Deliverable D1.2

Specification: Resource Modelling Platform

Deliverable

Number and title:	D1.2 – Specification: Resource Modelling Platform		
Work package:	WP1 (Energy and Resources)		
Lead authors:	H. Hermanns (USAAR), M. Köhl (USAAR)		
Contributors:	A. Hartmanns (UT), T. Noll (RWTH)		
Reviewers:	P. R. D'Argenio (UNC), M. B. Rodriguez (ASC)		
Due date:	M31 (2024-04-30)	*Dissemination level:*	Public
		Version:	1.0 (final)
Submission date:	2024-05-02	*Pages:*	16

Version history

Version	*Date*	*Notes*
1.0	2024-05-02	First official release

Project

Title:	Models in Space Systems: Integration, Operation, and Networking		
Acronym:	MISSION	*Start date:*	2021-10-01
GA no.:	101008233	*Duration:*	56 months
Call:	H2020-MSCA-RISE-2020	*Website:*	mission-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

1 Introduction

The MISSION project has a number of activities that revolve around resource modelling. Across these activities and in tight synchronisation with the development of the integrated formalism in WP4, the project has developed a resource modelling platform that integrates timed automata modelling for spacecraft and kinetic representations for battery dynamics. This is made possible through the use of a powerful and well-supported modelling backbone, the format JANI [9], together with a dedicated and user-friendly modelling environment called Momba [33].

2 JANI

This section introduces the core ingredients that make up the *JANI* approach, together with a presentation of the details of its use as a modelling backbone for resources in the new space domain.

The origin of JANI lies in the observation that the formal analysis of critical systems is supported by a vast space of modelling formalisms and tools, but that the variety of incompatible formats and tools poses a significant challenge to practical adoption as well as continued research. The JANI format is a metamodel based on networks of communicating automata and has been designed for ease of implementation without sacrificing readability. JANI uses the JSON data format, inheriting its ease of use and inherent extensibility. JANI targets, but is not limited to, quantitative model checking. Several existing tools support the verification of JANI models, and automatic converters from a diverse set of higher-level modelling languages have been implemented. The ultimate purpose of JANI is to simplify tool development, encourage research cooperation, and provide the foundation for the QComp [10, 23] competition in quantitative model checking.

As JANI provides flexible support for modelling quantitative aspects of resources of various kind – especially through the powerful concepts of transient variables and flexible reward structures – it is particularly well suited for the specification of resource parameters prevalent in the space domain, in a spectrum from electric power budgets to transmission bandwidths and data collection capabilities.

In the remainder of this section, we provide an overview of JANI format's most important features, in particularly highlighting the main differences to the well-known user-facing PRISM [34] modelling language. Different from the latter, JANI is not intended to be a user-facing language: it is rather geared for use by tools and for ease of use for tool implementers. We thus keep our overview brief and focus in more depth on our convenient user-facing modelling interface for JANI, Momba, in Section 3.

2.1 Implementer's Guide

The JANI specification defines the JANI model format: a direct JSON representation of networks of stochastic hybrid automata (SHA) [22] with variables, or special cases thereof. SHA combine nondeterministic choices, continuous system dynamics, stochastic decisions and timing, and real-time behaviour including nondeterministic delays. A wide range of well-known and extensively studied formalisms in modelling and verification such as (stochastic) timed automata, discrete and continuous Markov chains and many more can be regarded as special cases of SHA. By providing variables and parallel composition, models with large or infinite state spaces can be represented succinctly in JANI. The format also includes a basic set of variable types and expressions with most common operations, and allows the specification of probabilistic and reward-based properties for verification within a model.

The overriding goal of JANI is simplicity for implementers. The core specification fits on five printed pages. Where expressions over the model's variables are required (such as a guard, the probability of a destination of an edge, or the right-hand side of an assignment), they are represented as expression trees. This is in contrast to other representations of networks of automata, e.g. UPPAAL's [5] XML

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

```
... "features": [ "derived-operators" ],
    "variables": [ { "name": "i", "initial-value": 0,
                     "type": { "kind": "bounded", "base": "int",
                               "lower-bound": 0, "upper-bound": 7 } } ],
    "edges":
    [ { "location": "loc0",
        "guard": { "op": "∧",
                   "left": { "op": "<", "left": 0, "right": "i" },
                   "right": { "op": "<", "left": "i", "right": 7 } },
        "destinations": [
          { "location": "loc0", "probability": 0.8, "assignments": [
            { "ref": "i",
              "value": { "op": "+", "left": "i", "right": 1 } } ] },
          { "location": "fail", "probability": 0.2 } ] } ], ...
```

Listing 1: Excerpt of a JANI MDP model

format, where they are stored as expression strings. Using trees makes it entirely unnecessary to write any kind of expression parsing code to process JANI models.

Listing 1 shows a slightly simplified excerpt of an MDP model with two locations `loc0` and `fail`. It has one edge from `loc0` with guard $0 < i \wedge i < 7$ that loops back to `loc0` with probability 0.8, incrementing i by 1, and goes to `fail` with probability 0.2.

An important aspect of the format is its extensibility, which is based on the mentioned use of JSON in combination with an explicit extension mechanism: a model can list a number of *model features* that it makes use of. They are defined separately from the core JANI specification, and include a `derived-operators` features, which provides for e.g. `max` and `min` operations (which could be represented with comparisons and if-then-else in core JANI), an `arrays` and a `datatypes` feature that specify array types resp. functional-style recursive datatypes (e.g. to define an unbounded linked list type), and a `functions` feature that allows the definition of (mutually) recursive functions for use in expressions. Feature support naturally varies between tools; for example, BDD-based model checkers are typically not able to easily handle unbounded recursive datatypes.

While its syntax is completely different, the semantic concepts of JANI are based on the PRISM language. However, it is more general in some aspects:

Locations. Automata in JANI consist of local *variables* and *locations* connected by edges with action labels, guards, rates, probabilistic branches and assignments over the variables. While being natural for an automaton, having both locations and discrete variables is not strictly necessary as one can be encoded using the other. In fact, PRISM only supports the latter, necessitating the use of "program counter" variables to emulate locations if desired. By supporting both, JANI provides modelling flexibility; if a tool prefers one extreme, an automatic conversion can easily be implemented. Locations provide structural information for e.g. optimisations and static analysis as well as a natural point to store the time progress conditions ("invariants") of TA-based models.

Synchronisation vectors. A JANI model consists of a set of automata that execute in parallel. Edges are either performed independently, or two or more automata synchronise on an action label and perform an edge simultaneously. Inspired by CADP's [17] EXP.OPEN tool, JANI uses *synchronisation vectors* and sets of input-enabled actions as a general specification of synchronisation patterns. As an example, consider three automata. To specify CSP- or PRISM-style multi-way synchronisation on action `a`, we include the one vector $[\mathtt{a}, \mathtt{a}, \mathtt{a}]$. For CCS-style binary synchronisation between `a!` and `a?`, we need the six vectors

$$\{\, [\mathtt{a!}, \mathtt{a?}, -], [\mathtt{a?}, \mathtt{a!}, -], [\mathtt{a!}, -, \mathtt{a?}], [\mathtt{a?}, -, \mathtt{a!}], [-, \mathtt{a!}, \mathtt{a?}], [-, \mathtt{a?}, \mathtt{a!}] \,\}.$$

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

For UPPAAL-style broadcast synchronisation, we make all automata input-enabled on `a?` and use the three vectors $\{ [\texttt{a!}, \texttt{a?}, \texttt{a?}], [\texttt{a?}, \texttt{a!}, \texttt{a?}], [\texttt{a?}, \texttt{a?}, \texttt{a!}] \}$. Synchronisation vectors can express all common process-algebraic operations like renaming or hiding, too—they are a concise yet extremely powerful mechanism.

As a further difference to PRISM, JANI allows assignments to global variables on synchronising edges. Inconsistent concurrent assignments are a modelling error. This small extension removes a major modelling annoyance, but also has important implementation consequences.

Transient variables and assignments. When edges synchronise in a network of automata, the assignments of all participating automata are typically performed all at once, atomically. In JANI, we additionally allow each assignment to be annotated with an *index*. Assignments with the same index are executed atomically, but sets of assignments with different indices are performed sequentially in the indexed order. In combination with transient variables, which are not part of the state vectors and get reset before and after taking an edge so they do not blow up the state space, this allows e.g. efficient value passing: If two automata synchronise and want to pass a value v, the first one can "send" v by making an assignment $t := v$ to a global transient variable t with index i on its synchronising edge while the second one can "receive" v by making an assignment $l := t$ to the local variable l with index $i' > i$ on its own synchronising edge.

Rewards. Finally, reward structures in JANI are simply expressions over global (transient or non-transient) variables. Properties indicate whether they are instantaneous or steady-state rewards, or whether to accumulate when edges are taken (edge/transition rewards) or over time in locations (rate rewards). This is again a very simple but expressive way to specify rewards. As an example,

```
{ "op":"Emax", "exp":"i", "accumulate":["steps"], "step-instant":6 }
```

asks for the maximum expected reward, computed by accumulating the current value of variable `i` whenever a transition is taken, after exactly 6 transitions.

3 Momba

This section presents *Momba* as a tool enhancing resource modelling convenience when working with JANI.

Momba is a Python framework for dealing with quantitative models centered around the JANI-model interchange format. Momba strives to deliver an integrated and intuitive experience to aid the process of model construction, validation, and analysis. It provides convenience functions for the modular construction of models effectively turning Python into a syntax-aware macro language for quantitative models. Momba's built-in exploration engine allows gaining confidence in a model, for instance, by rapidly prototyping a tool for interactive model exploration and visualization, or by connecting it to a testing framework. Finally, thanks to the JANI model interchange format, several state-of-the-art model checkers and other tools are readily available for model analysis.

3.1 User Guide

Dealing with formal models encompasses a variety of tasks which can be challenging from time to time—especially for newcomers. Everything starts with the construction of a model or a family thereof. Often there already exists a textual or other, more formal, description of the scenario to be modeled, such as a rough sketch of the desired behavior or a circuit diagram. Then, after a formal model has been conceived, one has to validate that the model actually adequately models what should be modeled. In this regard models are just like any other human artifact, inadequate initially but over time it gets better. Only after confidence in the model has been established, one is able to harvest the benefits by

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

handing over the model to analysis tools, e.g., a model checker. Momba strives to deliver an integrated and intuitive experience to aid the process of model construction, validation, and analysis.

Example: Racetrack. In what follows, we demonstrate multiple facets of Momba using a variant of Racetrack, a well-known benchmark in autonomous AI decision making [4, 36] which has also found its use in several model checking contexts [19, 21, 2, 20]. Using the example of Racetrack, we go through the entire process from the programmatic construction of a family of models through their validation to their analysis. For each step, we highlight what Momba has to offer in terms of effectively supporting the process.

Originally Racetrack has been a pen-and-paper game [18]. A *track* is a two-dimensional grid comprising *start*, *goal*, *wall*, and *blank* cells (see Figure 1) [4]. A car starts off with some initial velocity from a start cell, with the objective to reach a goal cell as fast as possible without crashing into a wall. The car is controlled by nine possible actions modifying the current velocity vector. Racetrack naturally lends itself as a benchmark for sequential decision making in risky scenarios, in particular, when extended with probabilistic noise. In a variety of such noisy forms, it has been adopted as a benchmark for *Markov Decision Process* (MDP) algorithms in the AI community [4, 6, 35, 37, 36].

For our demonstration, we consider multiple *variants* of Racetrack giving rise to a family of MDPs, which have already been studied [2] from a feature-oriented perspective [12]. For example, there are different tank options and fuel is consumed according to various consumption models. In addition, there are different undergrounds inducing probabilistic noise modeling slippery road conditions. Clearly, this modeling scenario is beyond what is possible with mere model parametrization, especially so because we are interested in the car's performance on different tracks each inducing its own MDP [4].

```
dim: 12 35
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxggg
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
s..................................
s..................................
s..................................
s..................................
xxxx...............................
xxxxxxxx...........................
xxxxxxxxxxxx.......................
```

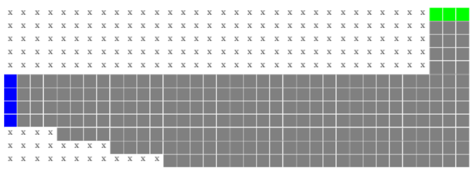


Figure 1: Textual representation (left) and picture of a track (right): start cells in blue (s), goal cells in green (g), and wall cells marked with x.

3.1.1 Model Construction

Typically, formal models are not constructed out of thin air but based on some kind of scenario description existing upfront. Such descriptions usually comprise an operational characterization of the behavior to model together with additional and sometimes more formal information about the specific case. Our Racetrack use case is no exemption, here a textual description of the behavior of the car is provided together with a specific track and a specification of the variant.

Naturally, Python can be used to nicely capture the formal parts of a scenario description in various data structures. Combined with a domain-specific parser for configuration files, scenario descriptions are interchangeable and easy to interface with the code for model construction. In our case, a textual representation of the track (cf. Figure 1) [4] is provided and parsed together with additional parameters, like the size of the tank and the type of the underground, into a data structure tailored to that purpose. Figure 2 shows an excerpt of this data structure for the track. It consists of two classes, `Coordinate` and `Track`, capturing coordinates of cells and entire tracks, respectively. A track has a height and a width, and consists of sets of coordinates for the respective types of cells. For further details, we refer the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

interested reader to the `racetrack` Python package, which is available online[1] and as part of Momba's artifact.

```
@dataclass(frozen=True, order=True)
class Coordinate:
    x: int
    y: int


@dataclass(frozen=True)
class Track:
    width: int
    height: int

    blank_cells: FrozenSet[Coordinate]
    blocked_cells: FrozenSet[Coordinate]
    start_cells: FrozenSet[Coordinate]
    goal_cells: FrozenSet[Coordinate]

    ...
```

Figure 2: Excerpt of the data structure for the track of a Racetrack model.

Now, how does Momba support the construction of models from such data structures? Momba provides an object-oriented modeling API together with convenience functions effectively turning Python into a syntax-aware macro language for the programmatic construction of models in a modular fashion while catching modeling errors early. For instance, based on an instance `track` of the `Track` class, Momba can be used to declare JANI constants for the track's width and height:

```
ctx.global_scope.declare_constant("WIDTH", INT, value=track.width)
ctx.global_scope.declare_constant("HEIGHT", INT, value=track.height)
```

Here, `global_scope` is an object representing the global scope for declaring constants and variables within a JANI model represented by `ctx`. Note that every JANI model has a global scope as well as local scopes for each automaton, respectively. Variables are declared analogously to constants providing an initial value instead of a constant value. Figure 3 shows a variable declaration encoding the cells of a track, again given by a `track` object, as a two-dimensional array in JANI. Each type of cell is represented by an integer between zero and three. By indexing the array with the coordinates of a cell, one obtains the type of the cell. Note that we cannot use a JANI constant here, as constants are not allowed to be arrays in JANI. Furthermore, we make the variable *transient*, which means that it only has a value when a transition is taken, i.e., the variable does not end up in the states of the resulting model, thereby reducing their size. We use the `ArrayValue` class provided by Momba to construct the two-dimensional array based on the `track` object. Clearly, such constructions go beyond what is possible with mere model parametrization.

In the case of Racetrack, we also declare variables `car_x` and `car_y` for the x and y position of the car. To determine whether the car is outside the track, we then use the declared constants and variables. Momba provides a function `expr` for constructing JANI expressions. To construct a JANI expression indicating whether the car is outside of the track, we can use the following code:

```
expr("car_x >= WIDTH or car_x < 0 or car_y >= HEIGHT or car_y < 0")
```

The resulting expression can then be used in a property, e.g., to indicate a crash when the car is outside of the track, or in guards of the edges of the automata to prevent the car from going outside of the track in the first place.

[1] https://pypi.org/project/racetrack/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

```
value = ArrayValue(
    tuple(
        ArrayValue(
            tuple(
                ensure_expr(track.get_cell_type(Coordinate(x, y)).number)
                for x in range(track.width)
            )
        )
        for y in range(track.height)
    )
)
ctx.global_scope.declare_variable(
    "map",
    typ=array_of(array_of(types.INT.bound(0, 3))),
    is_transient=True,
    initial_value=value,
)
```

Figure 3: Variable declaration encoding a track as a two-dimensional array.

Syntax-Aware Macros. A distinguishing feature of Momba is that it allows the interpolation of expressions in a syntax-aware fashion. For our Racetrack use case, we want to be able to use different fuel consumption models. We capture them in terms of *macros* mapping mapping JANI expressions to JANI expressions:

```
linear = lambda dx, dy: expr("abs($dx) + abs($dy)", dx=dx, dy=dy)
quadratic = lambda dx, dy: expr("$linear ** 2", linear=linear(dx, dy))
```

A macro is simply a Python function leveraging Momba's functionality. Upon execution, these macros construct JANI expressions using a straightforward syntax inspired by Python expressions. In this case, both functions take JANI expressions for the current velocity of the car in x and y dimension and return a JANI expression for the resulting fuel consumption, which is either *linear* or *quadratic* in the velocity. In contrast to how macros work in languages like C, syntax-aware macros using Momba's `expr` function prevent surprises from mere text-based expansion. For instance, in case of the quadratic fuel model, naive text-based expansion or interpolation would lead the last summand to being squared instead of the whole sum. Using Momba's `expr` function prevents that as it understands the structure of the individual expressions and combines them on the AST level.

Example: Tank Automaton. Figure 4 shows an example for constructing an entire automaton within a JANI model. Again, `ctx` represents the model. The automaton constructed here models the tank of the car in the Racetrack use case. To this end, the function takes a tick action and a fuel model as input. Here, `tick_action` is an action object which is used elsewhere to synchronize with the tank automaton. Whenever a tick happens, fuel is consumed. The parameter `fuel_model` is one of the earlier defined macros for the different fuel consumption models. Being Python functions, macros can be easily passed around. Based on the provided fuel consumption model, an expression for the fuel consumption is constructed taking the current velocity of the car into account. Based on the consumption, the `fuel` variable must be updated. This is achieved by constructing an assignment. To create this assignment, again Momba's `expr` function is used to construct an expression for the updated fuel level, that is, the current fuel minus the fuel consumption lower bounded to zero and upper bounded to the tank size. Note that the fuel consumption is also used as a guard for the constructed edge. As a result, the tank automaton may block transitions within the model in case the fuel is empty.

Model Construction API. As demonstrated by the examples, Momba provides an object-oriented API for the programmatic construction of models going well beyond what is possible with model parametrization. In particular, it enables the construction of models from preexisting scenario de-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

```python
def construct_tank(ctx, tick_action, fuel_model):
    automaton = ctx.create_automaton(name="tank")
    initial = automaton.create_location(initial=True)

    consumption = fuel_model(expr("car_dx"), expr("car_dy"))

    automaton.create_edge(
        source=initial,
        destinations=[
            create_destination(
                initial,
                assignments={
                    "fuel": expr(
                        "min(TANK_SIZE, max(0, fuel - floor($consumption)))",
                        consumption=consumption,
                    )
                },
            )
        ],
        action_pattern=tick_action,
        guard=expr(
            "fuel >= $consumption",
            consumption=consumption,
        ),
    )

    return automaton
```

Figure 4: Simplified version of the code for constructing the tank automaton.

scriptions. Most of these functions provide all kinds of comforts, for instance, directly checking the types of the involved expressions. For example, adding an edge to an automaton whose guard is not a Boolean expression will result in an immediate error. There is no conversion step necessary to turn the constructed model into JANI besides executing the Python code. Momba's internal model representation is based on the JANI specification and so is the provided API to construct models. Hence, every JANI model can be specified programmatically with Momba which is a great benefit because JANI does not offer any features for building models programmatically while Momba offers the whole range of possibilities of Python.

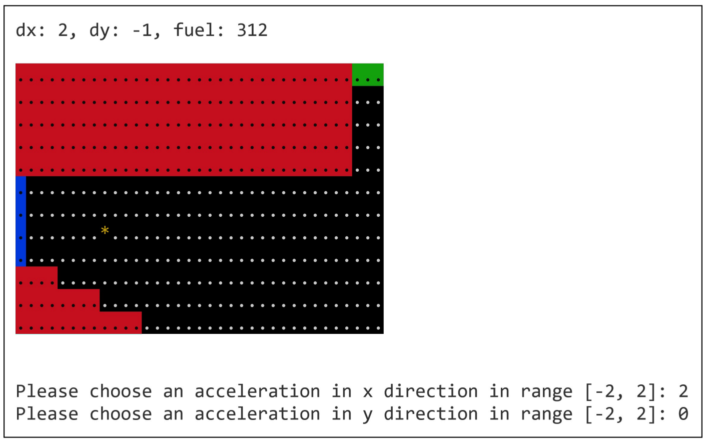
For Racetrack, using syntax-aware macros and Momba's model construction API, we arrive at a Python script for generating a collection of JANI models from scenario descriptions comprising a track and specifying a variant. Again, the script is part of the `racetrack` Python package. Iterating over possible scenario descriptions, hundreds of JANI models can be generated fully automatically and consequently be analyzed.

3.1.2 Model Exploration

Having our models ready, we have to somehow gain confidence that they actually model what we want them to, before handing them over to analysis tools. One way of gaining confidence into a model is by simulating its behavior and manually checking it for consistency with the own understanding of what the model should do. Just like any kind of debugging, this can be a tedious and frustrating process, especially with text-based traces generated by some generic simulator. Momba instead comprises a built-in simulation engine, enabling rapid development of interactive visualizations.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

Interactive Racetrack Game. In the case of Racetrack, we developed an interactive implementation of the game by visualizing the current state of the model and mapping user input to its transitions. This effectively allows a user to steer the car through a track thereby exploring a model's behavior, testing edge cases as in a racing game, and ultimately gaining confidence in the model. Figure 5 shows the interactive visualization. Here, the car is indicated by a yellow asterisk and the user can steer by entering acceleration values. Certainly, there is ample room for beautification of this simulator (see TraceVis [21] for example) but for rapid model development and testing this is not needed. After playing around with the interactive simulation for a while and testing various edge cases, we can be confident that the models we built are indeed adequate. The game is also available as part of the `racetrack` Python package for the reader to play around with.



```
dx: 2, dy: -1, fuel: 312

Please choose an acceleration in x direction in range [-2, 2]: 2
Please choose an acceleration in y direction in range [-2, 2]: 0
```

Figure 5: Interactive visualization using Momba's simulation engine.

State Space Exploration Engine. Momba's built-in state space exploration engine supports a variety of different JANI models, including timed models. It has been written completely from scratch in Rust with easy accessibility from Python in mind. Figure 6 shows an example of using the state space exploration engine for the Racetrack model, which is a discrete time model. After creating an instance of the `Explorer` class, the initial states of the model can be queried. In this case, the model simply has a single initial state. The state object then exposes the global and local environments binding values to variables. In addition, it can be used to query the locations of the individual automata and the outgoing transitions of a state. In Figure 6, a successor state is chosen by selecting a transition uniformly at random and then picking one of its destinations according to the probability distribution of the transition. Recall that in the case of Racetrack the underlying model is an MDP, so we have nondeterminism and in addition each transition comes with a probability distribution over potentially multiple successor states. In principle, nondeterminism can be resolved by uniform random sampling or by querying an external oracle such as, in the case of our interactive visualization, the user, a testing framework, or even a neural network as done for DSMC [19, 20]. For each step, the engine provides all the necessary information about the current state and possible transitions, including actions, probability distributions, and, in the case of timed models, possible time delays. This information can then be extracted and used to display whatever is of interest for understanding and investigating the behavior of the model under scrutiny. We conclude that Momba's state space exploration engine can indeed serve as a platform for model validation by simulation. In particular, we demonstrated how it can be used to prototype a tool for interactive model exploration and visualization.

3.1.3 Model Analysis

Having constructed the models and gained confidence in their adequacy, we are now ready to harvest the benefits of formal modeling and to apply various state-of-the-art analysis tools, exploiting the JANI-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

```
explorer = engine.Explorer.new_discrete_time(network)

(state,) = explorer.initial_states

print("x:", state.global_env["car_x"].as_int)
print("y:", state.global_env["car_y"].as_int)

state = random.choice(state.transitions).destinations.pick().state

print("x:", state.global_env["car_x"].as_int)
print("y:", state.global_env["car_y"].as_int)
```

Figure 6: A simple state space exploration example.

model interchange format. Again, Momba provides the necessary functions to define properties and hand our models, with the respective properties attached to them, over to common analysis tools.

For the Racetrack use case, imagine that we are interested in the maximal probability of reaching a goal cell with a non-empty tank from a given start cell: Using Momba's `prop` function, we can express this property as follows:

```
goal_property = (
    prop("min({Pmax(F(map[car_y][car_x] == 3 and fuel > 0)) | initial})")
)
```

Note that we use the earlier declared variable `map` here to determine the type of cell the car is currently on. The number 3 indicates that the cell is a goal cell. We can now pass this property together with the model to a model checker. For instance, to invoke `mcsta` of Modest Toolset [26], we can use:

```
modest_checker = tools.modest.get_checker(accept_license=True)
results = checker.check(model, properties={"goal": goal_property})
print("Probability:", results["goal"])
```

After generating a model with the car starting from position $(0, 7)$ on the track depicted in Figure 1 and with sand as underground, `mcsta` calculates a probability of $87.5\,\%$ when invoked by Momba with the model.

Note that Momba automatically takes care of downloading the necessary tool or invoking it within Docker. For end users, this translates to a fully integrated experience, where they do not have to worry about downloading and installing the right tools—Momba simply does that for them. Thanks to the JANI-model interchange format, Momba connects well to the established tools of the ecosystem.

4 Resource Analysis Machinery

This section presents the *Modest Toolset* and the *Storm* model checker as resource analysis engines. They both support JANI for defining input models.

4.1 Modest

The *Modest Toolset* [26] is designed for modelling and analysis of hybrid, real-time, distributed, and stochastic systems. Its approach is centred around the (networks of) stochastic hybrid automata (SHA) formalism [22], which combines nondeterministic choices, continuous system dynamics, stochastic decisions and timing, and real-time behaviour including nondeterministic delays. A wide range of well-known and extensively studied formalisms in modelling and verification such as (stochastic) timed automata, discrete and continuous Markov chains and many more can be regarded as special cases of SHA.

In addition to its original input language, which is also entitled Modest [22], the toolset supports the JANI model exchange format. As shown in the previous section, this is the key feature for interacting with Momba. For standalone usage, it is available for download free of charge.

The two main tools with the Modest Toolset supporting the analysis of resource-related aspects are the *mcsta* probabilistic model checker and the *modes* [8] statistical model checker. Both support the computation of reachability and reach-avoid probabilities as well as of expected accumulated and long-run average rewards. Resource usage and constraints can be encoded as rewards and used (i) as hard constraints such as in reward-bounded (i.e. resource-bounded) reachability probability queries, and (ii) as expected values (i.e. as querying for the expected resource usage).

- **mcsta** in particular uses a disk-based approach [25] that allows its explicit-state model-checking engine to handle large unstructured state spaces and thus complex models. It has high-performance algorithms for the analysis of Markov automata models [11]. In general, it specifically focuses on delivering *sound* results [24, 27], i.e. results that are guaranteed to be correct up to a user-specified approximation error.
- **modes** extend the standard statistical model checking (SMC) [1] approach with an automated rare event simulation mechanism [7] and support for lightweight scheduler sampling [15]. The latter in particular allows the analysis of practically unlimited-size resource models with nondeterminism, i.e. where the task is to optimise resource usage itself or other quantities under resource constraints. Specific variants of scheduler sampling are implemented for probabilistic timed automata to handle real-time aspects [16, 28].

4.2 Storm

The *Storm* model checker is another tool supporting the analysis of systems involving random or probabilistic phenomena. Given an input model and a quantitative specification, it can determine whether the model conforms to the specification. Here, models comprise both discrete- and continuous-time as well as deterministic and nondeterministic Markov models, such as discrete-time Markov chains, Markov decision processes, or Markov automata. While originally they had to be defined using the PRISM Language, the JANI modelling language is now supported as well for representing input models. The Momba tool also provides an interface to Storm. Moreover, Storm is publicly available for download and offers a python API named Stormpy to facilitate interfacing with other code bases.

With regard to analysis features, measures such as reachability and reach-avoid probabilities or expected accumulated and long-run average rewards can be computed. Moreover, property specifications using logics such as PCTL, CSL, and LTL are available for model-checking applications.

Noteworthy, Storm supports a number of specific features that are relevant with regard to resource-oriented analyses of space systems:

- Parametric Markov models [32] Many model descriptions contain constants, referred to as parameters, which may be specified only by an interval of possible values. This is the case, e.g., if exact error rates for specific components of a spacecraft are not available. To deal with such systems, Storm offers advanced verification techniques such as automatically proving that for all combinations of parameter values in some region, a property holds.
- Multi-objective model checking [39]: In practice, systems often have to be optimised with respect to different objectives that are mutually dependent and that exhibit trade-offs. A typical example are on-time-within-budget objectives, asking for guarantees to reach a goal state within a specific

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

deadline with a certain probability while keeping the allowed average costs below some threshold. For such scenarios, Storm provides algorithms to analyze several objectives simultaneously and approximate the corresponding to Pareto curves.

- Partially observable Markov models [30]: In many applications involving planning under uncertainty, such as robot navigation or machine maintenance, an agent does not have access to the complete state information of the system but only to specific aspects. Formally, this is represented by so-called Partially Observable Markov Decision Processes in which it is assumed that the system dynamics are determined by a Markov Decision Processes, but the agent cannot directly observe the underlying state. For such systems, Storm supports the (approximative) verification of quantitative reachability properties.

4.3 Case Studies

Both Modest and Storm come with smaller collection of examples to demonstrate the functionalities of the respective toolset. Various more realistic benchmarks can be found at the Quantitative Verification Benchmark Set [29]. They are accessible to Modest and Storm either directly, or via a variety of Jani converters.

With respect to our application domain, the following benchmarks in the collection are particularly interesting:

- Dynamic Power Management [38]
- Energy-Aware Job Scheduling [3]
- Embedded Control System
- Synchronous Leader Election Protocol [31]

Within the MISSION project, the modes tool's lightweight scheduler sampling engine in particular has been used to study routing in satellite networks [13, 14], where the distributed-information aspect posed research challenges as part of WP3. Based on our resource modelling platform, we are currently extending these routing models with resource aspects to be able to compare candidate routes not only in terms of success probability, but also w.r.t. energy usage (which is crucial given the limited batteries of the involved (nano-)satellites) and transmission times.

5 Conclusion

We have identified *JANI* as a highly useful resource modelling platform, with *Momba* as a tool enhancing resource modelling convenience, and the *Modest Toolset* and *Storm* as resource analysis engines. During the *MISSION Gathering* in Rio Cuarto in February 2024 the platform was presented to the MISSION audience, including a dedicated tutorial on *Momba*. The approach was enthusiastically received as a suitable resource modelling platform, that overarches timed automata modelling for spacecrafts as well as kinetic representations for battery dynamics.

References

[1] Gul Agha and Karl Palmskog. "A Survey of Statistical Model Checking". In: *ACM Trans. Model. Comput. Simul.* 28.1 (2018), 6:1–6:39. DOI: 10.1145/3158668.

[2] Christel Baier et al. “Components in Probabilistic Systems: Suitable by Construction”. In: *Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. X by Construction.* 2020.

[3] Christel Baier et al. “Energy-Utility Quantiles”. In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 8430. Lecture Notes in Computer Science. Springer, 2014, pp. 285–299. DOI: 10.1007/978-3-319-06200-6_24.

[4] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. “Learning to act using real-time dynamic programming”. In: *Artificial Intelligence* 72.1 (1995), pp. 81–138. ISSN: 0004-3702. DOI: 10.1016/0004-3702(94)00011-O.

[5] Gerd Behrmann et al. “UPPAAL 4.0”. In: *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. IEEE Computer Society, 2006, pp. 125–126. DOI: 10.1109/QEST.2006.59.

[6] Blai Bonet and Hector Geffner. “Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming”. In: *ICAPS*. 2003, pp. 12–21.

[7] Carlos E. Budde, Pedro R. D’Argenio, and Arnd Hartmanns. “Automated compositional importance splitting”. In: *Sci. Comput. Program.* 174 (2019), pp. 90–108. DOI: 10.1016/J.SCICO.2019.01.006.

[8] Carlos E. Budde et al. “An efficient statistical model checker for nondeterminism and rare events”. In: *Int. J. Softw. Tools Technol. Transf.* 22.6 (2020), pp. 759–780. DOI: 10.1007/S10009-020-00563-2.

[9] Carlos E. Budde et al. “JANI: Quantitative Model and Tool Interaction”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. 2017, pp. 151–168. DOI: 10.1007/978-3-662-54580-5_9.

[10] Carlos E. Budde et al. “On Correctness, Precision, and Performance in Quantitative Verification – QComp 2020 Competition Report”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12479. Lecture Notes in Computer Science. Springer, 2020, pp. 216–241. DOI: 10.1007/978-3-030-83723-5_15.

[11] Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns. “A Modest Approach to Markov Automata”. In: *ACM Trans. Model. Comput. Simul.* 31.3 (2021), 14:1–14:34. DOI: 10.1145/3449355.

[12] Philipp Chrszon et al. “ProFeat: feature-oriented engineering for family-based probabilistic model checking”. In: *Formal Aspects Comput.* 30.1 (2018), pp. 45–75. DOI: 10.1007/s00165-017-0432-4.

[13] Pedro R. D’Argenio, Juan A. Fraire, and Arnd Hartmanns. “Sampling Distributed Schedulers for Resilient Space Communication”. In: *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*. Ed. by Ritchie Lee, Susmit Jha, and Anastasia Mavridou. Vol. 12229. Lecture Notes in Computer Science. Springer, 2020, pp. 291–310. DOI: 10.1007/978-3-030-55754-6_17.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

[14] Pedro R. D'Argenio et al. "Comparing Statistical and Analytical Routing Approaches for Delay-Tolerant Networks". In: *Quantitative Evaluation of Systems - 19th International Conference, QEST 2022, Warsaw, Poland, September 12-16, 2022, Proceedings*. Ed. by Erika Ábrahám and Marco Paolieri. Vol. 13479. Lecture Notes in Computer Science. Springer, 2022, pp. 337–355. DOI: 10.1007/978-3-031-16336-4_17.

[15] Pedro R. D'Argenio et al. "Smart sampling for lightweight verification of Markov decision processes". In: *Int. J. Softw. Tools Technol. Transf.* 17.4 (2015), pp. 469–484. DOI: 10.1007/S10009-015-0383-0.

[16] Pedro R. D'Argenio et al. "Statistical Approximation of Optimal Schedulers for Probabilistic Timed Automata". In: *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by Erika Ábrahám and Marieke Huisman. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 99–114. DOI: 10.1007/978-3-319-33693-0_7.

[17] Hubert Garavel et al. "CADP 2011: a toolbox for the construction and analysis of distributed processes". In: *Int. J. Softw. Tools Technol. Transf.* 15.2 (2013), pp. 89–107. DOI: 10.1007/S10009-012-0244-Z.

[18] Martin Gardner. "Mathematical games". In: *Scientific American* 229 (1973), pp. 118–121.

[19] Timo P. Gros et al. "Deep Statistical Model Checking". In: *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Ed. by Alexey Gotsman and Ana Sokolova. Vol. 12136. Lecture Notes in Computer Science. Springer, 2020, pp. 96–114. DOI: 10.1007/978-3-030-50086-3_6.

[20] Timo P. Gros et al. "MoGym: Using Formal Models for Training and Verifying Decision-making Agents". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 430–443. DOI: 10.1007/978-3-031-13188-2_21.

[21] Timo P. Gros et al. "TraceVis: Towards Visualization for Deep Statistical Model Checking". In: *Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. From Verification to Explanation.* 2020.

[22] Ernst Moritz Hahn et al. "A compositional modelling and analysis framework for stochastic hybrid systems". In: *Formal Methods Syst. Des.* 43.2 (2013), pp. 191–232. DOI: 10.1007/S10703-012-0167-Z.

[23] Ernst Moritz Hahn et al. "The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models – QComp 2019 Competition Report". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer et al. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 69–92. DOI: 10.1007/978-3-030-17502-3_5.

[24] Arnd Hartmanns. "Correct Probabilistic Model Checking with Floating-Point Arithmetic". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 41–59. DOI: 10.1007/978-3-030-99527-0_3.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

[25] Arnd Hartmanns and Holger Hermanns. “Explicit Model Checking of Very Large MDP Using Partitioning and Secondary Storage”. In: *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*. Ed. by Bernd Finkbeiner, Geguang Pu, and Lijun Zhang. Vol. 9364. Lecture Notes in Computer Science. Springer, 2015, pp. 131–147. DOI: 10.1007/978-3-319-24953-7_10.

[26] Arnd Hartmanns and Holger Hermanns. “The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 2014, pp. 593–598. DOI: 10.1007/978-3-642-54862-8_51.

[27] Arnd Hartmanns and Benjamin Lucien Kaminski. “Optimistic Value Iteration”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 488–511. DOI: 10.1007/978-3-030-53291-8_26.

[28] Arnd Hartmanns, Sean Sedwards, and Pedro R. D’Argenio. “Efficient simulation-based verification of probabilistic timed automata”. In: *2017 Winter Simulation Conference, WSC 2017, Las Vegas, NV, USA, December 3-6, 2017*. IEEE, 2017, pp. 1419–1430. DOI: 10.1109/WSC.2017.8247885.

[29] Arnd Hartmanns et al. “The Quantitative Verification Benchmark Set”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 344–350. DOI: 10.1007/978-3-030-17462-0_20.

[30] Linus Heck et al. “Gradient-Descent for Randomized Controllers Under Partial Observability”. In: *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*. Ed. by Bernd Finkbeiner and Thomas Wies. Vol. 13182. Lecture Notes in Computer Science. Springer, 2022, pp. 127–150. DOI: 10.1007/978-3-030-94583-1_7.

[31] Alon Itai and Michael Rodeh. “Symmetry breaking in distributed networks”. In: *Inf. Comput.* 88.1 (1990), pp. 60–87. DOI: 10.1016/0890-5401(90)90004-2.

[32] Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. “Parameter Synthesis in Markov Models: A Gentle Survey”. In: *Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Ed. by Jean-François Raskin et al. Vol. 13660. Lecture Notes in Computer Science. Springer, 2022, pp. 407–437. DOI: 10.1007/978-3-031-22337-2_20.

[33] Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. “Momba: JANI Meets Python”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 389–398. DOI: 10.1007/978-3-030-72013-1_23.

[34] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[35] H. Brendan McMahan and Geoffrey J. Gordon. “Fast Exact Planning in Markov Decision Processes”. In: *ICAPS*. 2005, pp. 151–160.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

[36] Luis Enrique Pineda and Shlomo Zilberstein. “Planning Under Uncertainty Using Reduced Models: Revisiting Determinization”. In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. 2014.

[37] Luis Enrique Pineda et al. “Fault-Tolerant Planning under Uncertainty”. In: *IJCAI*. 2013, pp. 2350–2356.

[38] Qinru Qiu, Qing Qu, and Massoud Pedram. “Stochastic modeling of a power-managed system — Construction and optimization”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 20.10 (2001), pp. 1200–1217. DOI: 10.1109/43.952737.

[39] Tim Quatmann, Sebastian Junges, and Joost-Pieter Katoen. “Markov automata with multiple objectives”. In: *Formal Methods Syst. Des.* 60.1 (2022), pp. 33–86. DOI: 10.1007/S10703-021-00364-6.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.