A SysML v2 based Modeling Language and Tool for Task Planning and Runtime Verification with Digital Twins

Luca Cristoforetti[1][a], Alessandro Flori[1], Tommaso Fonda[1], Kostantinos Kapellos[2], Andrea Micheli[1][b], Stefano Tonetta[1][c], Alessandro Valentini[1]

[1]*Fondazione Bruno Kessler (FBK), Trento, Italy*

[2]*Trasys International, Belgium*

tonettas@fbk.eu, cristofo@fbk.eu, aflori@fbk.eu, tfonda@fbk.eu, konstantinos.kapellos@trasysinternational.com, amicheli@fbk.eu, alvalentini@fbk.eu

Keywords: Formal Methods, Digital Twins, MBSE, Planning, Monitoring

Abstract: Digital Twins (DTs) are key enablers for autonomous and adaptive systems, providing a virtual counterpart that mirrors, predicts, and verifies the behavior of a physical asset. In space applications, where communication latency and uncertainty are significant, DTs enable support for automated task planning and runtime verification with formal methods. This paper presents the model-based design approach developed within the ExploDTwin project, introducing the Digital Twin Formal Modeling Language (DTFML), a SysML v2-based modeling language for specifying and analyzing planning and monitoring problems on top of a DT. DTFML extends SysML v2 with constructs for plannable durative activities, observable variables that can be read from telemetries, external functions used to simulate the resource consumption of activities. In this way, the DTFML enables rigorous definition of planning, plan monitoring, and runtime verification problems. The associated SAWS[2] (Safety Analysis, Validation and Verification for SysML v2) tool enables automated validation and model checking through integration with formal verification engines. The approach is demonstrated on a space-exploration rover case study, showing its applicability to industrial applications.

1 Introduction

The operation of space missions involves the supervision and control of complex assets, such as rovers and spacecrafts, which execute sequences of activities in remote and uncertain environments. Mission operators rely on ground-based systems to plan tasks, monitor their execution through telemetry, and verify the correct behavior of the space application. In this context, *Digital Twins* (DTs) are increasingly adopted to support mission operations by maintaining a virtual counterpart of the physical asset. The DT continuously aligns its internal model with the telemetry received from the spacecraft and provides services such as simulation, diagnosis and prognosis.

While DTs have proven valuable for visualization and simulation, achieving *automation* in planning and monitoring the activities of the physical asset requires the DTs to be equipped with formal models to enable reasoning. In space applications, a main challenge is therefore the development of methods and tools that can exploit the DT to automatically plan activities, monitor their execution, and maintain alignment between the physical and digital systems. This calls for a model-based approach that enables a unified representation of the system and its environment, suitable for analysis by different reasoning engines.

The *ExploDTwin* project (Flori et al., 2025; Bonizzi et al., 2025), funded by the European Space Agency (ESA), addresses these challenges by developing a model-based design framework for DTs in space applications. A central component of this framework is the *Digital Twin Formal Modeling Language* (DTFML), a SysML v2-based modeling language designed to specify the structure and behavior of space assets and their environment. DTFML extends SysML v2 with constructs for activities, observables, external functions, and *durative transitions*, which represent time-extended actions relevant for task planning and monitoring. The resulting model acts as a common formal specification from which inputs for model checking, planning, and runtime veri-

[a] https://orcid.org/0000-0002-8519-6342
[b] https://orcid.org/0000-0002-6370-1061
[c] https://orcid.org/0000-0001-9091-7899

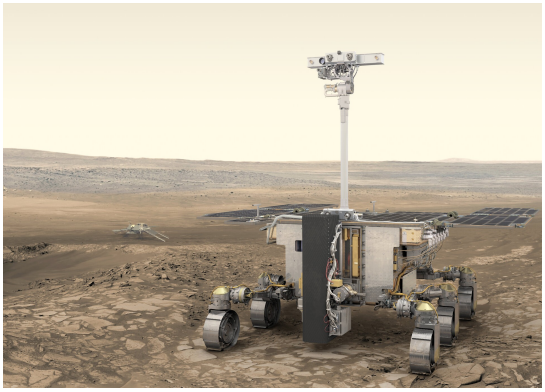

Figure 1: ExoMars rover.

fication can be automatically generated.

The DTFML is supported by *SAWS*^{2} (https://es-static.fbk.eu/tools/saws2/), an Eclipse-based environment that offers a textual SysML v2 editor, validation and navigation features, and automated translation to formal languages for model checking through integration with verification tools such as nuXmv (Cimatti et al., 2019a) and OCRA (Cimatti et al., 2013). Beyond model checking, the DTFML model serves as a common representation that can also be processed by dedicated components of the ExploDTwin framework to derive inputs for planning and runtime verification. In this way, the approach enables the consistent use of a single SysML v2 model across design-time and runtime reasoning tasks, promoting automation and traceability within the DT lifecycle.

The main contributions of this paper are:

1. We present an MBSE approach that uses a single SysML v2 model as the source for model checking, planning, runtime verification, and DTs configuration.
2. We introduce the DTFML language, extending SysML v2 with constructs for activities, observables, external functions, and durative transitions.
3. We describe SAWS^2, which provides model editing, validation, and automated translation for formal analysis using model checking tools.
4. We demonstrate the approach through a space-rover case study.

2 Background and Related Work

MBSE for DTs MBSE has been proposed in various works to support DTs engineering, providing a systematic and model-centric foundation for design, implementation, and lifecycle management (cfr., e.g., (Bordeleau et al., 2020)). A recent survey of the state of the art confirms that MBSE is increasingly applied to DTs, across a variety of domains, showing promising results in automation, transformation, and model reuse (Lehner et al., 2025). The book (Fitzgerald et al., 2024) offers a comprehensive view on how system models, metadata, simulation assets, and co-modelling/co-simulation techniques can be combined to support the engineering and evolution of DTs for cyber-physical systems. More advanced works have demonstrated the automated generation of DT instances from MBSE models (Heiler et al., 2024; Barbie et al., 2025). In other works, MBSE is used to provide DTs as services with also the possibility to combine them (Talasila et al., 2025). While these approaches focus on creating, generating or deploying the DT from MBSE models (structure, simulation, etc.), they often do not address the use of the MBSE model as a formal basis for automated reasoning. In contrast, the approach presented in this paper goes beyond generation: we use a single SysML v2–based model to support automatic reasoning — transforming the model into inputs for model checking (to validate architectural and behavioral consistency), task planning, and runtime verification.

SysML v2 and Formal Methods SysML v2 (https://www.omg.org/spec/SysML) is the next-generation modeling language designed by the OMG, with textual syntax and formal semantics of its kernel, KerML, making it significantly more suitable for formal analysis than SysML v1. Recent work has explored formal verification directly from SysML v2 models: (Molnár et al., 2024) demonstrates various approaches that connect SysML v2 with model checking, theorem proving, and contract-based reasoning through tools such as nuXmv, Imandra, and Gamma; other research has investigated translating SysML v2 behavioral models into formal frameworks such as BIP, enabling the verification of component interactions and system-level temporal properties (Khelifati et al., 2025). These efforts confirm the growing interest in exploiting SysML v2 not only for design documentation but also as an analyzable artifact supporting formal methods, an objective that aligns with the MBSE approach of ExploDTwin and with the design of DTFML.

Formal Methods for DTs Formal methods have increasingly been applied to DTs to ensure correctness, safety, and robustness in both design-time and runtime settings (see, e.g., (Wright et al., 2022; Kristensen et al., 2024; Johnsen, 2022; Naeem and Seceleanu, 2025)). Differently from these works, DTFML is used to describe the tasks that can be planned, ex-

ecuted and monitored on the physical twin, while the DT is used to simulate the resource consumption of the activities in the plan.

3 The Digital Twin Formal Modeling Language (DTFML)

3.1 SysML v2 Subset and Library

The Digital Twin Formal Modeling Language (DTFML) extends SysML v2 with a dedicated library, the **ExploDTwin library**. This library provides a set of predefined definitions and a strict structural hierarchy that the formal model designer must instantiate to specify the DT. The language design leverages the standard semantics of SysML v2 while introducing domain-specific constructs to support planning and formal verification.

3.1.1 Standard SysML v2 Modeling Constructs

The formal model is structured around the interaction between the *Real Asset*, which represents the physical system, and its *Environment*.

- **Subsystems:** The Real Asset is composed of *Subsystems*, each being a distinct module containing its own variables and behavior. Subsystems distinguish between *State Variables* (dynamic, potentially observable via telemetry) and *Calibration Parameters* (static configuration values).
- **Environment:** This component models the external world and its phenomena using state machines and variables, providing the context in which the Real Asset operates.

The framework relies on standard SysML v2 mechanisms for structural and behavioral modeling: the main elements (System, Real Asset, Environment) are defined using standard `part def` constructs; state machines are used to describe the behavior of the subsystems and the environment; connections between components are established using standard `connect` (for different port directions) and `bind` (for same port directions) statements to link ports and variables across the hierarchy.

3.1.2 Library-Specific Specifications

The library extends the standard language with specific types:

- **Predefined Structure:** The library enforces a specific hierarchy where the user must define a `SystemComponent` containing a `RealAsset` (the physical twin model) and an `Environment`.
- **Extended Data Types:** The library introduces types not found in the standard SysML v2 library, such as `BoundedScalar`, `BoundedInt`, and `RealRange`, which allow the definition of `minValue` and `maxValue` constraints essential for formal verification.
- **Metadata Annotations:** The library utilizes custom metadata tags (e.g., `@IsObservable`) to classify variables for external reasoning tools, distinguishing for example the variable that are observable from the telemetry.
- **Variable setters:** To support modular reasoning, the library provides a way to define *setter* interfaces for local variables. This ensures that a component's internal state can be written without violating encapsulation principles.

Activities and Durative Transitions A key contribution of the library is the modeling of *Activities* and *Durative Transitions* to support task planning, which differs from standard state machine semantics.

- **Activity Definition:** An `Activity` is defined as an abstract part containing specific attributes for duration (`lbDuration`, `ubDuration`), constraints (`precondition`, `postcondition`, `invariant`), and effects. It serves as a complex operation that the operator can command.
- **Durative Transitions:** Standard state machine transitions are instantaneous. To model time-extended actions, the library introduces `DurativeTransition`, a subtype of the standard `StateTransitionAction`. The Activity state machine has an *idle* and a *busy* state, with transitions from idle to busy and back, which have as guard and effect respectively the pre/post condition and pre/post effect of the activity. Moreover, a clock constrain the duration of the activity between `lbDuration` and `ubDuration`.
- **Mission State Machines:** This describes the behavior of a subsystem with high-level states and durative transitions to move from one state to another, triggered by activities. Semantically, the behavior of the subsystem is given by the composition of the mission state machine with the local state machines that describe the durative transitions.

External Computation To handle complex logic that exceeds the capabilities of static analytical modeling (e.g., machine learning predictions for power

consumption), the library specifies **Uninterpreted Functions**.

- These are defined using `calc` or `calc def` and tagged with `@UninterpretedFunction`.
- While the function signature (inputs and outputs) is defined within SysML v2, the actual computation is offloaded to external tooling during analysis or runtime. These functions can be bound to model attributes to dynamically update system states based on complex expressions.

3.2 SAWS^2 tool

To facilitate the specification of models in DTFML, we developed SAWS^2, a dedicated editing environment tailored to the subset of SysML v2 and the ExploDTwin library described above. SAWS^2 is a set of plugins to extend the Eclipse framework with the purpose of enabling validation, formal verification, and safety analysis of formal models specified in a fragment of SysML v2. The tool exploits the Eclipse Xtext technology (https://eclipse.dev/Xtext/) to provide editing support: language-aware syntax highlighting, autocompletion, content assist, and live validation, see Fig 2. SAWS^2 features an automatic Model-to-Model transformation, eased by the use of open-source FBK libraries and facilitates the use of formal methods tools.

4 Mapping to Analysis Tasks

4.1 Mapping DTFML to Formal Verification Models

To enable automated model checking, the SAWS^2 tool performs a translation of the DTFML model into TimedSMV, the input language of nuXmv (in the timed configuration (Cimatti et al., 2019a)). This translation follows the two-step approach introduced in the tool of CHESS for SysML v1 (Debiasi et al., 2021), separating the structural decomposition from the behavioral implementation: the architecture is first mapped into OCRA, while the state machine behavior into timed SMV, and then composed through the OCRA tool (Cimatti et al., 2013).

Architecture Translation (DTFML to OCRA) The first step translates the static architecture of the System, Real Asset, and Environment into the **OCRA** language. OCRA allows for the specification of component hierarchies and interfaces.

- **Component Decomposition:** Each DTFML component (e.g., `SystemComponent`, `RealAsset`, `Subsystem`) is mapped to an OCRA `COMPONENT`. The hierarchical structure defined in the SysML v2 `part def` tree is preserved, with sub-components declared recursively.
- **Interface Mapping:**
  - **Ports and Flow:** SysML v2 attribute and port usages and `connect`/`bind` statements are translated into OCRA ports and connections.

Behavioral Translation (DTFML to SMV) The second step translates the behavioral logic defined in DTFML (State Machines and Activities) into the **SMV** language, which serves as the input for the NuSMV/nuXmv model checkers. This implementation is then linked to the corresponding OCRA component leaf.

Each state machine is translated into SMV in a standard way:

- **States:** SysML v2 states are mapped to an enumerated variable `State` in the SMV module.
- **Transitions:** Transitions are translated into `next(State)` assignments within the SMV `TRANS` or `ASSIGN` blocks. Guards (boolean conditions) and triggers are mapped to conditional logic governing these state updates.
- **Clocks** are mapped into corresponding clocks in (timed) SMV.

Each state machine (either for the Mission State Machine or for the Durative Transition) is mapped into a separate module of SMV. The SMV of the subsystem results in the composition of the such module instances.

Asynchrony and Concurrency DTFML allows subsystems to operate asynchronously. In the SMV mapping, this is managed by extending the state machines with self-loops in which a component *stutters* while the other can take some actions.

External Functions `@UninterpretedFunction` definitions are mapped to uninterpreted functions in SMV. This allows the verifier to check safety properties against *any* valid return value of the external calculation.

4.2 Task Planning Problem

The formal DT model provides a direct mapping to a temporal planning problem expressed in terms of

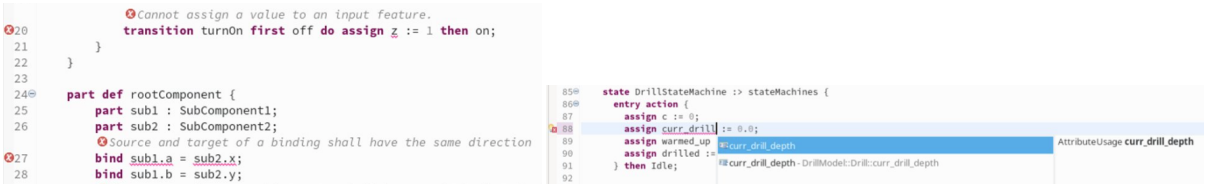

Figure 2: SAWS[2] validates errors and suggests elements during model editing

durative actions (like in PDDL 2.1 (Fox and Long, 2003)) using the Unified Planning library (Micheli et al., 2025). Each activity in the model corresponds to a durative action that the planner can select when constructing a plan. The model specifies the lower and upper bounds on the duration of the activity, conditions and effects at the start, conditions and effects at the end, and invariants that must hold throughout execution. Such constraints are defined over a set of variables in the planning model, called "fluents" in the planning jargon. This structure aligns naturally with the semantics of temporal planning, enabling formal reasoning about concurrent actions and resource usage over time.

State variables defined in the model become the fluents (i.e. the varibales) of the planning problem, including both discrete states and numeric quantities. Conditions and effects may involve numeric expressions, for example to ensure that sufficient energy is available before starting an activity or that thermal limits are respected during execution. These numeric conditions are critical for capturing resource interactions and operational constraints.

When certain parameters cannot be determined analytically, the planner relies on external prediction functions declared in the model. These functions provide estimates for quantities such as activity duration or energy consumption under specific environmental and configuration conditions (Flori et al., 2025). For instance, in the drilling scenario, power demand may depend on soil properties and depth, and the planner queries the corresponding prediction function to obtain an estimate during plan generation.

4.3 Runtime Verification Problems

The monitoring part of the ExploDTwin framework is powered by the NuRV tool, and more specifically by the Abstraction Based Runtime Verification algorithm (Cimatti et al., 2019b). We leverage the provided SysML v2 model and an activity plan to synthesize a monitor of the plan. The real asset performs the same activity plan and produces telemetry. Telemetry is received by the mission operators, and is delivered in an online or offline manner to the monitoring component. Telemetry must coincide with the state variables that are flagged as observable in the SysML v2 model. The component detects either successful plan execution, or faults, by evaluating the telemetry's consistency with the SysML v2 model and the activity plan. In particular, from the SysML v2 model we extract activities' constraints and effects, the state variables. From the activity plan we extract activities parameters' and the execution order. We combine this information to generate an SMV model that contains assumptions on the behavior of the real asset and the property to monitor - the plan ends successfully. We then receive the telemetry, and the NuRV tool verifies incrementally the SMV model against the telemetry, producing verdicts.

5 Case Study: Exploration Rover

5.1 Overview of the application

The case study demonstrates how the ExploDTwin framework can be applied to a simplified exploration scenario inspired by the ExoMars mission. A rover is tasked with collecting three distinct objects (a circle, a triangle, and a square) distributed across symbolic locations on a map. To achieve this, the rover must navigate between locations and perform collection activities while operating under a limited energy budget.

The scenario is modeled using the DT formalism, which captures the system state, activities, and resource interactions in a unified representation. The state includes attributes that track mission progress, such as which objects have been collected, the robot's current location, remaining battery charge, and payload weight. Two main activities are defined: Move, which transfers the robot from one location to another and consumes energy, and Collect, which retrieves the object at the current location and updates the state accordingly. The energy cost of movement is not fixed but estimated by an external prediction function implemented as a neural network.

With this model we can demonstrate the full range of ExploDTwin capabilities: from plan generation to runtime monitoring, and the integration of exter-

nal prediction functions such as ML-based estimators within a formal modeling framework.

5.2 Representation in DTFML

In the DTFML, component decomposition is expressed by nesting standard SysML v2 part usages (representing the subcomponent instances) into the component part definition. The following two figures show two sample decompositions.

```
part def ShapeCollectorRobot :> SystemComponent {

    part bot :>> realAsset : RoverModel::RoverModel;
    part env :>> env : EnvironmentModel::Environment;

    //...
```

Figure 3: The System is composed of the Real Asset and the Environment.

```
part def RoverModel :> ExploDTwinMetamodel::RealAsset {

    part collection_subsys: CollectionSubsystem;
    part movement_subsys: MovementSubsystem;

    // ...
```

Figure 4: The Real Asset is composed of two subsystems.

The same pattern is also used to model the activities belonging to each subsystem.

```
part def MovementSubsystem :> Subsystem {

    part act_move: Move :> activities;

    // ...
```

Figure 5: The Movement subsystem has got one activity.

Subsystem attributes are modeled as attribute usages having *out* as direction. They can be typed by standard data types such as `Real` and `Integer`, or custom ones, such as `BoundedReal`. If an attribute is typed by a bounded data type, it must specify the values for the bounds. Finally, attributes can be annotated by custom metadata definitions.

A subsystem's mission state machine contains durative transitions for the subsystem's activities. Each durative transition is explicitly bound to an activity via the transition's *act* feature.

An activity definition shall redefine the appropriate abstract elements (e.g., `precondition`, `preeffect`, etc.) inherited from the base activity definition.

- `precondition`, `postcondition` and `invariant` are modeled as constraint usages. An activity definition can override them to provide an actual expression for each of them.

```
out attribute square_collected: Boolean;
out attribute triangle_collected: Boolean;
out attribute circle_collected: Boolean;

out attribute weight: BoundedInt {
    @IsObservable;
    attribute :>> minValue = 0;
    attribute :>> maxValue = 15;
}
```

Figure 6: Sample attributes from the Collection subsystem.

```
transition 'move-durative' : DurativeTransition
first ready
then ready {
    :>> act = act_move;
}
```

Figure 7: Sample durative transition from the mission state machine of the Movement subsystem.

```
constraint :>> precondition {
    location_from == current_location and
        activity_id == MovementActivityIDs::NO_ACTIVITY
}
```

Figure 8: Sample precondition.

- `preeffect` and `posteffect` are modeled as action usages. An activity definition can override them to provide a body for each of them. Such body shall only contain send actions.
- `getLowerBoundDuration` and `getUpperBoundDuration` are modeled as calculation usages. An activity definition must redefine both, to provide a lower and upper bound to its duration. These bounds can either be provided explicitly, or they can be computed by means of an uninterpreted function.

  An uninterpreted function definition shall include:

- the `@UninterpretedFunction` tag,
- the list of input parameters, with their type,
- the returned value type.

5.3 Validation and Verification Experiments

To evaluate the effectiveness of the proposed framework, we performed a series of verification and validation activities on the Rover case study. These experiments covered the entire lifecycle supported by the ExploDTwin toolchain: from the static formal verification of the model to the automated generation of

```
action :>> preeffect {
    send battery_level - (battery_cost(current_location, location_to))
        via set_battery_level;
    send MovementActivityIDs::Move via set_all_included_activity_id;
}
```

Figure 9: Sample preeffect containing an uninterpreted function invocation.

```
calc :>> getLowerBoundDuration {
    20
}
```

Figure 10: An explicitly defined activity duration lower bound.

plans, and finally, the runtime monitoring of telemetry.

5.3.1 Model Verification

The first step involved validating the correctness of the DTFML model itself. After translating the SysML v2 specification into SMV using the SAWS[2] tool, we performed symbolic model checking to ensure the consistency and safety of the architectural model.

- **Reachability Analysis:** We verified that all mission-critical states (e.g., *Collecting*, *Moving*) were reachable from the initial state. This ensures that the model does not contain *dead code* or unreachable configurations due to conflicting constraints. These checks were expressed as invariant/reachability properties and verified with nuXmv, which was able to provide execution traces witnessing the reachability of the state.
- **Safety Properties:** We verified domain-specific safety invariants. For instance, we checked for example that the rover never attempts to collect an object while moving. These checks were expressed as Linear Temporal Logic (LTL) properties and verified with nuXmv. Note that, since we are using integer/real variables and uninterpreted functions, SMT-based algorithms with automated abstraction refinement techniques have been employed.

5.3.2 Plan Generation

Once the model was verified, we utilized the planner integrated into the toolchain to synthesize a mission plan. We defined a high-level goal, such as "acquire three different kinds of sample". The planner successfully generated a valid sequence of durative actions (e.g., `Move`(L_1, L_2), `Collect`(L_2), `Move`(L_2, L_3), `Collect`(L_3), `Move`(L_3, L_4), `Collect`(L_4)) along with their start times and durations. This plan guarantees compliance with all platform constraints defined in the DTFML model (e.g., power consumption and timing).

```
calc def battery_cost{
    @UninterpretedFunction;
    in l_from : LOCATIONS;
    in l_to : LOCATIONS;
    return : Integer;
}
```

Figure 11: Sample uninterpreted function definition.

5.3.3 Plan Monitoring and Fault Simulation

To validate the runtime verification capabilities, we simulated the execution of the generated plan against both nominal and faulty scenarios. The workflow applied is as follows:

1. **Plan-to-LTL Translation:** The generated plan π was automatically converted into a LTL formula, φ_π. This formula formally encodes the expected behavior of the system, specifying that each action must start at the planned time and complete within its allowed duration constraints.
2. **Trace Generation:** We used the underlying model checking engine to generate two distinct execution traces:
   - *Nominal Trace (σ_{ok}):* A trace that fully satisfies the plan formula φ_π, representing a successful mission execution.
   - *Faulty Trace (σ_{fail}):* A trace where a specific failure was injected (e.g., the `Collect` activity taking longer than the upper bound defined in the model, or a battery discharge rate higher than expected), causing a violation of the plan.
3. **Projection on Observables:** To realistically mimic a telemetry stream received from a physical asset, the full state traces were projected onto the subset of variables annotated with `@IsObservable` in the SysML v2 model. This hides internal model states (such as internal planner variables) from the monitor, testing its ability to reason based solely on available telemetry.
4. **Monitoring:** The projected traces were fed into the runtime monitor.

These experiments confirmed the ability of the monitor to approve the correct plan executions and to detect a fault in the erroneous one.

6 Conclusions and Future Work

This paper presents the model-based DT framework of ExploDTwin for supporting planning and monitoring critical space systems and, in particular, the Digital Twin Formal Modeling Language (DTFML), a SysML v2 extension that bridges the gap between standard systems engineering practices and rigorous formal methods for temporal planning and runtime verification. With a dedicated library, DTFML allows the specification of activities to be planned and related functions for predicting resource consumption. We detailed the language constructs and exemplify them on a rover example.

Future research will focus on the following directions: investigating techniques to automatically extract partial DTFML models from standardized mission documentation and requirements specifications; extending the platform modeling capabilities to support hybrid systems, integrating continuous physical dynamics; integrating contract-based design into the planning loop, by formally defining assumptions and guarantees of the activities in temporal logic; using DTFML to specify a *Platform-Aware Mission Planning* (PAMP) problem (Panjkovic et al., 2025), to generate mission plans that are robust against all possible non-deterministic evolutions of the platform.

Acknowledgments AI tools were used for editorial purposes in this paper, and all outputs were inspected by the authors to ensure accuracy and originality.

REFERENCES

Barbie, P., Pollom, A., Fischer, R.-P., and Becker, M. (2025). Automated Generation of Standardised Digital Twins Based on MBSE Models. In *MODELSWARD*, pages 75–84.

Bonizzi, A., Calza, D., Flori, A., Gobbi, A., Kapellos, K., Micheli, A., Pujatti, M., and Tonetta, S. (2025). Runtime Verification of Prediction Based on a Formal Specification of Assumptions. In *EASI Workshop*.

Bordeleau, F., Combemale, B., Eramo, R., van den Brand, M., and Wimmer, M. (2020). Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges. In *ICSMM*, pages 43–54. Springer.

Cimatti, A., Dorigatti, M., and Tonetta, S. (2013). OCRA: A tool for checking the refinement of temporal contracts. In *ASE*, pages 702–705. IEEE.

Cimatti, A., Griggio, A., Magnago, E., Roveri, M., and Tonetta, S. (2019a). Extending nuXmv with Timed Transition Systems and Timed Temporal Properties. In *CAV (1)*, pages 376–386. Springer.

Cimatti, A., Tian, C., and Tonetta, S. (2019b). NuRV: A nuXmv Extension for Runtime Verification. In *RV*, pages 382–392. Springer.

Debiasi, A., Ihirwe, F., Pierini, P., Mazzini, S., and Tonetta, S. (2021). Model-based Analysis Support for Dependable Complex Systems in CHESS. In *MODELSWARD*, pages 262–269. SCITEPRESS.

Fitzgerald, J., Gomes, C., and Larsen, P. G., editors (2024). *The Engineering of Digital Twins*. Springer.

Flori, A., Fonda, T., Gobbi, A., Gobbi, S., Kapellos, K., Micheli, A., Tonetta, S., Valentini, A., and Ntagiou, E. (2025). Planning and Scheduling with External Functions in the ExploDTwin Project. In *IWPSS*.

Fox, M. and Long, D. (2003). PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124.

Heiler, P., Pollom, A., Becker, M., Fischer, R.-P., and Psota, T. (2024). MBSE 4 DT: A Tool for the Automated Generation of Operational Digital Twins Based on MBSE Models. In *MODELS 2024 – Tools and Demonstrations Track*.

Johnsen, E. B. (2022). Digital Twins as Evolving Model-Centric Systems. Invited talk, FormaliSE 2022.

Khelifati, A., Hammad, A., and Boukala-Ioualalen, M. (2025). Combining SysML V2 and BIP to Model and Verify CPS Interactions. In *ICSOFT*, pages 400–409. SCITEPRESS.

Kristensen, M. H., Bonizzi, A., Gomes, C., Hansen, S. T., Martin, C. I. I., Iven, H., Kamburjan, E., Larsen, P. G., Leucker, M., Talasila, P., Tang, V. T., Tonetta, S., Vosteen, L. B., and Wright, T. (2024). Runtime Verification of Autonomous Systems Utilizing Digital Twins as a Service. In *ACSOS-C*, pages 121–127. IEEE.

Lehner, D., Zhang, J., Pfeiffer, J., Sint, S., Splettstößer, A., Wimmer, M., and Wortmann, A. (2025). Model-driven engineering for digital twins: a systematic mapping study. *Softw. Syst. Model.*, 24(5):1339–1377.

Micheli, A., Bit-Monnot, A., Röger, G., Scala, E., Valentini, A., Framba, L., Rovetta, A., Trapasso, A., Bonassi, L., Gerevini, A. E., Iocchi, L., Ingrand, F., Köckemann, U., Patrizi, F., Saetti, A., Serina, I., and Stock, S. (2025). Unified Planning: Modeling, manipulating and solving AI planning problems in Python. *SoftwareX*, 29:102012.

Molnár, V., Graics, B., Vörös, A., Tonetta, S., Cristoforetti, L., Kimberly, G., Dyer, P., Giammarco, K., Köthe, M., Hester, J., Smith, J., and Grimm, C. (2024). Towards the Formal Verification of SysML v2 Models. In *MoDELS (Companion)*, pages 1086–1095. ACM.

Naeem, M. and Seceleanu, C. (2025). Contract-Based Verification of Digital Twins. In *ICECCS*, pages 338–357. Springer.

Panjkovic, S., Cimatti, A., Micheli, A., and Tonetta, S. (2025). Platform-Aware Mission Planning. In *ICAPS*.

Talasila, P., Gomes, C., Vosteen, L. B., Iven, H., Leucker, M., Gil, S., Mikkelsen, P. H., Kamburjan, E., and Larsen, P. G. (2025). Composable digital twins on Digital Twin as a Service platform. *Simul.*, 101(3):287–311.

Wright, T., Gomes, C., and Woodcock, J. (2022). Formally Verified Self-adaptation of an Incubator Digital Twin. In *ISoLA*, pages 89–109.