A multi-view and programming language agnostic framework for model-driven engineering

Rodolfo Jordão, Fahimeh Bahrami, Rui Chen and Ingo Sander
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
jordao@kth.se, fahimehb@kth.se, ruich@kth.se, ingo@kth.se

***Abstract*—Model-driven engineering (MDE) addresses the complexity of modern-day embedded system design. Multiple MDE frameworks are often integrated into a design process to use each MDE framework's state-of-the-art tools for increased productivity. However, this integration requires substantial development effort.**

In this paper, we propose an MDE framework based on a formalism of system graphs and trait hierarchies for programming-language-agnostic integration between tools within our framework and with tools of other MDE frameworks. Implementing our framework for each programming language is a one-time development effort.

We evaluate our proposal in an MDE design process by developing a Java supporting library and an AMALTHEA connector. Then we perform an MDE industrial avionics case study with both. The evaluation shows that our framework facilitates the integration of different tools and the independent development of different system parts. Therefore, our framework is a reliable MDE framework that lowers the effort of integrating tools to benefit from their combined state-of-the-art.

***Index Terms*—Model-driven Engineering, System Modelling, Collaborative Tools**

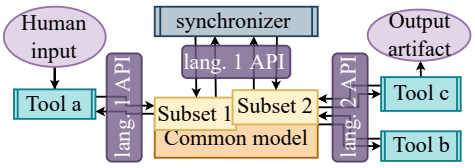
I. Introduction

Modern embedded systems design include a large number of aspects, from integrating different functionalities, to satisfying non-functional requirements and reducing production costs. This design complexity not only makes systems design harder, but also increases the penalty for wrong design decisions [1].

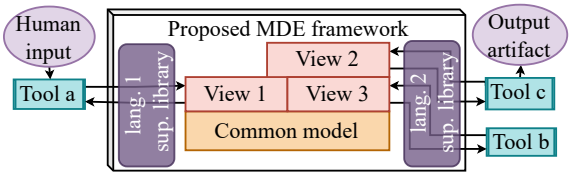
Model-driven engineering (MDE) is an approach that tackles this complexity by using *models* [2], [3]. These models then become the primary *inputs and results* of design activities involved in the embedded system design process. In this context, an MDE framework is a collection of models and *tools* that operate on these models to abstract design activities.

Frequently, an MDE embedded system design process has to integrate multiple tools from MDE frameworks in order to benefit from the state-of-the-art each tool provides at different phases of the design process. However, this integration between tools from different MDE frameworks requires substantial engineering effort [2].

An alternative to tool-to-tool integration that lowers the required engineering effort is to use a common model across the tools being integrated (Figure 1). These common models tend to be complex in order to properly abstract the design process, e.g. UML/SysML [4]; consequently, it has been observed that different tools might use different subsets of the same common model [2]. In this scenario, the integration between tools based on a common model requires keeping up-to-date the different subsets of the model (Figure 1a). This synchronization requirements between different subsets is an unintentional source of effort that hinders integration between tools via common model approaches. In the best case, the synchronizer is a tool that must be updated when any of the subsets that it keeps synchronized changes.



(a) Common model integration with different subsets kept up-to-date by a synchronizer. The conceptual overlap of the subsets is represented by a visual overlap.



(b) Common model integration in our framework without synchronization.

Fig. 1: Conceptual view of the integration approaches. "lang." is a short-hand of "Programming language".

In this paper we propose a novel extensible MDE framework based on a common model that addresses this integration limitation by combining the following two key properties:

1) the framework's common underlying model is based on a formalism of system graphs and trait hierarchies that

This research was partially funded by the Sweden's Innovation Agency (Vinnova), by the NFFP7 project 2019-02743: TRANSFORM - Design transformation for correct-by-construction design methodology, NFFP7 project 2017-04892: Correct by construction design methodology (CORRECT), and the ITEA3 project 2018-02228: PANORAMA - Boosting Design Efficiency for Heterogeneous Systems.

978-1-6654-7332-3/22/$31.00 IEEE

is consistent across different tools and supports cross-cutting views (Section III),
2) the framework's implementation for each programming language is a one-time engineering effort that eases the development of integrated tools in our framework and connections of tools in our framework to tools of other MDE frameworks (Section IV).

With these two properties, the tools are integrated in a programming language agnostic fashion by viewing parts of the unique common model without synchronization. To the best of our knowledge, our framework and its concepts are the first MDE framework to support both key properties mentioned.

We provide a proof-of-concept Java supporting library and an AMALTHEA framework [5] connector (Section V); and showcase our proposed framework through an industrial avionics embedded MDE design process case-study (Section V).

II. Related work

We survey in the related literature other frameworks and approaches that directly or indirectly address the effort of integrating tools in an MDE design process.

The ARCADIA methodology with its Capella tool is an integrative MDE framework that supports multi-stage design processes [6]. Cross-cutting concerns are expressed through the use of nested view points, in which the detailed system model is increasingly abstracted. Capella shares many goals and capabilities to our framework, but does not provide a language-agnostic implementation approach to integrate tools of other frameworks.

SPIRIT [7] is another MDE framework that promotes an interconnected multi-stage design process through a service-oriented approach. SPIRIT treats different models and tools in a top-down fashion and connects them by considering each tool as a service and orchestrating their execution. A distinction of our approach to SPIRIT is that the service-oriented architecture is optional: different tools connect to the same model conceptually, and not to each other directly. Moreover, our proposal provides a Java freely available supporting library. The Octopus toolset [8] framework is similar to SPIRIT in terms of integration, but with a narrower scope that focuses on Design Space Exploration (DSE).

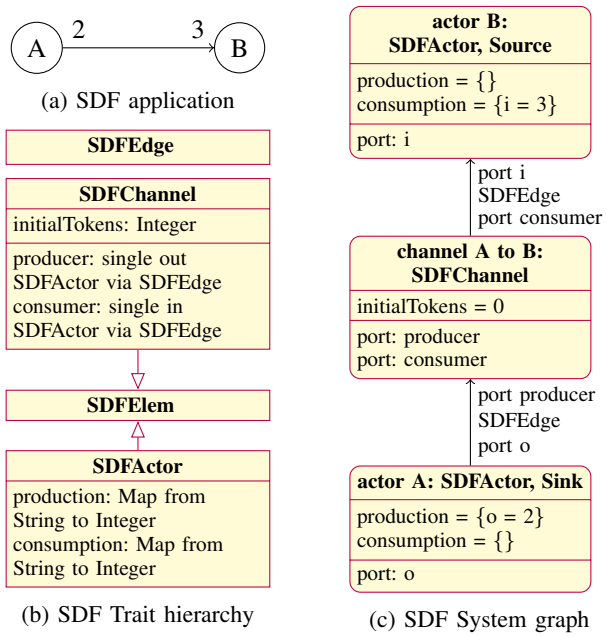
CrossEcore [9] is an MDE framework that extends the Eclipse modeling framework (EMF) with cross-platform capabilities: different MDE tools can share results by defining the same meta-model and connecting their data through CrossEcore. Although CrossEcore addresses the tool integration limitation as our framework, it does not conceptually incorporate cross-cutting viewing, as in SysML [4].

The Functional Mock-up Interface (FMI) is a simulation-focused standard that enables simulation tools to exchange a simulatable common model [10]; as FMI is a standard, tools that consume the common model must implement the standard directly or use a third-party development library. Similarly, the Ptolemy II project is an MDE framework with a simulatable common model that aids in the design of heterogenous embedded systems [11]. Contrary to FMI, the Ptolemy II project provides a common model called MoML [11], and a Java library for tools to interact with this model. The AMALTHEA model and its development platform App4mc [5] is another MDE framework that targets automotive embedded systems. AMALTHEA covers more aspects than functionality and simulation, in contrast to Ptolemy II and FMI, but does not provide a simulatable common model; rather, the model is used to exchange data between automotive tooling. Since App4mc is based on EMF, the integration is limited to other EMF-based tools or tools running in the Java virtual machine. In summary, these domain-specific MDE frameworks (or standards) follow the common model approach to integrate tools in their domains, but lack either of the two key properties our framework has.

Finally, EAST-ADL [12] and the MARTE profile for UML [13] are variants of UML/SysML which try to address the synchronization problem of Figure 1a by specializing UML/SysML. Like FMI, these variants are standards and do not provide first-party library for tools to be connected. There exist, however, established third-party tools for EAST-ADL and UML/MARTE such as Papyrus [14] that fill this gap. Therefore, their comparison with our framework is the same cited for AMALTHEA (App4mc) and FMI.

III. The framework overview

The proposed framework is split in two components: the *system graph* component and the *trait hierarchy* component. For simplicity, we consider $\mathbb{S}$ to be the set of all finite-length strings (made from `Unicode`) and $\mathbb{B} = \{\top, \bot\}$ to be the set of boolean values.



(a) SDF application

(b) SDF Trait hierarchy

(c) SDF System graph

Fig. 2: Example model of a SDF application (a), a trait hierarchy for SDF applications (b) and a system graph (c) for the application. Both trait hierarchy and system graph are shown in an UML-like format.

A. System graph component

A system graph is a tuple $G = (V, E)$ that includes the information about the system. This information can contain application, platform and design constraint information. A tutorial example is shown in Figure 2, where a system graph models a Synchronous Dataflow (SDF) application [15].

An SDF application is a network of concurrent processes, called *actors*, connected through unbounded FIFO *channels*. Each actor consumes and produces a fixed number of *tokens* in its input and output channels, respectively. An actor can *fire* as long as its input channels have enough tokens. In the SDF example of this paper, we ignore unconnected inputs and outputs of SDF applications as we focus on their analysability.

The example in Figure 2 also shows how the system information is concretely stored as a graph G. For each vertex $v \in V$, there is an *identifier* $n_v \in \mathbb{S}$, *nested properties* $D_v \in \mathbb{D}$ (defined next paragraph), *ports* $P_v \subset \mathbb{S}$, and *trait declarations* $T_v \subset \mathbb{S}$. For each edge $e \in E$, there are the *source* and *target identifiers* $s_e, t_e \in \mathbb{S}$, the *optional source and target ports* $sp_e, tp_e \in \mathbb{S} \cup \{\emptyset\}$, and trait declarations $T_e \subset \mathbb{S}$. The empty set is used for *optional* port existence, and it is not equal to an empty string.

The *property set* $\mathbb{D}$ of nested dictionaries can be inductively constructed by values d, where d is one, and exactly one, of the following:

1) a string, boolean or numeric literal: $d \in \mathbb{S} \cup \mathbb{R} \cup \mathbb{B}$,
2) a finite sequence of length $n \in \mathbb{N}$ in $\mathbb{D}$: $d \in \mathbb{D}^n$,
3) a dictionary of $n \in \mathbb{N}$ entries with integers or strings (exclusive or) as keys and other properties as values: $d \in (\mathbb{Z} \times \mathbb{D})^n$ or $d \in (\mathbb{S} \times \mathbb{D})^n$.

For any v, D_v is a dictionary with string keys $(\mathbb{S} \times \mathbb{D})^n$.

The string-centric definitions given for vertices and edges do not imply that edges in E connect vertices in V. We define the concept of *proper* system graphs to express this matched connectivity. A system graph $G = (V, E)$ is a proper system graph if, for any $e \in E$ there exists $v, v' \in V$ so that: $n_v = s_e$, $n_{v'} = t_e$, $sp_e \neq \emptyset \rightarrow sp_e \in P_v$ and $tp_e \neq \emptyset \rightarrow tp_e \in P_{v'}$. In other words, every edge links vertices that indeed exist in V of G and that the ports it declares indeed exist in both source and target vertices. That is, a proper system graph G is similar to a directed graph. System graphs are assumed to be proper in this text, unless otherwise specified.

B. Trait hierarchy component

A trait hierarchy $\mathcal{T}$ is a *tree of traits* where each *trait* is a set of *property* and *port requirements* for vertices of system graphs, or *connected vertices requirements* for edges of system graphs. Consider, for instance, the trait hierarchy of SDFs in Figure 2b. Four traits are defined in this hierarchy: `SDFEdge` for connections between vertices declaring `SDFElem`; `SDFActor` for actors, which requires the presence of two dictionaries with integer keys, production and consumption; `SDFChannel` for channels, which requires the number of initial tokens and two connected vertices via `SDFEdge` edges in ports producer and consumer.

Formally, a trait hierarchy is a partially ordered set $\mathcal{T} = (T, \prec_T)$ in which $(\prec_T)$ stands for the *refinement* relation. Whenever clear from context, We say for a trait t that $t \in \mathcal{T}$ as a short-hand for $t \in T$, and we use $\prec$ instead of $\prec_T$.

By convention, traits associated with vertices and edges not refine each other. Mathematically, for a vertex trait $t_v \in T$ and an edge trait $t_e \in T \setminus \{t_v\}$, neither $t_v \prec t_e$ nor $t_e \prec t_v$. This convention is exemplified in Figure 2b, where `SDFEdge` does not refine or is refined by the other (vertex) traits.

A vertex can declare a trait $t \in T_v$ that is not part of $\mathcal{T}$, in which case the trait is *opaque*. Opaque traits have no property and port requirements according to the hierarchy considered. This is fundamental to define the behaviour of a tool that work with system graphs containing traits outside the trait hierarchy of the tool (Section IV). For example, the system graph in Figure 2c conforms to the trait hierarchy in Figure 2b, while it contains two opaque traits: `Sink` and `Source`.

We formalize the requirements imposed by traits through four functions of a trait $t \in \mathcal{T}$ as follows.

First, the *vertex required ports* $Pr(t)$ returns a tuple (p, m, a, io, tn, te), where p is the port; m and a are boolean values signalling if there are *multiple* connected vertices to this port and if they are *ordered*, respectively; tn is the connected vertex(es) *optional trait*; te is the connecting edge(s) *optional trait*; $io \in \{in, out, inout\}$ describe the edge orientation of connected vertices to port p. The "producer" (Figure 2b) of `SDFChannel`, for example, is the tuple $(\text{producer}, \perp, \perp, out, \texttt{SDFActor}, \texttt{SDFEdge})$.

Second, the *vertex required properties* $Dr(t)$ returns a tuple (s, α) where $s \in \mathbb{S}$ is the required property's name and α is required property's *data type*. In our context, these data types are a restriction of allowed properties in $\mathbb{D}$. For example, the "consumption" (Figure 2b) required property of vertex trait `SDFActor` has data type $(\mathbb{S} \times \mathbb{R})^n$, for any $n \in \mathbb{N}$.

Third and fourth, the *allowed source trait* $Sr(t)$ and *allowed target trait* $Tr(t)$ define the allowed traits in $\mathcal{T}$ for both source and target vertices. These can be used for validations that are not possible with vertex port requirements.

Finally, we say a system graph G *conforms to a trait hierarchy* $\mathcal{T}$ if:

1) each vertex that declares a trait t has the properties in line with $Dr(t)$, and the ports with its connected vertices and edges in line with $Pr(t)$;
2) each edge that declares a trait t connects vertices in line with $Sr(t)$ and $Tr(t)$.

C. Multiple views

We extend the notion of *views* from SysML [4] to describe how tools are integrated in our framework. A view of the system graph is a subset of the information contained in the system graph, which discards ports, properties, vertices and edges that are not relevant for this view. Therefore, views can be used for assessment of cross-cutting concerns as in SysML [4], but also by tools. In our framework, a view of the system graph is given as a set of traits. For example, in Figure 3, the SDF application is the SDF view of the system graph (Figure 2a), given the SDF trait hierarchy of Figure 2b.

An important consequence of this notion is that a system graph view is invariant to new irrelevant information, allowing tools to be integrated seamlessly in our framework

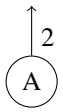
(Section III-E). For example, consider that the trait hierarchy of Figure 2b is extended through addition of a trait `FloatOpNeed`. `FloatOpNeed` requires the presence of an integer property "num flops" and does not refine any trait previously present in Figure 2b. If `actor A` additionally declares the trait `FloatOpNeed` and has a property "num flops" of 7, `actor A` becomes part of two loosely related views shown in Figure 3: the flops requirements view (Figure 3a) and the SDF view (Figure 3c).

Func.	Req.
actor A	7
⋮	⋮

(a) Flops req. view

actor A: SDFActor, extra::FloatOpNeed
production = {o = 2}
consumption = {}
num flops = 7
port: o

(b) Vertex being viewed



(c) SDF view

Fig. 3: A vertex being viewed in two different views.

To express these concepts in line with Section III-B, we define *vertex trait viewers*. A vertex trait viewer is a tuple $w(t) = (Dr(t), \mathcal{N}(t))$, where $Dr(t)$ is the vertex required properties of t as in Section III-B and $\mathcal{N}(t)$ is the set of *vertex view navigation* of t. Specifically, $\mathcal{N}(t)$ returns a set of functions $F(G, v)$ that returns *other vertex viewers* of vertices connected to vertex v in the system graph G. Each function F in $\mathcal{N}(t)$ follows directly from each required port in $Pr(t)$, and thus, returns the other viewers with correct ordering, multiplicity and orientation. Therefore, a viewer $w(t)$ navigates a system graph G as if G is exactly the graphical structure defined by $w(t)$. For example, $w($`SDFChannel`$)$, derived from Figure 2b, has two functions $F(G, v)$; one for the producer of v in G as a `SDFActor` viewer; and another for the consumer of v in G as a `SDFActor` viewer.

D. Composition and merging of system graphs

The composition of system graphs produces a new system graph with their combined information. The split-view example of Figure 3 can also be understood as the composition of two system graphs having a vertex with the same *identifier*, e.g. `actor A` which is shown in Figure 4. This interpretation also demonstrates how composition is possible across different views. The composition of system graphs is defined through the *merge operation* ($\oplus$), as described in the following paragraphs.

The merge operation ($\oplus$) merges the vertices and edges of two system graphs based on vertex identifiers. While merging properties of two vertices, there might be clashing information that cannot be solved trivially, e.g. two different literals that occupy the same nested position. Thus, we introduce the value **reject** as a possible result of ($\oplus$) to symbolize such non-trivial cases. If the merge results in any **reject** recursively, the merge is rejected.

For the sake of simplicity, let $d(s)$ return the value of dictionary d for the key s and a sequence l be denoted by $[l_1, l_2, \dots]$. Equations (1) to (5) define ($\oplus$) for all system graph definitions of Section III-A, save for vertex properties,

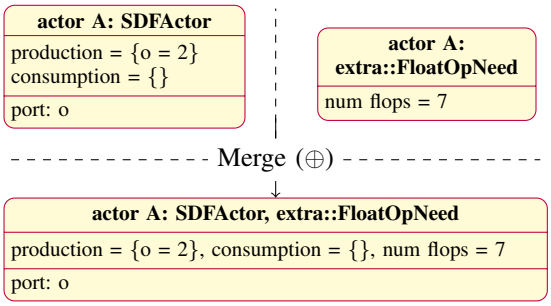


Fig. 4: Example composition across aspects through merging.

which is computed (and implicitly defined) by Algorithm 1. The ellipsis in Equation (5) is used as short hand for equality of the edge tuple elements minus its traits.

$$G \oplus G' = (V \oplus V', E \oplus E') \tag{1}$$

$$\begin{aligned} V \oplus V' = {} & \{v \oplus v' | v \in V,\, v' \in V',\, n_v = n_{v'}\} \\ & \cup \{v \in V |\ \nexists v' \in V',\, n_v = n_{v'}\} \\ & \cup \{v' \in V' |\ \nexists v \in V,\, n_v = n_{v'}\} \end{aligned} \tag{2}$$

$$\begin{aligned} E \oplus E' = {} & \{e \oplus e' | s_e = s_{e'}, \dots, tp_e = tp_{e'}\} \\ & \cup \{e \in E |\, \nexists e' \in E', s_e = s_{e'}, \dots, tp_e = tp_{e'}\} \\ & \cup \{e' \in E' |\, \nexists e \in E, s_e = s_{e'}, \dots, tp_e = tp_{e'}\} \end{aligned} \tag{3}$$

$$v \oplus v' = (s_v, P_v \cup P_{v'}, D_v \oplus D_{v'}, T_v \cup T_{v'}) \tag{4}$$

$$e \oplus e' = (s_e, t_e, sp_e, tp_e, T_e \cup T_{e'}) \tag{5}$$

Algorithm 1 $d \oplus d'$ computation

1: $d \oplus d' \leftarrow$ **reject**
2: **if** d, d' are literals ($\mathbb{S}$, $\mathbb{R}$ or $\mathbb{B}$) and $d = d'$ **then**
3: $\quad d \oplus d' \leftarrow d$
4: **else if** d, d' are dictionaries of equal key data type ($\mathbb{S}$ or $\mathbb{Z}$) **then**
5: $\quad (d \oplus d') \leftarrow$ empty sequence
6: $\quad$ **for** key s of either dictionary d or d' (union of keys) **do**
7: $\quad\quad$ **if** s is a key of both dictionaries (intersection) **then**
8: $\quad\quad\quad (d \oplus d')(s) \leftarrow d(s) \oplus d'(s)$
9: $\quad\quad$ **else if** s is a key of d only **then**
10: $\quad\quad\quad (d \oplus d')(s) \leftarrow d(s)$
11: $\quad\quad$ **else if** s is a key of d' only **then**
12: $\quad\quad\quad (d \oplus d')(s) \leftarrow d'(s)$
13: **else if** d, d' are sequences of size m and n with $m > n$ **then**
14: $\quad d \oplus d' \leftarrow d' \oplus d$
15: **else if** d, d' are sequences of size m and n with $m \leq n$ **then**
16: $\quad d \oplus d' \leftarrow [d_1 \oplus d'_1, \dots, d_m \oplus d'_m, d'_{m+1}, \dots, d'_n]$
17: **if** $d \oplus d' =$ **reject** **then**
18: $\quad$ **abort** with merge error of d and d'
19: **return** $d \oplus d'$

We remark that only $D_v \oplus D_{v'}$ might result in **reject**, since Equations (4) and (5) are only applied to cases when $n_v = n_{v'}$, i.e. associative cases. Thus, Algorithm 1 aborts in three cases: if any subdictionary or sequence is rejected (lines 4 and 15), if the properties have different types (lines 2, 4 and 15) or if two literals have different values (line 3).

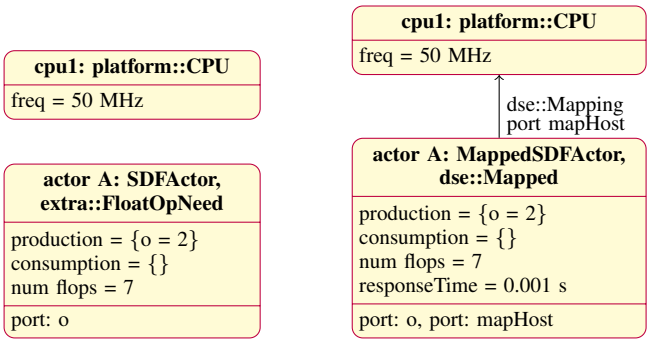
Therefore, we derive an important practical property of the

merge operation: $G \oplus G'$ is not rejected if all vertex property "leaf" literals are equal where they clash.

E. Tool integration

As outlined in Section III-D, system graphs can systematically be incremented with information without losses. This increment happens in two ways; first, by composing the current system graph with another; second, by refining the traits of a vertex or an edge. A refinement of trait still imposes at least the same requirements on the vertex or edge as the non-refined trait does. Consequently, the views of the refined system graph within a tool remain the same as they were before refinement; in turn, the tool integration is safely maintained.

For example, consider the two system graphs in Figure 5 and assume that `MappedSDFActor` refines `SDFActor`. `MappedSDFActor` requires a "responseTime" float property and a "mapHost" port to a `platform::CPU` via an edge declaring a `dse::Mapping` trait. The system graph in Figure 5b is the result of submitting the system graph in Figure 5a to a mapping and scheduling procedure for SDF applications in a multicore architecture, e.g. the DSE of [16]. An SDF simulator tool can simply ignore the added DSE information, since `actor A` is still a `SDFActor` through the refined `MappedSDFActor` trait.



(a) System graph before DSE (b) System graph after DSE

Fig. 5: Example of lossless information gain.

Tools that modify the system graph in a graph rewriting sense [17] can also be made lossless through appropriate trait definitions. For example, consider the transformation from SDFs to *job graphs* of [18], commonly performed for SDF synthesis. To express this transformation, we define a trait `JobFromSDF` with a port "actor" to a `SDFActor`. In Figure 2a, `actor A` executes three times in order to maintain memory bounded [18], so there will be three jobs created after this transformation, as shown in Figure 6. Then, the aforementioned SDF simulator tool can use the incremented system graph and produce the same results observed before the transformation.

F. External connections

The connection between our framework and others can be achieved by model-to-model (M2M) transformations in the sense of meta-modeling-based frameworks. These connections are implemented by using the supporting libraries available

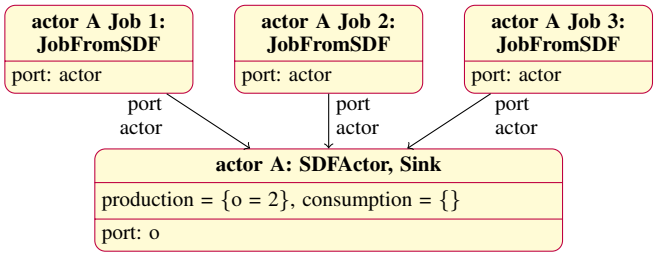


Fig. 6: Example of lossless system graph increment

for the implementation language of the other framework. For instance, to connect the SDF example of Figure 2 to the AMALTHEA framework, the connector would use the Java supporting library (Section IV) and the SDF trait hierarchy to produce Task graphs from both SDF actors and channels in system graphs.

IV. Implementation

An implementation of our proposed framework, i.e. complying to Section III, is a *supporting library*. Supporting libraries have a major goal: to guarantee that any tool based on our framework respects the definitions and consistencies of Section III across different programming languages. This relieves designers and tool vendors of the effort to guarantee such definitions and consistencies by themselves.

Each supporting library has three components to coincide conceptually with Section III and maximize the exploitation of the relevant work of other communities (Figure 7).

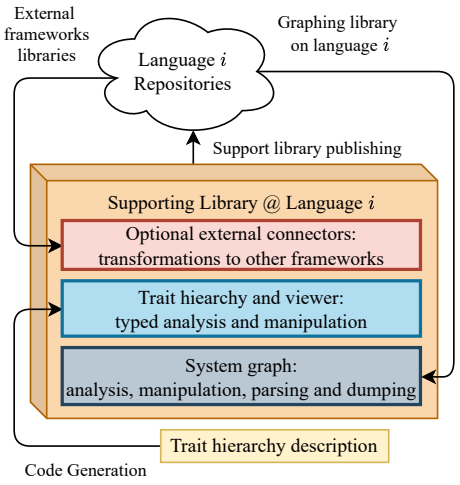


Fig. 7: Schematic representation of a supporting library.

The *system graph* component handles the in-memory representation and merging of system graphs as discussed in Sections III-A and III-D, as well as storing and restoring them from disk. It is also in this layer that merging and minimal consistency checks are implemented, raising errors whenever appropriate. If a graphing library is available, e.g. JGraphT [19] for the Java supporting library, then the system graph component represents system graphs in-memory through the graphing library's data structures. In all cases, the development of the system graph component is a one-time engineering

effort for each supported language, where the effort is the implementation of system graph data structures, e.g. vertices.

The *trait hierarchy* and viewer component contains the trait hierarchy, its viewers and the trait consistency validation discussed in Sections III-B and III-C. The trait viewers provide abstraction mechanisms for idiomatic access and manipulation of the system graph in the implementation programming language. For example, the trait viewers in the Java supporting library are classes with getter and setter methods that wrap the vertex being viewed. The java data types for these methods are derived from the Pr and Dr functions of Section III-B. If a trait is opaque, i.e. not in the bundled trait hierarchy, then there are no abstraction mechanisms, but the vertex information is maintained. Like the system graph component, the trait hierarchy and viewer component is a one-time development effort through automatic code generation: the component code is automatically generated from a *trait hierarchy description*.

The third and optional external connectors component contains *external connectors* to other MDE frameworks as discussed in Section III-F. It is not mandatory for these conversions to be a one-to-one mapping. For example, in the current Java supporting library, only a subset of AMALTHEA models is converted one-to-one.

The published supporting library does not need to have all external connectors bundled. In the Java supporting library, for instance, the system graph and trait hierarchy components are published as `forsyde-io-java-core` to Maven Central; on the other hand, the AMALTHEA external connector is published as `forsyde-io-java-amalthea` to Maven central and builds over `forsyde-io-java-core`.

A. Description languages

Supporting libraries can persist system graphs and trait hierarchies on disk in any format, but at least two human-readable formats are supported: a domain specific language (DSL) for system graphs and another DSL for trait hierarchies.

For the sake of brevity, we describe the DSLs with Listings 1 and 2 as examples, and argue that they contain all syntactical elements required to infer their formal context-free grammars. Their full ANTLR EBNF grammars can be found in the implementation repositories[1].

```
1   vertex actor_A [SDFActor] (o) {
2       "production": {"o": 2_i},
3       "consumption": {},
4       "num flops": 7_i
5   }
6
7   vertex channel_A_to_B [SDFChannel]
8   (producer, consumer) {"initialTokens": 0_i}
9
10  edge [DataEdge] from actor_A port o to
     channel_A_to_B port producer
```

Listing 1: System graph description excerpt of Figure 2a.

1) System graphs: The system graph DSL represents data in a flat structure as shown in example system graph of Listing 1. vertices (Lines 1 and 7) are declared with their traits (inside square brackets, Lines 1 and 7), their ports (inside parenthesis, Lines 1 and 7), and their properties (inside curly brackets, Lines 1-5 and 8). Edges are declared similarly, but with their ports declaration being optional (e.g. no **port** for target on Line 14). The vertex properties are declared in a JSON-like syntax, with two major differences: integers have optional suffixes that indicate if they are plain integers or long integers (e.g. Lines 2, 4 and 8), and strings can be written spanning multiple lines (not shown in Listing 1 or 2). We remark that there is no inherent separation in the description file as a result of the definitions of Section III.

[1]https://github.com/forsyde/forsyde-io

```
1   namespace execution {
2       vertex Stimulus
3       ...
4       vertex PeriodicTask refines Task {
5           port periodicStimulus is single in Stimulus
6       }
7       ...
8   }
9
10  namespace platform {
11      ...
12      vertex DigitalModule refines PlatformElem {
13          prop operatingFrequencyInHertz is long
14      }
15      ...
16      vertex GenericProcessingModule refines
     platform::DigitalModule
17
18      namespace runtime {
19          ...
20          vertex FixedPriorityScheduler refines
     AbstractScheduler {
21              prop preemptive is boolean
22          }
23          ...
24      }
25  }
```

Listing 2: Excerpt of the case study trait hierarchy description

2) Trait hierarchy: The trait hierarchy DSL is inspired by the object-oriented paradigm in that the vertices and edges declarations resemble class declarations, organized within namespaces. An example that expresses the domain of Section V is shown in Listing 2.

The separation convention between vertices and edges is enforced by the declaration keywords **vertex** and **edge**. The properties (e.g. Lines 13 and 21) and ports (e.g. Line 5) are directly translated to Pr and Dr. These, in turn, are directly translated to vertex trait viewers (Sections III-B and III-C).

The refinement identifiers (e.g. Line 4) can be declared both in absolute or relative forms. An absolute declaration contains "::" in the identifier (e.g. Line 16). The absolute form is the trait actual name in the trait hierarchy, as traits declared in the relative form are translated into their absolute forms during the parsing step of the DSL. For example, `Stimulus` in Line 2 is translated to `execution::Stimulus`.

B. Consistency validation and rejection

The DSLs give the supporting library the possibility of reporting consistencies of the description in an human-understandable format. These errors can be DSL syntax errors or failing to assert minimal consistencies. For example, if we change the source of edge in line 14 at Listing 1 to

`acrto_A` (note the incorrect spelling), the Java supporting library will throw the error: "`edge at 14:0 declares source 'acrto_A' that does not exist`".

Likewise, the trait hierarchy DSL rejects a trait hierarchy declaration and reports the error whenever a trait hierarchy cannot be built from the declaration. This occurs when trait declarations refine other traits that are not declared in the hierarchy. For example, if we change `PeriodicTask` to refine `Taska` instead of `Task`, the java supporting library will throw the following message: "`vertex trait 'execution::PeriodicTask' declared at 13:4 refines 'execution::Taska' which does not exist in the hierarchy`".

The goal of these mechanisms is to block tools that use supporting library from consuming inconsistent models created without a supporting library; for example, models created manually. In addition, these mechamisms can be used to cross-check if supporting libraries do not produce invalid models.

V. Illustrative Case Study

The industrial avionics case study that we perform is a safety-critical design scenario where different applications share the same platform. The objective is to find a mapping between the applications and the platform that guarantees that the design requirements are met, based on performance data.

To showcase how our framework fits this case study through an MDE approach, we develop a trait hierarchy that expresses the domain-specific information of the case study for automated DSE and external connectors, and four system graphs complying to this trait hierarchy (Figure 8).

The full case study details, along the system graphs and trait hierarchies produced are omitted for brevity but can be found in the implementation[2] and case study[3] repositories.

A. Applications, platform and constraints overview

The two applications are described as task graphs, A_1 and A_2, where the tasks represent code that must be executed. These tasks are activated periodically, with deadlines to be satisfied; might have precedence constraints between each other; and might communicate. The platform P is given as a connected graph of communication, processing, storage and I/O elements. Each processing unit executes its mapped tasks according to a fixed-priory preemptive policy. Moreover, the worst-case execution time (WCET) of executing a task in a processing unit is known; and the constraint C is that no processing element has utilization higher than 50%.

B. Case study trait hierarchy

We create a trait hierarchy $\mathcal{T}$ with a namespace for each view involved in Section V-A: a namespace *execution* for the applications; namespaces *platform* and its nested *runtime* for hardware and runtime software elements of the platform; a namespace *requirements* for the constraints; and a namespace *decision* to have the mapping and scheduling decisions. The last namespace emphasizes that the decisions regarding mapping and scheduling are part of this design process, and have

[2]https://github.com/forsyde/forsyde-io

[3]https://github.com/forsyde/panorama-kth-demonstrator

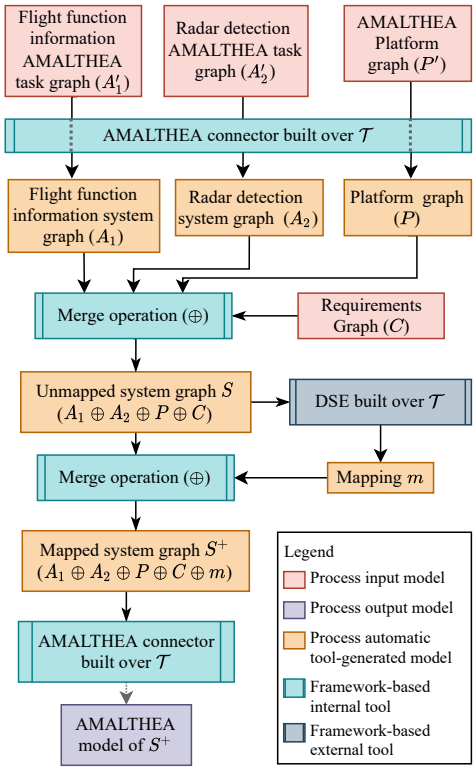


Fig. 8: Case study MDE process overview diagram

well-defined properties and relations to elements in the system graph. These declarations can be seen partially in the trait hierarchy of Listing 2. We remark that these declaration can co-exist with the ones in Figure 2b, to emphasize the multi-view extensibility of our framework.

C. Independent system graphs

Given the trait hierarchy $\mathcal{T}$ produced in Section V-B, we develop four models: one AMALTHEA model for each application (A_1' and A_2'), one AMALTHEA model for the platform P' (expressing both hardware and runtime software) and one system graph for the constraints C. The motivation to develop the system graphs after $\mathcal{T}$ is to ensure they are conforming to $\mathcal{T}$. The AMALTHEA models A_1', A_2' and P' are converted independently to system graphs A_1, A_2 and P through the AMALTHEA connector, described in Section V-E.

The four system graphs represent different design teams working in parallel without full knowledge of each other, aside from the identifiers of common elements. This is evident between P and C: the merge results are coherent if the identifiers between P and C's processing units match. This matching is equivalent to the design teams agreeing on common names for the system's processing elements.

D. Design space exploration

The merged unmapped system graph $S = A_1 \oplus A_2 \oplus P \oplus C$ is fed to a DSE tool that performs scheduling and mapping for S (DSE built over $\mathcal{T}$ in Figure 8). The DSE tool uses constraint satisfaction programming (CSP) and partitioned scheduling

equations in order to find a feasible solution where all the tasks meet their deadlines even when sharing the same processing element. In this work, we use the formulations in Forget *et al.* [20] and Khalilzad *et al.* [21], and defer the development of a customized DSE tool to future work. The current DSE tool is built using the Java supporting library (itself built over $\mathcal{T}$) and returns a system graph with mapping edges (m) between application and platform vertices once a mapping is found. These edges have traits `decision::Mapping` or `decision::Allocation` and are merged back with S to obtain the result of the DSE, $S^+ = S \oplus m$.

E. Connecting the results to AMALTHEA

In order to improve the reachability of the results of the case study design process, we choose AMALTHEA [5] as an secondary input and output model due to its maturity and presence in the automotive community. Consequently, we develop a connector between our framework and AMALTHEA, by using the freely available AMALTHEA modelling libraries.

The conversion algorithm, and its implementation, was tested for correctness by converting S and S^+ to AMALTHEA and back to system graphs, then checking for conversion and DSE inconsistencies. The full code can be found in the implementation repository[4].

VI. Conclusion

We present a novel model-driven engineering (MDE) framework based on a common model formalized by system graphs and trait hierarchies. Our framework's novel concepts can be implemented as a one-time-engineering effort for each language, enabling a language-agnostic integration of tools within our framework.

We evaluate our framework through an industrial avionics MDE case study. Our framework successfully performs four activities in the MDE case study as representatives of an MDE process: modelling of separate parts of the system, integration of these parts, integration of an external automated Design Space Exploration (DSE) tool and publication of results to another framework (AMALTHEA). These activities are performed with a Java supporting library that is developed as a one-time-engineering effort and is publicly available. We conclude that our framework constitutes a solid foundation to integrate MDE tools and exploit their state-of-the-art, including tools from different MDE frameworks.

A future direction for our approach is the automatic generation of external connectors for other MDE frameworks. Currently, developing external connectors is a manual one-time-engineering effort.

References

[1] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, Jan. 2012.

[2] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice," *Softw Syst Model*, vol. 17, no. 1, pp. 91–113, Feb. 1, 2018.

[3] M. Baleani, A. Ferrari, L. Mangeruca, *et al.*, "Correct-by-construction transformations across design environments for model-based embedded software development," in *Design, Automation and Test in Europe*, Mar. 2005, 1044–1049 Vol. 2.

[4] O. S. O. M. Group, *UML for Systems Engineering*, 2003.

[5] C. Wolff, L. Krawczyk, R. Höttger, *et al.*, "AMALTHEA — Tailoring tools to projects in automotive software development," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, Sep. 2015, pp. 515–520.

[6] Capella. "Capella Website." (2022), [Online]. Available: https://www.eclipse.org/capella/ (visited on 04/25/2022).

[7] J. Lu, D. Chen, J. Wang, and M. Torngren, "Towards A Service-oriented Framework for MBSE Tool-chain Development," in *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, Jun. 2018, pp. 568–575.

[8] T. Basten, M. Hendriks, N. Trčka, *et al.*, "Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems," in *None*, ser. Embedded Systems, T. Basten, R. Hamberg, F. Reckers, and J. Verriet, Eds., New York, NY: Springer, 2013, pp. 189–244.

[9] S. Schwichtenberg, I. Jovanovikj, C. Gerth, and G. Engels, "CrossEcore: An extendible framework to use ecore and OCL across platforms," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18, New York, NY, USA: Association for Computing Machinery, May 2018, pp. 292–293.

[10] T. Blochwitz, M. Otter, M. Arnold, *et al.*, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," in *Proceedings of the 8th International Modelica Conference*, C. Clauß, Ed., Dresden: Linköping University Press, Mar. 22, 2011, pp. 105–114.

[11] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, ser. EMSOFT '07, New York, NY, USA: Association for Computing Machinery, Sep. 30, 2007, pp. 114–123.

[12] H. Blom, H. Lönn, F. Hagl, *et al.*, "EAST-ADL: An architecture description language for automotive software-intensive systems," in *Embedded Computing Systems: Applications, Optimization, and Advanced Design*, IGI Global, 2013, pp. 456–470.

[13] O. M. O. M. Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, 2009.

[14] A. Lanusse, Y. Tanguy, H. Espinoza, *et al.*, "Papyrus UML: An open source toolset for MDA," in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, Citeseer, 2009, pp. 1–4.

[15] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[16] K. Rosvall and I. Sander, "Flexible and Tradeoff-Aware Constraint-Based Design Space Exploration for Streaming Applications on Heterogeneous Platforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 2, 21:1–21:26, Nov. 2017.

[17] B. Pinaud, O. Andrei, M. Fernández, H. Kirchner, G. Melançon, and J. Vallet, "PORGY : A Visual Analytics Platform for System Modelling and Analysis Based on Graph Rewriting," in *Extraction et Gestion de Connaissances*, ser. Extraction et Gestion de Connaissances 2017 (EGC2017), vol. RNTI-E-33, Grenoble, France: Revue des Nouvelles Technologies de l'Information, Jan. 2017, pp. 473–476.

[18] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. CRC press, 2017.

[19] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "JGraphT – A Java Library for Graph Data Structures and Algorithms," *ACM Trans. Math. Softw.*, vol. 46, no. 2, 16:1–16:29, May 18, 2020.

[20] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling Dependent Periodic Tasks without Synchronization Mechanisms," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2010, pp. 301–310.

[21] N. Khalilzad, K. Rosvall, and I. Sander, "A modular design space exploration framework for multiprocessor real-time systems," in *2016 Forum on Specification and Design Languages (FDL)*, Sep. 2016, pp. 1–7.

[4] https://github.com/forsyde/forsyde-io