: The SAFER module, function ControlCycle.

$$
\begin{array}{l}
\mathit{ThrusterConsistency} : \mathit{TS}`\mathit{ThrusterName}\text{-}\mathsf{set} \to \mathbb{B} \\
\mathit{ThrusterConsistency}\ (\mathit{thrusters}) \triangleq \\
\quad \neg\ (\{\mathrm{B1}, \mathrm{F1}\} \subseteq \mathit{thrusters}) \wedge \\
\quad \neg\ (\{\mathrm{B2}, \mathrm{F2}\} \subseteq \mathit{thrusters}) \wedge \\
\quad \neg\ (\{\mathrm{B3}, \mathrm{F3}\} \subseteq \mathit{thrusters}) \wedge \\
\quad \neg\ (\{\mathrm{B4}, \mathrm{F4}\} \subseteq \mathit{thrusters}) \wedge \\
\quad \neg\ (\mathit{thrusters} \cap \{\mathrm{L1R}, \mathrm{L1F}\} \neq \{\} \wedge \mathit{thrusters} \cap \{\mathrm{R2R}, \mathrm{R2F}\} \neq \{\}) \wedge \\
\quad \neg\ (\mathit{thrusters} \cap \{\mathrm{L3R}, \mathrm{L3F}\} \neq \{\} \wedge \mathit{thrusters} \cap \{\mathrm{R4R}, \mathrm{R4F}\} \neq \{\}) \wedge \\
\quad \neg\ (\mathit{thrusters} \cap \{\mathrm{D1R}, \mathrm{D1F}\} \neq \{\} \wedge \mathit{thrusters} \cap \{\mathrm{U3R}, \mathrm{U3F}\} \neq \{\}) \wedge \\
\quad \neg\ (\mathit{thrusters} \cap \{\mathrm{D2R}, \mathrm{D2F}\} \neq \{\} \wedge \mathit{thrusters} \cap \{\mathrm{U4R}, \mathrm{U4F}\} \neq \{\})
\end{array}
$$

Fig. 13: The SAFER module, function ThrusterConsistency.

```
def control_cycle(switch_positions: hcm.SwitchPositions, raw_grip:
     hcm.HandGripPosition, aah_cmd: aux.RotCommand) -> ts.ThrusterSet:
    grip_cmd = hcm.grip_command(raw_grip, switch_positions.mode)
    thrusters = ts.selected_thrusters(grip_cmd, aah_cmd, aah.active_axes(),
                                      aah.ignore_hcm())
    aah.transition(switch_positions.aah, grip_cmd, SAFER.clock)
    SAFER.clock += 1
    assert len(thrusters) <= 4
    assert thruster_consistency(thrusters), thrusters
    return thrusters
```

The ThrusterConsistency Function. The function *ThrusterConsistency* presented in Figure 13 checks thruster consistency: no two selected thrusters should oppose each other, i.e. cancel each other out by thrusting in opposite directions. The translation of the *ThrusterConsistency* function is interesting in the way ChatGPT translated the slightly complicated set operations in the two developments. In development B, ChatGPT generated the following code[5].

```
def thruster_consistency(thrusters: Set[ts.ThrusterName]) -> bool:
    if {ts.ThrusterName.B1, ts.ThrusterName.F1}.issubset(thrusters):
        return False
```

[5] The `&` operator in Python performs set intersection. Furthermore, a set is `True` in a Boolean context iff. it is non-empty.

```
    ...
    if ({ts.ThrusterName.L1R, ts.ThrusterName.L1F} & thrusters) and
       ({ts.ThrusterName.R2R, ts.ThrusterName.R2F} & thrusters):
        return False
    ...
    return True
```

The first four statements (of which the first is shown before the first dots ...) were not translated correctly at first, since instead of the `.subset` method it used `&` (set intersection). This was not detected during interaction with ChatGPT, but was later detected during execution due to a violation of the thruster consistency assertion in the `control_cycle` function. In development A we observed a quite different, and clever, translation as shown below.

```
def ThrusterConsistency(self, thrusters: ThrusterSet) -> bool:
    opposing_pairs = [
        ({"B1"}, {"F1"}),  # Back vs Front
        ...
        ({"L1R", "L1F"}, {"R2R", "R2F"}),  # Left vs Right
        ...
    ]
    for left, right in opposing_pairs:
        if left & thrusters and right & thrusters:
            return False
    return True
```

The Python code identifies first pairs of sets of opposing thrusters and then checks for the intersection of the activated thrusters with these sets. The translation in addition performed a pre-condition check on the number of thrusters being no more than 4, which we have removed, since this check is done elsewhere. However, also for this development there were errors in the logic for checking opposing thrusters, which was caught by an invariant violation at execution time.

9 Module TEST

This module contains functions to test the VDM specification. We translated these functions to Python for testing the Python code. However, we also generated classical unit tests and performed differential testing as described below.

9.1 Unit Testing

In development A, we asked ChatGPT how to test the SAFER code. It recommended two options: (1) unit testing and (2) property-based testing. We first asked for some unit tests. As a result it produced three unit tests, like the following.

```
class TestSAFERSystem(unittest.TestCase):
    def setUp(self):
        self.safer = SAFER()

    def test_control_button_up(self):
        switch_positions = SwitchPositions(
```

```
            mode=ControlModeSwitch.ROT,
            aah=ControlButton.UP
        )
        aah_cmd = {
            RotAxis.ROLL: AxisCommand.NEG,
            RotAxis.PITCH: AxisCommand.POS,
            RotAxis.YAW: AxisCommand.ZERO
        }
        raw_grip = HandGripPosition(AxisCommand.ZERO, AxisCommand.POS,
                                    AxisCommand.NEG, AxisCommand.ZERO)
        thrusters = self.safer.ControlCycle(switch_positions, raw_grip, aah_cmd)
        self.assertTrue(self.safer.ThrusterConsistency(thrusters),
            "Invariant violated for ControlButton.UP")
```

This unit test first sets the inputs of the hand controller, namely the switch positions and the hand grip position. It also defines a command from the AAH. Then, it calls the control cycle function to produce the selected thrusters. These thrusters are then evaluated for consistency in an assertion. Note, that this assertion is actually redundant, since it is already checked in each control cycle.

After a few iterations of producing unit tests with ChatGPT it became obvious that this method does not scale, given the many combinations of input values. Hence, we continued with the testing method described in the VDM specification of this module in [4].

9.2 Testing in VDM Style

The VDM tooling supports its own style of testing. It exploits the ability of the interpreter to process set comprehensions. These set comprehensions can then be used to generate test inputs in a very concise manner. The TS module defines such input sets for SAFER.

In development B, ChatGPT again had "lost" the document, and it was uploaded again. Once it finally found the section in the newly uploaded report, it produced a quite impressive translation of the whole module, which had 129 lines of non-trivial VDM specification (ignoring comments and blank lines), covering 7 definitions of constants (5 of which were set comprehensions), and 10 functions. There were some minor issues and observations that we will go through in the following.

Position Datatypes. Figure 14 shows definitions of the sets *switch-positions*, *all-grip-positions*, and *all-rot-commands*. These are defined as set comprehensions and serve as test inputs. In the translation to Python these were translated to list comprehensions.

```
switch_positions = [hcm.SwitchPositions(mode, aah)
    for mode in [hcm.ControlModeSwitch.TRAN, hcm.ControlModeSwitch.ROT]
    for aah in [hcm.ControlButton.UP, hcm.ControlButton.DOWN]
]
...
all_rot_commands = [{aux.RotAxis.ROLL: a, aux.RotAxis.PITCH: b, aux.RotAxis.YAW: c}
    for a in aux.axis_command_set
    for b in aux.axis_command_set
    for c in aux.axis_command_set
]
```

$$
\begin{aligned}
&\textit{switch-positions} = \{\mathsf{mk}\text{-}\textit{HCM`SwitchPositions}\,(\textit{mode}, \textit{aah}) \mid \\
&\qquad \textit{mode} \in \{\textsc{Tran}, \textsc{Rot}\}, \textit{aah} \in \{\textsc{Up}, \textsc{Down}\}\};\\
&\\
&\textit{zero-grip} = \mathsf{mk}\text{-}\textit{HCM`HandGripPosition}\,(\textsc{Zero}, \textsc{Zero}, \textsc{Zero}, \textsc{Zero});\\
&\\
&\textit{all-grip-positions} = \{\mathsf{mk}\text{-}\textit{HCM`HandGripPosition}\,(\textit{vert}, \textit{horiz}, \textit{trans}, \textit{twist}) \mid \\
&\qquad \textit{vert}, \textit{horiz}, \textit{trans}, \textit{twist} \in \textit{AUX`axis-command-set}\};\\
&\\
&\textit{all-rot-commands} = \{\{\textsc{Roll} \mapsto a, \textsc{Pitch} \mapsto b, \textsc{Yaw} \mapsto c\} \mid \\
&\qquad a, b, c \in \textit{AUX`axis-command-set}\};
\end{aligned}
$$

Fig. 14: The TEST module, Positions.

$$
\begin{aligned}
&69.0 \quad \textit{HugeTest} : () \to \\
&\quad .1 \qquad (\textit{HCM`SwitchPositions} \times \textit{HCM`HandGripPosition} \times \textit{AUX`RotCommand}) \xrightarrow{m} \\
&\textit{TS`ThrusterSet} \\
&\quad .2 \quad \textit{HugeTest}\,() \triangleq \\
&\quad .3 \quad \{\mathsf{mk}\text{-}(\textit{switch}, \textit{grip}, \textit{aah-law}) \mapsto \textit{SAFER`ControlCycle}\,(\textit{switch}, \textit{grip}, \textit{aah-law}) \mid \\
&\quad .4 \qquad \textit{switch} \in \textit{switch-positions}, \textit{grip} \in \textit{all-grip-positions}, \\
&\quad .5 \qquad \textit{aah-law} \in \textit{all-rot-commands}\};
\end{aligned}
$$

Fig. 15: The TEST module, Function HugeTest.

ChatGPT was then asked to turn this into sets, to match the VDM specification, which it did. However, during subsequent execution an exception was thrown since `SwitchPositions`, defined as a dataclass, is an unhashable type, meaning it cannot be used as an element type of a set. In Python, only immutable types (e.g., numbers, strings, tuples) are hashable by default. To fix this, we could make the class hashable by defining a `__hash__` method and, optionally, an equality method `__eq__` to ensure the correct behavior.

```python
@dataclass(frozen=True)
class SwitchPositions:
    mode: ControlModeSwitch
    aah: ControlButton

    def __hash__(self):
        return hash((self.mode, self.aah))
```

The translation of *all-rot-commands* as a set of dictionaries resulted in a similar problem in that dictionaries are mutable, and therefore not hashable, and therefore cannot occur as elements in sets. ChatGPT repeated the correct suggestion to use lists instead of sets, as shown above, to avoid this problem.

The HugeTest Function. The *HugeTest* function shown in Figure 15 is interesting since it shows a "clever" way of specifying property-based testing [10] in VDM. Given a function $f : D_1 \to D_2$ and a property that should hold, e.g.

$consistent(f(x))$, property-based testing consists of testing for some finite subset $F \subseteq D_1$:

$$\forall x \in F \bullet consistent(f(x))$$

This can instead be expressed as a set comprehension:

$$\{x \mapsto f(x) \mid x \in F\}$$

assuming that f will throw exceptions in case the *consistent* property is violated. In VDM this can be achieved via assertions. This is what happens in the function *ControlCycle* in Figure 15. The property to be tested here will be thruster consistency. ChatGPT translated this into the following.

```
def huge_test() -> Dict[
                    Tuple[hcm.SwitchPositions, hcm.HandGripPosition, aux.RotCommand],
                    ts.ThrusterSet]:
    return {
        (switch, grip, aah_law): safer.control_cycle(switch, grip, aah_law)
        for switch in switch_positions
        for grip in all_grip_positions
        for aah_law in all_rot_commands
    }
```

However, again we ran into the problem of non-hashable types, detected by executing the code. The types `hcm.SwitchPositions`, `hcm.HandGripPosition`, and `aux.RotCommand` are not hashable, and hence tuples of such cannot be used as keys in a dictionary. This code was therefore generated using a list of tuples.

```
def huge_test() -> list[Tuple[
                    Tuple[hcm.SwitchPositions,hcm.HandGripPosition,aux.RotCommand],
                    ts.ThrusterSet]]:
    return [
        ((switch, grip, aah_law), safer.control_cycle(switch, grip, aah_law))
        for switch in switch_positions
        for grip in all_grip_positions
        for aah_law in all_rot_commands
    ]
```

$$
\begin{array}{l}
ConvertTIds : TS`ThrusterSet \to \mathsf{char}^{**} \\
ConvertTIds\,(ts) \triangleq \\
\quad \mathsf{if}\ ts = \{\} \\
\quad \mathsf{then}\ [] \\
\quad \mathsf{else\ let}\ t \in ts\ \mathsf{in} \\
\qquad [ConvertTId\,(t)] \frown ConvertTIds\,(ts \setminus \{t\});
\end{array}
$$

Fig. 16: The TEST module, function ConvertTIds.

The ConvertTIds Function. The last function we shall mention is *ConvertTIds*, shown in Figure 16. This function returns a list of *ConvertTId(t)* for all

t in the argument ts, in no specified order. It is defined in a non-deterministic manner, selecting a t from ts and then calls itself recursively. This is translated into the following list comprehension, which iterates `t` through `ts` in order. It requires some "understanding" of this recursive approach in Figure 16 to perform this translation.

```
def convert_thruster_ids(thrusters: ts.ThrusterSet) -> List[str]:
  return [convert_thruster_id(tnm) for tnm in thrusters]
```

The two solutions are not equivalent due to the non-deterministic / underspecified choice in the VDM specification. However, this difference seems not important.

Test Results. With *HugeTest* we discovered a bug in development A. After execution of 4654 tests, the property of thruster consistency was broken. The fault was in the TS module, where wrong thrusters were selected. It turned out that ChatGPT had changed a part of the Python code after it was reviewed. Hence, it had changed already correct code. After fixing the code all 8748 tests passed.

Note, with only a handful of unit tests it would have been very unlikely that we would have discovered this bug. It demonstrates how important automated test case generation is. Nobody wants to write over 8000 unit tests by hand, even if assisted by an LLM. With the same method, we also detected a bug in the thruster consistency definition of development B. We refer to the discussion of the *ThrusterConsistency* function in Section 8 for the details.

9.3 Differential Testing.

Having established thruster consistency, one may still wonder if the system is actually firing the correct thrusters. An obvious method would be to test against the VDM specification. However, we had only the PDF document available. Hence, we decided for differential testing of the two separate Python developments. That is, we defined a property that the two versions of SAFER must produce the same output. Then, we used the same property-based testing method as for thruster consistency. The only problem was that we had to convert the enum return types, because the developments used different encodings for enums. ChatGPT came up with an immediate solution for this conversion.

Test Results. *HugeTest* discovered a difference in the outputs after 4377 tests. The problem was in the AAH module in development B, as already mentioned on page 12: the transition for the `(AAH_ON, UP)` case in the AAH state machine was wrong. After fixing this fault, all 8748 test cases passed. Hence, both separate developments produced the same output. This demonstrates the value of differential testing.

10 Lessons Learned

We can make a number of observations based on this exercise. First of all, we have generated code from a formal abstract mathematical specification with the use of an LLM. In general, one may expect that it is easier to review and understand an abstract specification than it is to understand the generated code, making formal specification a possible approach to advanced prompting.

In this case, however, the generated code is very similar to the formal specification due to the similarity between VDM and Python, as we have shown, which is interesting in itself. Some key differences include the following. VDM allows for infinite sets and universal and existential quantification over infinite sets, which Python does not. Maps in VDM are immutable, but dictionaries in Python are mutable, and not hashable, which means that one cannot create e.g. a set of dictionaries. VDM supports design-by contract in the form of pre- and post-conditions and state invariants, intended to hold between state updates, which Python does not. A minor issue is Python's very verbose way of expressing enumerated types. A not unimportant, although semantically shallow, difference is that VDM appears visually more elegant due to the rendering of its mathematical symbols. Python is code, with the usual rendering of code using colors. The Fortress programming language [13] tried to focus on mathematical rendering as well. Python's very flexible meta-programming features may be used to remove some of these differences.

With respect to the use of LLMs for code generation, we of course observed that the LLM occasionally hallucinated. ChatGPT even changed already correct code parts when other parts were being created or corrected. We observed that error messages can be used directly as prompts for correction. In particular, assertions become very important, and also make it easier to review code. A useful technique turned out to be to formulate a prompt in an informal free, but structured, notation similar to pseudo code or a DSL (Domain-Specific Language).

With respect to verifying generated code, AI-assisted pair programming, with separate development and equivalence runtime verification checks seems interesting. Also property-based testing, where tests are generated, are important. E.g. in some cases bugs were caught after several thousands of tests, which would have been impossible to create manually. Finally, an LLM can be used for code (and specification) explanation. We even used this technique to better understand the SAFER protocol, using the documents [11, 3, 4], including the formal VDM specification in [4], as prompts.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. J. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.
3. S. Agerholm and P. G. Larsen. Modeling and validating SAFER in VDM-SL. In *Lfm'97 - Fourth NASA Langley Formal Methods Workshop*, 1997.

4. S. Agerholm and P. G. Larsen. SAFER specification in VDM-SL. Technical report, IFAD, Forskerparken 10, Odense, Denmark, 1997.
5. B. Aichernig and K. Havelund. AI-assisted programming with test-based refinement. In B. Steffen, editor, *Bridging the Gap Between AI and Reality, track: AI Assisted Programming (AIAP) at AISoLA*, volume 14129 of *LNCS*, pages 385–411. Springer, 2025.
6. A. A. Avizienis. The methodology of N-version programming. *Software Fault Tolerance*, 1995.
7. E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, and G. Reger. An introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–23. Springer, 2018.
8. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
9. ChatGPT. `https://chat.openai.com`.
10. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.
11. J. Crow, B. D. Vito, R. Lutz, L. Roberts, M. Feather, J. Kelly, J. Rushby, and S. Owre. Formal methods specification and analysis guidebook for the verification of software and computer systems, volume II: A practitioner's companion. Technical Report NASA-GB-O01-97, NASA, 1997.
12. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.
13. Fortress. `https://en.wikipedia.org/wiki/Fortress_(programming_language)`.
14. A. Groce, G. Holzmann, R. Joshi, and R.-G. Xu. Putting flight software through the paces with testing, model checking, and constraint solving. In *Proceedings of the 5th International Workshop on Constraints in Formal Verification (CFV)*, 2008.
15. K. Havelund. Closing the gap between specification and programming: VDM^{++} and Scala. In M. Korovina and A. Voronkov, editors, *HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 of *EasyChair Proceedings*, December 2011. Manchester, UK.
16. K. Havelund and G. Reger. *Runtime Verification Logics - A Language Design Perspective*, volume 10460 of *LNCS*, pages 310–338. Springer, 2017.
17. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.
18. P. Larsen, B. Hansen, H. Brunn, N. Plat, H. Toetenel, D. Andrews, J. Dawes, G. Parkin, and et al. Vienna development method specification language part 1: Base language. Information Technology Programming Languages, Their Environments and System Software Interfaces, December 1996.
19. W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.
20. PVS. `http://pvs.csl.sri.com`.