TNO innovation for life

SysML v2 as foundation for ESI tools?

TNO Public ⟩ TNO 2024 R12436
6 December 2024

ICT, Strategy & Policy
www.tno.nl

TNO 2024 R12436 – 6 December 2024

SysML v2 as foundation for ESI tools?

Author(s)	Bram van der Sanden Pierre America
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	55 (excl. front and back cover)
Number of appendices	0

Acknowledgements
The research described in this report is supported by the Netherlands Organisation for Applied Scientific Research TNO.

Contents

1 Introduction

The systems modeling language (SysML) is a widely-used general-purpose language for systems engineering. The language has been around for over two decades, and is driven by both INCOSE and OMG as standard for systems engineering applications. It provides a language to specify system structure, using well-known formalisms like (internal) block diagrams to define components and interfaces, and languages to model behavior, using use cases, interaction diagrams, activity diagrams and state machines.

This year, the final specifications of SysML v2 will be released, and with that, commercial tooling support for SysML v2 will become available. As the language is no longer directly extending UML, the language has a better consistency and is more intuitive to system engineers. In contrast to SysML v1, the language now has both a graphical and a textual syntax. There is also a standardized API for exchanging model information. Finally, the new version of SysML v2 has been designed with extensibility in mind, such that domain-specific extensions can be created more easily.

Currently ESI has quite some tools based on the Eclipse modeling framework. Typical technology packages being used include Xtext, Sirius, and the Ecore metamodel. Looking at the language of tools like Eclipse LSAT, Eclipse POOSL, and Eclipse CommaSuite, we see that they are strongly inspired by formalisms also found in SysML (v2); activity diagrams in Eclipse LSAT, and block diagrams, interfaces, and interaction diagrams in Eclipse POOSL and Eclipse CommaSuite.

In this report, we investigate the applicability of SysML v2 as foundation/backbone for ESI tools. This is driven by the observation that there is quite some overlap in the underlying formalisms and the fact that SysML v2 promises to have more means to build domain-specific extensions on top. By aligning with SysML v2, we can embed our ESI tooling more easily within the different tool-ecosystems used by our partners such as Enterprise Architect and Cameo by Dassault. Instead of providing completely standalone solutions, we can integrate our tools as plug-ins in commercially supported solutions. Having a shared base eases adoption, as users do not have to learn a completely new notation and language for each new tool.

To explore the applicability of SysML v2, we take Eclipse LSAT, and model the language concepts of Eclipse LSAT based on SysML v2. Right now, there are already several prototype implementations available that enable us to do this exploration. Next to that, we want to see if the observations also hold for other ESI tools, including Eclipse POOSL and Eclipse CommaSuite.

2 Introduction to SysML v2

SysML, the Systems Modeling Language, is a general-purpose language to model systems. It provides the capabilities to create and visualize models capturing different aspects of a system, including requirements, structure, and behavior. In this section, we will not go into details on the language constructs available in SysML v2, as good introductions to both the textual [1] and graphical notation [2] are available online.

2.1 A brief comparison of SysML v1 and v2

SysML v1 was based on UML. SysML v2 is not based on UML any more. Instead, it is based on the new Kernel Modeling Language (KerML). KerML is an application-independent modeling language that has a formal semantics grounded in first-order logic. KerML is based on the following key concepts that provide the basis for SysML v2:

- **Elements** and **relationships**, and dependencies between model elements;
- **Specialization** of elements, including subclassification, subsetting, redefinition, and feature typing;
- **Namespaces** to contain and name elements, and packages to organize elements;
- **Annotations** to attach metadata to the model; and
- **Expressions** to specify calculations, case results, constraints and formal requirements.

Figure 2.1 shows the pillars of SysML v1 on the left-hand side, and the SysML v2 capabilities on the right-hand side. The pillar of parametrics in SysML v1 relates to the Analysis capability in SysML v2. SysML v2 additionally has the capability of views & viewpoints, and verification.

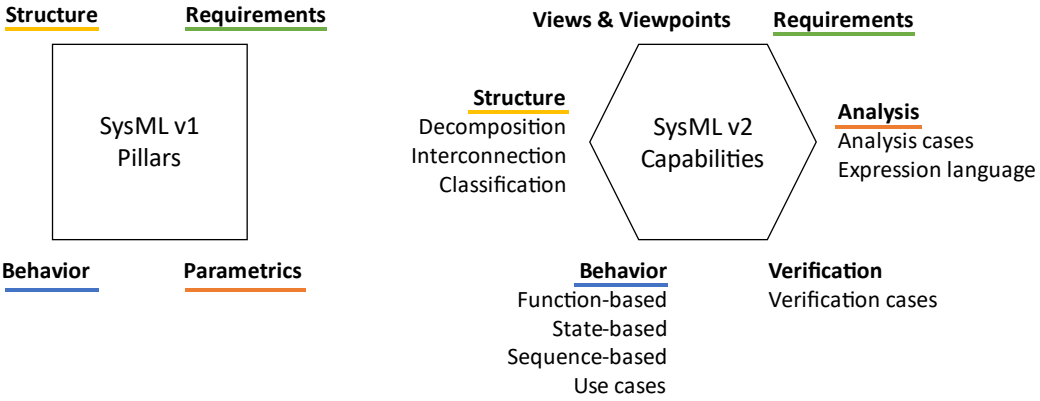

Figure 2.1: The pillars in SysML v1 and the capabilities in SysML v2.

SysML v2 has a number of advantages compared to SysML v1. Some of the most important advantages include:

- A new metamodel that is focused on systems modeling, with a **consistent language structuring**. This consistent structuring, based on the pattern of definition and usage is shown in Table 2.1.
- Both a **textual and graphical notation**.

- Native support for **variability modeling**, providing modeling constructs to express variation points, variants, and configurations.
- A standardized **API** to create, read, update, and delete SysML v2 models, where the models are in a Git-like **version control** system.
- **Package imports and namespaces** build into the language.
- 4D **modeling** of an object's life and spatial extent using the concepts of occurrences and snapshots.
- Means to describe **viewpoints and views** directly in the language, aligned with the ISO/IEC/IEEE 42010 standard.
- Means to **specify analysis cases**, like verification and simulation cases, next to use cases.

Table 2.1: Comparing SysML v1 terminology with SysML v2 terminology (non-exhaustive list). Source: [2]

SysML v1	SysML v2 (syntax keyword)	SysML v2 (metamodel concept)
part property | block	part | part def	PartUsage | PartDefinition
value property | value type	attribute | attribute def	AttributeUsage | AttributeDefinition
proxy port | interface block	port | port def	PortUsage | PortDefinition
action | activity	action | action def	ActionUsage | ActionDefinition
state | state machine	state | state def	StateUsage | StateDefinition
constraint property | constraint block	constraint | constraint def	ConstraintUsage | ConstraintDefinition
requirement	requirement | requirement def	RequirementUsage | RequirementDefinition
connector | association block	connection | connection def interface | interface def	ConnectionUsage | ConnectionDefinition InterfaceUsage | InterfaceDefinition
use case	use case | use case def	UseCaseUsage | UseCaseDefinition

There are various sources of information available when considering to migrate from SysML v1 to SysML v2. The OMG has a dedicated set of webpages with advice and Q&A on the migration path [3]. Documentation on an automated transformation from SysML v1 to SysML v2 is also available. This transformation will likely be implemented by various tool vendors, to support the migration [4].

2.2 Interacting with SysML models

SysML v2 models can be connected to external tools, and transformed to analysis models in various ways. In the current pilot implementation, there are at least four different ways to access a SysML v2 model:

- Using the SysML v2 API;
- By exporting the SysML v2 model to XMI or JSON;
- By parsing the SysML v2 model with a custom parser.

The first option is the preferred one, as the API provides a clean way to not only access, but also update the model, and comes with a version control mechanism. Parsing the SysML v2 model directly with a custom solution is the least desired option, as it does not leverage any of the existing functionalities already available. However, this approach is also used in practice, for example in PySysML2 [5].

2.2.1 Access and update models using the SysML v2 API

The OMG has created the Systems Modeling API and Services [6] to access, navigate, and operate on KerML-based models, and in particular, SysML models. The specification describes the types and details of requests that can be made (often CRUD operations; Create, Read, Update, Delete) and responses that can be received.

The specification includes a Platform Independent Model (PIM), and two Platform Specific Models (PSMs). The PIM provides a service specification, independent of the platform. For each of the services, the operations with their inputs and outputs are defined. A PSM is a binding of the PIM using a particular technology, like REST/HTTP, SOAP, Java, or Python. The specification includes a REST/HTTP PSM and OSLC PSM.

An important aspect of the Systems Modeling API is that it is a Git-inspired way of handling models, using concepts like branches, commits, and versions ("tags" in the specification). The API includes functionality for model management, and diffing to compare model versions of different commits.

Pilot server
As part of the pilot implementation, there is a public server where models can be published [7] (e.g., using the %publish command in the Jupyter Notebooks environment of the pilot implementation). Next to that, there is a server with the official SysML v2 models [8].

SysML v2 model elements as part of a digital thread
The SysML v2 REST API allows programmatic access to SysML v2 models, enabling users to access and update model elements via standard web protocols. The API can be used in connection with other standards like OSLC [9] (Open Services for Lifecycle Collaboration) that focus on data exchange between different tools, establishing a "digital thread". OSLC is a set of specifications that define how lifecycle management tools can integrate and exchange data, for example for requirements management, change management, and architecture modeling. With this integration, SysML v2 model elements can be linked to elements in other tools. For example, linking requirements specified in an external tool with their representation in the SysML v2 model. An example of an OSLC server for the SysML v2 REST API can be found on GitHub, see [10]. Other commercial tooling will also support this integration, like PTC Windchill [11].

SysML v2 API or custom version control?
As SysML v2 now also provides a textual syntax, there are two ways for version control of SysML v2 models. First, once can use well-known approaches like Git when only textual SysML v2 models are used. However, if also graphical models are created, then SysML v2 API seems the preferred approach, as the API works with stable identifiers for model elements. This provides also an advantage when restructuring the model, such that it is clear for example that only model elements moved to a different location, instead of the whole model being flagged as changed.

2.2.2 Exporting a SysML v2 model to XMI or JSON

The SysML v2 pilot implementation provides a utility `org.omg.sysml.xtext.util.SysML2XMI`, in the `omg.sysml.xtext` project [12], as discussed by Ed Seidewitz [13]. The utility can be executed by selecting a `.sysml` file using the `Save SysML2XMI` launcher. Alternatively, a directory can be selected, in which case it will recursively process all `.sysml` files in the selected directory and its subdirectories. The output XMI files have

`.sysmlx` extensions. The XMI is Eclipse specific, but it is quite close to the OMG-standard XMI. Next to the XMI transformation, there is also a JSON transformation available [12].

2.2.3 Viewpoints

An advantage of SysML v2 is that you can define custom viewpoints. These viewpoints can be used to filter what part of the model you want to show. A user of the viewpoint is often a particular stakeholder or set of stakeholders, but can also be a machine. When exporting a SysML v2 model to an analysis model, it can be very useful to first define a viewpoint and filter to include only the required information in the model.

An example can be found in model `11a-View-Viewpoint.sysml` in the pilot implementation, as shown in Figure 2.2.

```
viewpoint 'system structure perspective' {
    frame 'system breakdown';
}

view 'system structure generation' {
    satisfy 'system structure perspective';
    expose SystemModel::vehicle::**[@SysML::PartUsage];
    render asElementTable {
        view :>> columnView[1] {
            render asTextualNotation;
        }
    }
}
```

Figure 2.2: Example of a view and viewpoint, exposing only part usages within the context of the package SystemModel and the part vehicle within that package.

2.2.4 Examples of how to interact with SysML v2 models

As SysML v2 is not yet available as language in commercial tools, there are few concrete examples showing how SysML v2 model can be linked to analysis tools. In [14] Sanford Friedenthal, one of the SysML v2 core team members, gives an overview of some examples that have been worked out.

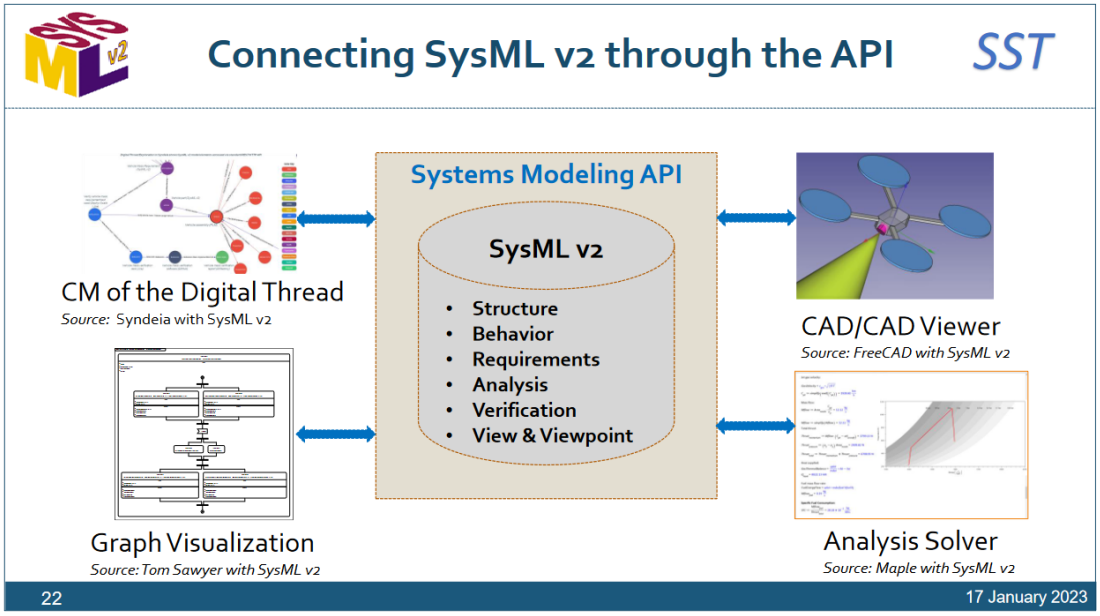

Figure 2.3: Slide by Sanford Friedenthal showing examples where SysML v2 is connected to other tools through the SysML v2 API. Source: [14].

FreeCAD integration

The integration with FreeCAD, an open-source CAD viewer, is illustrated using a quadcopter model. This model is part of the pilot implementation (`SimpleQuadcopter.sysml`, in the library Geometry Examples). The integration is currently done manually. This means that the model structure of the SysML v2 model mimics the structure of the FreeCAD model. At a later stage, an automatic integration with FreeCAD can be established using the SysML v2 API. With such an integration, modifications of values in either the FreeCAD or SysML v2 model will be automatically reflected in both models. The SysML v2 model can be used as abstract representation of the CAD model, capturing only the system-level aspects of relevance, and the CAD model can be refined to include more details.

Maple integration

Maple has created a SysML 2.0 connector plugin to connect to SysML v2 via OpenAPI [15]. The integration is illustrated using an example of a multi-stage jet engine, where the SysML v2 model describes the structural composition of the engine together with equations of the air flow and the air fuel mixture of the different components. Using Maple, the fuel composition can be analyzed and optimized.

A video is available that illustrates the steps used to extract information from the SysML v2 model and perform a subsequent analysis. This illustrates how SysML v2 can be used in combination with an analysis tool:

- Establish a connection to the server
- Load the model, by providing the project identifier and commit identifier
- Download the model and store the model in a Maple data structure
- Create a hash table based on the model element identifiers to link model elements
- Look up the model elements representing equations (type calculationDefinition)
- Look up all the equation usages and perform parameter substitution in Maple
- Causalize the model, such that it is clear which unknown is determined by which equation
- Check whether all variables can be computed, or whether there are missing variables

- Trim the equations, keeping only the equations that are needed to evaluate the fuel consumption
- Generate a Maple procedure that can compute the variables efficiently
- Use this procedure to compute the optimal values for fuel consumption

The example illustrates how the API can be used to retrieve a full model, but does not yet use the more advanced capabilities of SysML v2 to query on specific parts of the model or use viewpoints to filter parts of the SysML v2 model.

2.3 Restricting the language

It is currently an open question whether the SysML v2 language can be restricted. In a Q&A document released by ESA [16], it is indicated that *"SysML v2 will support so-called "conformance classes" to define coherent subsets of functionality, and permit verification by conformance test suites".* However, we did not find any proof or examples in the specification itself about the concept of conformance classes.

2.4 Extending the language

There are two main ways in which SysML v2 models can be extended; by using libraries, or using the "metadata" concept.

Libraries

The first way to extend and re-use SysML v2 models is by defining model elements in a library. These model elements can then be used by either instantiating them or by creating model elements that are specializations of the elements in the library. This approach is used heavily in SysML v2 itself. SysML v2 has several libraries for specialized modeling disciplines, including risk modeling, cause and effect modeling, geometrical modeling, analysis techniques, quantities, and SI units.

Semantic metadata

The second way to extend SysML v2 is by using the "metadata" concept. This concept replaces the stereotypes and profiling that were available in SysML v1 and UML. Using the "metadata" concept, new concepts with their own keyword can be added to the language.

For example, consider the use case where we want to add a concept, say a "Failure", that has a severity level. In SysML v2, we can add a metadata definition for the concept `Failure` and then use it as keyword `#failure`. In the definition, we link the metadata to the concept of Failure by making the link to the collection `failures` that will contain all instances in the model. With `meta SysML::Usage` we ensure that failures can be referenced as a usage. After a discussion with the SST team, it is likely that the specific syntax for using metadata will change in the future.

```
library package FailureModeling {
    import ScalarValues::Real;
    attribute def Level :> Real {
        assert constraint { that >= 0.0 and that <= 1.0 }
    }

    enum def LevelEnum :> Level {
        low = 0.25;
        medium = 0.50;
        high = 0.75;
    }
```

```
    abstract occurrence def Situation;
    abstract occurrence situations : Situation[*] nonunique;

    abstract occurrence def Failure {
        attribute severity : Level;
    }

    abstract occurrence failures : Failure[*] nonunique :> situations;

    metadata def failure :> SemanticMetadata {
        :>> baseType = failures meta SysML::Usage;
    }
}

package MyModel {
    import FailureModeling::*;
    #failure 'device shutoff' {
        :>> severity = LevelEnum::high;
    }
}
```

The downside of using semantic metadata is that it extends the SysML v2 language, rather than basing it on SysML v2 concepts as is done with libraries. By extending the language, the compatibility might be broken with transformations and analyses that support only vanilla SysML v2. There are not yet clear guidelines on which approach to use, but it seems advisable to use libraries if possible to mitigate this potential risk, and as the syntax of metadata has not yet been fully crystalized.

2.5 Model validations

SysML v2 is designed with extensibility in mind. Model validations play an important role to ensure that end users create models that adhere to the domain-specific extensions and refinements. Currently, we see the following options to add model validations.

Validation rules in the IDE

In the pilot implementation, SysML v2 models are validated using a set of validation rules programmed in Xtend [17]. A similar approach could be used to add extra custom validation rules to the SysML v2 IDE in Eclipse.

Validation rules as constraints in the SysML v2 model

SysML v2 has the concept of constraints to define predicates. A constraint can compute a Boolean-value result based on a set of input parameters. An example is shown below in Figure 2.4.

```
import SI::*;
import ScalarValues::Real;
import RealFunctions::sum;

constraint def MassConstraint {
    in partMasses : Real[0..*];
    in massLimit : Real;
}
constraint massConstraint : MassConstraint {
    sum(partMasses) <= massLimit
}

part def Engine;
part def Vehicle {
    assert constraint massConstraint : MassConstraint {
        in partMasses = (chassisMass, engine.mass, transmission.mass);
        in massLimit = 2500[kg];
    }
    attribute chassisMass : Real;
        part engine : Engine {
        attribute mass : Real;
    }
    part transmission : Engine {
        attribute mass : Real;
    }
}
```

Figure 2.4: An example of a constraint "massConstraint" that checks whether the total mass of the vehicle is below the mass limit.

2.6 Tool support

There are a number of open source tools already supporting (a subset of) SysML v2. Table 2.2 lists some of the well-known open source tools. Commercial tool vendors are also implementing SysML v2. Some well-known commercial tools that will support SysML v2 later on are shown in Table 2.3.

Table 2.2: Open source tools supporting SysML v2

Name	Textual, graphical, API	Platform	Vendor
SysIDE	Textual editing	Visual Studio Code	Sensmetry
SysON	Graphical editing	Web-based, based on Sirius	Obeo
SST Jupyter Notebooks plugin	Textual editing, graphical viewing	Jupyter Notebooks	SysML v2 Submission Team
SST Eclipse plug-in	Textual editing, graphical viewing	Eclipse, PlantUML for viewing	SysML v2 Submission Team

Table 2.3: Commercial tools supporting SysML v2, see also [18]

Name	Textual, graphical, API	Vendor
CATIA Magic & 3D Experience	Textual, graphical, API	Dassault
Rhapsody, Harmony	Textual, graphical, API	IBM
SysML v2 Modeler, Teamcenter integration		Siemens
Systems Architecture Modeler	Textual, graphical, API [19]	Ansys

Name	Textual, graphical, API	Vendor
Syndeia	API PLM implementation, integration to PLM and CAD systems, as well as Jira and management tools	INTERCAX
Model-Based Engineer	Graphical viewer, accessing models via API [20]	Tom Sawyer
PTC modeler	Graphical editing, OSLC server [11]	PTC
Enterprise Architect	Graphical editing, API	Sparx
System Composer	API [21] [22]	MathWorks

2.7 SysML v2 semantics

A limitation of SysML v1 was that it lacked an authoritative formal semantics that was commonly accepted. As a consequence, constructs in SysML v1 could be interpreted in different ways by different people, leading to many papers proposing different formalizations, and different choices made by the various tools implementing SysML v1 [23]. Recently, a paper has been published reviewing the current SysML v2 semantics [23]. This paper provides an accessible introduction to understand the semantics underlying SysML v2. As also indicated in the paper, the complexity of the SysML v2 specifications makes it very difficult to understand the semantic foundations.

Here, we would like to highlight some key conclusions of this analysis, without going into too much detail on the semantic nuances.

- **Alignment with other standards:** The formal semantics of the KerML core layer are defined in line with Tarski's model-theoretic semantics for predicate logic. This layer defines the syntactic foundation of SysML v2 models. As discussed in the paper, KerML's semantics depart from other standard model-theoretic semantics for structural modeling languages like RDFs, OWL, and Alloy. The authors suggest to consider bringing SysML v2 more in line with these established semantics.
- **Underspecification:** The semantics for some parts in the KerML core layer is underspecified. For example, concepts like **`composite`** and **`portion`** are only specified informally, as well as visibility qualifiers like **`public`**, **`private`**, and **`protected`**. Also, the authors conclude that the semantics of KerML is largely underspecified at the moment.
- **V&V of the specification:** There are no details on verification or validation of the formalization effort. Ideally a machine-readable version of the semantics should be available, that can be verified and validated.
- **3D vs 4D semantics for time and space:** The authors indicate that SysML v2 does not choose whether to use a 3D or 4D foundational to express time and space properties of objects. In the 3D view, objects exist in time, and their properties can change over time (by occurrences that happen in time), provided that essential properties are preserved. In the 4D view, the objects do not change, but instead, there are temporal parts with different properties.

As the authors have been in contact with the SysML v2 core team, we have good hope that the recommendations to improve the semantics of SysML v2 will be taken into account in the future developments of SysML v2.

3 Experiences with SysML v2

In this chapter, we reflect on our modeling experiences in Section 3.1 and our experiences while using the available tools in Section 3.2.

3.1 Modeling experiences

While modeling in SysML v2, we experienced that modeling guidelines and best practices are needed to create proper models. In this section we collected some of the questions we had and what guidance is available to help address them.

3.1.1 Modeling an element as item, part, or attribute?

It is not always immediately clear whether to use a "part" or "item" when modeling an element. Some general modeling advice is given by Ed Seidewitz on the SysML v2 forum [24]:

Parts are generally part of a system that interact with other and can perform actions. Items have an identity over time and may have a physical extend over space. All parts are items, but not all items are parts. An item may flow through or be acted on by a systems without be considered a "part" of the system.

For example, a car may be modeled as having parts like its chassis, engine, fuel tank, wheels, etc. However, while the fuel in the car is a physical item, it is not generally considered to be a "part" of the car. Rather, it is stored in the car and can flow within the car to the engine, where it is acted on to provide power.

In some cases, the same item may be considered a part in some contexts but not others. For example, in a system model of an assembly line for car engines, an engine under assembly is an item flowing through the assembly line, not a part of it. However, once the engine is assembled and installed in a car, it becomes a part of that car, performing a critical function in the overall behavior of the car.

The above text does not always provide clear guidance, so here are some heuristics that we propose to follow, even if they are not officially mentioned in the language definition.

- Persons: Use **item**, unless we are really interested in their internal structure, for example in medical applications.
- Raw materials, for example as fuel or in production processes
  - Use **item** when the material can be arbitrarily subdivided, e.g.,
    - liquid or paste, or
    - tape, ribbon, cable, or rope that is cut.
  - Use **part** when the components retain their identity in the production process. Note that the component would not become an aggregated part of the production equipment, but it can be referenced (using `ref part`).
- Data

- Use **attribute** when the data itself is immutable, e.g., integers, strings, records, or lists. In principle, this always applies to data that is transmitted.
- Use **part** when the data can internally be modified, e.g., array, file, digital twin, or database.

3.1.2 How to link structure and behavior?

There are multiple ways of linking structure and behavior in SysML v2. Based on the examples provided in [1], the preferred way is to have a structural decomposition and a separate action decomposition, and linking the actions to the structural elements using the **perform (action)** keyword. However, behavior can also be specified together with the structural decomposition.

3.1.2.1 Actions and parts

Here is an example of the first (preferred) approach:

```
action def 'Compress Air' {
    in 'low pressure air' : Air;
    out 'high pressure air' : Air;

    action 'sense pressure' : 'Sense Pressure';

    action 'control motor' : 'Control Motor';
    flow 'sense pressure'.pressure to 'control motor'.pressure;

    action 'generate torque' : 'Generate Torque';
    flow 'control motor'.voltage to 'generate torque'.voltage;

    action 'pump air' : 'Pump Air';
    bind 'low pressure air' = 'pump air'.'low pressure air';

    action 'store air' : 'Store Air';
    bind 'high pressure air' = 'store air'.'air outlet';
    flow 'pump air'.'high pressure air' to 'store air'.'air inlet';
    flow 'store air'.'pressure outlet' to 'sense pressure'.'air';
}

part def 'Air Compressor' {
    perform action 'compress air' : 'Compress Air';
    port 'air in' : 'Air Port';
    port 'air out' : ~'Air Port';

    part 'motor controller' : 'Motor Controller' {
        perform 'sense pressure' = 'compress air'.'sense pressure';
        perform 'control motor' = 'compress air'.'control motor';
    }

    part motor : Motor {
        perform 'generate torque' = 'compress air'.'generate torque';
    }

    part pump : Pump {
        perform 'pump air' = 'compress air'.'pump air';
    }

    part tank : Tank {
        perform 'store air' = 'compress air'.'store air';
    }
}
```

Here we see that the main action, `Compress Air`, is broken down into constituent actions, which are connected by flows. Similarly the main part, `Air Compressor`, is divided into components and each component specifies exactly which sub-action of the main action it performs.

What appears to be missing from the SysML v2 language is a canonical way to specify that the input and output of performed actions takes place via specific ports in the parts definition (in the example, `air in` and `air out`). There are accept and send actions, which can indicate ports in their **via** clauses, but these appear to be limited to the transfer of messages, not physical items like fluid or power. This is a pity, since this seems an unnecessary restriction.

A disadvantage of the above approach could be that it appears necessary to include the breakdown in constituent actions in the top-level action definition, where you may just want to talk about the top level abstractly. This could be solved by defining the top level separately:

```
action def 'Compressing Air' {
    in 'low pressure air' : Air;
    out 'high pressure air' : Air;
}
```

Then the concrete top-level action could be specified as a specialization:

```
action def 'Compress Air' :> 'Compressing Air' {
     …
    action 'pump air' : 'Pump Air';
    bind 'low pressure air' = 'pump air'.'low pressure air';

    action 'store air' : 'Store Air';
    bind 'high pressure air' = 'store air'.'air outlet';
     …
}
```

As mentioned above, a different approach would be to describe the actions completely within the structural definitions, as follows:

```
part def 'Air Compressor' {
    perform action 'compress air' {
       in 'low pressure air' : Air;
       out 'high pressure air' : Air;
        action 'sense pressure' {…}
        action 'control motor' {…}
        action 'generate torque' {…}

        action 'pump air' {
            in torque : Torque;
            in 'low pressure air' : Air;
            out 'high pressure air' : Air;
        }
        bind 'low pressure air' = 'pump air'.'low pressure air';

        action 'store air' {…}
          …
    }

    port 'air in' : 'Air Port';
    port 'air out' : ~'Air Port';
```

```
    part 'motor controller' : 'Motor Controller' {
        perform 'sense pressure' = 'compress air'.'sense pressure';
        perform 'control motor' = 'compress air'.'control motor';
    }

    part motor : Motor {
        perform 'generate torque' = 'compress air'.'generate torque';
    }

    part pump : Pump {
        perform 'pump air' = 'compress air'.'pump air';
    }

    part tank : Tank {
        perform 'store air' = 'compress air'.'store air';
    }
}
```

Here we see no action definitions at all, only action usages performed by specific parts. This description is more compact, but it does not allow you to reason about certain actions more abstractly, decoupled from the system structure. Here it would even be impossible to mention in the part definition of `Pump` that it performs a `pump air` action, since that action is defined only in the context of the `Air Compressor` part. Of course, it is always possible to adopt an intermediate approach.

3.1.2.2 Use cases

A similar issue is how to specify which part is doing which action in a use case. If you have a use case definition:

```
use case def 'Turn On Vehicle' {
    action 'send ignition on' : 'Send Ignition On';
    then
    action 'send vehicle on' : 'Send Vehicle On';
}
```

and a part or item definition:

```
item def Driver :> 'Vehicle Occupant' {
    perform action 'send ignition on' : 'Send Ignition On';
}
```

then you can have a use case usage:

```
include use case 'turn on vehicle' : 'Turn On Vehicle' {
    subject :> vehicle {
        perform action :> 'send vehicle on';
    }
    actor :> driver {
        perform action :> 'send ignition on';
    }
}
```

but only the definition in the Driver item definition applies, not the one in the 'Turn On Vehicle' use case definition. Visual Studio Code indicates this by highlighting the related identifiers:

```
actor :> driver {
    perform action :> 'send ignition on';
}

item def Driver :> 'Vehicle Occupant' {
    perform action 'send ignition on' : 'Send Ignition On';
}
```

and by omitting the highlighting were the same identifier is not referenced:

```
use case def 'Turn On Vehicle' {
    action 'send ignition on' : 'Send Ignition On';
    then
    action 'send vehicle on' : 'Send Vehicle On';
}
```

Fortunately, there is a way to indicate that the performed action in the use case specializes both the one from the action definition and the one from the structure definition. This works by binding them together:

```
include use case 'turn on vehicle' : 'Turn On Vehicle' {
    subject :> vehicle {
        perform action :> 'send vehicle on' = 'Turn On Vehicle'::'send vehicle on';
    }
    actor :> driver {
        perform action :> 'send ignition on' = 'Turn On Vehicle'::'send ignition
on';
    }
}
```

3.1.2.3 States and parts

The next question around behavior we might ask is whether it is possible to separate state definitions (hierarchical ones, which correspond to state machines) and part definitions. The reason to do this, again, could be the wish to talk about the behavior more abstractly than just in the context of a part. And it turns out to be possible.

We first define the behavior in terms of state definitions:

```
state def 'Vehicle States' {
    state 'vehicle off';

    state 'vehicle on' : 'Vehicle On States' {
    }

    entry; then 'vehicle off';
    transition …
}

state def 'Vehicle On States' {
    state neutral;
    state forward;
```

```
    state reverse;

    entry …
    transition …
}
```

Now we can state that a part should exhibit these states:

```
part def Vehicle {
    …
    exhibit state : 'Vehicle States';
}
```

But there is a complication: In some cases we want to specify guard conditions on transitions deep in the state machine. In our example, we may want to specify that the forward gear can only be engaged when the vehicle's speed is zero. That means that the state definition needs access to this attribute. We realize this by adding a parameter to the state definition:

```
state def 'Vehicle On States' {
    in ref vehicle: Vehicle::Structure::Vehicle;
    state neutral;
    state forward;
    state reverse;

    transition
        first neutral
        accept 'Forward Select'
        if vehicle.speed == 0
        then 'forward';
     …
}
```

That also means that the parent state must pass that parameter on:

```
state def 'Vehicle States' {
    in ref vehicle: Vehicle::Structure::Vehicle;
    state 'vehicle off';

    state 'vehicle on' : 'Vehicle On States' {
        in ref :> vehicle = 'Vehicle States'::vehicle;
    }
    …
}
```

And in the part definition we must also pass this parameter:

```
part def Vehicle {
    attribute speed : ISQ::SpeedValue;

    exhibit state : 'Vehicle States' {
```

```
        in ref :> vehicle = this;
    }
}
```

Note that we pass a reference to the vehicle as a parameter instead of just the speed, because the speed will vary over time and we want to refer to the speed at the instant of changing gears.

So we see that in theory it is possible to separate the state model from the structure, but in practical cases this may become very clumsy, leading to circular references. In these cases it may be better to define states in the context of the relevant structure.

3.1.3 Structuring a large model

When developing a large model, we typically want to subdivide it in such a way that multiple people can work on it in parallel without getting in each other's way too much. Among others, that also means that we want to manage versions of the various segments (let's use that word for now) separately. For this version management we may want to use well-known mechanisms such as Git. This fits nicely with the textual representation of SysML v2.

Since SysML has the notion of package, it makes sense to use packages to structure the model. But SysML does not specify clearly how packages can be represented in files: Since packages can be nested, it is clear that multiple packages can be stored in one file. But the SysML v2 specification does not preclude spreading a package over multiple files. We think this is not a good idea, since it would not be clear that the package representation in one of those files is not complete. It is also unclear how various tools would deal with this. Therefore we recommend to never spread a package over multiple files and to keep package nesting to a minimum.

In various books and other publications on working with SysML v1 (e.g., [25] [26]) we see a model structure like in Figure 3.1 or even like in Figure 3.2.

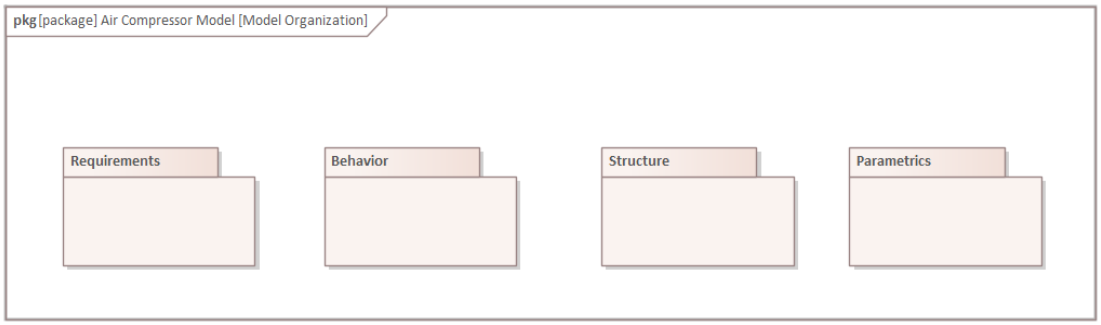

Figure 3.1: Structure of a simple model.

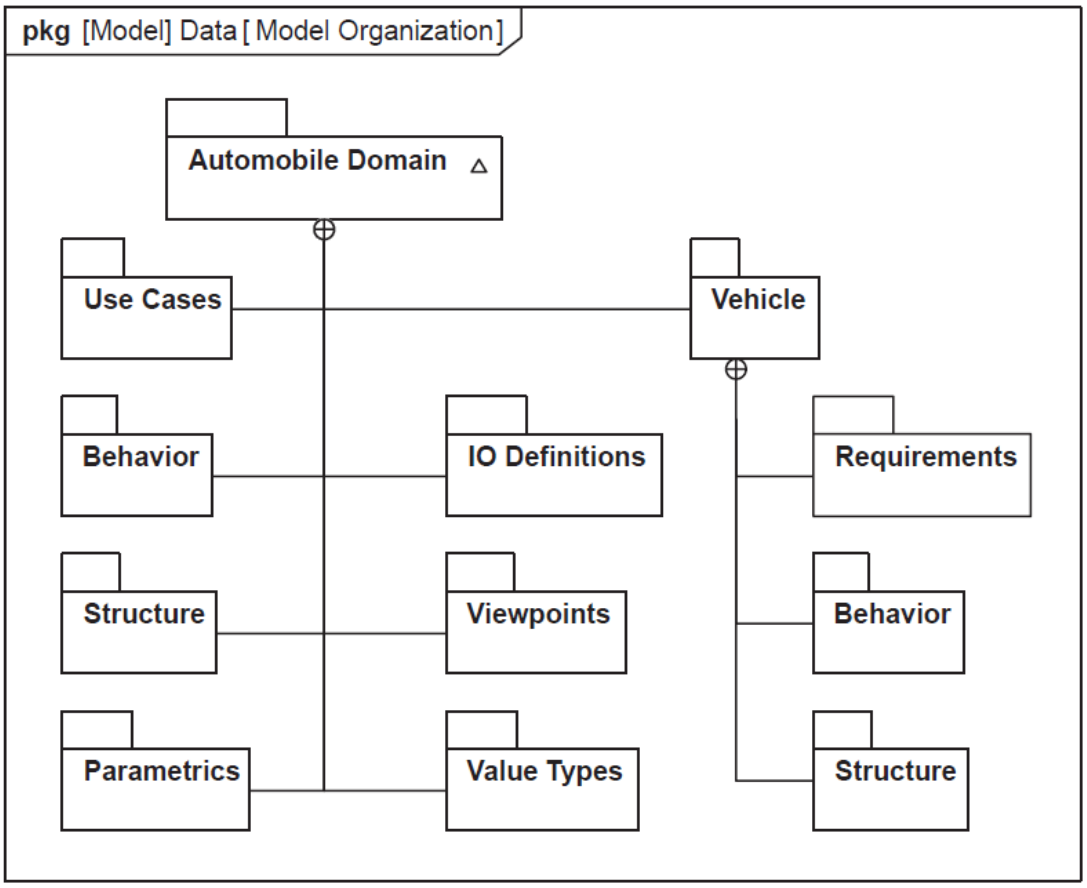

Figure 3.2: Structure of a more complex model.

However, when we try to put models like this into single-package files, we see many cross-references (`import`) among the packages. Very quickly, these imports become circular or, at the very least, strange. For example, we see that the Structure package needs to import the Behavior package (because it needs to include the performed actions, as shown in Section 3.1.2). But the Behavior package needs to talk about Air and Torque, so in order to avoid circular references, we need to split these out of the Structure package.

Therefore we propose a structure as sketched in Figure 3.3. Here we see a successive subdivision of requirements, structure, and behavior, followed by a bottom-up round of calculations, analysis, and verification. This allows developers to work independently on subsystems and components once the higher level has been defined. Of course, it is not necessary that higher levels are completely fixed, but a certain level of stability is desired.

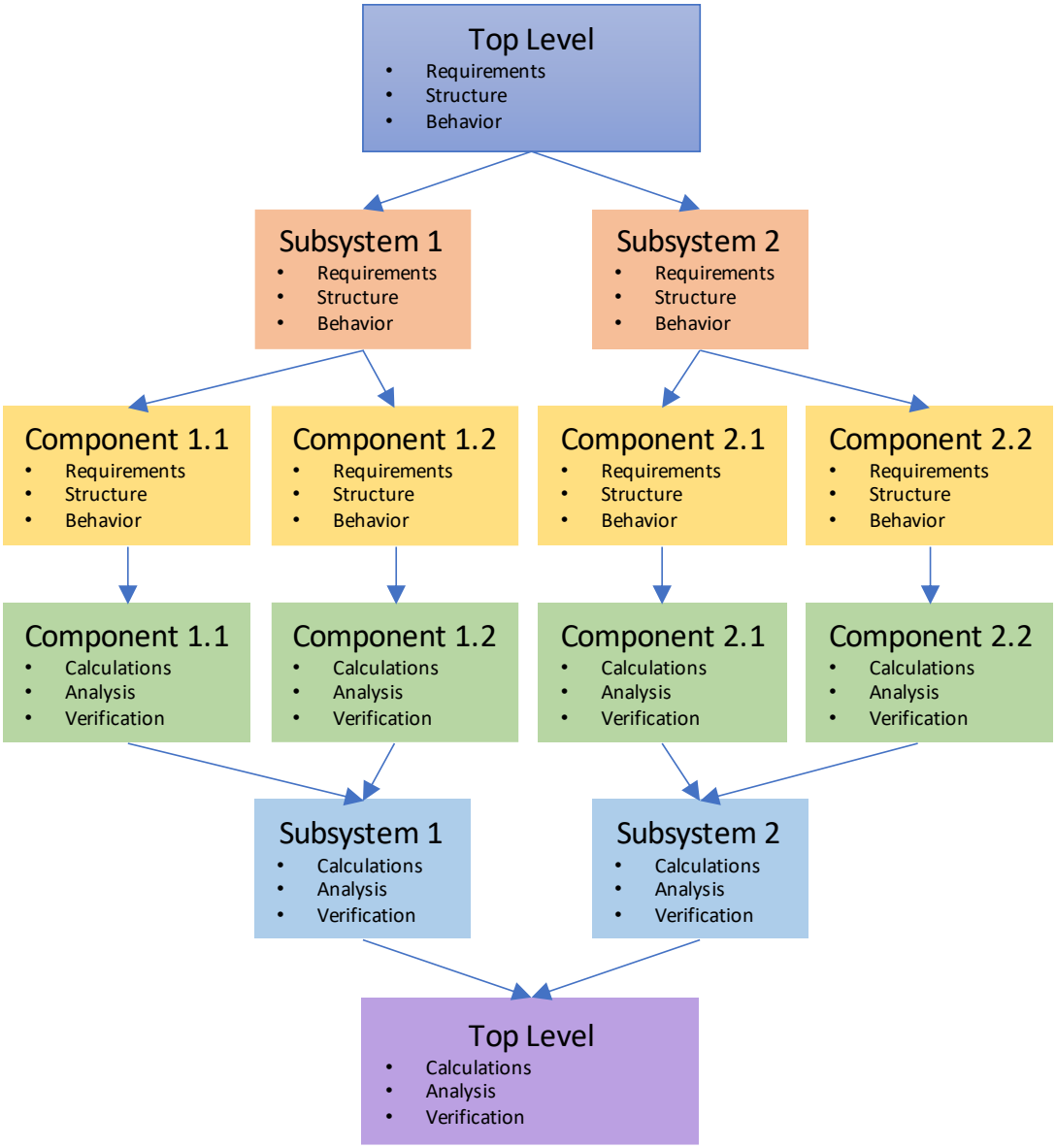

Figure 3.3: Proposed structure for larger models. Arrows indicate package dependencies.

3.2 Tooling experiences

While creating SysML v2 models, we also reflected upon the currently available tooling.

3.2.1 Eclipse

The official release of SysML v2 on GitHub [27] includes a plug-in for Eclipse that allows editing the textual representation of SysML (and KerML). Because of this provenance, we consider the Eclipse plug-in as a kind of reference implementation of SysML v2. This plug-in offers the typical features in Eclipse, such as syntax and error highlighting, project navigation, and finding declarations and references of symbols. The features are less comprehensive than the ones in Visual Studio Code (see Section 3.2.3).

The unique feature of the Eclipse plug-in is the possibility to generate **diagrams**. This works on the basis of PlantUML [28] and that means that you get a picture that is automatically generated from the code and you cannot change that. This reduces the usefulness considerably, because you cannot use a diagram to tell a specific story, but you get all the details in the source file (or a selected part of the source file). For a somewhat large file, the complete diagram does not even fit inside the picture. Therefore, the diagrams are mainly useful as a limited kind of feedback to see whether the model is expressing what you expected.

3.2.2 Jupyter and the SysML v2 API

The reference implementation of SysML v2 also contains a Jupyter plug-in. This plugin provides an interactive SysML v2 shell and Jupyter kernel to work with SysML v2 models in Jupyter notebooks. The implementation provides so-called "SysML v2 magic commands", listed in Table 3.1, which can be used to visualize (parts of) the model, inspect the abstract syntax tree, export the model, or publish the model to a model server.

Table 3.1: SysML v2 magic commands in Jupyter notebooks

Magic command	Functionality
%eval	Evaluate a given expression.
%export	Save a file of the JSON representation of the abstract syntax tree rooted in the named element.
%help	Get a list of available commands or help on a specific command
%list	List loaded library packages or the results of a given query
%show	Print the abstract syntax tree rooted in a named element
%publish	Publish to the repository the model elements rooted in a named element
%view	Render the view specified by the named view usage
%viz	Visualize the name model elements

Editing

Editing SysML v2 models in Jupyter notebook is less convenient compared to other IDEs that have better support for code completion and syntax checking. We also experienced that working with larger models is not convenient in Jupyter notebooks.

Analyzing

Jupyter notebooks provides support to visualize the SysML v2 models using PlantUML, in the same way as in the Eclipse implementation. No graphical editing support is provided. Some basic support for evaluating expressions is available (e.g., on natural and real numbers), but this does not yet cover the full language. Exporting the model to JSON works as expected.

Publishing

A benefit of the Jupyter notebooks environment is that you can interact with a model server. In the pilot implementation, models are published to the pilot server as mentioned in Section 2.2.1. We have tried publishing some simple models. These models did indeed become available on the pilot server, and could be queried with the SysML v2 API. We performed a few simple queries on the model server using the OpenAPI visualization of the API. We noted that making more complex queries would benefit from the use of viewpoints, and guidelines on how to efficiently query the model. At the moment, viewpoints are not yet implemented, and more advanced model queries were a bit slow on the server.

3.2.3 Visual Studio Code

For Visual Studio Code there is a plug-in called SysIDE CE, which makes editing textual SysML v2 projects a real pleasure.
It provides

- easy navigation with a file explorer and breadcrumbs
- extensive syntax highlighting in multiple colors
- immediate error feedback
- autocompletion
- highlighting of the same symbol in the current file, including markers in the scrollbar
- navigation to definition or references of the same symbol
- easy renaming (including other occurrences of the symbol)
- reasonable pretty printing (Format Document). It is only unfortunate how comments (/* ... */) are always spread out over multiple lines, even if they would fit on a single line.

We have not found any discrepancies in error messages between Visual Studio Code and the reference implementation in Eclipse.

We recommend to use Visual Studio Code in a folder structure that also includes the complete SysML and KerML standard libraries. In that way the editor can do complete checking of referenced definitions and it is easy to look up the definitions when desired.

In our opinion this makes Visual Studio Code the preferred tool for editing SysML v2 projects at this moment. Its only drawback is that it does not offer any diagramming facilities.

3.2.4 SysON

SysON [29] is an open source tool being developed specifically for SysML v2. It is being developed by Obeo [30], the company that is also developing the Capella tool for MBSE. SysON is still in an early phase of its development and it does not yet implement the complete SysML v2 language. Nevertheless we have tried it out extensively and we intend to follow its development.

SysON is based on web technology, more specifically on Sirius Web [31]. This means that it has a central server with a database to store the model repository. Users can access the models using their browser and the central server will retain consistency among the models that concurrent users are editing.

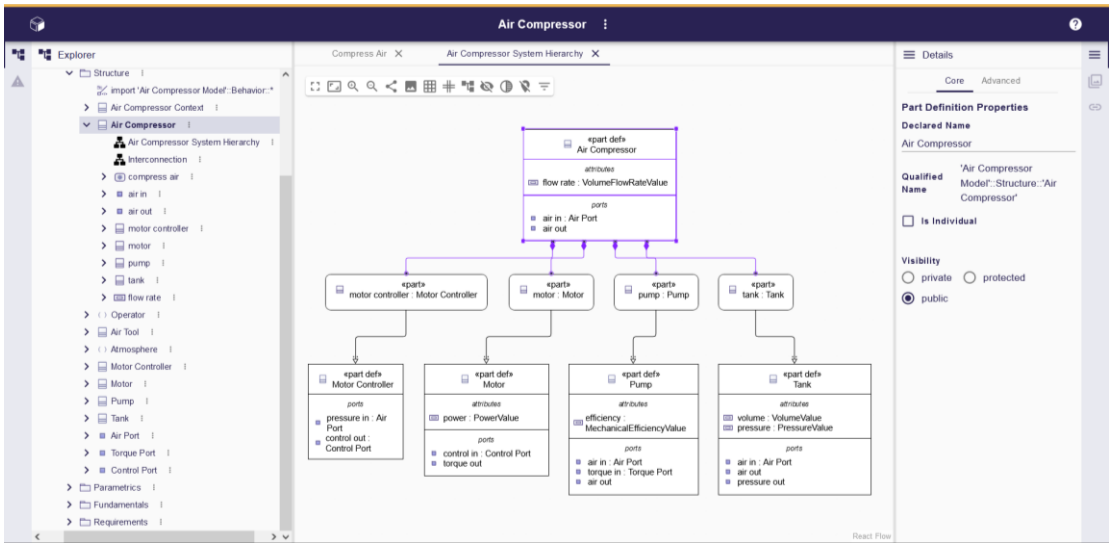

Figure 3.4: SysON screenshot.

In Figure 3.4, we see that the main way of interacting with a model is the graphical representation (the middle pane), whereas the left part of the screen allows navigation via a hierarchical structure and the right part allows editing of some properties of the currently selected item. There are significant limitations in the graphical editor, but currently it is the best way to create diagrams for SysML v2.

It is possible to upload a textual SysML v2 model to SysON, but in the current version some of the information in the model is lost, even though it could be represented in SysON. Similarly it is possible to download a model in the standard textual representation, but again some of the information is lost, not only the diagrams. It is also possible to download a complete model, including diagrams, in a SysON-specific file format (a zipped collection of JSON files) and to upload it again to a different (or the same) server, but these files cannot be readily understood and edited by a human user. Because of these issues with uploading and downloading, SysON cannot currently be used for round-trip engineering of SysML models. Therefore we hope that these issues will be fixed soon.

The roadmap shows that the SysON creators intend to have a full implementation of SysML v2 by the end of 2024. This would allow users to do general modelling work using SysON. The web-based character of SysON would then also allow a team of engineers working together on a single model. Unfortunately, the standardized SysML v2 API is not yet mentioned in the roadmap, so cooperation with other tools will not yet happen smoothly. Moreover, version management of models, as implied by the API, will also take longer to be implemented.

4 Use case: Eclipse LSAT

We have selected Eclipse LSAT as ESI tool to evaluate SysML v2, as Eclipse LSAT is used to model both structural and behavioral aspects of production systems and uses several concepts that match closely to concepts found in SysML.

4.1 Eclipse LSAT

4.1.1 Overview

Eclipse LSAT (Logistics Specification and Analysis Tool) [32] [33] is a tool for rapid design-space exploration of supervisory controllers that steer the product logistics and orchestrate the behavior in flexible manufacturing systems. LSAT enables lightweight modeling of system resources, system behavior, and timing characteristics using a set of domain-specific languages. The tool can show the system behavior related to specific product flow using a Gantt-chart visualization. Eclipse LSAT also provides analysis capabilities to compute a throughput- or makespan-optimal product flow.

4.1.2 Modeling in Eclipse LSAT

LSAT is based on four integrated domain-specific languages, shown in Figure 4.1, to model flexible production systems, describing the platform and the application.

- The **platform** refers to the structural decomposition of the system, as well as the behavior and timing characteristics at the lowest level. It is described using the **machine language** and **settings language**.
  - The **machine language** has concepts to describe the system resources, like robots and process stations, and their peripherals that can execute actions. The machine language also describes the movements between symbolic locations.
  - The **settings language** is used to describe the physical locations, the motion profiles to move between locations, and how much time actions take, either as constants or distributions.
- The **application** refers to the high-level operations and the production flow. It is described using the **activities language** and the **logistics language.**
  - The **activities language** has the concept of activities to model operations that have a set of actions and action dependencies, as well as explicit claiming and releasing of the required resources.
  - The **logistics language** provides a way to model a production flow in the form of an activity sequence. A set of production flows can also modeled, using a network of automata, where each state transition links to executing an activity.

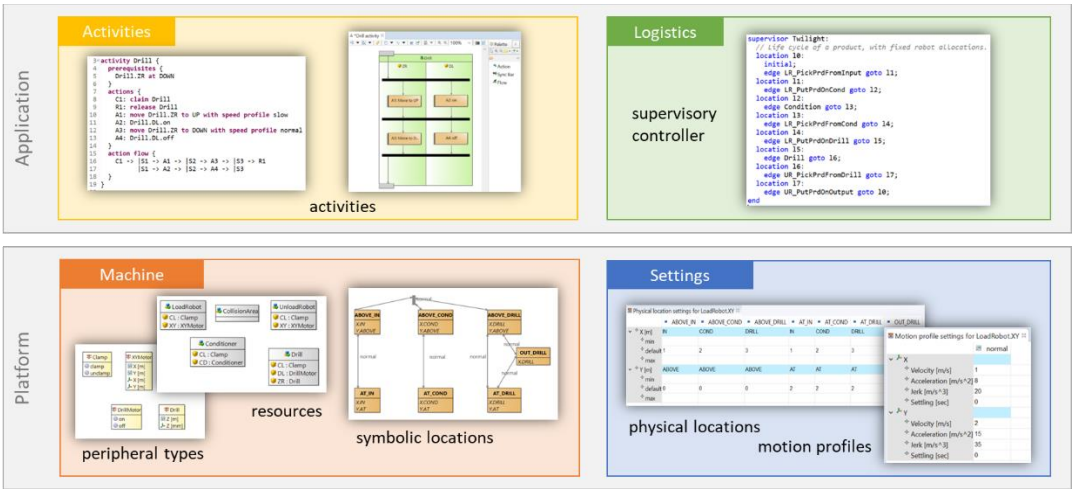

Figure 4.1: Languages in Eclipse LSAT.

Eclipse LSAT provides both graphical and textual editors to help with understanding and developing the model.

Model validations
Eclipse LSAT has many model validations to help the modeler create correct models [33], showing a warning or error at the specific location in the model when there is a problem. There are syntax validations to check references to model elements like actions, peripherals, and resources. This helps the user to prevent issues caused by typographical errors. LSAT also has domain validations, to ensure that no invalid motion profiles or destinations are specified. Each activity is checked for a proper structure, adherence to the resource claiming rules, and being acyclic. Analysis validations check completeness of the model, for example making sure that no physical settings are missing.

4.1.3 Performance analysis in Eclipse LSAT

Eclipse LSAT provides performance analysis to analyze the timing behavior of a model. For a single activity sequence, a Gantt chart can be computed that shows which actions are executed at what time, and what is the total makespan of the activity sequence. As mentioned before, the timing of actions is either explicitly modelled, or in case of motion actions, derived from a movement specification between two locations.

When the logistics is specified as a network of automata, then both the minimum and maximum performance over all described sequences can be computed. In case of finite sequences, the computed performance is a makespan value. In case of infinite sequences, the computed performance is a throughput value.

4.2 Research questions

In the study, we investigated the following questions to evaluate the suitability of SysML v2 as foundation for Eclipse LSAT.

1. What **language extension points** are available to add domain-specific concepts?
2. Can we **restrict the SysML v2 language**? For example excluding the concept of choice from activity diagrams?
3. Can we add **static and/or dynamic model validations** to the extended language?

4. How can we **integrate analysis capabilities** to the extended language? For example, can we generate a Gantt chart that can be visualized using Eclipse TRACE4CPS? How much effort is required compared to an approach with Xtend?
5. Can we **mimic the LSAT languages** based on SysML v2? How much can we re-use of the SysML v2 core?

The first four questions are largely addressed in Section 2. In the next sections, we will focus mostly on the last question, and evaluate the full set of questions in Section 4.6.

4.3 Expressing LSAT concepts with SysML v2

To evaluate SysML v2 as foundation for Eclipse LSAT, we have considered both the language concepts, as well as how a concrete model would look like.

4.3.1 Use case

As case study, we used the bowling ball production system. In this production system, balls available at the input buffer are turned into bowling balls by heating the balls at the conditioner (COND), and subsequently drilling three gripping holes in the balls at the drill station (DRILL). Finished bowling balls are put in the output buffer. Transportation of the balls in the system is done by two robots (the load robot (LR) and unload robot (UR)) that move along a shared rail. Figure 4.2 graphically illustrates the modelled system. Peripheral movements are shown with arrows, and atomic peripheral actions are shown with rounded rectangles.

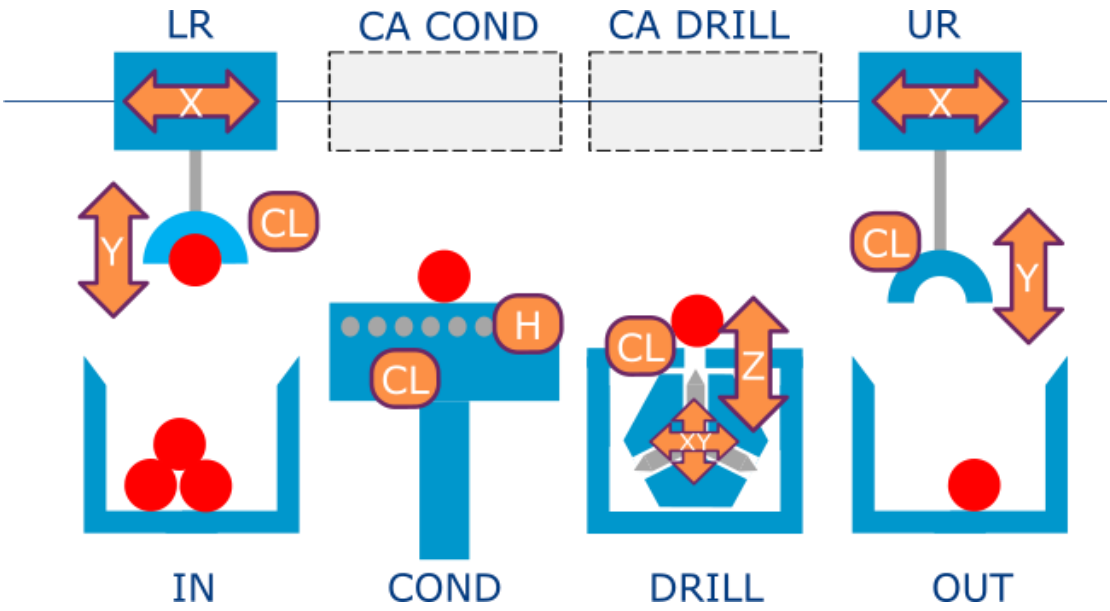

Figure 4.2: Bowling balls production system.

4.3.2 SysML v2 libraries for Eclipse LSAT generic concepts

As discussed in Section 2.4, it is preferred to extend SysML v2 using the library mechanism. We have created two libraries to capture core elements of the platform. The first library is the machine library, describing the machine language concepts like resource, peripheral, machine, but also the means to describe movements in terms of symbolic locations, axes, and paths between symbolic positions. This library is shown in Figure 4.3.

```
library package LSATMachineLibrary {
    doc /* LSAT machine library describing the machine specification concepts. */

    import ISQSpaceTime::TimeValue;
    import MeasurementReferences::MeasurementUnit;
    import ISQBase::LengthValue;

    action def TimedAction {
        attribute time : TimeValue;
    }

    part def Peripheral;
    part peripheraltypes : Peripheral[*];
    part def MoveablePeripheral :> Peripheral {
        attribute axes : Axis[*];
        attribute positions : SymbolicPosition[0..*] ordered;
        attribute paths : Path[0..*] ordered;
        attribute motionProfiles : MotionProfile[0..*] ordered;
    }

    part def Resource {
        doc /* A resource is a logical unit, containing a set of peripherals,
               that can be claimed and released. */
        part peripherals : Peripheral[*];

        action def claim :> TimedAction {
            assign time := 0;
        }
        action def release :> TimedAction {
            assign time := 0;
        }
    }

    part def Machine {
        part resources : Resource[*];
    }

    attribute def AxisPosition {
        doc /* An axis position is a position on a single axis. */
        attribute pos : LengthValue;
    }
    attribute def XAxisPosition :> AxisPosition;
    attribute def YAxisPosition :> AxisPosition;

    attribute def Axis {
        doc /* Axis positions are defined on an axis, but are specific to a peripheral. */
        attribute movementUnit : MeasurementUnit;
    }

    attribute def SymbolicPosition {
        doc /* A symbolic position is defined in terms of a single axis position
               on each involved axis. */
    }

    abstract attribute def XYPosition :> SymbolicPosition {
        doc /* An XYPosition is composed of an X-axis position and Y-axis position. */
        attribute x : XAxisPosition;
        attribute y : YAxisPosition;
    }

    abstract attribute def MotionProfile;

    attribute def Path {
        doc /* A path describes a movement from a source to target position according
               to a motion profile. */
        attribute source : SymbolicPosition;
        attribute target : SymbolicPosition;
        attribute motionProfile : MotionProfile;
    }
}
```

Figure 4.3: SysML v2 library describing the Eclipse LSAT machine concepts.

We feel that all concepts of the machine language can be described intuitively with SysML v2. One disadvantage of SysML v2 is that it is not possible to add [34] or modify features after declaring a part in the model. This means that we cannot add or modify the timing information of the claim and release actions at another place in the model after declaring them. In Eclipse LSAT, the machine language can be fully separated from expressing timing information. Note that we included the claim and release actions here, instead of specifying them in a separate hierarchy. We have chosen this way of modeling, such that in the activities it is immediately clear which actions are provided by which resources and peripherals. More details on this are described in Section 4.3.3.

We have created a second SysML v2 library describing motion profiles that are part of the setting language, shown in Figure 4.4. Linking symbolic positions to physical locations is not done in the library, in contrast to in Eclipse LSAT. Instead, the symbolic positions are assigned with the concrete location values. We have chosen for this solution, as the physical locations are nothing more than assigning concrete values to the parameters of a symbolic location.

Next to the motion profiles, the library also includes calculations to compute the timing for a movement based on two physical locations and a third-order motion profile. In Eclipse LSAT, a default motion calculator is provided that provides the same capability. For more complex computations, currently a Matlab interface is available in Eclipse LSAT. Using SysML v2, we feel that it becomes easier to model motion calculations, and also add extensions to represent settling time of movements, or software overhead. The benefit of modeling the motion calculations directly in SysML v2 is that they can then also be computed while simulating the SysML v2 model, providing the required information for a timed simulation.

```
library package LSATSettingsMotionLibrary {
    doc /* Defines how the timing of movements is computed based on a
           VAJ motion profile and two locations. */

    import ComplexFunctions::abs;
    import ScalarValues::Real;
    import RealFunctions::*;
    import LSATMachineLibrary::SymbolicPosition;
    import LSATMachineLibrary::MotionProfile;
    import LSATMachineLibrary::XYPosition;

    abstract attribute def MotionProfile1D :> MotionProfile;
    abstract attribute def MotionProfile2D :> MotionProfile {
        attribute x : VAJ;
        attribute y : VAJ;
    }

    attribute def VAJ :> MotionProfile1D {
        attribute V : Real;
        attribute A : Real;
        attribute J : Real;
    }

    private calc def ComputeTiming {
        doc /* Compute the required timing based on the distance to travel and the
               VAJ motion profile. Computation is based on the pseudo code in
               Section 3.6 of "Characterizing Performance Variability in Manufacturing
               System Configurations" by Izgi Ozbay
               (https://pure.tue.nl/ws/portalfiles/portal/283013643/Ozbay_I.pdf)
            */
        in distance : Real;
        in vMax : Real;
        in aMax : Real;
        in jMax : Real;
        return : Real =
            if (aMax - sqrt(vMax * jMax) < 0) ?
                if (distance - ((vMax^2)/aMax) - ((vMax*aMax)/jMax) > 0) ?
                    (aMax/jMax) + (vMax/aMax) + (distance/vMax)
                else if (distance - ((2*aMax^3)/jMax^2) > 0) ?
                    (aMax/jMax) + sqrt((aMax^2/jMax^2)+(4*distance/aMax))
                else
                    4 * (distance/2*jMax)^(1/3)
            else
                if (distance - 2 * sqrt((vMax^3)/jMax) < 0) ?
                    4 * (distance/2*jMax)^(1/3)
                else
                    2 * sqrt(vMax/jMax) + (distance/vMax);
    }

    calc def ComputeTimingForMovement {
        doc /* Compute the distance */

        in startPosition : XYPosition;
        in endPosition : XYPosition;
        in profile : MotionProfile2D;

        return : Real = max(
                ComputeTiming(abs(startPosition.x.pos,endPosition.x.pos),
                  profile.x.V, profile.x.A, profile.x.J),
                ComputeTiming(abs(startPosition.y.pos,endPosition.y.pos),
                  profile.y.V, profile.y.A, profile.y.J)
        );
    }
}
```

Figure 4.4: SysML v2 library describing the motion profiles as part of the Eclipse LSAT settings language, and motion timing computations.

4.3.3 Modeling the production system with SysML v2

4.3.3.1 Modeling the platform

We have modelled the bowling ball production system in the same way as in Eclipse LSAT, by first modelling the structural components of the system; the peripherals and resources, together with their movement axes. We start by modeling the peripheral types in SysML v2 in the same way as in Eclipse LSAT. Figure 4.5 shows the Eclipse LSAT model fragment and Figure 4.6 the SysML v2 model fragment.

```
Machine Bowling

PeripheralType Clamp {
  Actions { clamp unclamp }
}
PeripheralType Conditioner {
  Actions { condition }
}
PeripheralType XYMotor {
  SetPoints {
    X [m]
    Y [m]
  }
  Axes {
    X [m] moves X
    Y [m] moves Y
  }
}
PeripheralType Drill {
  Actions { on off }
  SetPoints { Z [m] }
  Axes { Z [mm] moves Z }
  Conversion "Z=Z/1000"
}
PeripheralType Carousel {
  SetPoints { Theta [degrees] }
  Axes { Theta [degrees] moves Theta }
}
```

Figure 4.5: First part of the bowling ball production system in Eclipse LSAT, describing the peripherals with their movement axes and actions they provide.

As we want to model explicitly that certain actions are provided by a specific resource or peripheral, we need to directly assign the timing in this part of the SysML v2 model. An advantage of SysML v2 is that you can use the International System of Units (abbreviated to SI) to assign well-defined timing units like seconds or milliseconds. In contrast, in Eclipse LSAT, the timing values are dimensionless. Similarly, movement units can be specified in terms of SI-units.

```
package BowlingMachineSpec {
    doc /* This package contains the model of the bowling ball production system. */

    import ISQSpaceTime::TimeValue;
    import LSATMachineLibrary::*;
    import LSATSettingsMotionLibrary::*;

    // Peripheral types being used in the bowling balls production system.
    part def Clamp :> Peripheral {
        doc /* Clamp that can clamp or unclamp to hold or release a bowling ball. */
        action def clamp :> TimedAction {
            assign time := 0.250 [SI::second];
        }
        action def unclamp :> TimedAction {
            assign time := 0.200 [SI::second];
        }
    }
    part def Conditioner :> Peripheral {
        doc /* Conditioner that can condition a bowling ball to the
               right temperature for drilling. */
        action def condition :> TimedAction {
            assign time := 5.0 [SI::second];
        }
    }
    part def XYMotor :> MoveablePeripheral {
        doc /* Motor peripheral that can move over an X and Y axis. */
        attribute xAxis :> axes {
            :>> movementUnit = mm;
        }
        attribute yAxis :> axes {
            :>> movementUnit = mm;
        }
    }
    part def Drill :> MoveablePeripheral {
        doc /* Drill peripheral that can be turned on/off and move along the Z axis. */
        action def on;
        action def off;
        attribute zAxis :> axes {
            :>> movementUnit = mm;
        }
    }
    part def Carousel :> MoveablePeripheral {
        doc /* Carousel peripheral that can rotate on the theta axis.*/
        attribute thetaAxis :> axes {
            :>> movementUnit = degree;
        }
    }
```

Figure 4.6: First part of the bowling ball production system in SysML v2, describing the peripherals with their movement axes and actions they provide.

After modeling the peripherals, we describe the parameters that describe the physical locations of the system, shown in Figure 4.8. Note that this is almost identical to how you describe these locations in LSAT using expressions, shown in Figure 4.7. The only difference is that in SysML v2 you also have the capability to add a dimension after the expression, e.g., `[SI::m]`, to denote that the unit is meters.

```
val RailLength = 4
val RailHeight = 2

val InXPosition = 0.25 * RailLength
val ConditionerXPosition = InXPosition + 0.25 * RailLength
val DrillXPosition = ConditionerXPosition + 0.25 * RailLength
val OutXPosition = RailLength
val CollisionAreaConditioner = 0.4 //From center
val CollisionAreaDrill = 0.4 //From center

val HomeYPosition = 0
val OutDrillYPosition = 0.4 * RailHeight
val AtYPosition = RailHeight

val BallWeight = 6
```

Figure 4.7: Parameters in Eclipse LSAT that determine the physical locations in the bowling ball production system, described by expressions.

```
// Machine description of the bowling balls production system.
part bowlingBallProductionSystem : Machine {
    // Parameters that determine the physical locations of the system.
    attribute RailLength = 4 [SI::m];
    attribute RailHeight = 2 [SI::m];

    attribute InXPosition = 0.25 * RailLength [SI::m];
    attribute ConditionerXPosition = InXPosition + 0.25 * RailLength [SI::m];
    attribute DrillXPosition = ConditionerXPosition + 0.25 * RailLength [SI::m];
    attribute OutXPosition = RailLength [SI::m];
    attribute CollisionAreaConditioner = 0.4 [SI::m]; //From center
    attribute CollisionAreaDrill = 0.4 [SI::m]; //From center

    attribute HomeYPosition = 0 [SI::m];
    attribute OutDrillYPosition = 0.4 * RailHeight [SI::m];
    attribute AtYPosition = RailHeight [SI::m];

    attribute BallWeight = 6 [SI::kg];
```

Figure 4.8: Parameters in SysML v2 that determine the physical locations in the bowling ball production system, described by expressions.

The next part of the specification is to model the different resources. The load robot specification in Eclipse LSAT is shown in Figure 4.9, and the other resources in Figure 4.11. Figure 4.10 and Figure 4.12 show the corresponding SysML v2 specification. Each resource contains the peripheral instances attached to it, the physical locations for these peripherals, and the motion profiles to move between these locations. The SysML v2 model also keeps track of the current position for each moveable peripheral, such that a check could be performed whether the end position of a move is also the start position of the next move. Eclipse LSAT performs such checks once a production sequence is modelled, and in this way SysML v2 might also be able to perform a similar model validation.

When zooming in to the differences between the SysML v2 model and the Eclipse LSAT model, we see that the next position and the motion profile are explicit input parameters when specifying a move. In Eclipse LSAT, this linking is done in the activity and not yet when specifying the platform; `A1: move LoadRobot.XY to AT_IN with speed profile normal.`

```
Resource LoadRobot {
  CL: Clamp
  XY: XYMotor {
    AxisPositions {
      X (IN, CA_COND_L, COND, CA_COND_R, CA_DRILL_L, DRILL)
      Y (ABOVE, AT)
    }
    SymbolicPositions {
      ABOVE_IN (X.IN, Y.ABOVE)
      ABOVE_COND (X.COND, Y.ABOVE)
      ABOVE_DRILL (X.DRILL, Y.ABOVE)
      ABOVE_CA_COND_L (X.CA_COND_L, Y.ABOVE)
      ABOVE_CA_COND_R (X.CA_COND_R, Y.ABOVE)
      ABOVE_CA_DRILL_L (X.CA_DRILL_L, Y.ABOVE)
      AT_IN (X.IN, Y.AT)
      AT_COND (X.COND, Y.AT)
      AT_DRILL (X.DRILL, Y.AT)
      OUT_DRILL (X.DRILL)
    }
    Profiles (normal)
    Paths {
      FullMesh { profile normal
        ABOVE_IN ABOVE_COND ABOVE_DRILL
        ABOVE_CA_COND_L ABOVE_CA_COND_R ABOVE_CA_DRILL_L
      }
      ABOVE_IN <-> AT_IN profile normal
      ABOVE_COND <-> AT_COND profile normal
      ABOVE_DRILL <-> AT_DRILL profile normal
      ABOVE_DRILL <-> OUT_DRILL profile normal
      OUT_DRILL <-> AT_DRILL profile normal
    }
  }
}
```

Figure 4.9: Specification of the load robot resource in Eclipse LSAT.

```
part loadRobot :> resources {
    part cl : Clamp :> peripherals;
    part motorXY : XYMotor :> peripherals {
        // X axis positions.
        attribute IN : XAxisPosition {:>> pos = InXPosition;}
        attribute CA_COND_L : XAxisPosition {
          :>> pos = ConditionerXPosition - CollisionAreaConditioner;}
        attribute COND : XAxisPosition {:>> pos = ConditionerXPosition;}
        attribute CA_COND_R : XAxisPosition {
          :>> pos = ConditionerXPosition + CollisionAreaConditioner;}
        attribute CA_DRILL_L : XAxisPosition {
           :>> pos = DrillXPosition - CollisionAreaDrill;}
        attribute DRILL : XAxisPosition {:>> pos = DrillXPosition;}

        // Y axis positions.
        attribute ABOVE : YAxisPosition {:>> pos = HomeYPosition;}
        attribute AT : YAxisPosition  {:>> pos = AtYPosition;}
        attribute OUT : YAxisPosition  {:>> pos = OutDrillYPosition;}

        // Symbolic positions.
        attribute ABOVE_IN : XYPosition {:>> y = ABOVE; :>> x = IN;}
        attribute ABOVE_COND : XYPosition {:>> y = ABOVE; :>> x = COND;}
        attribute ABOVE_DRILL : XYPosition {:>> y = ABOVE; :>> x = DRILL;}
        attribute ABOVE_CA_COND_L : XYPosition {:>> y = ABOVE; :>> x = CA_COND_L;}
        attribute ABOVE_CA_COND_R : XYPosition {:>> y = ABOVE; :>> x = CA_COND_R;}
        attribute ABOVE_CA_DRILL_R : XYPosition {:>> y = ABOVE; :>> x = CA_DRILL_L;}
        attribute AT_IN : XYPosition {:>> y = AT; :>> x = IN;}
        attribute AT_COND : XYPosition {:>> y = AT; :>> x = COND;}
        attribute AT_DRILL : XYPosition {:>> y = AT; :>> x = DRILL;}
        attribute OUT_DRILL : XYPosition {:>> y = OUT; :>> x = DRILL;}

        // Motion profiles.
        attribute normalProfileX : VAJ {:>> V = 2;
          :>> A = 8 * AccelerationWeightFactor; :>> J = 20;}
        attribute normalProfileY : VAJ {:>> V = 2;
          :>> A = 15 * AccelerationWeightFactor; :>> J = 35;}
        attribute normalProfile : MotionProfile2D { :>> x = normalProfileX;
          :>> y = normalProfileY; }

        // Movements.
        // (note: in LSAT, you create move actions directly within an activity)
        attribute currentPosition : XYPosition = ABOVE_IN;

        action def move :> TimedAction {
            in toPos : XYPosition;
            in profile : MotionProfile2D;

            assign time := ComputeTimingForMovement(currentPosition, toPos, profile);
            // Update current XY position.
            currentPosition = toPos;
        }
        action def passingMove :> TimedAction {
            in toPos : XYPosition;
            in profile : MotionProfile2D;

            // Note: we didn't consider passing moves in the computation yet,
            // only point-to-point moves.
            assign time := ComputeTimingForMovement(currentPosition, toPos, profile);

            // Update current XY position.
            currentPosition = toPos;
        }
    }
```

Figure 4.10: Specification of the load robot resource in SysML v2.

```
Resource DrillTable {
  CL: Clamp
  XY: XYMotor {
    SymbolicPositions {
      AT_THUMB // shorthand for AT_THUMB (X.AT_THUMB, Y.AT_THUMB)
      AT_INDEX_FINGER
      AT_MIDDLE_FINGER }
    Profiles (normal)
    Paths {
      FullMesh { profile normal
        AT_THUMB AT_INDEX_FINGER AT_MIDDLE_FINGER }
    }
  }
  CA: Carousel {
    Profiles (normal)
    Distances { INDEX FULL_TURN }
  }
}
Resource Drill (Thumb, IndexFinger, MiddleFinger) {
  DR: Drill {
    SymbolicPositions { UP DOWN }
    Profiles (drill, retract)
    Paths {
      DOWN --> UP [Z] profile drill
      UP --> DOWN profile retract }
  }
}
Resource Conditioner {
  CL: Clamp
  CD: Conditioner
}
Resource UnloadRobot {
  CL: Clamp
  XY: XYMotor { .. }
}
Resource CollisionAreaAboveConditioner {}
Resource CollisionAreaAboveDrill {}
```

Figure 4.11: Specification of the other resources in Eclipse LSAT.

```
            part drillTable[1] :> resources {
                part cl : Clamp :> peripherals;
                part xy : XYMotor :> peripherals;
                // Note: symbolic positions are not yet modeled.
                part ca : Carousel :> peripherals;
                // Note: symbolic positions are not yet modeled.
            }
            part drill :> resources {
                part dr : Drill :> peripherals;
                // Note: symbolic positions are not yet modeled.
            }
            part conditioner :> resources {
                part cl : Clamp :> peripherals;
                part cd : Conditioner :> peripherals;
            }
            part unloadRobot :> resources {
                part cl : Clamp :> peripherals;
                part xy : XYMotor :> peripherals;
                // Note: symbolic positions are not yet modeled.
            }
            part collisionAreaAboveConditioner :> resources;
            part collisionAreaAboveDrill :> resources;

        }
    }
}
```

Figure 4.12: Specification of the other resources in SysML v2.

4.3.3.2 Modeling the application

Activities in Eclipse LSAT, like the one shown in Figure 4.13, can be represented in a similar way in SysML v2, shown in Figure 4.14. As explained before, we specify the action definitions that peripherals can execute as part of the peripheral definitions, in a similar way as is done in Eclipse LSAT. In the activity, we instantiate these action definitions. If needed, the same action can be instantiated multiple times. For movement actions, we use the `move` and `passingMove` action definitions, and model the target position and motion profile as input parameters.

After modeling the activities, we can model activity sequences in the same way in SysML v2, as shown in Figure 4.16. There, we model the second activity to condition a ball, and an activity that captures the start of the production process. In this way, a hierarchy of activities can be modeled. This is currently not yet supported by Eclipse LSAT. In contrast, in Eclipse LSAT only an activity dispatching sequence can be modeled as shown in Figure 4.15.

```
activity LRBallFromInToCond {
   prerequisites {
      LoadRobot.XY at ABOVE_IN
   }
   actions {
      CLR: claim LoadRobot
      RLR: release LoadRobot
      CC: claim Conditioner
      RC: release Conditioner
      CCAC: claim CollisionAreaAboveConditioner
      RCAC: release CollisionAreaAboveConditioner

      A1: move LoadRobot.XY to AT_IN with speed profile normal
      A2: LoadRobot.CL.clamp
      A3: move LoadRobot.XY to ABOVE_IN with speed profile normal
      A4a: move LoadRobot.XY passing ABOVE_CA_COND_L with speed profile normal ALAP
      A4b: move LoadRobot.XY to ABOVE_COND with speed profile normal
      A5: move LoadRobot.XY to AT_COND with speed profile normal
      A6: Conditioner.CL.clamp
      A7: LoadRobot.CL.unclamp
      A8: move LoadRobot.XY to ABOVE_COND with speed profile normal
   }
   action flow {
      CLR -> A1 -> A2 -> A3 -> A4a -> CCAC -> A4b  -> A5 -> CC -> A6 -> A7 -> RC ->
      A8 -> RLR -> RCAC
   }
}
```

Figure 4.13: Eclipse LSAT specification of an activity to move a bowling ball from the input to the conditioner.

```
package BowlingActivitiesSpec {
    import BowlingMachineSpec::*;

    // Short hand to define the production system.
    alias bowling for bowlingBallProductionSystem;

    action def LRBallFromInToCond {
        doc /* Bring a ball from the input buffer to the conditioner. */
        action CLR : bowling::loadRobot::claim;
        action RLR : bowling::loadRobot::release;
        action CC : bowling::conditioner::claim;
        action RC : bowling::conditioner::release;
        action CCAC : bowling::collisionAreaAboveConditioner::claim;
        action RCAC : bowling::collisionAreaAboveConditioner::release;

        action A1 : bowling::loadRobot::motorXY::move {
            in toPos = bowling::loadRobot::motorXY::AT_IN;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }
        action A2 : bowling::loadRobot::cl::clamp;
        action A3 : bowling::loadRobot::motorXY::move {
            in toPos = bowling::loadRobot::motorXY::ABOVE_IN;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }
        action A4a : bowling::loadRobot::motorXY::passingMove {
            in toPos = bowling::loadRobot::motorXY::ABOVE_CA_COND_L;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }
        action A4b : bowling::loadRobot::motorXY::move {
            in toPos = bowling::loadRobot::motorXY::ABOVE_COND;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }
        action A5 : bowling::loadRobot::motorXY::move {
            in toPos = bowling::loadRobot::motorXY::AT_COND;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }
        action A6 : bowling::conditioner::cl::clamp;
        action A7 : bowling::loadRobot::cl::unclamp;
        action A8 : bowling::loadRobot::motorXY::move {
            in toPos = bowling::loadRobot::motorXY::ABOVE_COND;
            in profile = bowling::loadRobot::motorXY::normalProfile;
        }

        first start then CLR;
        first CLR then A1;
        first A1 then A2;
        first A2 then A3;
        first A3 then A4a;
        first A4a then CCAC;
        first CCAC then A4b;
        first A4b then A5;
        first A5 then CC;
        first CC then A6;
        first A6 then A7;
        first A7 then RC;
        first RC then A8;
        first A8 then RLR;
        first RLR then RCAC;
        first RCAC then done;
    }
}
```

Figure 4.14: SysML v2 specification of an activity to move a bowling ball from the input to the conditioner.

```
// Activity specification
activity Condition {
   actions {
      CC: claim Conditioner
      RC: release Conditioner
      A1: Conditioner.CD.condition
   }
   action flow { CC -> A1 -> RC }
}
// Production sequence
import "bowling.activity"
activities {
    LRBallFromInToCond
    Condition
}
```

Figure 4.15: Eclipse LSAT specification of the `Condition` activity and the start of the production sequence.

```
action def Condition {
    doc /* Condition a ball at the conditioner. */
    action claim : bowling::conditioner::claim;
    action cond : bowling::conditioner::cd::condition;
    action release : bowling::conditioner::release;

    first start then claim;
    first claim then cond;
    first cond then release;
    first release then done;
}

action ProductionSequence {
    doc /* Start of the production sequence, describing that a ball is taken from the
           input buffer to the conditioner, and then conditioned. */
    action moveProductToCond : LRBallFromInToCond;
    action condition : Condition;

    first start then moveProductToCond;
    first moveProductToCond then condition;
    first condition then done;
}
```

Figure 4.16: SysML v2 specification of the `Condition` activity and the start of the production sequence.

4.4 Performance analysis

There are various ways to export the SysML v2 model to a model format for performance analysis, as explained in Section 2.2. An elegant way seems to be to first define a viewpoint for the performance analysis, that contains only the relevant data. Given such a viewpoint, the SysML v2 API can be used to extract the data and link to a performance analysis tool.

As the moment of writing, the existing tools do not yet support SysML v2 viewpoints. Next to that, at the moment there are hardly any examples available on how viewpoints are used. Given that we foresee that it will be technically feasible, we decided to not investigate the coupling to a performance analysis yet, but rather leave that as future work. Guidelines and best practices will likely be needed on how to define viewpoints, and what is the most efficient way to interact with the API.

4.5 Opportunities

SysML v2 is a much more generic modeling language compared to Eclipse LSAT. We have identified several possible extensions where Eclipse LSAT might benefit from having SysML v2 as a foundation:

- **Variability management:** SysML v2 provides native language concepts to express a variation point at some definition in the model together with a set of concrete variants. Assertion constraints can be used to model restrictions on the choices that can be made in selecting certain combinations of variants. This mechanism could be used add variability management to Eclipse LSAT, for example to model specific configurations of resources and peripherals.
- **Use cases** SysML v2 has a concept of use cases to define the usage of the system by actors. In this way, the interaction of the modeled system with its environment can also be modeled explicitly.
- **Full activity models for logistics:** Eclipse LSAT currently provides only activities and activity sequences, and does not yet allow for a hierarchy of activities or other ways of activity composition. With SysML v2, the modeling side could be easily provided. However, there will like be restrictions on the allowed activity compositions to ensure that models can be analyzed. For example, self-recursion will not be allowed, and the proper claiming and releasing of resources has to be ensured over activities. Adding such additional validations is discussed in Section 2.3.

4.6 Evaluation

Based on the generic SysML v2 evaluation and our experiences on modeling Eclipse LSAT concepts based on SysML v2, we now come back to the research questions and provide a brief answer on each.

> ***Research question 1:*** *What* ***language extension points*** *are available to add domain-specific concepts?*

As explained in Section 2.4, there are two possible ways to add domain-specific concepts; defining model elements in libraries and using semantic metadata. For Eclipse LSAT, we have used model libraries and not used semantic metadata. We did not see added benefits yet for using semantic metadata over defining a library. The use of the library mechanism proved to be sufficient for now.

> ***Research question 2:*** *Can we* ***restrict the SysML v2 language****? For example excluding the concept of choice from activity diagrams?*

As explained in Section 2.3 restricting the language might be supported in the future, but is not yet supported at the moment of writing. We have not been able to explicitly write down restrictions. Note however that restrictions can always be checked as preprocessing step in a transformation from a SysML v2 model towards an analysis model.

> ***Research question 3:*** *Can we add* ***static and/or dynamic model validations*** *to the extended language?*

Eclipse LSAT has a lot of validations to ensure that models are well-founded. For example, to check whether resources are properly claimed and released in activity, and whether motions used in an activity are defined in the machine specification. The current SysML v2 tooling helps you to create syntactically valid models. However, there is not support yet for adding additional model validations. A possible way to add these validations is by extending the IDE validator, as mentioned in Section 2.5.

Research question 4: *How can we* ***integrate analysis capabilities*** *to the extended language? For example, can we generate a Gannt chart that can be visualized using Eclipse TRACE4CPS? How much effort is required compared to an approach with Xtend?*

There are various ways to export the SysML v2 model to another model format for performance analysis. Our suggested approach would be to define a dedicated viewpoint in SysML v2 for performance analysis, containing only the relevant data. Given such a viewpoint, the SysML v2 API can be used to extract the data and link to a performance analysis tool. At the moment of writing this report, there is no support yet for viewpoints and little examples on using the API. As indicated in Section 4.4, we foresee we can answer this question as soon as commercial tool support for SysML v2 is available.

Research question 5: *Can we* ***mimic the LSAT languages*** *based on SysML v2? How much can we re-use of the SysML v2 core?*

Based on our experiences, we have found that we can mimic the LSAT languages based on SysML v2, albeit with a slightly different modeling style at some points. By basing LSAT on SysML v2, we can re-use existing modeling notations for defining the structural decomposition in terms of the system, resources, and peripherals. We can also re-use the action definitions and actions to model activities and activity sequences. An advantage of SysML v2 is that it would also already provide the notation to make the step towards modeling full logistics with guards, effects, and state machines to model and update system state.

It is important to remark that we have worked out a single mapping to represent LSAT models in SysML v2. This mapping is shown in Figure 4.17. As the SysML v2 models describe the same system structure and behavior as the LSAT models, both can be used as input for timing analysis. Note that an equivalence proof would be required to formally prove this. We have not worked out such a proof due to the amount of effort required. The figure also shows the subset of SysML v2 that contains alternative ways to represent LSAT models. As an example, consider the example given in Section 3.1.2, that shows that there are alternative ways to link structure and behavior. Multiple of these ways could serve as valid input for a timing analysis. We suggest exploring alternative ways of modeling in SysML v2 as interesting follow-up research.

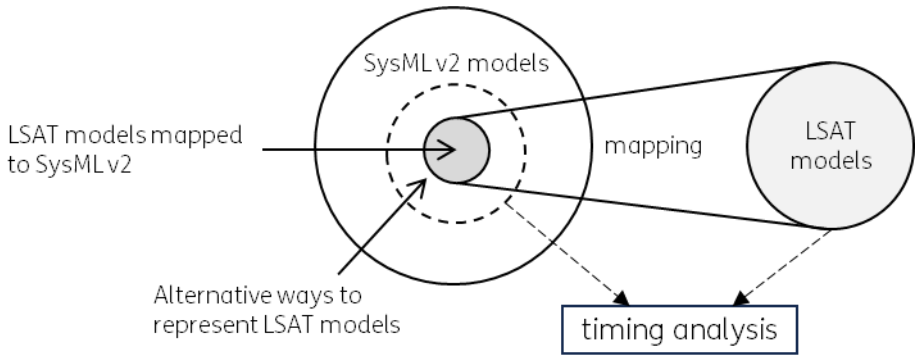

Figure 4.17: Mapping of LSAT models onto SysML v2 models.

In the longer term, we see potential to base LSAT on SysML v2, as the modeling language provides sufficient means to express the same models. However, at the moment various important capabilities are not yet supported. One of them is the availability of simulation tooling for SysML v2 models as well as documentation on the semantics. Currently it is very

difficult to assess the behavior of SysML v2 models, as the documentation is not easily readable and simulation tools are not yet available. We suggest to re-evaluate the research questions once there is more support available to restrict the language, add model validations, support for viewpoints, and support for simulating SysML v2 models.

5 Use case: Eclipse CommaSuite

5.1 Eclipse CommaSuite

The ComMA (Component Modeling and Analysis) approach is based on a domain-specific language to describe software interfaces. Each interface is described in terms of four aspects:

1. **The interface signature:** the set of commands, signals, and notifications that a server offers to its clients;
2. A **protocol state machine:** describes the interaction protocol between the client and server, i.e., the allowed sequence of interaction events including commands, signals, and notifications.
3. **Timing constraints:** describing constraints on the occurrence of events, for example lower and upper bounds on response times, or periodicity requirements.
4. **Data constraints:** describing constraints on parameters of subsequent events.

Based on the models different artifacts can be generated, including visualization, documentation, test cases, and interface monitoring. With monitoring via logging or sniffing, interface conformance can be checked, for example during nightly tests or after performing component updates. The monitoring checks whether execution traces conform to the state machine behavior of the interfaces, as well as to the timing and data constraints.

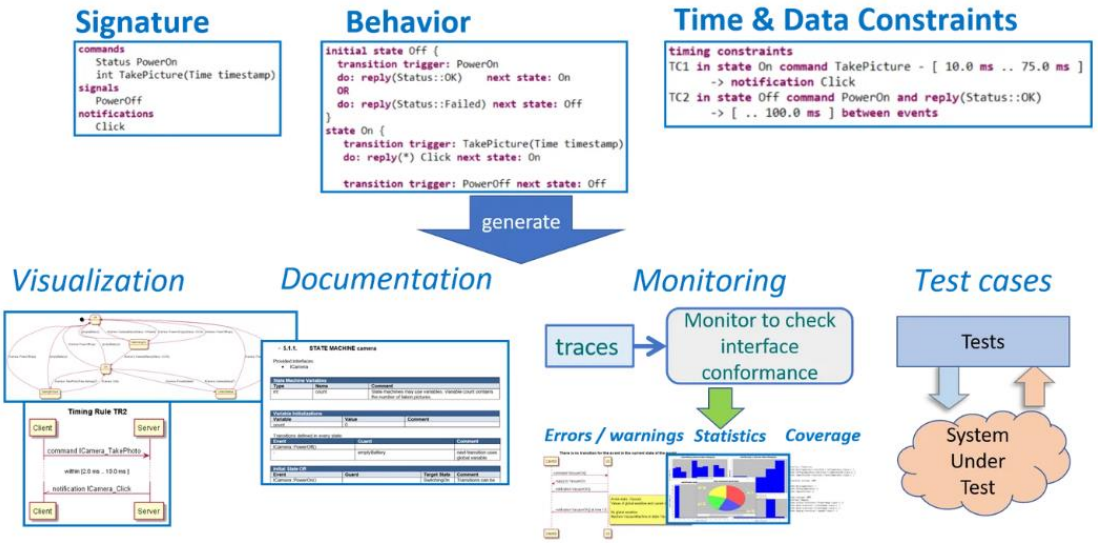

Figure 5.1: ComMA modeling and generation of artifacts, figure taken from https://esi.nl/research/output/tools/comma .

5.2 Mapping between ComMA and SysML v2

Similar to ComMA, SysML v2 also provides means to describe interfaces and state machines. An initial investigation has already been carried out that considered the question whether SysML v2 can express the same concepts as ComMA [35]. Overall, it is clear that SysML v2 provides similar concepts to the concepts used by ComMA.

Table 5.1: Mapping between ComMA concepts and SysML v2 concepts [35].

ComMA concept	SysML v2 concept	Limitations
Types		
Basic types	Empty Attributes	
Enum	Enum	1. No manual values
Map	Key Values Attribute	2. Naming conventions
Record	Attribute	
Vector	Attribute array	3. No multiple dimensions
Signature		
Command	Calculation	
Signal	Action	4. Differentiating
Notification	Action	5. Differentiating
Interfaces and components		
Component	Part def	6. Ports do not have a current state
Parts keyword	Multiple part usages	
Interface	Port def	
Provided port	Port usage	
Required port	Conjugated port usage	
Connection	Interface	

Some limitations were observed, that we re-evaluated.

Types: In [35], it was indicated that SysML v2 does not the assignment of values to elements of an enumeration (limitation 1). However, SysML v2 does provide this feature. One could write for example

```
enum def LevelEnum :> Level {
    low = 0.25;
    medium = 0.50;
    high = 0.75;
}
```

Regarding maps, it was indicated that maps in SysML v2 cannot be typed (limitation 2). However, the SysML v2 collections library provides this functionality:

```
datatype KeyValuePair {
    feature key: Anything[0..*] ordered nonunique;
    feature val: Anything[0..*] ordered nonunique;
}
datatype Map :> Collection {
    feature elements: KeyValuePair[0..*] :>> Collection::elements;
}
```

A vector in SysML v2 is one-dimensional. Supporting multiple dimensions in SysML v2, like modeling a matrix, can be done using a vector-of-vectors or alternatively as a single vector with all entries. The latter approach is used in the quantities and units library of SysML v2. We agree with [35] that there is no native support for modeling multi-dimensional vectors (limitation 3).

Signatures: In [35], signals are mapped onto an action. In the SysML v2 examples (e.g., in example *22. State Definitions*), it is advised to use attributes instead to model signals, as they do not have behavior themselves. We agree that in SysML v2 there is no dedicated concept for signals (limitation 4). Commands are translated to calculations in [35]. We suggest to use actions instead, that also have input and output parameters.

Interfaces and components: In [35], it is observed that in SysML v2 ports do not have a current state, whereas in ComMA this is possible. Such a state could be used in guard conditions for transition, so that certain transitions or triggers would only be allowed when a specific port is in a specific state.

5.3 Discussion

In ESI, we had a discussion with the current tool owner of CommaSuite on what are important considerations in the context of connecting or integrating ComMA with SysML v2.

- **SysML v2 API:** The SysML v2 API is seen as a very strong benefit compared to SysML v1. An envisioned future is where ComMA models serve as intermediate format, and are hidden from the end user. Instead, ComMA would directly interface as a plugin with SysML v2 models via the API.
- **Graphical and textual modeling:** SysML v2 provides both graphical and textual modeling. This is an advantage compared to ComMA, where only textual modeling is supported. There is a wish to have both input formats, as there have already been initiatives to connect graphical models to ComMA.
- **Commercial tool support:** industrial-grade tooling with commercial support is key for industrial adoption. Currently, no commercial tools are available that support SysML v2, but it is expected that these will become available in the near future.
- **Viewpoint for interface:** ComMA targets interface modeling and analysis. For users, it would be valuable if they only have to look at interfaces, and can abstract from other aspects that are modeled. A dedicated viewpoint for interfaces would be needed. Native support for viewpoints is part of the SysML v2 specification, but is not yet implemented in the current tooling.
- **Abstraction level:** SysML v2 targets system engineers as key users of the language. System-level models are typically more abstract compared to the level considered with ComMA, that focuses mostly on software interfaces. The ESI tool owner however indicated that ComMA can also be used to describe aspects of hardware interfaces, and in that light also connects to SysML v2 interface modeling. Note however that ComMA at the moment of writing does not yet support the modeling of continuous interfaces.
- **User workflow:** an important consideration whether to integrate with SysML v2, or other methodologies like Capella or CocoTech are what is predominantly used in the existing workflow of target users. An important question is whether it fits with other languages, tools, and methods used in the workflow of the company. Also, whether it is feasible to integrate with the workflow, e.g., with a new implementation or by linking a plugin. In light of the integration, it is also interesting to consider other standards next to SysML v2 being developed, like the Language Server Protocol that has a lot of momentum.
- **Semantics:** Well-defined, unambiguous semantics is key for formal analysis. Therefore, this is one of the key aspects to consider in connecting to SysML v2. SysML v2 is claimed to have a formal semantics, but accessible documentation on this aspect is not available yet. Various questions need to be answered to conclude whether the semantics can be preserved in the analysis:

  - How to define the semantics of SysML v2 models where only a subset of the modeling concepts is used?
  - If only a slice of the model is analyzed, does the property then also hold for the full model?
  - What is the semantics of concepts like run-to-completion, multi-threading, queueing, and asynchronous/synchronous communication in SysML v2?

  Next to these questions, one has to be aware that there is a risk for "abuse" of the SysML v2 language. This might occur if users do not fully understand the semantics and/or certain details of the semantics.
- **Model validation:** ComMA provides specific analyses, and it has validation rules on the model to ensure that the models are semantically valid before starting an analysis. An important consideration is whether user-defined validation rules can be added to SysML v2 models. This aspect has been addressed in Section 2.5
- **Restricting the language:** ComMA has specific restrictions in the language, to enable formal analysis of certain properties. To link such analyses to SysML v2 interface models, it is an important consideration whether the SysML v2 language can be restricted. This aspect has been addressed in Section 2.3.
- **Availability of documentation:** the student that evaluated SysML v2 for ComMA had a lot of difficulty getting into SysML v2, knowing what is possible and not possible. We see that documentation already has improved with clear examples on modeling in both the textual and graphical notation. With the availability of more realistic use cases, it becomes easier to see what SysML v2 can offer, and how to model certain types of systems.

6 Use case: Eclipse POOSL

6.1 Eclipse POOSL

POOSL [36] is an object-oriented modeling language, which is tightly integrated with the Rotalumis simulating engine. POOSL originated from the same time as when UML was being introduced, and as such shared various concepts. In a nutshell, POOSL provides language concepts to model process objects and data objects.

- **Process objects**
  - **Processes** that contain ports, messages, internal variables, and methods
  - **Clusters** that are an abstraction of processes and other clusters, containing ports, process instances, and channels.
  - **System** specification that contains process instances and channels.
- **Data objects**
  - Data objects contains variables and methods to create, read, and update these variables.

6.2 Mapping between POOSL and SysML v2

POOSL shares a lot of commonality with UML. As SysML v2 is close to SysML v1, which in turn is defined in terms of UML, we first discuss POOSL in the light of UML. Later, we come back to the link between POOSL and SysML v2.

POOSL and UML both find its origins in the mid-nineties, an were created around the same time. Both languages are influenced by earlier object-oriented modeling languages, as is well-illustrated in Figure 6.1. Not surprisingly, the languages therefore have many similarities, with POOSL linking to a subset of all UML modeling concepts.

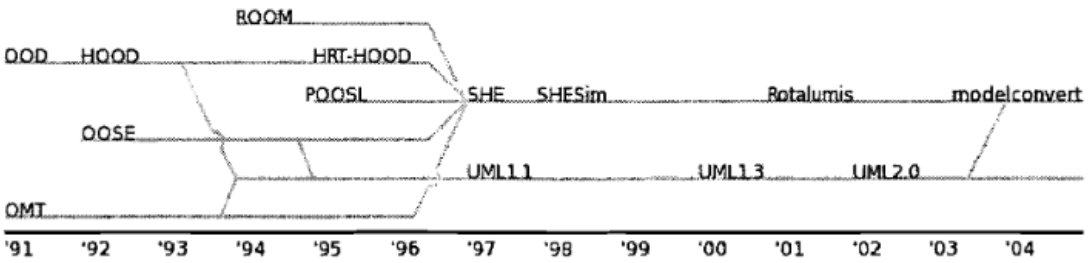

Figure 6.1: Time line of object-oriented modeling languages. Figure taken from [37].

Friedrichs [37] worked out a mapping between UML-class diagrams and state machines and POOSL. A schematic view of the mapping is shown in Figure 6.2.

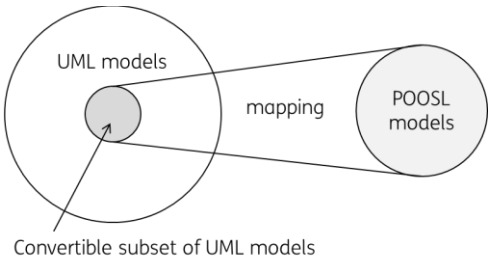

Figure 6.2: Schematic view of the UML to POOSL mapping, adapted from Fig. 1 in [37].

Mapping the static model elements of POOSL to UML is straightforward. For POOSL classes, corresponding classes in UML can be created. Stereotypes are added to indicate whether the classes are process, data, or cluster classes. Also translating variables, methods, and interfaces is straightforward.

The more challenging part is correctly transforming the behavioral elements of POOSL. For example, POOSL different types of transitions between states, including timing delays, aborts, and interrupts. Both POOSL and UML have support for aborts and interrupts, but [37] indicates that it is not clear whether the semantics of the concepts in both languages have the same semantics. The translation does not cover the mapping between POOSL statements and UML.

6.3 Discussion

In ESI, we had a discussion with the current tool owner of POOSL. He also identified the similarities between POOSL and UML/SysML, and noted that POOSL might be fully based on UML/SysML. We agree, with the important remark that a correct mapping between POOSL and UML/SysML that preserves the semantics might be challenging.

The maintainers of POOSL, Obeo, see a coupling between POOSL and SysML v2 as a desirable future. SysML v2 is expected to have a large audience, and could become the integration point for various other tools and methods, including POOSL. The integration could be done using the SysML v2 standard REST API, to ensure compatibility with SysML v2 tools. POOSL based on SysML v2 is also considered as a possibility, but this would be a big shift in the front-end used to specify models. To draw conclusions about the benefits and drawbacks of such a step, it is suggested by the maintainers to perform a study on what is missing in SysML v2 to support what POOSL is already doing.

7 Proposed ESI vision

7.1 Evaluation

Based on our experiences described in the previous chapters, we will now address the main question of this report:

Does SysML v2 provide the right capabilities to be used as a foundation for ESI tools?

In general, we see SysML v2 as very promising. SysML v2 has substantial improvements over SysML v1. The language is much more intuitive and better structured, there is both a textual and graphical modeling interface, and there is support for model management and model access via the API.

Based on the evaluation for Eclipse LSAT, Eclipse ComMA, and Eclipse POOSL, we conclude that many of the modeling concepts found in ESI tools are also present with similar/related concepts in SysML v2. Considering the extensibility of SysML v2, we think that SysML v2 could in the future serve as a front-end for ESI tools, and potentially also as foundation to base the languages on that are now part of ESI tools.

However, we do not advise the move towards SysML v2 at this moment in time, given the following aspects that should be addressed first:

- Commercial tool support is not yet available, and is key for industrial adoption;
- There is not yet any support on how to restrict the SysML v2 language;
- There is no documentation yet on how to add model validations, and what is the advised way to do this;
- There should be an assessment on whether the semantics are well-defined and unambiguous. Accessible documentation on the semantics is currently lacking. Furthermore, there is no support yet for simulating SysML v2 models. More work is needed to assess whether the semantics of SysML v2 is formal enough for analyses found in ESI tools. Concrete follow-up questions:
  - How to define the semantics of SysML v2 models where only a subset of the modeling concepts is used?
  - If only a slice of the model is analyzed, does the property then also hold for the full model?
  - What is the semantics of concepts like run-to-completion, multi-threading, queueing, and asynchronous/synchronous communication in SysML v2?
- There is no support yet for graphical editing of models. As this is a key aspect in for example Eclipse LSAT, and to some extent also in Eclipse POOSL, we see this also as prerequisite.

As soon as progress is made in multiple of these aspects, then an evaluation should be performed, to assess these issues are sufficiently resolved.

7.2 Vision

Given our experiences and based on the discussions with various ESI tool owners, our vision is that **SysML v2 should be considered as foundation for ESI tools** once commercial tool

support is available, and the semantics of SysML v2 models is deemed close enough to the semantics of languages in ESI tools. A strategy with concrete steps to move towards this vision is described]in the next section.

7.3 Strategy

We advise the following steps to move towards using SysML v2 as a foundation for ESI tools:

- Stay up to date on development regarding SysML v2, especially on the availability of commercial tooling and demonstrations of industrial use cases.
- Become a member of the OMG and join the working group(s) around SysMLv2 to stay aware of the latest developments and influence the direction with input from the high-tech equipment industry.
- Strengthen our connections with the main MBSE-tool providers to get insights in their SysMLv2 status and roadmaps, and discuss integration of our tools in their SysMLv2 suites.
- Evaluate commercial tooling, once these tools become available. Next to evaluating usability, these tools should be evaluated on more specific aspects like support for viewpoints, simulation capabilities, support to add model restrictions or model validations, and integration with the SysML v2 API.
- Analyze the formal semantics of SysML v2, answering the questions mentioned in the previous section.
- Perform multiple follow-up case studies for ESI tools. Candidates include Eclipse ComMA and Eclipse LSAT. These case studies should focus on the following aspects:
  - Create an automatic transformation from SysML v2 model to an ESI tool model to see if all concepts are transformed and properly translated. Investigate how to identify the subset of SysML v2 that can be supported in such an automated transformation.
  - Add model restrictions and validations to guarantee that the SysML v2 models are transformed to valid ESI tool models.
- Consider the use of SysMLv2 in running projects. Based on these learnings, execute a study to combine the learnings of these projects and develop a common MBSE methodology at ESI. In this study, the value of SysML v2 should be evaluated as language for systems modeling. ESI tools should be complementary to such an approach.

7.4 Recommendation: an ESI MBSE methodology vision

We see a lot of value in working out a dedicated vision on an MBSE methodology for the Dutch high-tech equipment industry. Companies are facing many similar challenges in how to model system designs, model specific aspects, and reason on those aspects. They could benefit from solutions on how to use model-based methodologies that are tailored to the characteristics of high-tech systems. In this context, we advise to assess the added benefit of adopting SysML v2 as default modeling language. This makes it easier to enable synergies across the ecosystem, re-using best practices, guidelines, models, and analysis tools. Furthermore, libraries of SysML v2 can be used with standardized interfaces and components. The ESI tools should be part of such an overarching MBSE methodology, complementing the overall systems modeling with analysis capabilities for specific system aspects.

8 Bibliography

[1] OMG, “Introduction to the SysML v2 Language Textual Notation,” 07 2024. [Online]. Available: https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/Intro%20to%20the%20SysML%20v2%20Language-Textual%20Notation.pdf. [Accessed 23 08 2024].

[2] OMG, “Introduction to the SysML v2 Language Graphical Notation,” 07 03 2023. [Online]. Available: https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/Intro%20to%20the%20SysML%20v2%20Language-Graphical%20Notation.pdf. [Accessed 23 08 2024].

[3] INCOSE, “INCOSE SysML v1 to SysML v2 Transition Guidance Activity Team,” OMG, 08 2024. [Online]. Available: https://www.omgwiki.org/MBSE/doku.php?id=mbse:sysml_v2_transition.

[4] OMG, “OMG Systems Modeling Language Part 2: SysML v1 to SysML v2 Transformation,” 07 2024. [Online]. Available: https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/2b-SysML_v1_to_v2_Transformation.pdf.

[5] K. L. Lucas, T. C. Ford, J. L. Stern and J. X. Situ, “PySysML2: Building Knowledge from Models with SysML v2 and Python,” in *The Proceedings of the 2023 Conference on Systems Engineering Research*, 2024.

[6] OMG, “Systems Modeling Application Programming Interface (API) and Services,” OMG, 2024.

[7] Intercax, “Systems Modeling API and Services (public),” Intercax, 2024. [Online]. Available: http://sysml2.intercax.com:9000/. [Accessed 23 08 2024].

[8] Intercax, “Systems Modeling API and Services (SST),” Intercax, 2024. [Online]. Available: http://sysml2-sst.intercax.com:9000/docs/. [Accessed 23 08 2024].

[9] OASIS, “Open Services for Lifecycle Collaboration | OSLC,” 2024. [Online]. Available: https://open-services.net/. [Accessed 23 08 2024].

[10] GitHub, “SysML OSLC Server,” 2024. [Online]. Available: https://github.com/oslc-op/sysml-oslc-server. [Accessed 23 08 2024].

[11] PTC - Systems and Software Engineering, “Windchill Modeler 10.0 and SysML 2.0 (Youtube video),” 2024. [Online]. Available: https://youtu.be/5-VG9TsywVU?t=5583. [Accessed 23 08 2024].

[12] E. Seidewitz, “SysML v2 Pilot Implementation - SysML2XMI and SysML2JSON,” 2024. [Online]. Available: https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/tree/master/org.omg.sysml.xtext/src/org/omg/sysml/xtext/util. [Accessed 23 08 2024].

[13] E. Seidewitz, “SysML v2 Release Google Groups - sysml to xmi in Eclipse,” 29 03 2024. [Online]. Available: https://groups.google.com/g/sysml-v2-release/c/glcQISm97nI. [Accessed 23 08 2024].

[14] S. Friedenthal, “SysML v2 Overview & Demo,” 30 01 2023. [Online]. Available: https://www.incose.org/docs/default-source/working-groups/mbse-initiative/sysml-

2-documents/sysml_v2_overview_demo.pdf?sfvrsn=50b971c7_2. [Accessed 23 08 2024].

[15] Maplesoft, "Engineering Analysis using Maple, driven by SysML 2.0 Semantics (Youtube video)," Maplesoft, 2023. [Online]. Available: https://www.youtube.com/watch?v=HdkCHFPqzIM. [Accessed 23 08 2024].

[16] H.-P. d. Koning, "Model Based Space Systems and Software Engineering - MBSE2021," ESA, 29 09 2021. [Online]. Available: https://indico.esa.int/event/386/contributions/6223/attachments/4266/6464/1015%20-%20Q&A.pdf. [Accessed 23 08 2024].

[17] M. Wilson and E. Seidewitz, "SysML 2 Pilot Implementation - SysMLValidator," 2023. [Online]. Available: raw.githubusercontent.com/Systems-Modeling/SysML-v2-Pilot-Implementation/ef3a378c3df678a683eb23be3d0fa4eac0646af8/org.omg.sysml.xtext/src/org/omg/sysml/xtext/validation/SysMLValidator.xtend. [Accessed 23 08 2024].

[18] OMG, "INCOSE 2024 International Workshop - SysML v1 to SysML v2 Transition Information Session," 28 01 2024. [Online]. Available: https://www.omgwiki.org/MBSE/doku.php?id=mbse:sysml_v2_transition:sysml_v1_to_sysml_v2_transition_information_session. [Accessed 23 08 2024].

[19] Ansys, "Scade One - An Open Model-Based Ecosystem, Ready for MBSE," 15 02 2024. [Online]. Available: https://innovationspace.ansys.com/knowledge/forums/topic/scade-one-an-open-model-based-ecosystem-ready-for-mbse/. [Accessed 23 08 2024].

[20] J. Feingold, "Growing Pains: Learning from SysML v1," Tom Sawyer, 25 02 2021. [Online]. Available: https://blog.tomsawyer.com/growing-pains-learning-from-sysml-v1. [Accessed 23 08 2024].

[21] MathWorks, 2024.

[22] MathWorks, "SysML v2 Modeling in System Composer".

[23] L. F. Pires, G. Guizzardi, G. Wagner and J. P. A. Almeida, "An Analysis of the Semantic Foundation of KerML and SysML v2," in *The 43rd International Conference on Conceptual Modeling*, Pittsburgh, USA, 2024.

[24] E. Seidewitz, "SysML v2 Release Google Groups - item and part usage," 15 01 2024. [Online]. Available: https://groups.google.com/g/sysml-v2-release/c/z1eOr5oH6pY. [Accessed 23 08 2024].

[25] L. Delligatti, SysML Distilled - A Brief Guide to the Systems Modeling Language, Addison-Wesley, 2014.

[26] S. Friedenthal, A. Moore and R. Steiner, A Practical Guide to SysML, Morgan Kaufmann, 2015.

[27] OMG, "OMG Systems Modeling Language (SysML) v2 Release," [Online]. Available: https://github.com/Systems-Modeling/SysML-v2-Release. [Accessed 4 9 2024].

[28] "PlantUML," [Online]. Available: https://plantuml.com/. [Accessed 4 9 2024].

[29] "SysON | The NextGen SysML Modeling Tool," [Online]. Available: https://mbse-syson.org/. [Accessed 4 6 2024].

[30] "Obeo," [Online]. Available: https://www.obeosoft.com/en/. [Accessed 4 9 2024].

[31] "Sirius Web," [Online]. Available: https://eclipse.dev/sirius/sirius-web.html. [Accessed 4 9 2024].

[32] B. van der Sanden, "LSAT," 2024. [Online]. Available: https://esi.nl/research/output/tools/lsat. [Accessed 23 08 2024].

[33] B. van der Sanden, Y. Blankenstein, R. Schiffelers and J. Voeten, “LSAT: Specification and Analysis of Product Logistics in Flexible Manufacturing Systems,” in *IEEE International Conference on Automation Science and Engineering (CASE)*, 2021.

[34] E. Seidewitz, “SysML v2 Release Google Groups - Adding features after declaring parts,” 2024. [Online]. Available: https://groups.google.com/g/sysml-v2-release/c/EsiPonftP7w. [Accessed 23 08 2024].

[35] T. Maassen, “Translating ComMA to SysML v2,” Radboud University, 2024.

[36] J. Voeten, POOSL: An object-oriented specification language for the analysis and design of hardware/software systems, Eindhoven University of Technology, 1995.

[37] B. Friederichs, “Automatic conversion of UML-class diagrams to POOSL,” Eindhoven University of Technology, Eindhoven, 2004.

[38] “Sparx Systems,” [Online]. Available: https://sparxsystems.com/. [Accessed 4 9 2024].

ICT, Strategy & Policy

www.tno.nl

TNO innovation for life