R&I

Universitat Oberta de Catalunya

# 5TH DELIVERABLE: VISUALIZABLE AND PROPAGABLE SYSMLV2 TRACES FOR METADATA ANALYSIS

**Edouard R. Batot**

SOM Research Lab
Av. Carl Friedrich Gauss, 5. Building B3
08860 Castelldefels

SOM

# Contents

# List of Acronyms

| | |
|---|---|
| UOC | Fundacio per a la Universitat Oberta de Catalunya |
| SOM | Systems, Software and Models Lab |
| CEA | Commissariat à l'énergie atomique et aux énergie alternatives |
| SST | SysML v2 Submission Team |
| MDE | Model-Driven Engineering |
| SYSML-V2 | Systems Modeling Language (SysML®) v2 |
| KERML | Kernel Modeling Language (KerML) |

## List of Figures

## Listings

# 1 Introduction

1. SysMLv2 related limitations
   - Metadata feature library - a promising idea
   - Model level expression - a (still) deceitful implementation
2. Data structure components
   (a) Concrete implementation
      - From SysMLv2 metadata library to JSon: extraction and re-injection of meta-data
      - JSon variability : multi-types, multi end, multi AnnotatingFeatures *et al.*

   (b) Conditions for use
      - Separated files
      - Named connections
      - Programmatic version
3. Syntax representation: JSon for all
   - Raw base extraction
   - Visualization using D3-JS
   - Matrix view
4. Implementers' guide
   - Impact analysis scenario
   - Energy consumption scenario
   - Toward multi criteria apprehension
      - Multiplicity/cardinality decisions
      - Integration target.. THEIA ?

This document is organized as follows. Section **??**, first briefly introduces the language dedicated to traceability that we have developed as well as the KerML / SysMLv2 ecosystem on which we integrate it. This section also defines the high level needs for quality traceability. Section **??**, presents which elements of KerML / SysMLv2 languages are of interest and how we have adapted and used them through the different integration strategies considered. We will detail our choices of architecture and implementation and the limits that we encountered in Section **??**[1]. Finally, we conclude this document in Section 6 before pointing at the resulting software artefacts and their examples of use in Section 7.

[1]We are integrating Trace*a* into SysMLv2 while it is being formalized by the SST.

## 2 SysMLv2 related decisions

During this fifth deliverable we ran into the limitations pointed out in the fourth deliverable. In this section, we present these limitations.

### 2.1 Metadata feature library – a promising idea

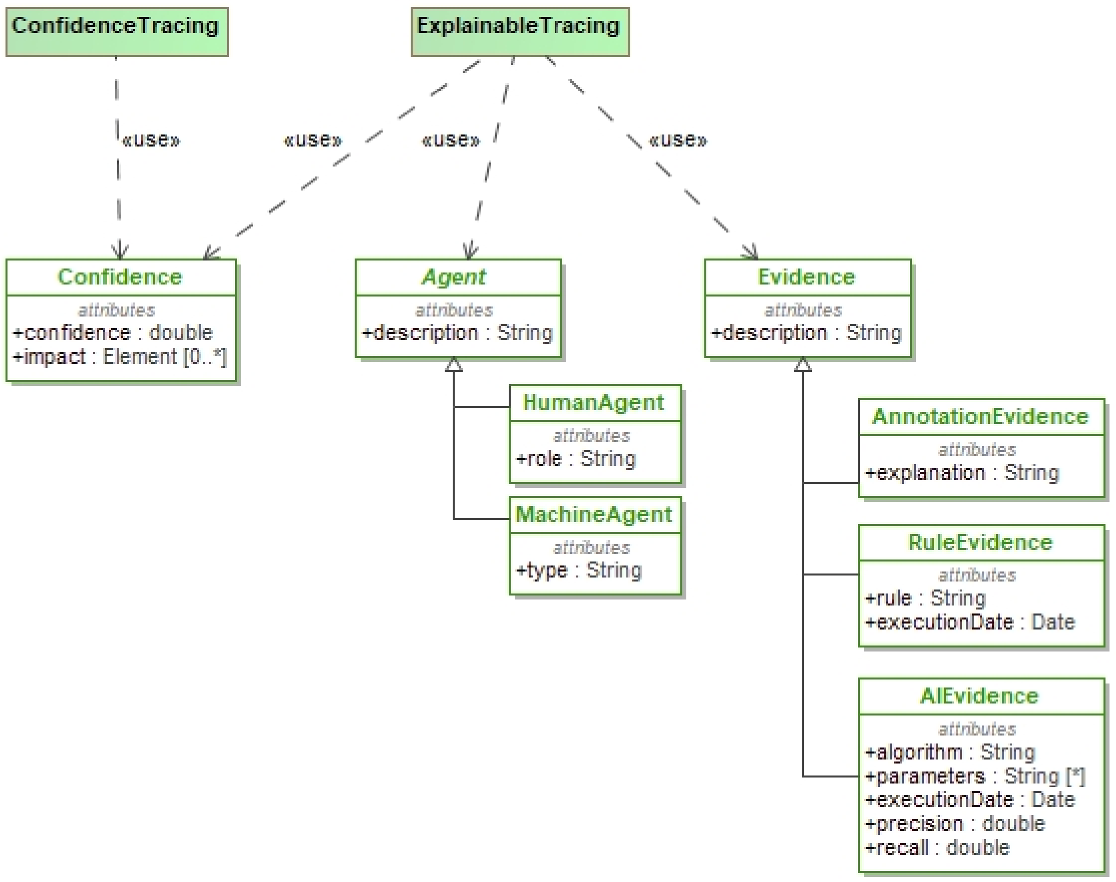


FIGURE 1: Metadatatypes for quality traceability including an assessment of confidence and monitoring of the identification processes used.

The Listings 1 and 2 present our feature library in SysML. Listing 1 contains the library declaration; Listing 2 contains an example of application of such datatype defined on a concrete example. It illustrates the allocation of a confidence value of 0.7 to a connection between a requirement (req) and a package. The second case (line 16 to 20) attributes a description and points to an element impacted (by the evaluation of the confidence),

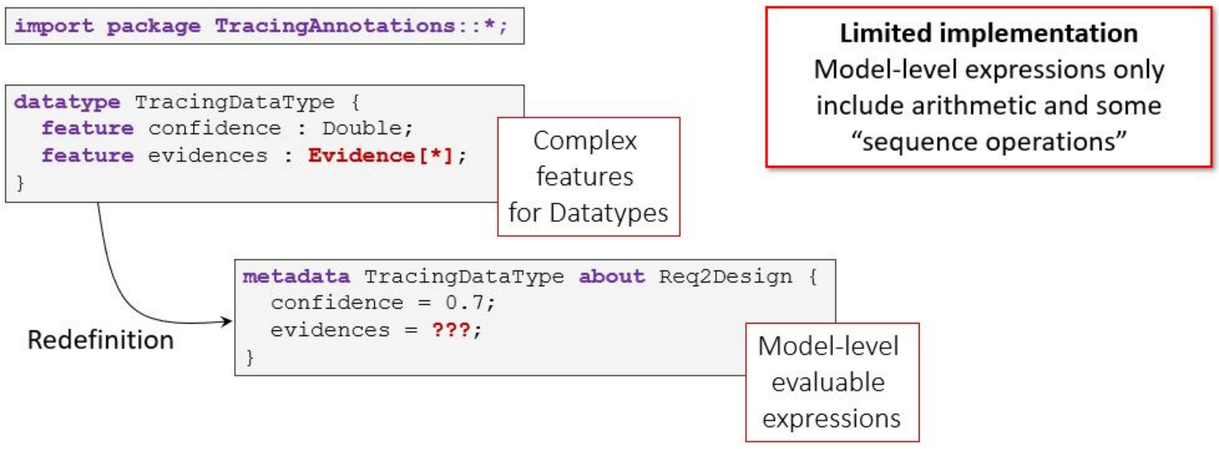
```
package TracingAnnotations {
 attribute def ConfidenceTracing {
   attribute confidence : Real;
   attribute impact : Anything[*]
   assert constraint
     { confidence >= 0.0 && confidence <= 1.0 }
 }

 attribute def ExplainableTracing {
   attribute description : String;
   attribute evidence : Evidence;
   attribute agent : Agent;
 }
}
```

Listing 1: Definition of a datatype dedicated to traceability (partial listing).

```
import package TracingAnnotations::*;

/*Definition of the target system. */
part vehiculetest {}
req RE01_MLV {}
package UMLCD_CORE {}

/*Assignment of a confidence of 0.7 to the Req2Design link.
  */
connection Req2Design connect RE01_MLV to UMLCD_CORE {
  @ConfidenceTracing {
    confidence = 0.7;
  }
}

/*Assigning a description and an impact to the Req2Design
  link.*/
connection Req2Design connect RE01_MLV to UMLCD_CORE {
  @ExplainableTracing {
    description = "Something";
    impact = (vehiculetest);
  }
}
```

Listing 2: Use of metadata features for traceability

## 2.2 Model level expression – a deceitful implementation

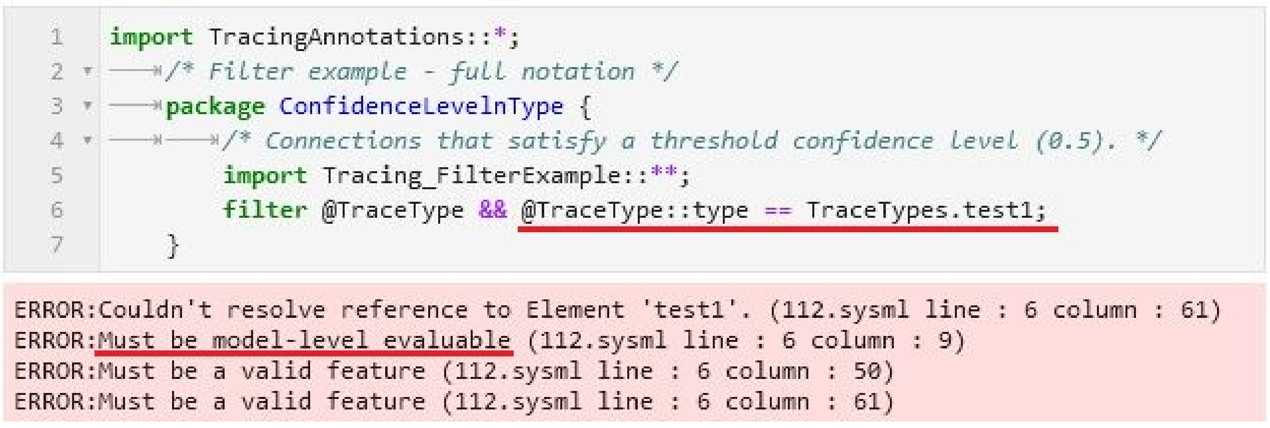


```
import package TracingAnnotations::*;

datatype TracingDataType {
  feature confidence : Double;
  feature evidences : Evidence[*];
}

metadata TracingDataType about Req2Design {
  confidence = 0.7;
  evidences = ???;
}
```

Limited implementation
Model-level expressions only include arithmetic and some "sequence operations"
Complex features for Datatypes
Redefinition
Model-level evaluable expressions

FIGURE 2: Annotations and complex datatypes.

When implementing these feature we ran into a number of disillusions. Fisrt and foremost, SysMLv2 implementation of the *Model-level evaluable expressions* is for the least unsure. The concrete classes of the pilot implementation have been designed and written by only one developer (namely, Ed Seidewitz) and do not offer any safe guards. As showed in the previous deliverable, Fig. 2 points to where lies this limitation: when using a metadata feature, there is no guarantee for the use of "complex structures", *i.e.,* that are not of basic types (int, double, String, ...). If used, these complex structure will at best show an error as pictured in Fig. 3, and at worst keep the fallacious state of the system silent and leave the system out of its requirements.



```
import TracingAnnotations::*;
    /* Filter example - full notation */
    package ConfidenceLevelnType {
        /* Connections that satisfy a threshold confidence level (0.5). */
        import Tracing_FilterExample::**;
        filter @TraceType && @TraceType::type == TraceTypes.test1;
    }
```

```
ERROR:Couldn't resolve reference to Element 'test1'. (112.sysml line : 6 column : 61)
ERROR:Must be model-level evaluable (112.sysml line : 6 column : 9)
ERROR:Must be a valid feature (112.sysml line : 6 column : 50)
ERROR:Must be a valid feature (112.sysml line : 6 column : 61)
```

FIGURE 3: Limitation of the implementation of model-level evaluable expressions.

# 3 TraceaML support architecture

This section introduces the decisions we made to activate Tracea's functionalities with SysMLv2 in its present state. We present the design decisions related to the traces and their trace links in a first place. Then we precise the conditions to the use of this implementation.

## 3.1 Decisions related to traceability features

First, to overcome the limitations of the implementation of SysMLv2, we decided to concentrate on the use of basic types. These are sufficient to express the confidence degree, the types of traces and the (energy) cost attributed to a connection in the system.

Then, we had to choose between a certain number of options while implementing the features related to confidence and cost metadata. If confidence is a unique value for a link[2], a link may have one or more source(s) and one or more target(s). As shown in For example, links may have one or more type(s). Yet, the visualization and semantics associated to such multi-valued attribute remains to discuss for annotating features do not restrict their usages. Since there is no way to restrict the definition of confidence annotations, what if more than one confidence is attributed to the same link? These variability points should be clearly and exhaustively understood to put our artefact to production. Meanwhile, multi-end and multi-typed links are allowed for persistence. Only the *first* type and the first source and target are considered for the pilot visualization product.

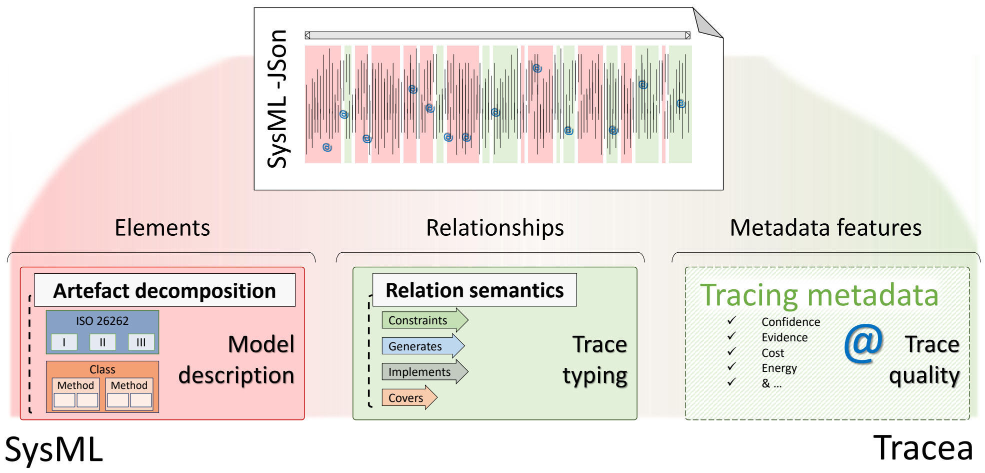


FIGURE 4: Organisation of SysMLv2 JSon representation.

[2]We consider *link* and *connection* full synonym in this document.

## 3.2 JSonTransformer: from SysMLv2 to TraceaML

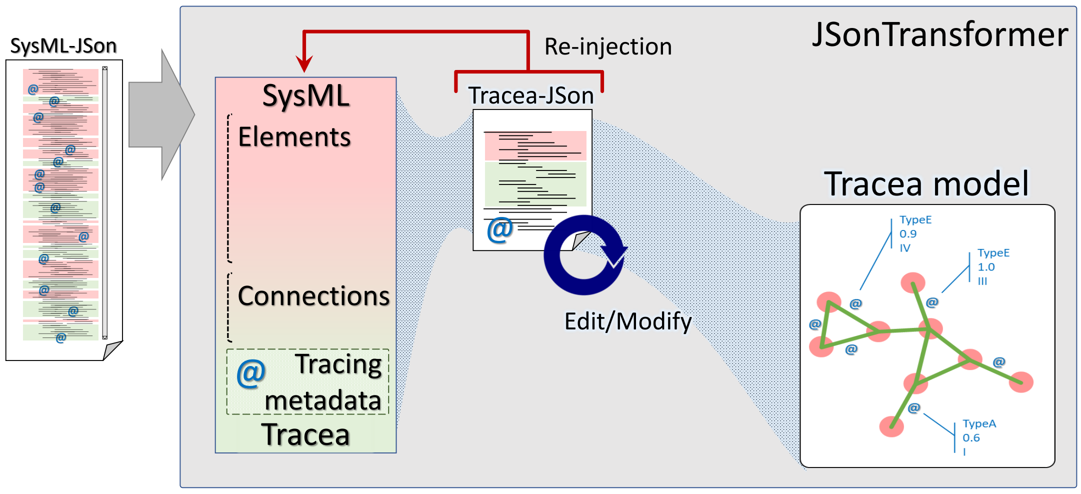


FIGURE 5: JSonTransformer: integration of Tracea with SysMLv2.

Fig. 5 shows the big picture of the integration of Tracea and SysMLv2 with JSonTransformer[3]. To remain orthogonal to the system, and to asynch ourselves from the changes in the language itself, we export a JSon snapshots of the SysMLv2 model. Export can be made from the Jupyter implementation or the Eclipse pilot implementation independently. From this very voluble expression of the model (say 50kLoC for a minimalist model of a few elements and a couple of links), we extract a core model to transform and manipulate easily into other format (for more details about Tracea-JSon see Section 4.1). This format allows the use of multi-end and multi-type links, using the IDs of the elements in the same manner as the original (SysML) file.

*[Most of the work here lies in the transformation from this raw JSon to the prettied "Tracea" version we target.]*

## 3.3 Extraction and re-injection of metadata

Transforming the JSon persistence of the SysMLv2 model comes at a cost. The entire JSon is used as input and our tool extract the necessary information. After editing this information, we re-inject it in the model through a (model-to-text) transformation. We take the *Tracea* model and transform it to SysMLv2 textual notation.

This method allows us to remain independent from the evolution of the pilot implemen-

[3]`https://github.com/ebatot/TraceaingJson`

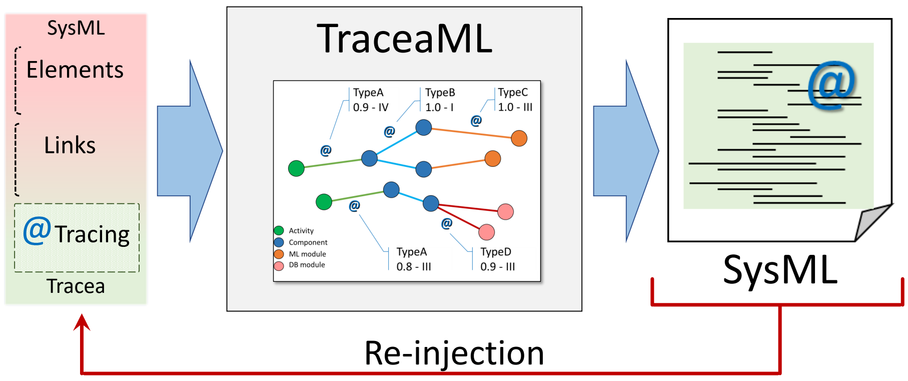


FIGURE 6: Re-injection of tracing metadata into a SysML model.

tation. We act directly on the concrete artefacts – which in the eyes of the SST is bond to its present version and will not change (*much*) any soon. This method follows a bottom up understanding of the model: we reason from examples (at the instance level) to build and browse the feature paths of interest of the SysMLv2 metamodel.

## 3.4 Implementation details

### 3.4.1 Conditions of use

Using a JSon snapshot requires that the instances of the links (or *ConnectionUsages* in SysMLv2 terminology) be stored separately, and named. Indeed, metadata is affected to elements through their name. We only target (and source) tracelink with coarse grain elements to avoid the engineering complexity to browse feature paths. Summarized:

- Separated files – in order to re-inject the connections and their metadata, these must be kept separated in SysML (textual).
- Named connections – metadata attribution from external source (the keyword "metadata" requires a named target).
- Coarse grain targets – browsing features' path is a coding challenge on its own that is not addressed in the current version of our implementation.
- External metadata – in order to be able to re-inject metadata without collision with existing ones, all metadata must be "external", *i.e.,* declared with "metadata" keyword in SysML.
- Feature IDs "bug" – Identification of SysMLv2 features is buggy: the same feature called from two connections distinct show two IDs. We bypass this limitation using names as identifier for features.

### 3.4.2 Unexpected SysMLv2 identifier allocation

As mentioned previously, implementing JSonTransformer, we ran into an unexpected behavior related to the allocation of identifiers to targets and sources of connections. We reported the ”bug” to the SST and recorded it in our Git repository at `https://github.com/modelia/traceea/tree/master/4-sysml-json-transformer/sysml_id_allocation_bug`

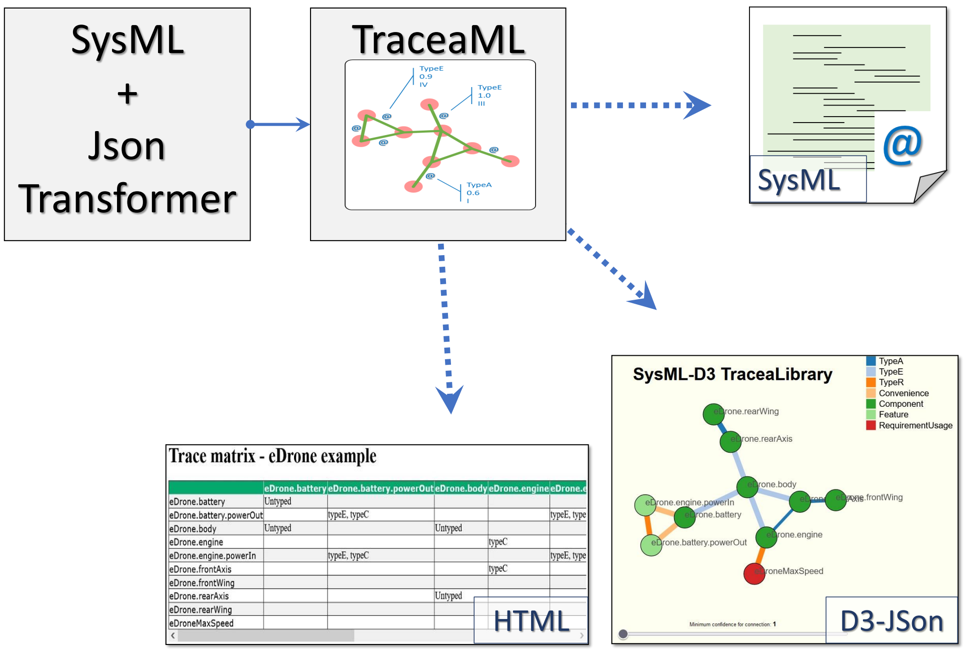
### 3.4.3 Programmatic version

We envision a programmatic version of the current implementation. The use of the API in the pilot implementation has a steep learning curb and we opted in an hybrid version: using the JSon persistence of the models to work on propagation features. This way we work on concrete cases that we can craft to better suit our needs without requiring the actual implementation to be completely finished.

The current implementation will serve as a guide to implement a programmatic version. In this more interactive version, browsing elements will follow the exact same path as the JSon-snapshoted version.

## 3.5 Summary

Need text ! Fig. 7



Trace matrix - eDrone example

| | eDrone.battery | eDrone.battery.powerOut | eDrone.body | eDrone.engine | eDrone.e |
|---|---|---|---|---|---|
| eDrone.battery | Untyped | | | | |
| eDrone.battery.powerOut | | typeE, typeC | | | typeE, type |
| eDrone.body | Untyped | | Untyped | | |
| eDrone.engine | | | | typeC | |
| eDrone.engine.powerIn | | typeE, typeC | | | typeE, type |
| eDrone.frontAxis | | | | typeC | |
| eDrone.frontWing | | | | | |
| eDrone.rearAxis | | | Untyped | | |
| eDrone.rearWing | | | | | |
| eDroneMaxSpeed | | | | | |

FIGURE 7: SysMLv2 Tracing components, a big picture.

# 4 Syntax representation

In this section we present the different means a user can take a trace to work with. The first version shows a key representation based on Traces's metamodel properties expressed in JSon. The second version is slightly augmented to allow its visualization using the D3-JS API[4]. Last but not least, a matrix-based representation facilitates the analysis of a trace and gives a useful HTML visualization.

## 4.1 Trace*a* model

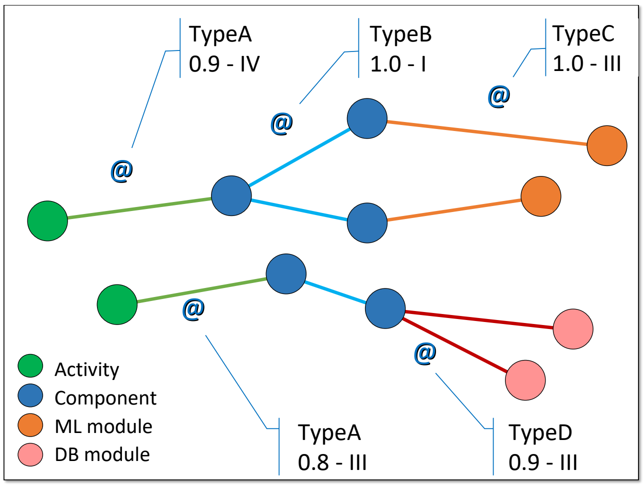


FIGURE 8: An illustrative example of a Tracea model.

The first representation reflects directly the complex nature of the trace using Trace*a*-like representation. Links are encoded with multiple types and can be multi-ended (source and target alike). This raw representation is the working base for future transformation into specific visualizations, characterized representations, or further mathematical analysis. An illustrative example of a Tracea model is given in Fig. 8. Its textual representation is shown in the excerpt of Listing 3.

[4]`https://d3js.org/`

```
{
 "links": [
   {
     "id": "1085",
     "name": "eng2front",
     "qualifiedName": "eDrone_example::eng2front",
     "types": ["TypeE"],
     "sources": [{ "id": "b9bb"}],
     "targets": [{ "id": "3adb"}],
     "confidence": 0.85
   }
    /* ... */
 ], "nodes": [
   {
     "id": "d83f",
     "name": "eDrone.battery",
     "type": "Feature"
   }
   /* ... */
]
}
```

Listing 3: Excerpt of a trace written in Tracea-JSon.

## 4.2 SysML reinjection

The Tracea model allows the generation of SysML code to reinject modification, or to populate a model from sources external from the target languages (here SYsML). Listing 4 shows a (modified) trace that can be reinjected in the original the model as depicted earlier in Fig. 6.

```
/* Trace links */
connection eng2front connect eDrone.engine to eDrone.frontAxis;
connection frAx2frWi connect eDrone.frontAxis to eDrone.frontWing;
connection reAx2reWi connect eDrone.rearAxis to eDrone.rearWing;
connection bod2bat connect eDrone.body to eDrone.battery;
connection bod2eng connect eDrone.body to eDrone.engine;
connection bod2frAx connect eDrone.body to eDrone.frontAxis;
connection bod2reAx connect eDrone.body to eDrone.rearAxis;
connection eng2bat_Typed connect eDrone.battery.powerOut to eDrone.engine.
    powerIn;
connection rq1ToWing connect eDroneMaxSpeed to eDrone.engine;
connection refine1 connect eDrone.battery to eDrone.battery.powerOut;
connection refine2 connect eDrone.battery to eDrone.engine.powerIn;

/* Tracing metadata */
metadata m6093: ConfidenceTracing about eng2front { confidence = 0.85;}
metadata m4103: TraceType about eng2front { tracetype = "TypeE";}
metadata m5096: ConfidenceTracing about frAx2frWi { confidence = 0.65;}
metadata m9040: TraceType about frAx2frWi { tracetype = "TypeE";}
metadata m7556: ConfidenceTracing about reAx2reWi { confidence = 0.75;}
metadata m9270: TraceType about reAx2reWi { tracetype = "TypeE";}
metadata m8202: TraceType about bod2bat { tracetype = "Internal";}
metadata m9711: TraceType about bod2eng { tracetype = "Internal";}
metadata m8159: TraceType about bod2frAx { tracetype = "Internal";}
metadata m7086: TraceType about bod2reAx { tracetype = "Internal";}
metadata m7259: ConfidenceTracing about eng2bat_Typed { confidence = 0.65;}
metadata m3388: TraceType about eng2bat_Typed { tracetype = "TypeA";}
metadata m9094: ConfidenceTracing about rq1ToWing { confidence = 0.45;}
metadata m5621: TraceType about rq1ToWing { tracetype = "TypeA";}
metadata m340: ConfidenceTracing about refine1 { confidence = 0.45;}
metadata m8719: TraceType about refine1 { tracetype = "Convenience";}
metadata m5065: ConfidenceTracing about refine2 { confidence = 0.55;}
metadata m8879: TraceType about refine2 { tracetype = "Convenience";}
```

Listing 4: SysML code generated from a Tracea model.

## 4.3 Graph visualization

The literature mention extensively the need for higher level representation to apprehend the inherent complexity of traces. In the end of the day, they are graphs and it sounds of utmost importance to represent them graphically. Fig. 9 shows a graph-based representation of a trace. Each link is an edges between two node-elements. Here, representing multi-ended links is arduous and would require strong decisions. The multi-typed nature of links would need also a further investigation to decide whether to decide the types to show beforehand, or to allow a more interactive manipulation in which there is options to allow users to change them on the fly.

*Interactivity: confidence/cost thresholds ; tickers for types ; ...*

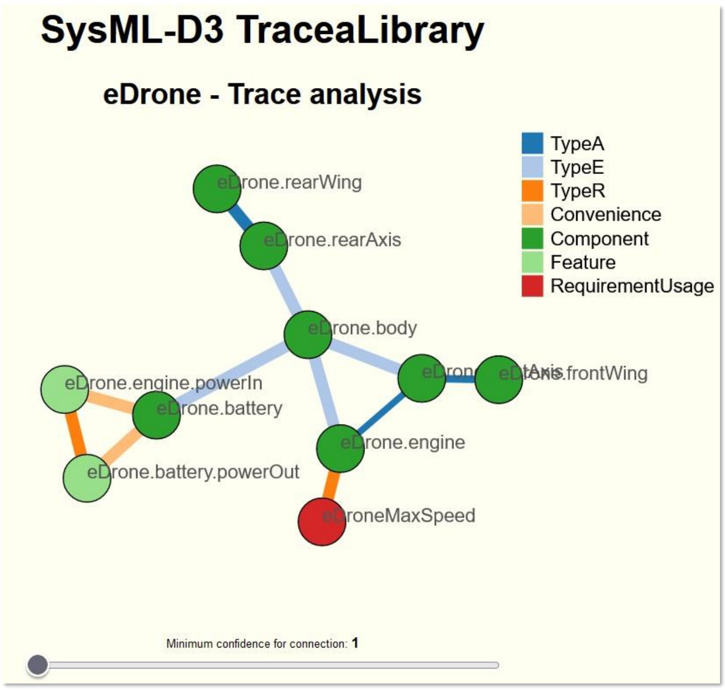
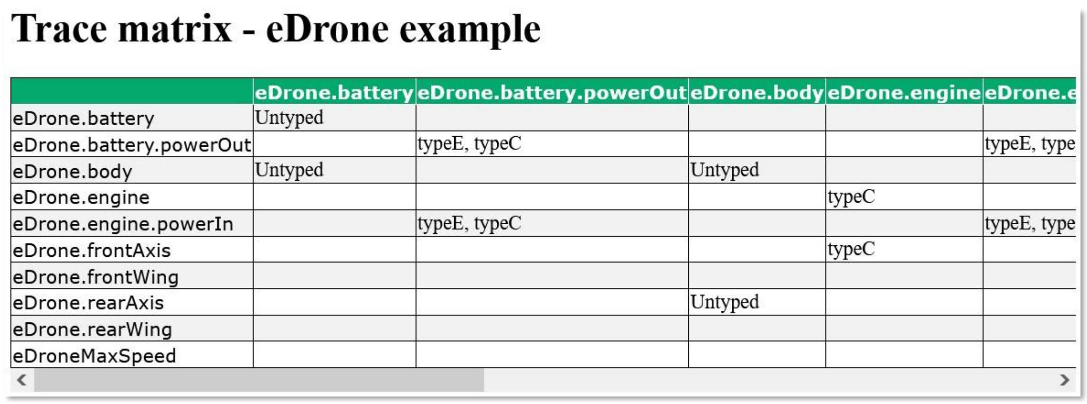


FIGURE 9: A graph-based visualization of a Trace.

## 4.4 Matrix-based view

Finally, we offer the alternative matrix-based representation of the traces. We export the Trace*a model* as text and HTML tables. The latter is rather interesting. The use of interactive instances of HTML in conjunct use with CSS-JS will allow a valuable set of tools to explore the trace.


Trace matrix - eDrone example

| | eDrone.battery | eDrone.battery.powerOut | eDrone.body | eDrone.engine | eDrone.e |
|---|---|---|---|---|---|
| eDrone.battery | Untyped | | | | |
| eDrone.battery.powerOut | | typeE, typeC | | | typeE, type |
| eDrone.body | Untyped | | Untyped | | |
| eDrone.engine | | | | typeC | |
| eDrone.engine.powerIn | | typeE, typeC | | | typeE, type |
| eDrone.frontAxis | | | | typeC | |
| eDrone.frontWing | | | | | |
| eDrone.rearAxis | | | Untyped | | |
| eDrone.rearWing | | | | | |
| eDroneMaxSpeed | | | | | |

FIGURE 10: A matrix-based representation of a Trace.

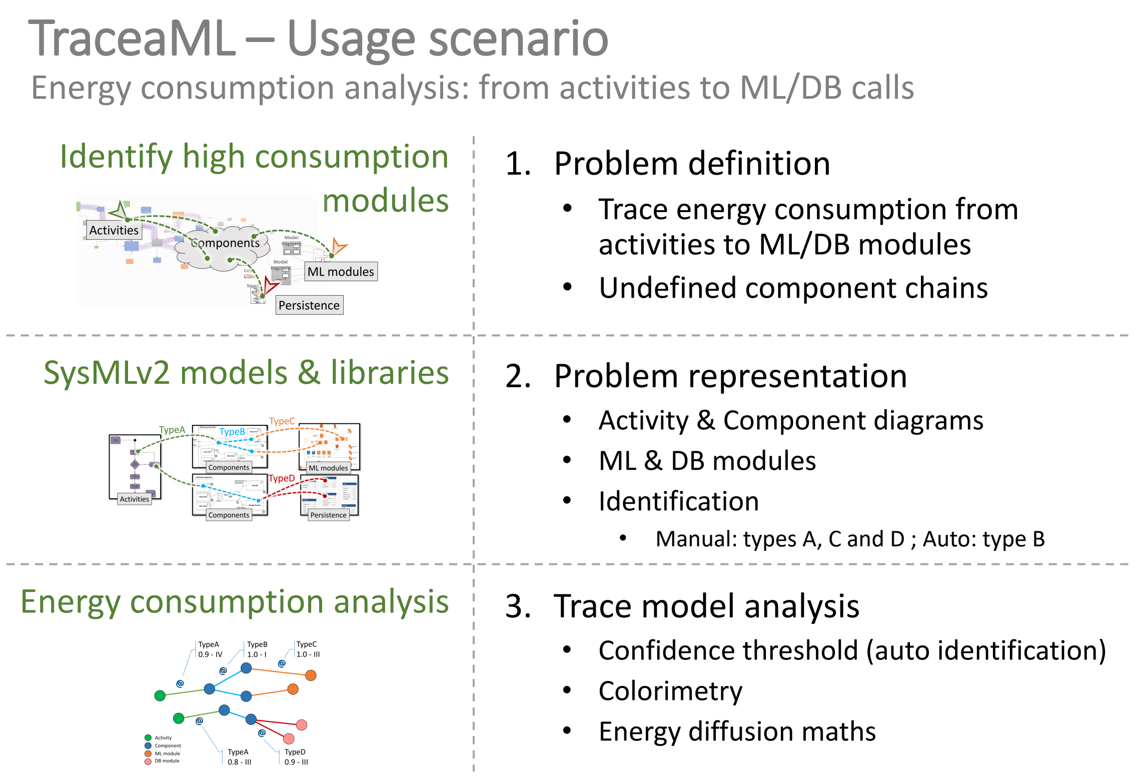
## 4.5 Interactive features planned

Using an external tool to handle traces then serves to populate the trace from external sources. In this fashion, an interactive UI shall offer:

- To assign trace types to links and define which ones are shown (tickers)
- To identify and target ends for links (potentially from a base of tagged elements ?)
- To show metadata features (parametered on demand)
- To filter links based on thresholds: confidence, cost, energy consumption; or on their type, name, end types...
- To export result after manipulation: results of the selection of links / elements, and to export the edited links.

# 5 Implementers' guide

## 5.1 Example usage scenario



TraceaML – Usage scenario

Energy consumption analysis: from activities to ML/DB calls

Identify high consumption modules

Activities
Components
ML modules
Persistence

1. Problem definition
   - Trace energy consumption from activities to ML/DB modules
   - Undefined component chains

SysMLv2 models & libraries

TypeA
TypeB
TypeC
TypeD
Activities
Components
ML modules
Persistence

2. Problem representation
   - Activity & Component diagrams
   - ML & DB modules
   - Identification
     - Manual: types A, C and D ; Auto: type B

Energy consumption analysis

3. Trace model analysis
   - Confidence threshold (auto identification)
   - Colorimetry
   - Energy diffusion maths

FIGURE 11: Use case: Energy consumption analysis.

## 5.2 Problem definition

## 5.3 Problem representation

## 5.4 Trace model analysis

## 5.5 Toward multi-criteria apprehension

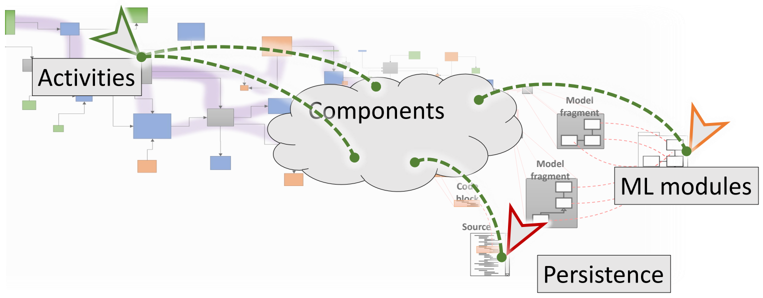


FIGURE 12: Problem statement.

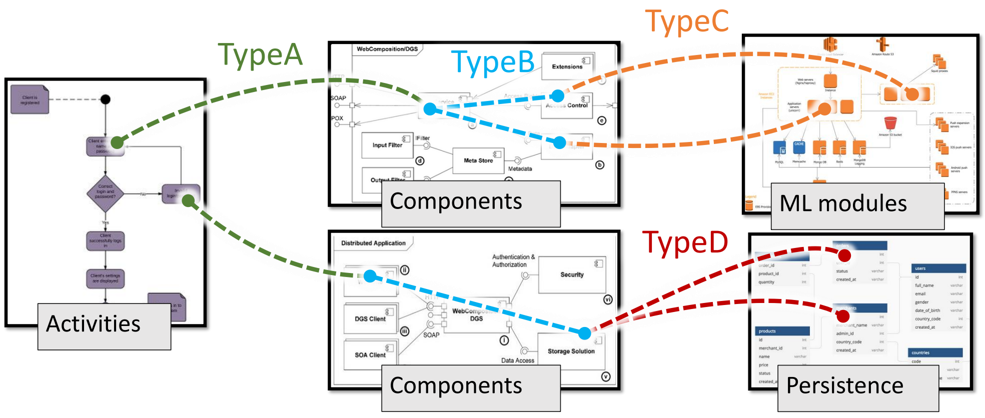


FIGURE 13: Problem representation.

## 6 Conclusion

D5 - Conclusion !! This deliverable presents means to integrate quality traceability into SysMLv2 using Trace*a*. With Trace*a* datatypes, SysMLv2 relationships can be annotated with valuable information related to their quality – *i.e.,* their *trustability*. The degree of confidence as well as the information necessary to justify it can be associated to SysML links (connections) through metadata definition.

This integration is *orthogonal*. It does not impact the structure of the language itself – changes happen at the model level with new feature libraries, not at the metamodel level. This allows the (re)use of the machinery supporting (meta)annotations and eases the (re)definition of tracing structures specific to a certain project or company. We showcase these benefits in one small example that also reveals the current limitations of SysMLv2's implementation.

The SST confirms that annotating features are valuable and salient artefacts in the development of SysML. They shall take more and more importance in the future releases. Work on the evaluation of expressions at the model level is highly required and will be part of the agenda of the fourth quarter of 2021. Finally, as a sign of encouragement, the SST invites us to further investigate in the direction we took.

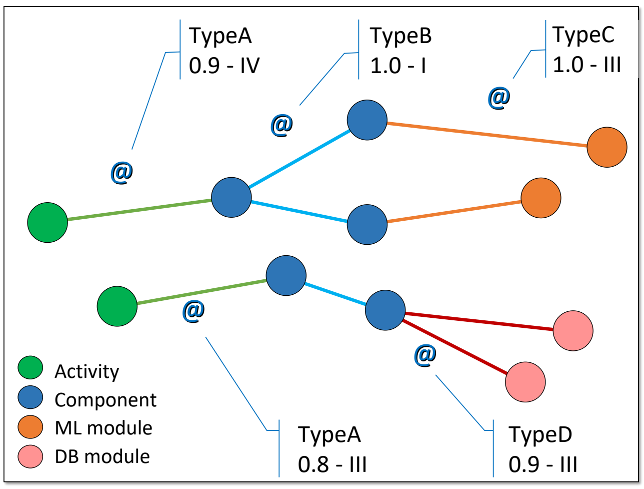


FIGURE 14: Trace model representation.

## 7 Software artefacts

The software artifacts we have implemented have been uploaded to the Modelia Git repository and can be accessed at: `https://github.com/modelia/tracea/tree/master/4-sysml-json-transformer`.

The folder contains:

– **TraceaLibrary** in two versions: with trace types as *Strings* and as *Enum literals* ;

– An **example definition**: the definition of an eDrone in SysMLv2: architecture, requirements, definitions, and a set of traces ;

– The **example output** files: JSonTransformer outputs 4 files for the eDrone example, respectively to the four formats mentioned in Section 4 ;

– A **D3 script** that plots a graphical visualization of Tracea models ;

– A **”bug” record** that depicts of a strange SysMLv2 behavior related to the allocation of IDs to ”dot-notation features”.

This deliverable gave birth to the JSonTransformer[5].

[5]`https://github.com/ebatot/TraceaingJson`

# REFERENCES