From Transpilers to Semantic Libraries

Formal Verification With Pluggable Semantics

Ármin Zavada
Budapest University of Technology and Economics
Budapest, Hungary
zavadaarmin@edu.bme.hu

Kristóf Marussy
Budapest University of Technology and Economics
Budapest, Hungary
marussy@mit.bme.hu

Vince Molnár
Budapest University of Technology and Economics
Budapest, Hungary
molnar.vince@vik.bme.hu

ABSTRACT

In the field of model-based systems engineering, there is an increasing demand for the application of formal methods. However, this requires expertise in formal methods, which cannot be expected from systems engineers. While several attempts have been made to bridge this gap, there are still open questions. (1) With the trend shifting towards ontological languages, systems are modeled as classes of 4D occurrences, rather than a 3D system evolving with time, which hinders the application of state-of-the-art model checking algorithms. (2) Ontological reasoning cannot handle the state space explosion problem, and can even make it harder for verifiers to operate efficiently. (3) When operationalizing ontological languages, we need to validate the conformance of the two semantics, even in the presence of optimizations. (4) On top of all, these challenges must be solved for every new engineering language, version, or variant. In this paper, we propose a new approach to address the aforementioned challenges. To validate its feasibility, we present a prototype tool and evaluate it on a SysML model.

CCS CONCEPTS

• **Software and its engineering** → **System modeling languages**; **Semantics**; • **Theory of computation** → *Formalisms.*

KEYWORDS

model-based systems engineering, kernel modeling language, formal verification, declarative interpretation, metaprogramming, semantic libraries, operational libraries

ACM Reference Format:
Ármin Zavada, Kristóf Marussy, and Vince Molnár. 2024. From Transpilers to Semantic Libraries: Formal Verification With Pluggable Semantics. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652620.3686251

1 INTRODUCTION

The application of formal methods in model-based systems engineering (MBSE) is becoming increasingly prevalent [15], as the introduction of models in the design process provides a straightforward opportunity to use automatic reasoning techniques on the knowledge base describing the system. The expected benefit is a better understanding and, often, the formal verification of the system's properties throughout the design process, starting from the requirement engineering phase and extending to the implementation and even the operation phases (e.g., diagnostics [38]).

A well-known challenge in formal verification is using formal languages to describe the design in a sufficiently precise form. Using such languages and the tools processing them generally requires a high level of expertise in formal methods, which is not expected from systems engineers [26]. There have been several attempts to bridge this gap by establishing mappings from high-level engineering languages to formal languages [9, 25, 28, 39]. While these approaches definitely helped in increasing the adoption of formal methods in MBSE, there are many open questions and repeating challenges that prevent its widespread application [18, 24, 35].

One of the fundamental theoretical challenges comes from the fact that engineering languages are mostly ontological in nature. They focus on the structure of models, with declarative semantics based on classification. While behavior models traditionally had operational semantics (even in UML [21] and SysML v1 [20], except Sequence Diagrams [27]), the latest trends display a shift towards ontological behavior modeling [6]. Version 2 of the Systems Modeling Language (SysML v2) [32] and the underlying Kernel Modeling Language (KerML) [31] have fully declarative semantics formalized with a set of axioms.

Ontological behavior modeling treats models as classes of 4D occurrences rather than a 3D system evolving with time [6]. This kind of semantics makes it easy to check if a given trace (i.e., a complete execution of a system) conforms to the model but does not give guidance on how to compute such an execution. On the contrary, operational semantics specify an algorithm to compute a temporal unfolding of the system behavior, which makes it easy to execute a model but does not provide an easy way to check the conformance of an execution and the model [29]. While there are ontology reasoners that can deal with declarative semantics, the efficiency of verification tools requiring operational semantics is generally vastly superior, partly due to decades of research improving their performance.

Another typical challenge for formal verification is the state space explosion problem [10, 37], i.e., even small models may produce a state space too large to represent in memory. To address this phenomenon, verification algorithms usually try to exploit the high-level structure of models (e.g., symmetries or common patterns) to compress the state space or optimize its computation [2].

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3686251

Another typical (partial) solution is to limit the engineering language to a subset that can be efficiently operationalized, excluding model elements that could inflate the state space (e.g., arbitrary concurrency).

A permanent question that usually remains unanswered is how to prove that the mapping from the engineering language to the formal language preserves the semantics. This is even harder in the presence of the mentioned optimizations, which can also happen during the transformation from one language to the other.

On top of all the challenges mentioned above, an extra layer of technical difficulties comes from the fact that these challenges have to be solved with every new engineering language, a new version or variant, or even for custom semantics used by companies (mostly because of an existing tool integration, such as a simulator).

In this vision paper, we will use SysML v2 [32] and KerML [31] to examine these challenges and formulate a vision about a way forward, ultimately leading to efficient verification of ontology-based modeling languages. In the context of these languages, we concretize the above open challenges as follows.

(1) How can we operationalize the (temporal) declarative semantics of an ontology-based language and keep the two semantics synchronized?
(2) How can we optimize the operationalization based on the domain-specific information encoded in the high-level model?
(3) How can we prove the conformance of the declarative and operationalized semantics, as well as the optimized and unoptimized operational semantics?
(4) How can we support the use of semantic variants of the same language or a family of similar languages?

Our proposed vision is based on encoding the operationalized semantics in the ontological knowledge base. This way, we can still use ontological derivations to reduce high-level models to a suitable abstraction level, where instead of further reduction and unfolding of the temporal aspects, we can map into the concepts of a formal language with operational semantics instead. This level can vary with the target formal language and the capabilities of the corresponding verification engine. We identify several opportunities to cross-check the semantic conformance of the different derivation and verification paths. By encoding the operationalization in the ontology, we reduce the transpilation aspect between the two languages to a simple syntactical translation, while the transformation of the semantics is left to the ontology reasoner, with the necessary reduction rules encoded in semantic libraries. These libraries become part of the knowledge base, resulting in flexibility when defining or customizing the operational semantics for a given model.

2 RELATED WORK

Supporting Challenge (1). Ontology-based concept modeling is getting increasingly popular in systems engineering. In [40], Lan Yang et. al. provide a thorough overview of the literature regarding the use of ontology in systems engineering. While structural models are commonly ontological, ontological behavior modeling is a relatively new direction, discussed in, e.g., [6, 7]. Ontological behavior models classify 4D traces of a system, encompassing both the structural aspects (3D) and the completely unrolled temporal aspects (4th dimension). This perspective is the basis of the openCAESAR MBSE framework and methodology developed at JPL [12], as well as the KerML [31] and SysML v2 [32] languages. None of these languages considered execution as a primary aspect, which makes it challenging to analyze these models with simulation or model checking (and other kinds of formal verification techniques building on operational semantics).

Supporting Challenge (2). Ontology-based reasoning and verification have been investigated, e.g., in [1, 4, 12]. While these approaches are sound, they generally do not scale to large models, because ontology reasoners are not designed to handle the size of the knowledge graph that the encoding of a fully unfolded state space would require. Model checking and related techniques have been addressing the state space explosion problem for decades [10, 37], e.g., with abstraction, symbolic model checking, and compositional verification, often exploiting symmetries or high-level patterns in the models [2, 11, 30, 34]. Contrary to ontology reasoners, model checkers are highly specialized tools optimized to handle huge state spaces in a highly compressed format, while still maintaining the efficiency of reasoning over the state graph. However, these tools generally expect a low-level formal language as input, which usually has well-defined (mostly) operational semantics.

Supporting Challenge (3). The gap between high-level ontological and low-level formal languages is often bridged by model transformations or transpilations [9, 25, 28, 39]. As mentioned, on top of the syntactic mapping, these model transformations also have to map from a declarative, ontology-based semantics to an operational one. Preserving the semantic equivalence (or at least conformance) of the two representations is crucial for the credibility of the whole analysis process. This is especially challenging when the specification of the semantics of the high-level language is not well-defined, as reported in [13], where the authors analyzed the specification of the PSSM [22] standard, and reported various discrepancies between the specification and its test set, indicating under-specified language semantics.

Supporting Challenge (4). The process of transforming engineering models into formal, analyzable models is intricate, often necessitating individual effort for each pair of languages. Although attempts have been made to simplify the N×M transformations into N+M by utilizing an intermediate language (e.g., [5, 8, 23, 28]), this approach also proves challenging: modifications to the intermediate language are often necessary to support specific high-level languages, making its maintenance difficult.

3 ONTOLOGICAL BEHAVIOR MODELING

The Domain layer of Figure 1 denotes a simple State Machine with two states and a state transition between them. State S1 has an *exit*, while S2 has an entry action. Both increment the variable x by one. The transition in between them *accepts* a *toggle* event, in which case it executes the *reset* action, then transitions to the S2 state.

Since KerML [31] – and by extension SysML v2 [32] – is defined as an ontology, the specified elements can be decomposed into lower-level concepts defining their meaning in a finer granularity. The decomposition and the corresponding reduction rules are defined in model libraries, building up the standard library from a few

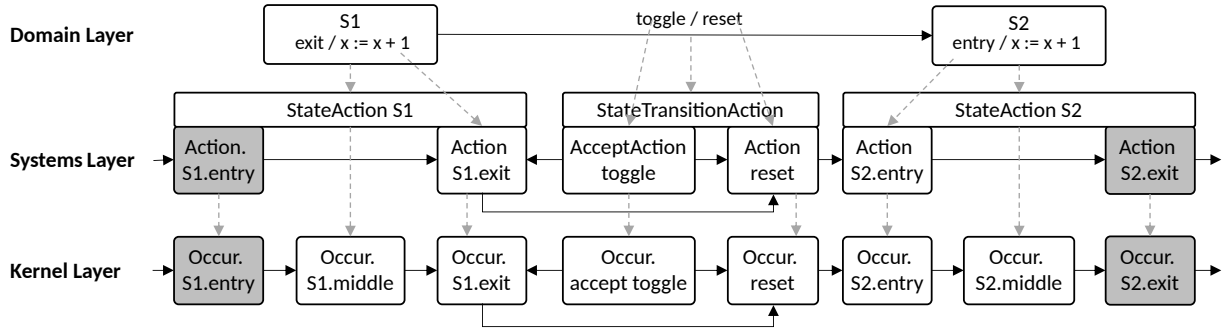

Figure 1: SysMLv2 representation of a simple State Machine in increasing granularities.

fundamental, core elements rooted in first-order logic. Excerpts of the SysML v2 Model Library related to State Machines can be found in Listing 1.

```
// System Library
abstract state def StateAction:>Action,
StatePerformance{
 entry action entryAction :>> 'entry';
 do action doAction: Action :>> 'do';
 exit action exitAction: Action :>> 'exit';
 //...
}
action def StateTransitionAction :> TransitionAction
abstract action def TransitionAction :> Action {
 action accepter : AcceptMessageAction :>> 'accept';
 action effect: Action :>> effect;
}
abstract action def Action :> Performance{...}
// Kernel Library
abstract behavior Performance :> Occurrence{...}
abstract class Occurrence {
 feature successors:Occurrence[0..*];
}
```

Listing 1: Excerpts from the SysML v2 System and Kernel Libraries.

Figure 1 also shows the various semantic layers (Listing 1) of the State Machine. The Systems layer defines *StateActions*, which model State Machines and States. States have entry-, do- and exit actions. The transition between the states is a *StateTransitionAction*, which has a *TriggerAction* of an *AcceptAction*, and an *effect* of reset *Action*. In this layer, the arrows denote the ordering between the actions, e.g., *S1 exit* must happen after *toggle accept* and *S1 entry*. Note that the assignment actions are not visualized for the sake of clarity, but would introduce a lot of additional complexity.

In the Kernel layer, Actions are defined as *Occurrences*. Occurrences are things that exist in space and *time*. Temporal relations are defined using *succession* connections between occurrences: succession enforces that instances of the earlier occurrence are finished before the matching later occurrence begins.

An execution of this model requires the instantiation of these concepts respecting the relationships and constraints. This will lead to an execution graph, which can be viewed as a refinement of the original model, resolving all the undefined aspects in the open-world semantics into a concrete instance. In this sense, formal verification is reasoning about the existence of an instance based on the knowledge base encoded in the model and the libraries.

4 FROM TRANSPILERS TO SEMANTIC LIBRARIES

A high-level overview of our proposed approach is illustrated in Figure 2. Components of the approach offer solutions to the challenges specified in section 1, respectively. The Figure shows the relationships between user models and libraries, which are both models (meta-level M1) expressed in a language (meta-level M2), as well as execution traces, which are instances of these models (meta-level M0). The language layer is not shown, as the power of the ontological approach stems from the use of libraries defining the (M1) concepts of the language, which are then assigned (M2) keywords in a higher-level language (e.g., KerML and SysML).

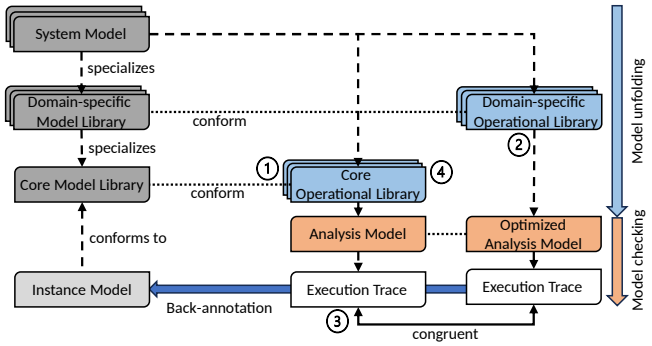

Figure 2: Vision Overview.

1) Core Operational Library. We propose to extend high-level ontological modeling languages with the ability to specify operational semantics as a first-class citizen. Doing so would allow language designers to specify the exact operational semantics of their language in a Core Operational Library as part of the ontological model library knowledge base. This allows the same model reasoning techniques current ontological modeling languages allow but also provides explicit operational semantics for model execution, simulation, and model checking. System models created in such a language could be directly mapped to the low-level analysis formalism.

2) Domain-specific Operational Library. The core operational library contains all the necessary information about the high-level

language's semantics. However, there are times when a more abstract representation is desired, e.g., in the state machine domain, the execution of guard expressions or assignments is better left to the verification tool (ontological reasoners would instantiate the execution of operands hierarchically). In such cases, a more abstract domain-specific operational library may be used.

Using such a library results in the analysis model merging some execution traces that would otherwise be separate in the original language. Model checking is thus a simpler problem with such models since the model checker has fewer execution traces to check. Figure 3 depicts an example of the execution traces of the example state machine introduced in Figure 1. The top trace corresponds to the domain-specific library, while the lower ones correspond to the Systems and Kernel layers. The lower layers contain more information about the exact unfolding of the behavior, but this information is not necessarily required for the verification. To enforce conformance between the core and domain-specific libraries, the execution traces must be congruent with each other, meaning the domain-specific execution traces must be stutter-equivalent [19] with the execution traces of the core layer.

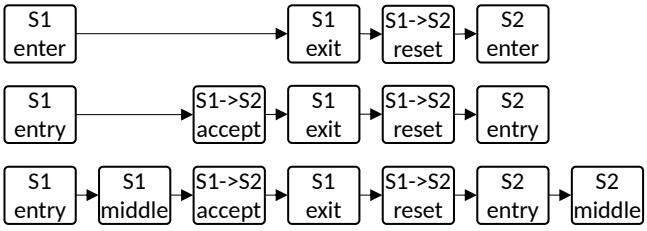

Figure 3: Execution traces of the Figure 1 State Machine in the Domain, Systems and Kernel layers.

3) Semantic conformance validation. It is crucial to validate the semantic conformance of the core and domain-specific libraries to the original ontological semantics. Since ontologies allow us to check the conformance of an instance model to the Ontology itself (i.e., none of the axioms are violated), any trace model can be conformance checked. The resulting witness execution trace of a verification process can be mapped back into the original language using back-annotation and then validated using the original Ontology.

4) Semantic variance support. With our proposed approach, semantic variances can be supported by implementing multiple variations of core- or operational libraries. Since the libraries are ontologies, the semantic variant libraries could reuse parts from a shared base library. For languages built on top of the same base library (e.g., SysML v2 and a new version of AADL [14] are both based on KerML), analysis models may interface with each other along the common base, allowing multi-domain verification with heterogeneous models.

5 PROTOTYPE IMPLEMENTATION

Validating the proposed approach requires the extension of an existing ontological modeling language with operational parts. However, since the SysML/KerML pilot implementation is not yet finished, there is no working model reasoner capable of producing valid instances of KerML models. For this reason, we implemented a prototype language with an instantiation mechanism that allows us to evaluate our approach.

5.1 Language

We introduce Ontological XSTS, a declarative-operational language built upon the XSTS [17] formalism. The language is made up of two parts. The first part supports ontological modeling similar to KerML, while the second grounds the semantics with composable operational atoms. Essentially, the ontological universe of the language is the set of all atomic XSTS statements, and the valid models are the statements resulting from the composition of the atoms.

5.1.1 Ontological Part. OXSTS supports similar modeling constructs as KerML, i.e., types and features, inheritance, and usage-oriented modeling. Listing 2 shows the State type implemented in OXSTS. *Types* are *classifiers* in OXSTS, that can reference each other through the use of *features*. *References* are special features without composition. *Containment* relations are *Features* that signify composition. Features have *multiplicities*, specifying the cardinality of their instances. The language supports classic ontological modeling techniques, such as feature *subsetting* and *redefinition*.

```
1 type State {
2   reference parent : Region[0..1]
3   reference parentState : State[0..1]
4   containment regions : Region[0..*]
5   containment entryActions : Action[0..*]
6   containment exitActions : Action[0..*] }
```

Listing 2: The State type in OXSTS.

The key in Ontological XSTS is that it only allows explicit and deterministic instantiation, while also allowing similar modeling techniques as KerML.

5.1.2 Operational Part. To operationalize the language semantics, we use XSTS as the base language. XSTS models can be reduced to SMT [3] formulae, but the language allows direct execution as well via its statements. XSTS is suitable to be an intermediate language [28], and there are model checkers [36] that can directly process it to create proofs with respect to formal requirements.

XSTS supports *variables*, *assignments*, n*on-deterministic assignments*, and *assumptions* to constrain variables. Operations are **atomic**, meaning they execute entirely, or not at all. Composing operations can be achieved with *choices* and *sequences*. Choices **choose** non-deterministically from their branches, while sequences execute **all** operations in the given order.

Listing 3 depicts the state machine introduced in Figure 1 in OXSTS on the domain-specific level to exploit high-level structure. Rather than modeling each Action defined in Listing 1, the states are encoded with an enumeration and an *activeState* variable, thus most of the succession steps can be eliminated from the model. Doing so reduces the state space of the model, thus increasing verification performance by decreasing the traces to be explored. The execution of the model starts by choosing a random value for *toggleEvent* (simulating the environment). Next, depending on the *activeState* value, the first or the second branch is selected from the choice.

```
enum States { S1, S2 }
var activeState: States = S1; // track current state
var toogleEvent: Boolean; var resetEvent: Boolean;
var x: Integer = 0;
tran { // main transition
 havoc (toogleEvent) // non-deterministic assignment
 choice {
  // S1 is active, and toggleEvent is received
  assume (activeState == S1); assume (toggleEvent);
  x := x + 1; resetEvent := true; // S1.exit action
  activeState := S2; x := x + 1; // S2.entry action
 } or {
  assume (activeState==S2) // no transitions from S2
}}
```

Listing 3: The high-level implementation of the Figure 1 state machine in OXSTS.

The S1 to S2 transition is modeled in the first choice, by executing the S1.exit, transition, and S2.entry actions respectively.

5.1.3 Combining the Parts. To extend the ontological language with operational semantics, we used constructs from metaprogramming to manipulate the atomic XSTS elements. Types and features can define variables and transitions (not to be confused with state transitions from the example). With the introduction of *inline operations*, the model can reference transitions to be unfolded by inlining their content at the call site. Composition is supported by *composite inline transitions*, which unfold into simple inline transitions placed into the appropriate composite transition: choice or sequence. Transitions, much like features, can be redefined, thus the language supports polymorphism. Static decision points can be modeled with *inline if* operations, that inline their bodies conditionally. Finally, a variable's domain can be a feature, resulting in that variable selecting over the instances in that feature.

```
type Transition {
 reference from : State[1..1]
 reference to : State[1..1]
 feature trigger : Trigger[1..1]
 feature actions : Action[0..*]
 feature guards : Guard[0..*]
 tran {
  inline trigger.isTriggered()
  inline from.isActive()
  inline seq guards -> main
  inline from.exit(commonRegion)
  inline seq actions -> main
  inline to.enter(commonRegion)
}}
```

Listing 4: The operational implementation of the Transition type.

Listing 4 shows the operationalized version of the Transition in OXSTS. Transitions have triggers, actions, guards, a source and a target state. Executing the transition entails checking the guards, triggers, and source state being active, by inlining the corresponding transitions, which contain the necessary assume operations. Next, the source state is exited, the transition actions are executed, and the target state is entered. State entry and exit actions are part of the *state.enter* and *state.exit* transitions, which produces operations that correctly enter or exit the state hierarchy respectively.

5.2 Semantifyr

Semantifyr is an open-source[1] proof-of-concept tool capable of instantiating OXSTS models and unfolding them into XSTS, mapping them directly to the operational analysis formalism. In our vision, a similar tool would implement the unfolding mechanism from the ontology model to the analysis formalism introduced in Figure 2.

OXSTS unfolding is done by first instantiating the model along the *containment* features, and then resolving all references in the model. Once the whole instance graph is calculated, model rewrite rules are repeatedly applied until the model no longer contains *non-xsts* constructs. Finally, the resulting XSTS model is serialized.

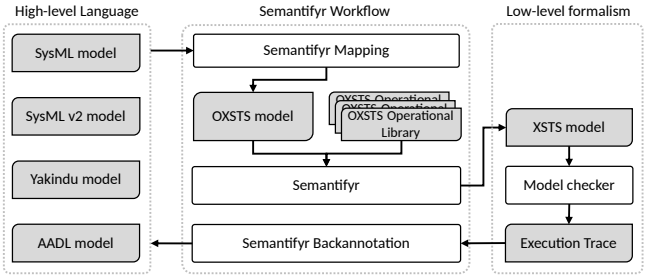

Figure 4: The end-to-end model transformation workflow of the Semantifyr tool.

The Semantifyr workflow depicted in Figure 4 is a simplified version of the vision defined in Figure 2. Parts ①, ② and ④ of our vision are realized using OXSTS Operational Libraries. By modeling the various language constructs available in the high-level engineering language (e.g., in KerML/SysML), a semantic mapping is possible between the language constructs and the types defined in the operational library. Each modeling language (and semantic interpretation variant) has a corresponding OXSTS operational library, thus the resulting OXSTS model is a mapping of the original model to an OXSTS model importing and reusing the necessary operational library types (such as *States*, *Transitions*, *Actions*). The constructed OXSTS models are unfolded into the XSTS formalism by Semantifyr. The resulting XSTS model is then passed through a model checker, after which ③ the execution traces are back-annotated to the original high-level language to describe the computed instance models, allowing for conformance checking. Note, that currently back-annotation is done by hand, automating which is future work.

6 SIMPLE SPACE MISSION

We evaluated the feasibility of our proposed workflow by using the prototype OXSTS language and Semantifyr. We modeled the Simple Space Mission model [33] created by the OpenMBEE community in SysML, as a substitute for SysML v2, since at this point, we cannot use a KerML-based ontological reasoner anyway.

The SysML model describes the communication of a satellite with limited battery charge with a ground station, where data transfer costs power. The state-based behavior of the station and spacecraft can be seen in Figure 5. The station is Idle initially, only starting *Operation* when the *Start* event is received or after 30 seconds.

[1] https://github.com/ftsrg/semantifyr

When the Operation state is entered, the *Spacecraft* is pinged – asking for data – and the *Receive data* internal behavior is activated. The ping signal is resent every 10 seconds. The Spacecraft State Machine has two parallel regions: one responds to ping signals, and the other handles battery recharging. The spacecraft only responds until the battery becomes low (under 30%), at which point it stops transmitting. Recharging begins under 80% and is done until fully charged.

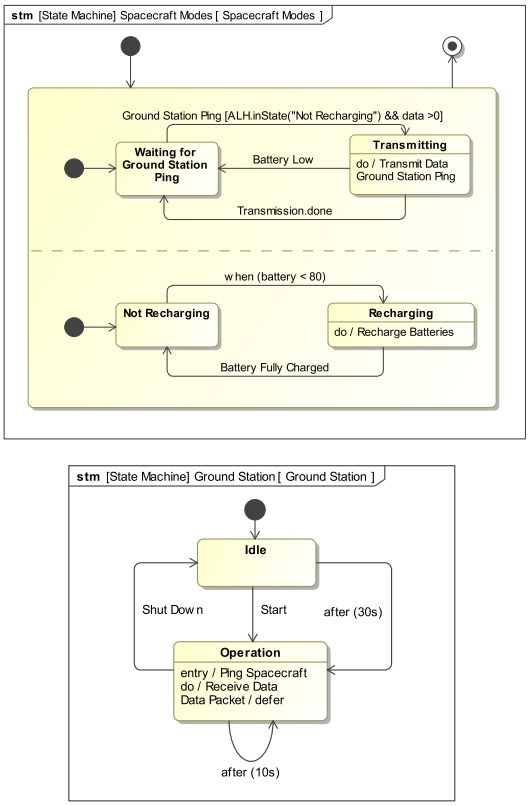

Figure 5: The State Machine behavior of the Station component.

The OXSTS implementation of the system entails the modeling of the ontology (States, Transitions, Spacecraft, Station) and operational semantics (how state transition is executed). The full models are available online [41]. Listing 5 shows excerpts from the used domain-specific operational library. *Regions* contain states and have an *activeState* variable with a domain of the *states* feature; this way the execution model can keep track of which state is active in a given region. States modify their parent regions' active states when entered or exited, e.g., exit is done by first executing the exit actions, clearing the *activeState* variable, then recursively exiting all parent states until the common region is reached. The Spacecraft component is a *Statechart* with ping and data events, and a few timeouts. An excerpt of the model is depicted in Listing 6.

To validate the semantic mapping of the model, we ran formal verification with an existing tool called Theta [16]. We compared the results to the direct verification of the XSTS model output by

```
type Region {
 containment states : State[0..*]
 var activeState : states[0..1] = Nothing }
type State {
 tran exitRecursive(commonRegion : Region) {
  inline seq exitActions -> main
  parent.activeState := Nothing
  inline if (commonRegion != parent) {
   inline parentState.exitRecursive(commonRegion) }}}
```

Listing 5: Excerpt from the State Machine semantic library.

```
type Spacecraft : Statechart {
 containment pingEvent :> inputEvents : Event
 containment transmitTimeout :> timeouts : Timeout
 var batteryCharge : Integer = 100
 var data : Integer = 100
 var recharging : Boolean = false
 containment Battery :> regions : Region {
  containment NotRecharging :> states : State // ...
  containment r :> transitions : Transition // ..
  containment Recharging :> states : State }
 containment Communication :> regions : Region {
  containment WaitingPing :> states : State }}
```

Listing 6: Excerpt from the Spacecraft component modeled in OXSTS.

Semantifyr. We checked 40 formal properties based on the SysML specification, for which all traces were consistent with the behavior described by the SysML model. While this is only a small experiment, it provided an initial validation for the key parts of our envisioned approach.

7 CONCLUSION

In this paper, we concretized open challenges in the context of using formal methods in model-based systems engineering with ontology-based modeling languages and proposed a vision in response to these challenges. Our approach is based on a novel combination of ontological modeling with operational modeling: we proposed the extension of ontological languages with operational semantic libraries. Creating a core semantic library for the language allows the explicit operationalization of all high-level language elements by tracing them back to the core axiomatic semantics of the language. To support optimizations, domain-specific operational libraries can abstract irrelevant details of the execution model. To check semantic conformance, the resulting execution traces can be validated against the original ontological semantics. Finally, language variants can be supported by specializing the core operational library in variant libraries. This approach can reuse some of the ontological aspects of state-of-the-art languages while leaving temporal aspects to a base language with operational semantics, thus inheriting the performance of state-of-the-art model checkers. We validated the feasibility of the approach through the prototype tool Semantifyr using a SysML model. We plan to further validate the approach by evaluating it in the context of KerML, since it allows the definition of high-level domain-specific languages, e.g., SysML v2, AADL.

ACKNOWLEDGMENTS

Supported by the **ÚNKP-23-2-III-BME-47** New National Excellence Program of the Ministry for Culture and Innovation from the National Research, Development and Innovation Fund.

REFERENCES

[1] 2018. *Ontology Based Behavior Verification for Complex Systems*. International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Vol. Volume 1B: 38th Computers and Information in Engineering Conference. https://doi.org/10.1115/DETC2018-85689

[2] Islam Abdelhalim, Steve Schneider, and Helen Treharne. 2012. An Optimization Approach for Effective Formalized fUML Model Checking. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 248–262.

[3] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[4] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. 2005. Reasoning on UML class diagrams. *Artificial Intelligence* 168 (10 2005), 70–118. https://doi.org/10.1016/j.artint.2005.05.003

[5] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. 2002. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In *Performance Evaluation of Complex Systems: Techniques and Tools*, Maria Carla Calzarossa and Salvatore Tucci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 236–260.

[6] Conrad Bock and James Odell. 2011. Ontological Behavior Modeling. *Journal of Object Technology* 10 (01 2011), 3: 1–36. https://doi.org/10.5381/jot.2011.10.1.a3

[7] Alexander Borgida and Ronald Brachman. 2003. Conceptual Modeling with Description Logics. 349–372.

[8] Ajay Chhokra, Sherif Abdelwahed, Abhishek Dubey, Sandeep Neema, and Gabor Karsai. 2015. From system modeling to formal verification. In *2015 Electronic System Level Synthesis Conference (ESLsyn)*. 41–46.

[9] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi, and Tullio Vardanega. 2012. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Automated Software Engineering*. ACM, 362–365.

[10] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Model Checking and the State Explosion Problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1

[11] Philippe Dhaussy, Jean-Charles Roger, and Frederic Boniol. 2011. Reducing State Explosion with Context Modeling for Model-Checking. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*. 130–137. https://doi.org/10.1109/HASE.2011.24

[12] Maged Elaasar, Nicolas Rouquette, Klaus Havelund, Martin Feather, Saptarshi Bandyopadhyay, and Alberto Candela. 2023. Autonomica: Ontological Modeling and Analysis of Autonomous Behavior. *INCOSE International Symposium* 33, 1 (2023), 1570–1585. https://doi.org/10.1002/iis2.13099 arXiv:https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/iis2.13099

[13] Márton Elekes, Vince Molnár, and Zoltán Micskei. 2023. Assessing the specification of modelling language semantics: a study on UML PSSM. *Software Quality Journal* 31, 2 (01 Jun 2023), 575–617. https://doi.org/10.1007/s11219-023-09617-5

[14] Peter Feiler, David Gluch, and John Hudak. 2006. *The Architecture Analysis & Design Language (AADL): An Introduction*. Technical Report CMU/SEI-2006-TN-011. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[15] Mario Gleirscher and Diego Marmsoler. 2020. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering* 25, 6 (01 Nov 2020), 4473–4546. https://doi.org/10.1007/s10664-020-09836-5

[16] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. 2020. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling* 19, 6 (01 Nov 2020), 1483–1517. https://doi.org/10.1007/s10270-020-00806-5

[17] Bence Graics, Milán Mondok, Vince Molnár, and István Majzik. 2024. Model-Based Testing of Asynchronously Communicating Distributed Controllers. In *Formal Aspects of Component Software*, Javier Cámara and Sung-Shik Jongmans (Eds.). Springer Nature Switzerland, Cham, 23–44.

[18] Henson Graves and Yvonne Bijan. 2011. Using formal methods with SysML in aerospace design and engineering. *Annals of Mathematics and Artificial Intelligence* 63, 1 (01 Sep 2011), 53–102. https://doi.org/10.1007/s10472-011-9267-5

[19] Jan Friso Groote and Frits W. Vaandrager. 1990. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *International Colloquium on Automata, Languages and Programming*. https://api.semanticscholar.org/CorpusID:267813466

[20] Object Management Group. 2012. Systems Modeling Language (SysML). https://www.omg.org/spec/SysML/1.6/About-SysML

[21] Object Management Group. 2017. Unified Modeling Language (UML-v2.5.1). https://www.omg.org/spec/UML/2.5.1/About-UML

[22] Object Management Group. 2019. Precise Semantics of UML State Machines (PSSM-v1.0). https://www.omg.org/spec/PSSM/1.0/About-PSSM

[23] Benedek Horváth, Vince Molnár, Bence Graics, Ákos Hajdu, István Ráth, Ákos Horváth, Robert Karban, Gelys Trancho, and Zoltán Micskei. 2023. Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering* (2023). https://doi.org/10.1002/sys.21679

[24] John C. Knight. 1998. Challenges in the utilization of formal methods. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Anders P. Ravn and Hans Rischel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[25] Martin Kölbl, Stefan Leue, and Hargurbir Singh. 2018. From SysML to Model Checkers via Model Transformation. In *Proc. of the 25th International Symposium on Model Checking Software (Lecture Notes in Computer Science, Vol. 10869)*. Springer, 255–274.

[26] Qin Ma, Monika Kaczmarek-Heß, and Sybren de Kinderen. 2022. Validation and verification in domain-specific modeling method engineering: an integrated life-cycle view. *Software and Systems Modeling* (18 Oct 2022). https://doi.org/10.1007/s10270-022-01056-3

[27] Zoltán Micskei and Hélène Waeselynck. 2011. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling* 10, 4 (01 Oct 2011), 489–514. https://doi.org/10.1007/s10270-010-0157-9

[28] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. 2018. The Gamma statechart composition framework: Design, verification and code generation for component-based reactive systems. In *Proceedings of ICSE'18: Companion Proceeedings*. ACM, 113–116. https://doi.org/10.1145/3183440.3183489

[29] D. Nardi and R. J. Brachman. 2007. *An Introduction to Description Logics*. Cambridge University Press, 1–44.

[30] Faranak Nejati, Abdul Azim Abdul Ghani, Keng-Yap Ng, and Azmi Jaafar. 2017. Handling state space explosion in verification of component-based systems: A review. *CoRR* abs/1709.10379 (2017). arXiv:1709.10379 http://arxiv.org/abs/1709.10379

[31] Object Management Group. 2023. *Kernel Modeling Language (KerML)*. Object Management Group. ptc/23-06-01.

[32] Object Management Group. 2023. *OMG System Modeling Language (SysML v2)*. ptc/23-06-02.

[33] OpenMBEE. 2019. OpenMBEE Simple Space Mission. https://github.com/OpenMBEE/OMGSpecifications. Accessed: 2024-03-29.

[34] Radek Pelánek. 2009. Fighting State Space Explosion: Review and Evaluation. In *Formal Methods for Industrial Critical Systems*, Darren Cofer and Alessandro Fantechi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–52.

[35] Charlotte Seidner and Olivier H. Roux. 2008. Formal Methods for Systems Engineering Behavior Models. *IEEE Transactions on Industrial Informatics* 4, 4 (2008), 280–291. https://doi.org/10.1109/TII.2008.2008998

[36] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: A Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of FMCAD'17*. FMCAD Inc., Vienna, Austria, 176–179.

[37] Antti Valmari. 1998. *The state explosion problem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 429–528. https://doi.org/10.1007/3-540-65306-6_21

[38] Emile van Gerwen, Leonardo Barbini, and Thomas Nägele. 2022. Integrating System Failure Diagnostics Into Model-based System Engineering. *INSIGHT* 25, 4 (2022), 51–57. https://doi.org/10.1002/inst.12412

[39] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. 2012. The hidden models of model checking. *Software and Systems Modeling* 11, 4 (2012), 541–555.

[40] Lan Yang, Kathryn Cormican, and Ming Yu. 2019. Ontology-based systems engineering: A state-of-the-art review. *Computers in Industry* 111 (2019), 148–171. https://doi.org/10.1016/j.compind.2019.05.003

[41] Ármin Zavada, Kristóf Marussy, and Vince Molnár. 2024. OpenMBEE Simple Space Mission in OXSTS. https://github.com/ftsrg/semantifyr/blob/main/engine/Test%20Models/Automated/Gamma/Spacecraft. Accessed: 2024-06-15.