Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

INTO-CPS Traceability Design

Deliverable Number: D4.2d

Version: 0.4

Date: 2016

Public Document

http://into-cps.au.dk

Contributors:

Kenneth Lausdahl(AU)
Peter Niermann (TWT)
Jos Höll (TWT)
Carl Gamble (UNEW)
Oliver Möller (VSI)
Etienne Brosse (ST)
Tom Bokhove (CLP)
Luis Diogo Couto (UTRC)
Adrian Pop (LIU)

Editors:

Christian König (TWT)

Reviewers:

Ken Pierce (UNEW)
Ana Cavalcanti (UY)
Alie El-Din Mady (UTRC)

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	12-07-2016	C. König (TWT)	Initial document version
0.2	08-09-2016	J. Höll (TWT)	Add suggestion of message specification for daemon
0.3	25-10-2016	C. König (TWT)	Revised after first internal review
0.4	10-11-2016	C. König (TWT)	Incorporated review comments

Abstract

This deliverable covers the design of the traceability and model management features in INTO-CPS. At the end of year 2, an architecture for handling traceability is drafted and development of prototypes has started. An outlook on the work that is planned for the third year of INTO-CPS is given. This document is closely related to Deliverables D3.1b [FGPP15] and D3.2b [FGPP16], where the foundations for traceability in INTO-CPS are described.

Contents

1 Introduction **6**
- 1.1 Purpose and goals for traceability within INTO-CPS 6
- 1.2 Scope 7
- 1.3 State of the art of the baseline tools 7

2 Specification **8**
- 2.1 Requirements and design decisions 8
- 2.2 Traceability architecture 9
- 2.3 Design decisions 12
- 2.4 Traceability database 13
- 2.5 Traceability actions 13
- 2.6 Message structure and syntax 14
- 2.7 Example messages 15
- 2.8 Format of the URI 20

3 Use cases **21**
- 3.1 Model Creation 22
- 3.2 Model Modification 22
- 3.3 Model Destruction 22

4 Querying and Visualisation **23**

5 Status of the tools at M24 **24**

6 Summary and Outlook **25**

Appendices **26**

A Used libraries **26**

B Abbreviations **27**

1 Introduction

This deliverable presents the design of the traceability and model management functions in INTO-CPS at the end of year 2. It is to a large extend connected with and based on the foundational work presented in Deliverables D3.1b [FGPP15] and D3.2b [FGPP16]. In the introduction (section 1), the purpose and scope of traceability in INTO-CPS is discussed, closing with a summary of the traceability features of the INTO-CPS baseline tools. In section 2, the requirements to traceabilty that are given in INTO-CPS are listed, followed by the architecture that was chosen, relevant design decisions and major components of the architecture and the message format that was defined for exchanging data. In the following section 3, relevant actions are divided into single steps that are performed by the tools. While so far the focus was on saving traceability data, the next section 4 discusses querying of the database and visualisation of the results. This deliverable is concluded with the status of the traceability support in the tools at the end of year 2 (section 5) and an outlook for the work that is planned for year 3 (section 6).

1.1 Purpose and goals for traceability within INTO-CPS

Traceability and, in particular, requirements traceability can be defined as in [GF94] as:

> Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its specification and development, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases).

This definition is closely related to the the tool-chain concept of INTO-CPS, where information about entities (such as requirements) needs to be exchanged between different tools, which were initially not connected, and are mostly designed as stand-alone tools. The benefits of traceability in general, and in particular the benefits that are aimed for in the framework of the INTO-CPS tools and methods, include:

- Checking the realisation of requirements in models or code,

- Enabling collaborative work by connecting artifacts and knowledge from different users,
- Decreasing redundancy by connecting different tools to a single requirements source and allowing a system-wide view that is not only limited to single tools.

Of course, these benefits come at a cost. In the case of traceability, this means mostly overhead by the users to create and maintain the traceability links. Having only some traceability links in a project up to date significantly reduces the value of the whole exercise. Therefore, one of the goals for traceability in INTO-CPS is to automatize most of the traceability work, so that the user has to perform as few manual tasks as possible.

1.2 Scope

The primary scope for traceability in INTO-CPS is demonstration of the basic traceability tasks across the tool-chain. This includes mostly tracing of the requirements and connecting them with the models (and their parts, such as classes), the simulation results and the produced code. Furthermore, analysing the impact of changes (e.g. in requirements) to the overall system can be supported through the traceability efforts.

It is however not in the primary scope of the traceability work in INTO-CPS to be able to trace back all steps of the development and revert to each step in the development's history. For this, other parts of the INTO-CPS project and tools are seen as more appropriate, such as versioning tools (SVN, git) or functionalities of the INTO-CPS Application (see Deliverable D4.2a [BLL$^+$16]). Therefore, as will be described below, some of the tools support git for versioning.

1.3 State of the art of the baseline tools

The goal of INTO-CPS is to form a tool-chain from existing tools, each with their own features and development history. In the following, the state of traceability integration in the INTO-CPS tools before the project start (i.e. the baseline tools) is listed.

Modelio: In the baseline version, Modelio allows to model requirements in various languages (UML or BPMN for example). By permitting link creation between requirements and these other formalisms, Modelio is able to trace the realisation, verification, satisfaction of these requirements.

OpenModelica: Traceability in OpenModelica is so far (ie. in the baseline version) only supported in terms of tracing generated C code back to Modelica code, and it is mostly used for debugging. More details can be found in [PSA+14].

20-sim: In the baseline version, 20-sim does not support traceability.

Overture: In the baseline version, Overture does not support traceability.

RT Tester: In the baseline version, RT-Tester does not support traceability in the OSLC sense, where traceability data is exchanged with external tools through a standardized interface and format (such as OSLC). It should be pointed out, however, that traceability in terms of test-case and requirement management is a core feature of RT-Tester tool chain. Connecting test cases and requirements, the associated status accounting (including reporting) during a test campaign are described in detail in [Ver15a] and [Ver15b].

In summary, it can be said that some of the baseline tools do not support traceability at all, and some support traceability in their own ecosystem, but do not support a common interface of method for exchanging traceability information in the tool-chain sense that is targeted in INTO-CPS. For all of the tools, an architecture and a common interface for exchanging traceability information must therefore be defined. This is explained in the following section.

2 Specification

In this section, the main functionalities of the traceability features that are being implemented in INTO-CPS are described.

2.1 Requirements and design decisions

The requirements to the traceability functionalities are listed in Deliverable D7.5 [LPO+16]. These requirements, which are listed in the following table 1, are being updated over the progress of the project.

Req. No.	Description
0013	The COE must link the co-simulation configurations together with any results produced
0015	It must be possible to trace which FMU instances contribute to the satisfaction of which requirements
0016	It must be possible to trace an FMU implementation back to the SysML architecture component which it implements
0017	It must be possible to trace test cases to test/simulation results
0089	The INTO-CPS toolchain must support the traceability from requirements down to the code
0090	Impact analysis
0103	Traceability Service in the INTO-CPS Application Download Manager
0104	Traceability Service controlled through INTO-CPS application
0105	Predefined Traceability Queries
0106	Expert Traceability Queries

Table 1: Requirements with relation to traceability, from D7.5 [LPO+16]

These requirements reflect the use-cases that are envisioned for the traceability in INTO-CPS, as well as the targeted technical realisation. The goal is to develop a traceability service that can be integrated into the INTO-CPS application, in order to offer the users of the application a relatively easy access to its features through predefined queries. Furthermore, this traceability service should allow expert users or developers the full potential of traceability.

2.2 Traceability architecture

This section introduces an outline of the tool and file system elements that could support the traceability activities described in the line follow robot use case (see Deliverable D3.2b [FGPP16]) and is shown in Figure 1.

The central element of the traceability architecture is the daemon, along with an interface (a REST-API[1]) that can be used in two ways: The tools (e.g. 20-Sim, OpenModelica, Overture, Modelio, RT Tester, the COE, the INTO-CPS application) write data to the interface (e.g. they send traceability

[1]REST: Representational State Transfer; API: Application Programming Interface

information from actions that happened within the tools to the daemon), the INTO-CPS application queries the interface (e.g. for retrieving information from the database). The daemon sends this data to the database, and queries the database for data.

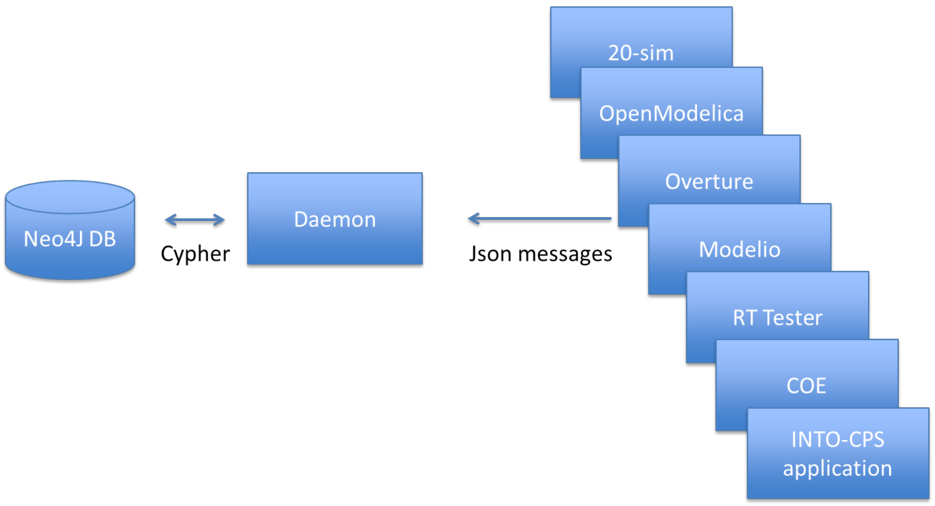

Figure 1: Schematic architecture of the traceability-related tools.

2.2.1 Traceability Daemon

The daemon is a essentially the core of the traceability tool support. It is launched or terminated by the INTO-CPS Application or the other tools. The daemon's primary function is to create an OSLC compliant HTTP port and listen for the POST and GET actions. It will store data sent with a POST and will return a suitable response to a GET request. The daemon provides an interface that allows that app query interface (section 2.2.2) to retrieve the required data. This interface basically passes cypher queries to the Neo4J database.

Regarding the dependencies to other software, the libraries in annex A are currently used by the traceability daemon.

2.2.2 Query Interface

The query interface enables the user to query the data stored by the traceability database. The list below only indicates the activities that the interface handles and does not indicate the all the data required. The complete description of the ontology that forms the basis of the traceability features in INTO-CPS can be found in Deliverable D3.1b ([FGPP15]), Annex 4.

- Impact Analysis
- Retrieve entities associated with X where X could be a simulation result

Querying the traceability database is done in the Cypher query language (the equivalent to the structured query language in relational databases, see `https://neo4j.com/developer/cypher/`). In principle, cypher queries can be written freely, which the daemon passes on to Neo4J. To facilitate usage however, we plan to pre-define queries for the actions listed above so that they can be easily integrated in the INTO-CPS application. This satisfies requirements 0105 and 0106 (see table 1).

2.2.3 Activity Interface

The interface enables the user to record events that take place outside of the INTO-CPS application, such as modelling in the modelling tools (Modelio, Overture, 20-sim, OpenModelica). The list below only indicates the activities that the interface handles and does not indicate all the data required. For a more detailed description of the activities, see [FGPP16].

- Requirements Management
- Architecture Modelling
- Model Description Export
- Model Description Import
- Simulation Modelling
- Model Check Modelling
- Model Checking
- Test Creation
- FMU Export
- HiL FMU Generation
- Code Generation

- Configuration Creation

Furthermore, some activities are specific to the INTO-CPS application:

- Simulation
- DSE Config Creation
- DSE
- Design Note Creation
- Library FMU Import

Each of these activities is sent from the tools to the daemon in a message, which are explained below in section 2.7.

2.3 Design decisions

In a previous deliverable D3.1b (sections 3.1 and 3.2, [FGPP15]), an extensive review of research projects with respect to their handling of traceability and requirements engineering is given. In summary, it can be said that there is currently not one standard approach or technology to traceability, and its implementation depends largely on the use-case and the related tools. Within the traceability efforts of INTO-CPS, two standards are mostly used, PROV and OSLC. Both have been introduced already in deliverable D3.1b [FGPP15], and therefore only some information is repeated here.

The Provenance (PROV)[2] set of documents builds on the notation of *entities*, *activities* and *agents*. In addition, relations are defined that describe the connection between the entities, activites and agents.

Over the last years, the "Open Services for Lifecycle Collaboration" (OSLC)[3] specifications emerged, aiming at the integration of development tools. For traceability purposes, in particular the OSLC Requirements Management (RM) specification is relevant[4]. An example for usage of OSLC for integration of modeling tools in general, and with aspects of traceability in particular, can be found in [EN13]. In order to design the traceability functionalities of the INTO-CPS toolchain in an open way, which should be compatible with

[2]see `http://www.w3.org/TR/prov-n/`

[3]see `http://open-services.net/`

[4]see `http://open-services.net/specifications/requirements-management-2.0/`

other tools, we use the PROV and OSCL specifications for describing the relations between entities, actions and agents.

2.4 Traceability database

As shown in section 2.2 in detail, the traceability data consists essentially of triples, made up of *subject*, *predicate* and *object*, where *subject* and *object* are either an entity, action or agent. The *predicate* is an OSLC or Prov[5] relationship. The triples that make up the traceability data are stored is a Neo4J database, which is a graph database[6]. This is advantageous, since the traceability data itself is in principle also a graph relationship between the different elements. Therefore, Neo4J promises to be a suitable choice for larger sets of traceability data. For example, for querying and retrieval of highly heterogeneous data (models, annotations and simulation results) in systems biology, a Neo4J database was successfully used [HWW15].

The synchronization of the databases of multiple users can be realized using a text file containing one line for each JSON message. Whenever traceability data is sent to the daemon one line can be written to this file which then can be merged in the repository. When pulling the git repository new lines in the pulled text file can be sent to the daemon.

In the following figure 2, different elements of a traceability graph are displayed. Entities (e.g. files) are shown in blue, actions (e.g. saving a file, executing a simulation) are shown in red and agents (e.g. a user with the name "Carl") are shown in green. All these elements are linked with relations between them.

2.5 Traceability actions

The following diagram (Figure 3) shows a possible flowchart of the actions performed by different tools after a user action. After the user has started the INTO-CPS application (abbreviated as "app" in the figure), the daemon is instantiated. When the user opens a project, a local database is opened with the project-relevant traceability data. Then, the work in the different tools (e.g. 20-sim, Modelio, OpenModelica, Overture, RT Tester...) creates new traceability relations from activities. The relevant activites are listed in

[5]see `https://www.w3.org/TR/prov-n/`
[6]see `https://neo4j.com/`

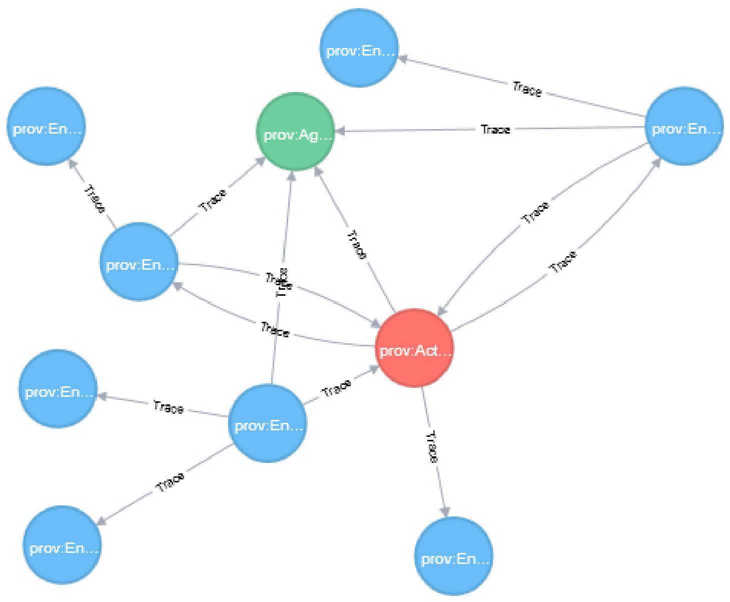

Figure 2: Visualisation of relations between different entities, actions and agents in Neo4J.

section 2.2.3. These traces are then sent trough the daemon to the database, where they are stored. In order to view and analyze traceability data, it is later retrieved from the INTO-CPS application, through the appropriate queries from the daemon to the database.

The PROV and OSLC relations that are mainly used in the context of INTO-CPS traceability, can be found in Deliverable D3.1b [FGPP15] (Annex 2.2 & 3.6).

2.6 Message structure and syntax

To implement the structure that is proposed in Deliverable D3.2b [FGPP16], the traceability data is structured in a set of activities, entities and agents and is encoded in a syntax that is described below in section 2.7. Initially, the data is exchanged in the JSON format, while other formats (e.g. RDF/XML, XML, Atom, Turtle) are also supported by OSLC. JSON (JavaScript Object Notation)[7] is a format to describe name/value pairs, and ordered lists of values (e.g. arrays).

[7]see `http://www.json.org`

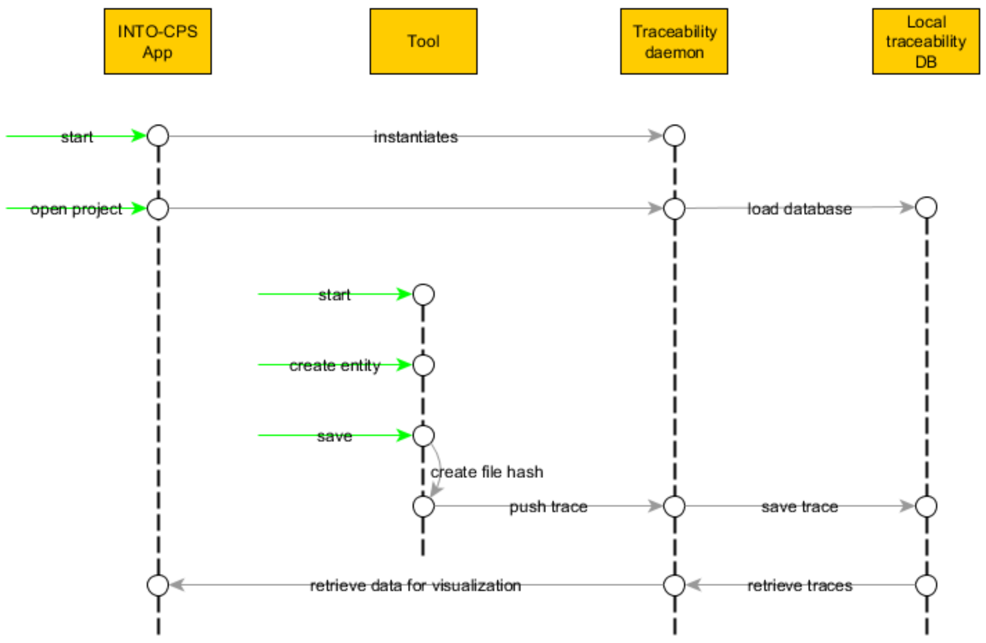

Figure 3: Flow-chart of actions in the different tools related to a traceability-relevant action.

2.7 Example messages

To illustrate the message format described above, we present in the following some examples of messages that are sent from the modelling tools to the traceability daemon. The message format as it is presented in the messages below is considered to be the first version (0.1) of the format, and it is possible that the format evolves in the third year of INTO-CPS.

We consider two people (PROV:AGENT), Carl and Richard, to be working on the project using the tool (PROV:ENTITY) Modelio. Each of these entries needs to have a RDF:ABOUT entry. Additional information is contained in other tags. This is graphically shown in the following figure 4.

In the traceability database these three nodes would be created by the following JSON message:

```
{
  "rdf:RDF": {
    "xmlns:rdf": "http://www.w3.org/1999/02/22-rdf-syntax-
        ns#",
    "xmlns:prov": "http://www.w3.org/ns/prov#",
```

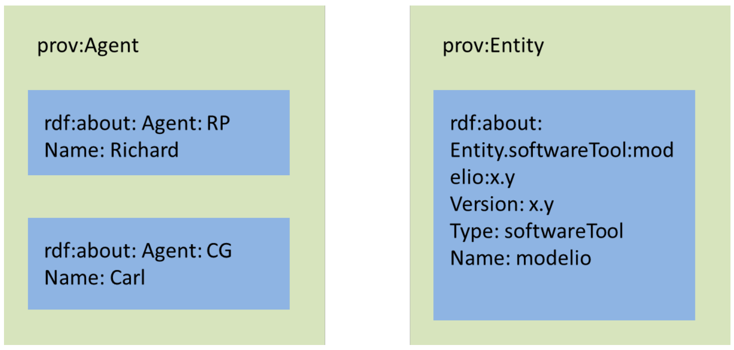

Figure 4: Example of messages that are sent to the traceability database.

```
      "messageFormatVersion": "0.1",
      "prov:Agent": [{
         "rdf:about": "Agent:RP",
         "name": "Richard"
      },
      {
         "rdf:about": "Agent:CG",
         "name": "Carl"
      }],
      "prov:Entity": {
         "rdf:about": "Entity.softwareTool:modelio:x.y",
         "version": "x.y",
         "type": "softwareTool",
         "name": "modelio"
      }
   }
}
```

It is important that every agent, entity (e.g. a tool or a file) and action that is meant to create a node in the traceability database has the property RDF:ABOUT which contains the unique URI (Uniform Resource Identifier). In order to later identify a node (agent, entity or action) the URI needs to be used, so a strict format for the URIs is necessary. The URI format is described below in section 2.8

Now consider for example the document "/sourceDocs/stakeHolderNeeds.pdf". This document contains two requirements in sections 1.2 and 1.3 and is attributed to the Richard. The connections between the nodes are created with the PROV:HADMEMBER and the PROV:WASATTRIBUTEDTO relations. This is graphically shown in the following figure 5.

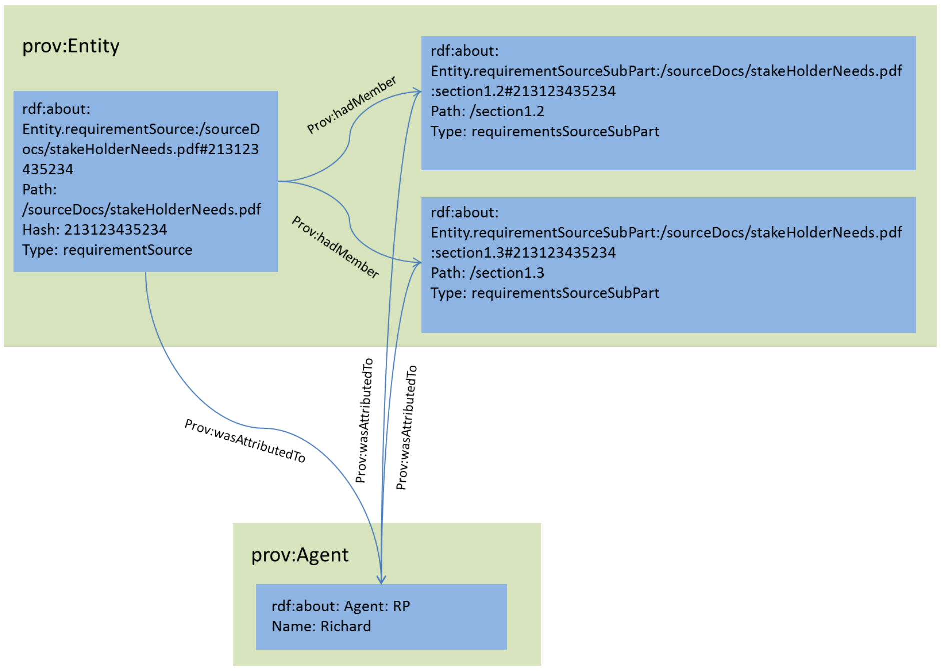

Figure 5: Example of messages that are sent to the traceability database, including relations between the nodes.

The following json message contains this information as well as the fact that the requirements in sections 1.2 and 1.3 are attributed to Richard too.

```
{
  "rdf:RDF": {
    "xmlns:rdf": "http://www.w3.org/1999/02/22-rdf-syntax-
       ns#",
    "xmlns:prov": "http://www.w3.org/ns/prov#",
    "messageFormatVersion": "0.1",
    "prov:Entity": [
      {
        "rdf:about": "Entity.requirementSource:/sourceDocs/
           stakeHolderNeeds.pdf#213123435234",
        "path": "/sourceDocs/stakeHolderNeeds.pdf",
        "hash": "213123435234",
        "type": "requirementSource",
        "prov:hadMember":
          {"prov:Entity": [
                {"rdf:about": "Entity.requirementSourceSubPart
                    :/sourceDocs/stakeHolderNeeds.pdf:section1
                    .2#213123435234"},
                {"rdf:about": "Entity.requirementSourceSubPart
                    :/sourceDocs/stakeHolderNeeds.pdf:section1
                    .3#213123435234"}
            ]},
            "prov:wasAttributedTo":
              {"prov:Agent": {"rdf:about": "Agent:RP"}}
        },
        {
            "rdf:about": "Entity.requirementSourceSubPart:/
                sourceDocs/stakeHolderNeeds.pdf:section1
                .2#213123435234",
            "path": "/section1.2",
            "type": "requirementSourceSubPart",
            "prov:wasAttributedTo": {"prov:Agent": {"rdf:about
                ":"Agent:RP"}}
        },
        {
            "rdf:about": "Entity.requirementSourceSubPart:/
                sourceDocs/stakeHolderNeeds.pdf:section1
                .3#213123435234",
            "path": "/section1.3",
            "type": "requirementSourceSubPart",
            "prov:wasAttributedTo": {"prov:Agent": {"rdf:about
                ":"Agent:RP"}}
        }
      ]
  }
}
```

Note that every object (such as PROV:HADMEMBER) of a potential node (such as PROV:ENTITY) is treated as a relation, if its value not a string but another json-level ({ ... {) containing another node (such as PROV:ENTITY).

The following example demonstrates the requirements management using Modelio. A requirement with two subrequirements R1 and R2 is generated by the activity requirementsManagement by Richard using Modelio and the original requirements contained in the document above. R1 and R2 thus elaborate the requirements formulated in section 1.2 and 1.3 of the document above.

```
{
```

```
"rdf:RDF": {
    "xmlns:rdf": "http://www.w3.org/1999/02/22-rdf-
      syntax-ns#",
    "xmlns:prov": "http://www.w3.org/ns/prov#",
    "messageFormatVersion": "0.1",
    "prov:Entity": [{
        "rdf:about": "Entity.requirements:/sysML/
          requirements#213123435235",
        "path": "/sysML/requirements",
        "hash": "213123435235",
        "type": "requirements",
        "prov:hadMember": {"prov:Entity": [
                {"rdf:about":"Entity.requirement:/sysML
                  /requirements:R1#213123435235"},
                {"rdf:about":"Entity.requirement:/sysML
                  /requirements:R2#213123435235"}
          ]},
        "prov:wasAttributedTo": {"prov:Agent": {"rdf:
          about":"Agent:RP"}},
        "prov:wasGeneratedBy": {"prov:Activity": {"rdf:
          about":"Activity.requirementsManagement
          :2016-09-19-13-53-06#b81f95c3-56aa-4189-baac
          -070631bd7957"}}
    }, {
        "rdf:about": "Entity.requirements:/sourceDocs/
          stakeHolderNeeds.pdf:R1#213123435235",
        "path": "/R1",
        "type": "requirement",
        "prov:wasAttributedTo": {"prov:Agent": {"rdf:
          about":"Agent:RP"}},
        "prov:wasGeneratedBy": {"prov:Activity": {"rdf:
          about":"Activity.requirementsManagement
          :2016-09-19-13-53-06#b81f95c3-56aa-4189-baac
          -070631bd7957"}},
        "oslc_elaborates": {"prov:Entity": {"rdf:about
          ":"Entity.requirementSourceSubPart:/
          sourceDocs/stakeHolderNeeds.pdf:section1
          .2#213123435234"}}
    }, {
        "rdf:about": "Entity.requirements:/sourceDocs/
          stakeHolderNeeds.pdf:R2#213123435235",
        "path": "/R2",
        "type": "requirement",
            "prov:wasAttributedTo": {"prov:Agent": {"rdf:
                about":"Agent:RP"}},
            "prov:wasGeneratedBy": {"prov:Activity": {"rdf:
                about":"Activity.requirementsManagement
                :2016-09-19-13-53-06#b81f95c3-56aa-4189-baac
                -070631bd7957"}},
            "oslc_elaborates": {"prov:Entity": {"rdf:about
                ":"Entity.requirementSourceSubPart:/
                sourceDocs/stakeHolderNeeds.pdf:section1
                .3#213123435234"}}
        }],
        "prov:Activity": {
            "rdf:about": "Activity.requirementsManagement
                :2016-09-19-13-53-06#b81f95c3-56aa-4189-baac
                -070631bd7957",
            "type": "requirementsManagement",
            "time": "2016-09-19-13-53-06",
            "guid": "b81f95c3-56aa-4189-baac-070631bd7957",
            "prov:wasAssociatedWith": {"prov:Agent": {"rdf:
                about":"Agent:RP"}},
            "prov:used": {"prov:Entity": [
                {"rdf:about":"Entity.softwareTool:modelio:x
                    .y"},
                {"rdf:about":"Entity.requirements:/
                    sourceDocs/stakeHolderNeeds.pdf:R1
                    #213123435235"},
                {"rdf:about":"Entity.requirements:/
                    sourceDocs/stakeHolderNeeds.pdf:R2
                    #213123435235"}
            ]}
        }
    }
}
```

2.8 Format of the URI

In the section above it is mentioned that a URI (the value in the RDF:ABOUT entry) must have a fixed structure such that it can be reconstructed later. We fix the structure of the URIs as follows:

1. The URI for an entity representing a submodel (Here <entity type>

can be something like requirement, requirementSource, ..):
Entity.<entity type>:<git relative path>:<subpart name>#<githash of the document>

2. The URI for an entity:
Entity.<entity type>:<git relative path>#<githash of the document>

3. The URI for an entity representing a tool:
Entity.softwareTool:<toolname>#<tool version>

4. The URI for an activity:
Activity.<activity type>:<time in format yyyy-mm-dd-hh-mm-ss>#<unique identifier for this activity such as a guid containing unique username, time (maybe computer ...)>

5. The URI for an agent:
Agent:<unique username>

The reason to use the git-hash of a document rather then the git-hash of the git revision to identify a file is the following: Using the git revision hash would create a new URI for every file in the project on a new commit, even if the file did not change and thus would give a new node in the graph. This node would have to be connected to the node of the document in the previous revision. This would create a large amount of nodes and edges in the database.

Since the traceability data should be able to link submodels, every submodel entity should automatically be linked to its parent model/modelfile.

3 Use cases

This section describes a sequence of actions that could be used during file creation, modification and destruction to create and record traceability information. These steps assume that the user has reached a significant point that is worthy of recording. They also assume at each step we must communicate the changes to the traceability database. Note that the support for git (or any other versioning system) is considered to be optional in this project.

3.1 Model Creation

The modeling process begins with the creation of a model file. The single steps occur in the modelling tool (e.g. Overture, 20-sim, OpenModelica). This can be performed in the following steps:

1. Modelling tool creates model file entity named model.extension.
2. Model file entity is committed to git. A tool records the git version number.
3. OSLC triples describing the activity are generated using the URIs.
4. OSLC triples are communicated to the traceability daemon.

3.2 Model Modification

Changes in the model also need to be communicated from the modelling tool to the traceability database.

1. Modelling tool updates model file entity.
2. Model file entity is committed to git. A tool records the git version number.
3. OSLC triples describing the activity are generated using the URIs in the metadata file and generating updated URI using the git version number obtained in step 2.
4. OSLC triples are communicated to the traceability daemon.

3.3 Model Destruction

Finally, the deletion of a model also needs to be communicated to the traceability database.

1. Modelling tool / file system deletes model file entity.
2. OSLC triples describing the activity are generated using the URIs.
3. OSLC triples are communicated to the traceability daemon.

4 Querying and Visualisation

In order to bring a benefit to the user, the traceability data not only needs to be recorded, but also analysed and presented in an way that is helpful to the user. The tools therefore must have a way of querying the database, through the traceability daemon, for specific information, such as relations between requirements, models or simulation results. While the implementation of the traceability functions is in progress at the end of year 2, a first list of queries is collected below:

- Query the database for all requirements that are related to a specific FMU.
- Query the database for the code files that are associated with a model.
- Query the database for all the Co-Simulation results that are associated with a multi-model.

The traceability information not only needs to be received from the database, but also needs to be presented to the user in a clear way, visualisation is also an important part. Here, it is logical to provide a visualisation function in the INTO-CPS application, while visualisation within the single tools can also be conceived. While it is not reasonable in large projects (with a large amount of traceability relations) to visualise all the traceability data, only certain relations of the traceability data, such as the ones listed above, should be displayed. Typically, two views are considered, the matrix view and the tree view.

The matrix view displays the relation between two sorts of traceability artifacts, such as requirements and test cases. Here, the matrix should display information, which requirements are tested in which test case, to ensure that all requirements are tested.

The tree view follows the relation between artifacts. An example of the tree view is shown above in Figure 2. In the case of INTO-CPS, this could for example be the relation between a requirement, its related high-level model (in SysML), the related FMU and generated simulation results or code. However, such tree views can become very extensive, and therefore filtering becomes necessary (showing for example only relations to the n-th level, or only showing relations to certain kinds of artifacts).

In year 3, the analysis and visualisation of the traceability information will be in the focus of the efforts.

5 Status of the tools at M24

In the following list a short overview of the status of the traceability implementation at M24 is given. In general, it can be said that most tools offer a prototype support for collecting traceability information (such as the ones described above in section 2.2.3) and sending them to the traceability daemon.

INTO-CPS application: No support for traceability yet.

Traceability daemon: A prototype of the daemon is working, writing traceability data to a database. It implements version 0.1 of the message format and supports simple queries (traces to, traces from).

Overture: The Overture tool has been extended with a traceability tool that is capable of extracting and submitting traceability information about model evolution, import of model descriptions and FMU exports. The tool can extract this information from an existing git repository going through the complete history or it can be connected to the git post_commit hook. The latter keeps the traceability daemon in sync with the repository during development. A full sync can always be performed to synchronise a git branch if changes are merged or other git operations alters the repository in a way where post_commit hook is not called.

20-sim: Detects user actions such as creating a new model, modifying a model, deleting a model, generating code (including FMU export) and importing FMI modelDescription.xml files. These actions are sent to the traceability daemon via external Python scripts. The supported traceability format is JSON (and where possible also RDF/XML). Submodels are addressed via a URI that is a combination of the hash of the overall 20-sim model and the unique path within the model to that submodel.

OpenModelica: Preliminary prototype to push traceability information to daemon implemented, work on queries and visualisation is ongoing.

Modelio: A prototype was implemented, supporting git and pushing test queries.

RT Tester: Support for OSCL Traceability has been implemented as a prototype in the VSI Tools. Traceability of Requirements to Test results is work in progress, details can be found in [PLM16].

6 Summary and Outlook

This deliverable presents the status of the traceability and model management efforts in INTO-CPS at the end of year 2. In summary, an architecture for traceability was presented, which allows the connected tools to send data to a central repository (i.e. the database). A daemon was created that receives this data and stores it. A format for the messages, containing the traceability data, was also defined.

For year 3, the focus of the work will be on the completion of the tool integration and on the data retrieval, i.e. the visualisation and presentation of the traceability data. Only then will the benefits of traceability become evident to the users. Therefore, the development efforts will be closely coupled to evaluation of the user's needs, primarily coming from the four case studies in Work Package 1 of INTO-CPS.

Appendices

A Used libraries

The table 2 below lists all the libraries that are used for the traceability daemon. Its purpose is to illustrate any potential licensing / open-source issues that might arise from conflicting licenses.

Library	Comment	License
typescript	Microsofts TypeScript, a super set to JavaScript, is used to ensure type-safeness during development and to benefit from its integration with the Visual Studio Code IDE.	Apache
node-js / restify	The restify library supports the definition of a RESTful web server in node.js. (Interface for the clients)	MIT
node-js / bluebird	The bluebird library is used to syncronize parts of program execution while retaining source code readability (escaping callback hell).	MIT
node-js / neo4j	The neo4j library offers an interface to the neo4j database server.	Apache 2.0
node-js / xml2js	The xml2js library is a xml document parser for node.js we use to parse rdf/xml messages.	MIT
Neo4J Graph DB	The neo4j graph database (community edition) is used to store the traceability information.	GPL / commercial
Elektron	Elektron is the framework for the INTO-CPS application	MIT

Table 2: Libraries used for the traceability daemon

It should be noted that the Neo4J Graph database offers two licenses: the GPL for the community edition, and a commercial license for the enterprise edition. For more information, see `https://neo4j.com/licensing/`

B Abbreviations

BPMN	Business Process Model and Notation
COE	Co-Simulation Orchestration Engine
CPS	Cyber-Physical System
DSE	Design Space Exploration
FMI	Functional Mockup Interface
FMU	Functional Mockup Unit
HiL	Hardware in the Loop
JSON	JavaScript Object Notation
OSLC	Open Services for Lifecycle Collaboration
RDF	Resource Description Framework
SVN	Apache Subversion
UML	Unified Markup Language
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

References

[BLL+16] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Casper Thule, Anders Franz Terkelsen, Carl Gamble, Adrian Pop, Etienne Brosse, Jrg Brauer, Florian Lapschies, Marcel Groothuis, Christian Kleijn, and Luis Diogo Couto. INTO-CPS Tool Chain User Manual. Technical report, INTO-CPS Deliverable, D4.2a, December 2016.

[EN13] Maged Elaasar and Adam Neal. *Integrating Modeling Tools in the Development Lifecycle with OSLC: A Case Study*, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[FGPP15] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 1. Technical report, INTO-CPS Deliverable, D3.1b, December 2015.

[FGPP16] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 2. Technical report, INTO-CPS Deliverable, D3.2b, December 2016.

[GF94] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, April 1994.

[HWW15] Ron Henkel, Olaf Wolkenhauer, and Dagmar Waltemath. Combining computational models, semantic annotations and simulation experiments in a graph database. *Database*, 2015, 2015.

[LPO+16] Peter Gorm Larsen, Ken Pierce, Julien Ouy, Kenneth Lausdahl, Marcel Groothuis, Adrian Pop, Miran Hasanagic, Jörg Brauer, Etienne Brosse, Carl Gamble, Simon Foster, and Jim Woodcock. Requirements Report Year 2. Technical report, INTO-CPS Deliverable, D7.5, December 2016.

[PLM16] Adrian Pop, Florian Lapschies, and Oliver Möller. Test automation module in the INTO-CPS Platform. Technical report, INTO-CPS Deliverable, D5.2a, December 2016.

[PSA+14] Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. Integrated Debugging of Modelica Models. *Modeling, Identification and Control*, 35(2):93–107, 2014.

[Ver15a] Verified Systems International GmbH, Bremen, Germany. *RT-Tester 6.0: User Manual*, 2015. `https://www.verified.de/products/rt-tester/`, Doc. Id. Verified-INT-014-2003.

[Ver15b] Verified Systems International GmbH, Bremen, Germany. *RT-Tester Model-Based Test Case and Test Data Generator – RTT-MBT: User Manual*, 2015. `https://www.verified.de/products/model-based-testing/`, Doc. Id. Verified-INT-003-2012.