Can Large Language Models Support Modeling Systems with ASMETA? A Case Study with a Planetary Rover

Andrea Bombarda[1][0000−0003−4244−9319], Silvia Bonfanti[1][0000−0001−9679−4551], Angelo Gargantini[1][0000−0002−4035−0131], and Nico Pellegrinelli[1][0009−0000−4944−6845]

University of Bergamo, Bergamo, Italy {andrea.bombarda, silvia.bonfanti, angelo.gargantini, nico.pellegrinelli}@unibg.it

Abstract. Formal modeling languages provide strong support for the specification, analysis, and validation of cyber-physical systems, but their adoption in practice is often hindered by the effort and expertise that are required to produce correct and complete models. In this paper, we investigate whether Large Language Models (LLMs) can support the modeling process by assisting in the generation and refinement of ASMETA specifications from natural language requirements. We propose an iterative and human-in-the-loop workflow in which an LLM is used to derive an initial ASMETA model, progressively refine it, and support scenario-based validation using existing ASMETA tools. The approach explicitly combines automated assistance with human inspection to mitigate modeling errors and potential biases introduced by the LLM. We evaluate the feasibility and effectiveness of this workflow through a case study based on the ABZ 2026 planetary rover problem, using GPT-5.2, accessed via the ChatGPT interface and leveraging the Projects functionality to support persistent and multi-iteration interactions. Our experience suggests that LLMs can significantly lower the entry barrier to formal modeling and support engineers by accelerating the creation of analyzable ASMETA artifacts, but expert oversight is necessary to ensure correctness, completeness, and alignment with stakeholder intent.

Keywords: ASMETA, Abstract State Machines, Large Language Models, Human-in-the-loop modeling

1 Introduction

Formal languages provide a disciplined way to describe software-intensive and cyber-physical systems with precision [2]. When requirements are captured in a mathematically grounded notation, a model becomes more than documentation: it becomes an analyzable artifact that can be used to run automated activities such as simulation and validation, formal verification, and even code generation. In this perspective, modeling is not only a design support, but also a practical enabler for early detection of inconsistencies, missing cases, and unintended behaviors, before implementation decisions make changes costly.

The ASMETA [8] framework supports system modeling through the Abstract State Machines (ASM) paradigm, offering a lightweight yet expressive language and a user-oriented environment. ASMETA specifications can be executed and analyzed with dedicated tools, allowing engineers to explore system scenarios, check properties, and derive implementation-oriented artifacts [1]. Despite these advantages, the adoption of a formal notation still poses a barrier: even when the syntax is relatively simple and the tooling is user-friendly, writing a correct and complete model requires familiarity with formal abstractions, disciplined structuring choices, and attention to details such as state representation, controlled/monitored functions[1], and rule design [11,13]. As a result, practitioners may not fully benefit from ASMETA's capabilities because the initial modeling effort is perceived as difficult or time-consuming.

This paper investigates whether Large Language Models (LLMs) can reduce this entry cost by supporting the automatic generation of ASMETA models from textual requirements. The core idea is to use natural language requirements, which are the most available and accessible form of specification in industrial settings, as input, and produce an initial formal model that can then be assessed and, possibly, refined by the engineer. We believe that this approach can both lower the skills threshold for formal modeling and support experienced engineers by reducing manual effort, shortening the time needed to obtain an analyzable artifact, and making model-based validation and verification more readily applicable in early development phases. Concretely, we envision an iterative modeling process in which an LLM assists engineers across multiple phases of the ASMETA workflow. Starting from natural language requirements, an initial ASM model is generated and progressively refined. The model is then exercised through scenarios using existing ASMETA tools, with support from the LLM. Violations and inconsistencies introduced at any of these stages are identified through human inspection, deliberately performed without LLM assistance in order to reduce potential biases, and the process is repeated. In this way, the LLM is integrated into a closed-loop workflow that combines automated analysis with human oversight, complementing existing formal modeling practices.

To investigate the efficacy of our approach, we conducted an evaluation using the planetary rover description and requirements, as provided by the ABZ 2026 case study [10]. We focus on GPT-5.2 from OpenAI, accessed via the ChatGPT interface [20], in particular leveraging its *Projects* functionality, which enables a structured workflow across multiple iterations, persistent context, and reusable artifacts. Rather than treating the LLM as a one-shot code generator, we explore an interactive process in which requirements are progressively clarified, modeling decisions are made explicit, and the produced ASMETA specification is incrementally improved. This reflects how formal modeling typically occurs in practice: through iterative refinement, feedback, and alignment with user intent.

Our evaluation shows that, even from a minimal prompt, GPT-5.2 was able to quickly produce a complete first draft of the rover model derived from the ABZ

[1] *Controlled* functions are modified by the machine and read by the environment. *Monitored* functions are read by the machine and modified by the environment.

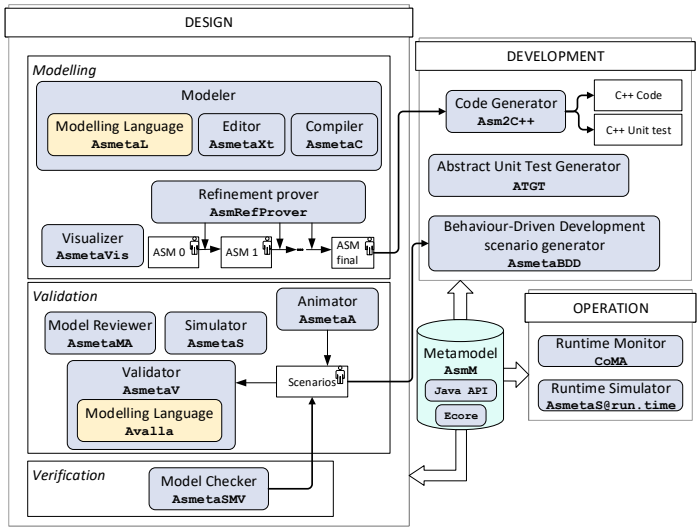

Fig. 1. The ASMETA toolset supporting the ASM development process, including modeling, validation, verification, code generation, and runtime operations.

case study, although the resulting artifacts required further iterations and some manual corrections before they became correct and complete. Scenario-based validation further showed that automatically generated scenarios were generally plausible, but often failed to exercise key behaviors (e.g., rover movement), which motivates using the LLM to craft more targeted scenarios and improve behavioral coverage. As with the ASMETA specifications, however, test scenarios generated by the LLM also required manual fixes to make them executable.

The remainder of this paper is structured as follows. Section 2 presents the ASMETA framework. Section 3 details the planetary rover case study, with focus on the part of the system subject to our analysis. Section 4 shows the envisioned LLM-based modeling and validation methodology, while Section 5 discusses on the challenges we faced and the lessons we learned from its adoption. Finally, Section 6 presents related work and Section 7 concludes the paper.

2 The ASMETA-based development process

This work builds on Abstract State Machines (ASMs), which extend Finite State Machines (FSMs) by replacing unstructured control states with states characterized by arbitrarily complex data. In particular, we rely on the capabilities that are provided by the ASMETA toolset [1], which supports developers throughout the entire system life cycle. This life cycle is typically structured into three main phases: design, development, and operation. Each of these phases is supported by dedicated analysis tools (see Figure 1).

For this case study, we limit ourselves to the *design* phase, in particular on the modeling and validation activities. The development process starts with the *modeling* phase. At its core is `AsmetaL`, the executable modeling language used to define ASMETA models. `AsmetaL` is designed to allow system requirements to be expressed in a form that resembles pseudo-code over abstract data structures, while maintaining a well-defined semantics. Models written in the `AsmetaL` language can be edited using the `AsmetaXt` editor, which provides editing functionalities, and compiled with the `AsmetaC` compiler.

Once the model is available, its correctness with respect to the requirements can be assessed through *validation*. The `AsmetaS` simulator supports both random and interactive execution of ASMETA models, while the `AsmetaA` animator enhances simulation with a graphical interface that offers a structured view of the system state at each step. Scenario-based validation is provided by `AsmetaV`: users specify expected behaviors in the `Avalla` language, and the tool executes them against the model to detect possible deviations from requirements.

Beyond the design phase, ASMETA supports model-driven *development* activities, such as automatic generation of test cases with `ATGT` [5,12], which exploits a model checker or random simulation to derive `Avalla` scenarios.

3 The planetary rover case study

In this section, we report some of the details of our case study, i.e., the planetary rover proposed for the ABZ 2026 Case Study track [10]. It is a semi-autonomous planetary rover undertaking an inspection mission. In particular, in this paper, we focus on modeling and validating the components concerning the rover battery. Thus, we exclude from our analysis the components used for the communication, the validation of a map, and vision. While we believe that some of them could be considered as well, to validate our approach and evaluate its efficacy, we have chosen to limit to more simple behavior and components of the planetary rover. More specifically, we modeled the components shown in Figure 2 and the requirements that we report in Table 1.

More specifically, we consider the following aspects:

- Selection of a charger as a current goal, when the battery is low (the `goal` message exchanged among GRA and `ComputePlan2Charging` components);
- Computation of the plan to reach a charger (the `plan2C` message exchanged among the `Planner` and `HardwareInterface` components);
- Execution of a plan step-by-step, implying the consumption of the battery;
- Handling of the battery charging status (`Battery+Hardware` component).

In the following, we detail the approach that we envisioned to model and validate such a system by the collaboration between human-in-the-loop and LLMs.

4 Modeling with LLMs

In this section, we provide a demonstration of our proposed modeling and validation approach based on the use of LLMs and a human-in-the-loop philosophy. We

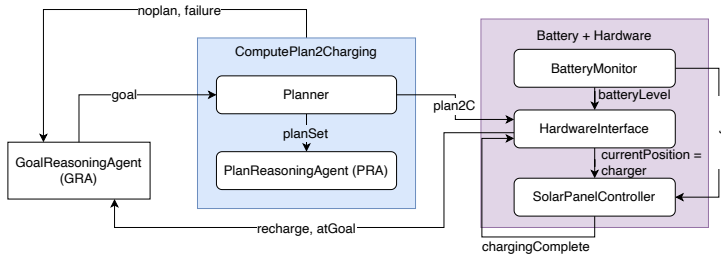

Fig. 2. System architecture.

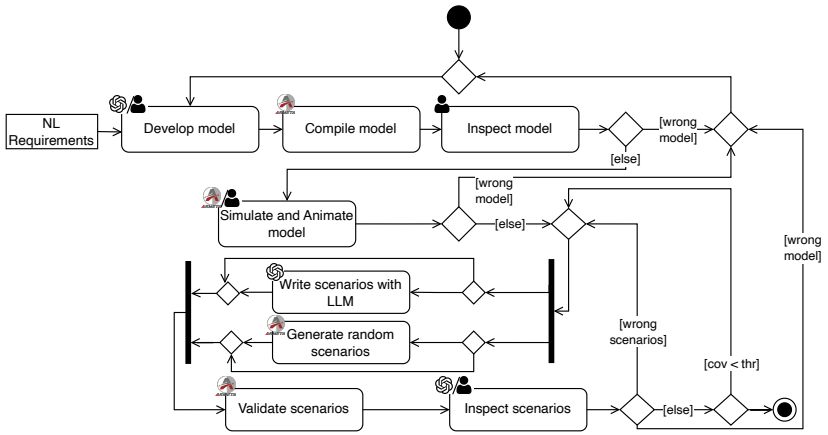

Fig. 3. The proposed modeling approach.

first introduce our methodology (Section 4.1). Then, we delve into the modeling (Section 4.2) and validation (Section 4.3) activities for the planetary rover.

4.1 Methodology

Figure 3 summarizes the workflow that we propose to integrate an LLM (e.g., ChatGPT) into the ASMETA-based design phase as an iterative companion to the modeler. The workflow is explicitly human-in-the-loop and alternates between LLM-assisted artifact generation/refinement and tool-supported assessment with manual inspection. We assume that users interact with an LLM over multiple iterations, progressively improving results by applying one, or a combination of, the two different strategies:

- Enriching the document knowledge base that is available to the model (e.g., adding requirements excerpts, modeling guidelines, standard-library fragments, or previous versions of the model);
- Revising the instructions (e.g., domain/type conventions, preferred idioms, or project-specific modeling constraints).

Table 1. Requirements targeted by our ASMETA implementation.

Req	Description
SL1	The rover shall never run out of battery.
G2	Whenever the `recharge` flag is set to true then the GRA shall set the `goal` as the nearest charging position.
G5	If either planning component ([...]) returns `noplan` the GRA shall [...] await instructions before proceeding.
CPC3	The shortest path to the charger, `plan2C`, shall be selected by the PRA.
CPC4	If there are no viable plans then the `ComputePlan2Charging` component shall return `noplan`, otherwise the `plan2C` shall be sent to `Battery+Hardware`.
CPC5	If a failure in planning occurs (e.g., timeout) the `ComputePlan2Charging` shall notify the GRA.
HI1	The `HardwareInterface` shall set the `recharge` flag to true if the `batteryLevel` is not high enough to reach the `goal` and subsequently reach a charging location.
HI2	Once at a charging position, the rover shall remain there until the battery has been recharged.
HI6	When the `currentPosition` is a `charger` position then the solar panels shall be opened.
HI7	When charging is complete the solar panels shall be closed. [...]
BM1	The `BatteryMonitor` shall monitor the battery level of the rover.
BM2	The `BatteryMonitor` shall return the battery level as 5% less than the measured level. [...]

This enables a closed-loop process where each iteration incorporates feedback from compilation, simulation, and analysis outcomes, and where the LLM can reuse a stable context across steps (e.g., modeling conventions and architectural assumptions). In our approach, we take advantage of the Projects functionality, which enables a structured workflow across multiple iterations, persistent context, and reusable artifacts. Our approach is based on a two-step process:

Step 1 - Cooperative modeling and refinement. Starting from the natural-language requirements, the LLM proposes an ASMETA model. The user then compiles the generated specification using the `AsmetaC` compiler. The compiler feedback helps to identify syntactic and structural issues during the subsequent manual inspection of the model. If errors or inconsistencies are detected, a refinement iteration is triggered. The user may decide to be assisted by the LLM to fix the errors. This may involve adjusting the model, adding missing context to the knowledge base, or tightening instructions or prompts (e.g., concerning types/domains and library usage). Alternatively, during a refinement step (especially in the case of minor issues such as isolated syntax errors), the user may choose to modify the model directly.

Step 2 - Model validation. Once the model compiles and the manual inspection does not reveal issues, the workflow proceeds with validation activities. The ASMETA model is simulated (with `AsmetaS`) and/or animated (with `AsmetaA`) to detect potential behavioral issues or runtime errors. If undesired behaviors or inconsistencies emerge, a new refinement iteration is started. Alter-

natively, scenario-based validation is performed. In particular, the user adopts two different strategies to obtain `Avalla` scenarios from the model: (a) automatically generating scenarios via the random test generator that is available in ASMETA, which exploits the random simulation offered by `AsmetaS`; (b) leveraging the LLM to generate scenarios. Scenarios are then executed with `AsmetaV`. The feedback from `AsmetaV` (PASS/FAIL verdict and coverage information [4]) supports the subsequent inspection activity. Scenarios that are generated with the aid of the LLM should be inspected manually. Conversely, randomly generated scenarios may be inspected with the support of the LLM. Therefore, the LLM plays a twofold role. First, it can support the creation and refinement of executable scenarios in the `Avalla` language. Second, during inspection, it can explain their semantics in natural language and help to assess whether they describe plausible behaviors of the modeled system. This reduces effort in constructing and interpreting scenarios. If the inspection reveals faults in the model, a new model refinement iteration is triggered, accordingly to Step 1. If the scenarios are faulty, they are corrected in a new scenario refinement iteration, either manually or with LLM guidance. When validation and inspection do not reveal errors in either the model or the scenarios, the achieved coverage is compared against a target threshold. If coverage is insufficient, a new refinement iteration is initiated with the goal of extending existing scenarios or generating new ones.

We emphasize that human oversight is an explicit de-biasing mechanism. A key design choice of the process that we envision and report in Figure 3 is that every step involving the LLM is systematically followed by manual inspection, supported by the feedback from ASMETA tools. This ensures that corrections are grounded in human judgment and tool evidence (e.g., compiler errors and simulation traces), mitigating risks such as wrong model implementation, inconsistent type choices, accidental misuse of library functions, or missing model behavior. In practice, the user's primary role is to arbitrate intent and correctness, while the LLM accelerates drafting, restructuring, and repair across iterations.

Setup description. To implement the previously described methodology, we use the GPT 5.2 LLM, accessed via the ChatGPT interface and its *Projects* functionalities. As a document knowledge base, we initially included a tutorial on the ASMETA language [8], the Egon Börger Festschrift paper describing all phases and activities within the ASMETA-based development process [1], the ASMETA StandardLibrary[2] the description of the ABZ 2026 case study [10], and the PDF of Figure 2. The aim is to provide ChatGPT with the relevant context on the `AsmetaL` language, the case study, and the part to be modeled. Additionally, we provide the *project* with the instructions reported in Figure 4.

Replication material. We provide all ASMETA specifications with models, scenarios, and prompts in our replication package at `https://github.com/asmeta/asmeta/tree/master/asmeta_models/ABZ2026_CaseStudy`.

[2] `https://github.com/asmeta/asmeta/blob/master/asmeta_models/STDL/StandardLibrary.asm`

The file TutorialAsmeta.pdf is a tutorial of the ASMETA formal method. It describes how systems should be modeled by using the AsmetaL notation and how validation-based scenario is performed by using the Avalla language. Further information on the AsmetaL language, together with its own metamodel are available at `https://asmeta.github.io/material/AsmetaL_quickguide.html`.

Additional information on the ASMETA framework, and on the purpose of each activity is described in the file FestschriftEgon75.pdf. The StandardLibrary.asm includes standard domains and functions. All domains not defined there should be defined in the specification.

The description of the case study under development is given within the document-v2.pdf, while the simplified architecture of the subportion of the system we consider is given in Architecture.pdf.

You should consider the information coming from these source documents to derive ASMETA models and Avalla scenarios.

Fig. 4. ChatGPT project instructions.

Generate a set of ASMETA modules for the planetary rover case study. Please model the components involved in the reduced architecture, and focus on the requirements SL1, G2, G5, CPC3, CPC4, CPC5, HI1, HI2, HI6, HI7, BM1, and BM2.

Fig. 5. First modeling prompt.

4.2 Modeling

Once we defined the methodology outlined above in Section 4.1, we started modeling the planetary rover, including its requirements listed in Table 1.

At the beginning, we provided to the LLM a very simple prompt, as in Figure 5. While it just specifies the target and the requirements of interest, we found it to be quite effective. Indeed, the LLM generated six ASMETA modules and one ASMETA main specification: 1. `RoverDomains` defines the basic domains of interest and utility static functions; 2. `HardwareInterface` provides the interface to the hardware, such as motors and battery; 3. `SolarPanelController` defines how and when the solar panels are opened or closed; 4. `BatteryMonitor` implements the monitoring activity of the battery (e.g., the BM2 requirement - see Table 1); 5. `ComputePlan2Charging` defines rules and functions used by the rover to define the path to reach a charger; 6. `GoalReasoningAgent` implements the reasoning capabilities of the planetary rover, such as the handling of no plans, or the definition of the next goal location; 7. `RoverReducedMain` imports all other modules, defines the main rule, and initializes all controlled functions.

We compiled and manually inspected the generated ASMETA specifications and we found, in each of them, some issue. Table 2 lists such issues, provides a description for each of them, and reports which ASMETA specifications manifested those issues. Most of these issues (I1, I2, I3, I4, I6, I7, I8, and I9) are syntax or compilation errors, while I5 concerns with an additional requirement that we would like to have in our specifications, since using `Real` instead of `Integer` may prevent the model checker to work with the ASMETA specification.

Considering the issues we identified, we modified the prompt and tried to fix each specification or module by adding additional details regarding the previously mentioned issues. Our second prompt is reported in Figure 6. With this prompt, we were able to fix all previously identified issues for the `RoverDomains`, `SolarPanelController`, and `HardwareInterface` ASMETA modules. However, as shown in Table 3, some issues remained and new ones emerged.

Table 2. Issues after the first ASMETA specification generation.

ID	Description	Specs.
I1	Missing `export` statement in modules	RoverDomains, HardwareInterface, SolarPanelController, BatteryMonitor, ComputePlan2Charging, GoalReasoningAgent
I2	Useless `par` when a single update rule is executed	ComputePlan2Charging
I3	Wrong usage and syntax of the `chooseone` function	ComputePlan2Charging, GoalReasoningAgent
I4	Wrong definition of domains where the product of two domains is used. The LLM uses `D1*D2` instead of `Prod(D1,D2)`	RoverDomains, HardwareInterface, ComputePlan2Charging
I5	Usage of function with `Real` domain or codomain	BatteryMonitor, RoverDomains
I6	Usage of non existing functions (such as `fst` to extract the first element of a sequence, or `snd` to extract the second)	RoverDomains
I7	Missing concrete domain initialization	RoverDomains
I8	Wrong positioning of invariants	RoverReducedMain
I9	Missing controlled functions initialization	RoverReducedMain

Generate a set of ASMETA modules for the planetary rover case study. Please model the components involved in the reduced architecture, and focus on the requirements SL1, G2, G5, CPC3, CPC4, CPC5, HI1, HI2, HI6, HI7, BM1, and BM2.

For the module RoverDomains.asm you previously generated, you must fix all following issues, without changing anything else: - The module must export all (export *) functions and domains - When defining products of two domains, use Prod(D1, D2) instead of D1*D2 - Use only functions that are defined (e.g., use first instead of fst) - Initialize all concrete domains in the definitions section of the module. - Use only Integers instead of Real [...]

Fig. 6. Second modeling prompt.

Given the exploratory nature of this study, and since the issues we identified were minor and further prompt refinements may introduce new inconsistencies, we corrected the models manually and verified that ASMETA could successfully parse and compile the generated specifications. An excerpt of the main ASMETA module for the planetary rover is reported in Listing 1. Note that, for example, the requirement SL1 in Table 1 is captured by an invariant.

4.3 Validation

Following the general approach introduced in Section 4.1, once the ASMETA specifications could be compiled and no issues were revealed by manual inspection, we proceeded with the model validation phase. In this section, we first describe model simulation and then present scenario-based validation activities.

Table 3. Issues after the second ASMETA specification generation.

ID	Description	Specs.
I3	Wrong usage and syntax of the `chooseone` function	ComputePlan2Charging, GoalReasoningAgent
I8	Wrong positioning of invariants	RoverReducedMain
I10	Missing `rtoi` function when real values are used in integer terms	BatteryMonitor

```
asm RoverReducedMain
import ../../STDL/StandardLibrary
import RoverDomains
import BatteryMonitor
import SolarPanelController
import ComputePlan2Charging
import HardwareInterface
import GoalReasoningAgent
signature:
definitions:
    invariant inv_SL1 over batteryLevel: (batteryLevel > 0)
    main rule r_Main = par
        r_BatteryMonitor[]
        r_ComputePlan2Charging[]
        r_GRA[]
        r_HardwareInterface[]
        r_SolarPanelController[]
    endpar
default init s0:
    function recharge = false
    function atGoal = false
    [...]
```

Listing 1. Excerpt of the ASMETA main module for the planetary rover case study.

Model simulation During the model simulation phase, we made extensive use of the `AsmetaS` simulator and the `AsmetaA` animator. Notably, although the ASMETA specifications compiled successfully, we identified several issues that emerged only at runtime. In the following, we describe the major issues that we identified and how we addressed them:

1. **Duplicated functions**: Although `AsmetaC` permits functions with the same name to appear in different modules, `AsmetaS` raises an error when such specifications are simulated. For example, we discovered that the `batteryLevel` function was declared as a *controlled* function in the `BatteryMonitor` module and as a *monitored* function in the `HardwareInterface` module. We inspected the LLM-generated output and found that this issue recurred multiple times, stemming from a misinterpretation of the PDF in Figure 2. Specifically, whenever an arrow connected two components, the LLM assumed that the function labeled on the arrow was *controlled* by the source component and *monitored* by the destination component. To solve this issue, we manually revised the affected modules and added all necessary imports.

2. **Inconsistent updates**: Although the ASMETA code produced by the LLM was syntactically correct, we observed that its coding style more closely resembled that of conventional programming languages, where updates take effect immediately as they are executed, rather than the ASM style, in which updates are collected during a step and applied only at the end of the step. For this reason, we found several inconsistent[3] updates to occur. For example, Listing 2 reports an excerpt of the `HardwareIntefrace` module, where an incon-

[3] We say that an update is *inconsistent* when the same location is updated to multiple different values during the same execution step.

```
rule r_loadPlan =
    if (not(noplan) and recharge) then par
        activePlan := plan2C
        moving := true
    endpar endif
rule r_moveOneStep = if moving then
    if not(isEmptyPlan(activePlan)) then par
        currentPosition := headPos(activePlan)
        activePlan := tailPlan(activePlan)
    endpar else moving := false endif endif
```

Listing 2. ASMETA rules generating inconsistent updates.

```
signature:
    domain Coord subsetof Integer
    // 2D discrete position (simple abstraction)
    domain Position subsetof Prod(Coord, Coord)
    // A plan is a finite sequence of positions
    domain Plan subsetof Seq(Position)
    [...]
definitions:
    // Initialize concrete domains
    domain Coord = {−10:10}
```

Listing 3. Concrete domains within the `RoverDomains` module.

sistent update occurs. It can be noticed that the `moving` function can be updated to `true` and, simultaneously, to `false` when the rover is already moving, needs to recharge, and has finished its movement plan. To address this issue, we manually revised the affected ASMETA modules and strengthened the guards on conditional rules. Although we performed such fixes manually, we may explore in the future a way to extend the LLM context (e.g., by providing additional semantic knowledge and some basic design rules for ASMs) to avoid the generation of ASMETA specifications with inconsistent updates.

3. **Concrete domain initialization**: We discover that the LLM declared some *concrete* domains, as subsets of other previously defined domains. For instance, the `RoverDomains` module in Listing 3 defines the concrete domains `Position` and `Plan` as subsets of structured domains. While this modeling choice is conceptually justified (e.g., only *Position*s within the rover's operating area are admissible), adopting it in ASMETA would require explicitly enumerating all of the values in these domains, resulting in a combinatorial explosion. Additionally, we noticed that the LLM-generated ASMETA modules did not contain such enumeration. To address this issue, we manually revised the affected ASMETA modules by redefining function domains and co-domains without relying on subset domains, for example, replacing `Position` with its supertype `Prod(Coord,Coord)`.

Given the preliminary nature of our investigation, we opted for manual corrections rather than further LLM iterations; this allowed us to obtain a set of ASMETA modules that could be simulated and animated without triggering runtime errors, and to move on to scenario-based validation.

Scenario-based validation To apply scenario-based validation to the ASMETA models developed and exercised through simulation, we adopted a two-pronged strategy that combines an LLM with a human-in-the-loop. First, we generate `Avalla` scenarios using `ATGT` and random test generation, and we ask the LLM to explain these scenarios and assess whether they capture plausible system behavior. Next, we investigate the use of the LLM to produce `Avalla` scenarios directly from textual descriptions. For both activities, because we had manually revised the ASMETA models, we updated GPT-5.2's documental knowledge

Table 4. Excerpt of the evaluation of random scenarios through GPT-5.2.

N.	Description	Plausibility
0	It repeatedly changes `batteryLevel` and planner output; at each step the rover selects a new `plan2C` but never moves: `currentPosition` remains (0,0) and `moving` stays false. When `planningFailure` is true it sets `failure`; when it becomes false, it clears `failure` and set `awaitingInstructions`.	The battery evolution is compliant with BM2, the failure signaling is plausible.
1	It repeatedly changes `planSet`, `batteryLevel`, and `planningFailure`; the rover updates `plan2C` and `batteryLevel` but never moves or recharges. `failure` mirrors `planningFailure`, it may switch to `awaitingInstructions`.	Plausible for the BatteryMonitor mapping and planner failure signaling, but overall not very representative.
3	It repeatedly changes the sensed `battery` and planner outputs; the rover updates `batteryLevel` and `plan2C` each step but never moves or recharges. `failure` follows `planningFailure` (true at steps 2 and 5), while `awaitingInstructions` briefly becomes true when planning succeeds again under low battery.	Plausible for the BatteryMonitor relation (battery value is compliant with the model's offset) and for planner-failure signaling.

The following is an Avalla scenario for the ASMETA specification of the planetary rover. [...]

Provide a concise but clear textual explanation of its behavior (maximum 3 lines) and evaluate whether it represents plausible behavior of the modeled system, i.e., if it is compliant with system requirements.

Fig. 7. Scenario evaluation prompt.

base, by including the corrected specifications, and the instructions accordingly (see the replication package for details on the new instructions).

Random scenario generation. We configured `ATGT` to generate 5 `Avalla` scenarios with 5 steps each and asked GPT-5.2, by using the prompt reported in Figure 7, to explain in natural language the system behavior that is represented by each of the scenarios and their plausibility w.r.t. system requirements, as in [10]. The more significant results of our interaction with our LLM are reported in Table 4. By reading the textual description and the plausibility evaluation, we can notice that although all scenarios are plausible, most of them are only representative for battery management and failures, while none of them address the movement of the rover. Thanks to the manual inspection of the random scenarios we noticed that the evaluation performed by GPT-5.2 is correct: `goal` is never updated in all of the randomly generated scenarios and always remains equal to the initial position of the rover. For this reason, although the coverage achieved by the random scenarios was satisfactory (all macro rules for all modules are covered, and most of behavior, except of the movement of the rover and its arrival to a charging location, is exercised), we proceeded with further guided scenarios.

LLM-based scenario generation. Considering the weaknesses of the scenarios that were randomly generated by `ATGT`, we instructed GPT-5.2 to generate a new

Generate an `Avalla` scenario for the planetary rover ASMETA specification. In this scenario, the rover starts from the position (0,0) and moves to the goal (2,3), which is a charger. While moving, the rover consumes its battery and, when arrived to the goal location, it needs to recharge. Since the goal is a charger location, it opens the solar panel and starts recharging.
While writing the scenario, initialize all monitored functions at each step to reasonable values and perform, at each step, the necessary checks. Remember that in Abstract State Machines the updates are collected and performed at the end of each execution step. Thus, you cannot expect that the updated value is already available in the same step when it is assigned to a function. `Avalla` allows for setting the value of monitored function and checking that of controlled ones. It is not possible to do viceversa.

Fig. 8. `Avalla` scenario generation prompt.

```
scenario rover_to_charger_2_3_recharge
load ./../RoverReducedMain.asm
check currentPosition = (0,0); // State 0
[...]
step
set chargers := {(2,3)}; // State 1
set planSet := {[(1,1),(2,2),(2,3)]};
set measuredBatteryLevel := 7;
[...]
check currentPosition = (2,3);
check atGoal = true;
check isAtCharger = true;
set measuredBatteryLevel := 2; // State 9
step
check panelState = OPEN; // State 10
check moving = false;
```

Listing 4. Excerpt of the `Avalla` scenario generated by GPT-5.2.

`Avalla` scenario including the movement of the planetary rover, its arrival to a charger, and the movement of solar panels. For such a scenario to be generated, we designed the prompt in Figure 8. This scenario is a simple scenario in which, starting from a low-battery condition in the position (0,0), the rover moves trough the environment and reaches a charging location in (2,3) where it stops and opens the solar panels to start the battery charging process. An excerpt of such a scenario is reported in Listing 4.

Initially, the validation of the rover ASMETA specification with the scenario in Listing 4 failed. Thanks to the manual inspection of both the `Avalla` scenario and the ASMETA modules we found two issues:

- The LLM partially ignored our request of not setting the value of *controlled* functions. In particular, in some steps, the function `chargingComplete` was used within a `set` statement.
- Some of the guards that we manually introduced to fix the ASMETA specification to prevent inconsistent updates (see Section 4.3) turned out to be overly restrictive and, in some cases, incorrect, causing the scenario to fail.

After resolving the two issues and manually confirming that the generated scenario matched our intent, we executed it and successfully validated the rover specification. With this scenario, additionally, we covered all missing macro rules.

5 Discussion

In this section, we discuss lessons learned and threats to validity emerging from our study. Our results and considerations should be considered as a proof of concept aimed at validating the feasibility of the proposed approach and further experimentation on additional subjects and settings is needed. However, we con-

sider our experience to be a solid starting point from which to derive lessons and identify possible concerns.

5.1 Lessons learned

Here, we report the lessons that we learned during the use of an LLM to support the modeling and validation process of the planetary rover case study.

L1 - Underrepresented ASMETA constructs and LLM usability. We observed that `AsmetaL` constructs that are rarely used in available examples or are only briefly documented can be challenging for an LLM to apply correctly. When the provided documentation does not include (or does not include sufficient) references, usage patterns, or working examples for these constructs, the model tends to produce ambiguous or incorrect ASMETA specifications, or to fall back on more common constructs that only partially capture the intended semantics. This happened, for example, for the `chooseone` function or for model invariants. This suggests that improving documentation coverage (e.g., with minimal and executable examples for less common features) is essential to make LLM-assisted modeling and validation reliable.

L2 - Clearer error feedback is essential when integrating LLM-generated code. We found it necessary to significantly improve the error messages of `AsmetaS` and `AsmetaV`, since LLM-generated code often contains multiple errors which are revealed only during simulation or validation, and fixing these issues requires first identifying where the fault originated and why it occurred. Without actionable diagnostics (e.g., precise locations or context-specific hints), debugging became slow and error-prone. This highlights that high-quality and developer-oriented feedback is a key enabler for efficiently integrating LLMs in the modeling and validation process with ASMETA.

L3 - Update semantics in ASMETA specifications and LLM-generated code. We learned that LLM-generated ASMETA specifications can be syntactically correct yet still reflect a "sequential programming" mindset, where state updates are assumed to take effect immediately when encountered. This style clashes with ASM semantics, in which updates are collected throughout the execution of a step and applied only at the end of the step. As a consequence, generated models may compile but still exhibit unintended runtime behavior, making simulation-based validation essential to uncover and correct semantic mismatches.

L4 - LLM-generated scenarios help validate manual fixes. After using an LLM to generate the ASMETA specification, we manually fixed several issues to obtain a compilable model. However, an LLM-generated `Avalla` scenario then failed because some of our fixes introduced overly restrictive (and sometimes wrong) guards that blocked the intended behavior. This shows that automatically generating scenarios with an LLM is valuable not only for increasing validation coverage, but also for stress-testing human-made adjustments and revealing unintended side effects. Overall, an iterative workflow that combines human-in-the-loop interventions with LLM-assisted model and scenario generation and explanation provides an effective mechanism to detect these inconsistencies.

5.2 Threats to validity

In this section, we discuss potential threats to the validity [21] of our work. A possible threat to *internal* validity is that our observations may be influenced by the specific interaction protocol used with the LLM (prompt structure, conversation history, and any injected documentation). To mitigate this threat, we provide replication material including all prompts, reference to documentation, instructions, and generated outputs. Additionally, several issues in the generated ASMETA artifacts were corrected manually; while this reflects the intended human-in-the-loop workflow, it introduces the possibility that outcomes (e.g., the time to a working model) partly depend on the modelers' expertise and debugging choices rather than on the LLM alone. *Construct* validity is limited since our study evaluates the proposed methodology through compilation, simulation/animation, and scenario-based validation outcomes. However, these proxies do not fully capture model quality: a specification may compile and simulate without runtime errors while still deviating from intended semantics because it omits relevant and untested behaviors. To mitigate this threat, we manually reviewed all generated artifacts. *External* validity is threatened because we performed our evaluation on a single case study, LLM, and modeling notation. Different domains, modeling styles, tools, and LLMs may lead to different error patterns and different degrees of assistance. For example, we did not consider non-determinism [6] in our experiments. To mitigate this threat, we provide a set of lessons learned, which may generalize across different contexts.

6 Related work

In this paper, we have explored the use of LLMs to support the modeling process of ASMETA specifications, including their validation activities, starting from a textual description of requirements. Our work fits within recent research efforts on supporting user adoption of formal methods through LLMs.

A growing set of approaches uses LLMs to translate natural-language requirements into formal logics or program specifications [3,23,14]. NL2CTL [24] proposes an LLM-based framework to translate natural language into CTL specifications, aiming to reduce the manual burden of formalization. Similarly, SYNTHTL [19] combines LLMs with model checking and human guidance to translate natural-language intent into temporal-logic specifications, emphasizing decomposition into simpler sub-requirements and iterative correction. Such an iterative correction is what we envision in our approach, where LLMs and humans collaborate to obtain system formal specifications. In a different but related direction, SpecGen targets automated generation of formal program specifications using LLMs, leveraging code understanding to infer precise behavioral constraints, which are expressed in JML [17]. Similarly, in [15,18], the authors proposed a way to formally model communication and security protocols using LLMs. Our work differs in the target formalism and artifacts: instead of temporal logic or program contracts, we generate ASMETA models and `Avalla` scenarios, enabling immediate simulation and validation inside the ASMETA toolchain.

Emerging research is also using LLMs with formal notations in the opposite direction, i.e., they translate formal models into understandable textual requirements. For example, in [22], the authors proposed a methodology for translating Event-B specifications into english, while in [7] the authors use an LLM to explain temporal properties.

A substantial body of work shows that LLMs trained on code can synthesize executable programs from natural language prompts and partial specifications. Codex [9] established an early baseline for NL-to-code generation and evaluated the functional correctness of the generated code, highlighting both the promise and brittleness of it when assessed via tests. Accordingly, more recent studies emphasize that small failures are common: generated programs may look plausible yet fail under rigorous behavioral checks, motivating iterative generation–validation–repair workflows [16], as we envision in our proposed approach: these findings motivate our choice to treat ASMETA and `Avalla` generation as an iterative modeling task rather than a one-shot generation.

7 Conclusion

In this paper we investigated whether Large Language Models (LLMs) can reduce the entry cost of formal modeling by assisting engineers in the generation and iterative refinement of ASMETA specifications starting from natural-language requirements, within an explicit human-in-the-loop workflow grounded on tool feedback (compilation, simulation/animation, and scenario-based validation). To evaluate the feasibility of this approach, we focused on a reduced architecture of the ABZ 2026 planetary rover case study and used GPT-5.2 through the ChatGPT interface and Projects functionality to support multi-iteration interactions and persistent context. Overall, our experience suggests that LLMs can effectively accelerate the production of an initial set of analyzable ASMETA artifacts, but expert oversight remains essential and LLMs can not be used as one-shot tools: even when generated specifications compile, important issues may only surface during simulation/animation and manual fixes may be required. This work is a proof of concept on a single case study; additional experiments on different systems, requirement styles, and modeling patterns are needed to assess generalizability and to quantify benefits more systematically. Nevertheless, we have identified a set of lessons learned which may be useful when applying the same approach to other case studies or other formalisms. As future work, we intend to apply the proposed approach to additional case studies and assess whether richer background knowledge (e.g., using RAG) or more specialized LLMs (e.g., a custom model fine-tuned on ASMETA specifications) can improve the results. We also plan to investigate integrating an LLM directly into the ASMETA toolset and extending the workflow with additional activities, such as manually crafting `Avalla` scenarios and using LLMs for their debugging.

Acknowledgments. The work of Andrea Bombarda is supported by the project ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - PNC0000003 – CUP: B53C22006700001.

References

1. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
2. Askarpour, M., Ghezzi, C., Mandrioli, D., Rossi, M., Tsigkanos, C.: Formal Methods in Designing Critical Cyber-Physical Systems, pp. 110–130. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-30985-5_8
3. Beg, A., O'Donoghue, D., Monahan, R.: Leveraging LLMs for Formal Software Requirements – Challenges and Prospects (2025), https://arxiv.org/abs/2507.14330
4. Bombarda, A., Bonfanti, S., Cornejo, C., Gargantini, A., Pellegrinelli, N.: Evaluating coverage and fault detection capability of scenarios for the validation of asmeta specifications. In: Deshmukh, J., Havelund, K., Pinto, A. (eds.) NASA Formal Methods. Springer Nature Switzerland, Cham (2026)
5. Bombarda, A., Bonfanti, S., Gargantini, A.: Automatic test generation with asmeta for the mechanical ventilator milano controller. In: Clark, D., Menendez, H., Cavalli, A.R. (eds.) Testing Software and Systems. pp. 65–72. Springer International Publishing, Cham (2022)
6. Bombarda, A., Bonfanti, S., Gargantini, A., Pellegrinelli, N.: Eliminating flakiness: Deterministic control for validating nondeterministic asmeta specifications. In: Dutle, A., Humphrey, L., Titolo, L. (eds.) NASA Formal Methods. pp. 100–115. Springer Nature Switzerland, Cham (2025)
7. Bombarda, A., Bonfanti, S., Gargantini, A., Pellegrinelli, N.: Formalizing and validating properties in asmeta with large language models (extended abstract) (2026), https://arxiv.org/abs/2603.15375
8. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: ASMETA Tool Set for Rigorous System Design. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods. pp. 492–517. Springer Nature Switzerland, Cham (2025)
9. Chen, M., Tworek, J., Jun, H., Yuan, Q., et al.: Evaluating Large Language Models Trained on Code (2021), https://arxiv.org/abs/2107.03374
10. Farell, M., Kobayashi, T.: ABZ 2026 Case Study: A Planetary Rover, https://github.com/trarse-nii/ABZ2026-case-study/blob/main/doc/document_v2.pdf
11. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: ter Beek, M.H., Ničković, D. (eds.) Formal Methods for Industrial Critical Systems. pp. 3–69. Springer International Publishing, Cham (2020)
12. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using spin to generate tests from asm specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) Abstract State Machines 2003. pp. 263–277. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
13. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. Software and Systems Modeling **22**(6), 1737–1749 (Aug 2023). https://doi.org/10.1007/s10270-023-01124-2
14. Klimek, R.: RE-oriented Model Development with LLM Support and Deduction-based Verification. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering. p. 1297–1304. FSE Companion '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696630.3730562

15. Li, Q., Han, J., Yuan, L., Li, X., Wang, X.: Constructing formal models of cryptographic protocols from alice & bob style specifications via llm. Scientific Reports **15**(1) (Apr 2025). https://doi.org/10.1038/s41598-025-93373-y
16. Liventsev, V., Grishina, A., Härmä, A., Moonen, L.: Fully Autonomous Programming with Large Language Models. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 1146–1155. GECCO '23, ACM (Jul 2023). https://doi.org/10.1145/3583131.3590481
17. Ma, L., Liu, S., Li, Y., Xie, X., Bu, L.: SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 16–28 (2025). https://doi.org/10.1109/ICSE55347.2025.00129
18. Mao, Z., Wang, J., Sun, J., Qin, S., Xiong, J.: LLM-Aided Automatic Modeling for Security Protocol Verification . In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 642–654. IEEE Computer Society, Los Alamitos, CA, USA (May 2025). https://doi.org/10.1109/ICSE55347.2025.00197
19. Mendoza, D., Hahn, C., Trippel, C.: Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers. In: 2024 Formal Methods in Computer-Aided Design (FMCAD). pp. 1–11 (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_17
20. OpenAI: ChatGPT (2026), `https://chatgpt.com/`, accessed: 2026-02-18
21. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Engg. **14**(2), 131–164 (Apr 2009). https://doi.org/10.1007/s10664-008-9102-8
22. Vanhari, F.K.: Translating Formal Specs: Event-B to English. In: 2024 34th International Conference on Collaborative Advances in Software and COmputiNg (CASCON). pp. 1–6 (2024). https://doi.org/10.1109/CASCON62161.2024.10838142
23. Wang, Y., Ge, N., Liu, J., Cao, Z., Chen, Z., Hu, C.: Generating sysml behavior models via large language models: an empirical study. In: Proceedings of the 16th International Conference on Internetware. p. 366–377. Internetware '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3755881.3755926
24. Zhao, M., Tao, R., Huang, Y., Shi, J., Qin, S., Yang, Y.: NL2CTL: Automatic Generation of Formal Requirements Specifications via Large Language Models. In: Ogata, K., Mery, D., Sun, M., Liu, S. (eds.) Formal Methods and Software Engineering. pp. 1–17. Springer Nature Singapore, Singapore (2024). https://doi.org/10.1007/978-981-96-0617-7_1