The Beauty of MBSE
Reducing Efforts by extending the Scope

Florian **Beer**
Robert Bosch GmbH, Stuttgart, Germany
Contact: florian.beer@bosch.com

Abstract

Model-based system and software engineering aims to provide a single source of truth for engineers, facilitating alignment between partners and disciplines. By formalizing information in models, it becomes machine-readable and persistent.

However, in many cases, MBSE is seen as a hurdle by development teams. The introduction of MBSE often comes with "big-upfront" ideas driven by architects, which can hinder the innovation process. Adopting agile approaches with pull-request-based iterations and quality gates with automated verification builds connection points to the experience of code-centric engineers.

To demonstrate the benefits of MBSE without imposing large-scale rollouts, we provide an example in the context of safety-relevant products. AbRA extends existing standard models with a meta-element for failure modes, the development team can perform safety analysis without the need for extensive post-development workshops. This approach simplifies the process and shifts the workload from analysis workshops to the development teams. It also promotes competence within the team, reduces efforts, and improves risk mitigation.

By adding a package-based architecture exchange, architects from different organizations can work on one product. The relevant parts of model for the next integration level can be easily shared while the design details, i.e. of safety analysis can be kept confidential.

Contents

Architecture-based Risk Analysis

Normative Requirements towards Safety Analysis

If we follow the established approaches for safety analysis, we usually have 3 architectures:

- The real architecture of the implemented product
- The documented architecture of the product, e. g. the system and software architecture documentation
- The analyzed architecture, e. g. in the FMEA documentation

As these three architectures in the real word often have differences, for ASIL-C and ASIL-D not only inductive but also a deductive analysis is required. As the inductive and deductive way of thinking help to detect weaknesses in the design, there is no requirement to maintain the analysis in different documents.

The core of AbRA

The goal in designing AbRA has been to facilitate risk analysis directly in the architecture model. The concept of cause-effect-chains has been taken over from the classical approaches. But as the role Fault/Error/Failure is regularly depending on the element under analysis, only one new element type is the meta-model has been introduced. To prevent failure propagation, occurrence can be either prevented by measures in advance or the negative effect be controlled after the initial fault occurred.

For prioritizing derived actions and judging if the residual risk is acceptable, the severity of the failure, the probability of occurrence and the confidence in the taken measures have to be taken into account.

This leads to the simplified meta-model for AbRA as shown in Fig. 1.

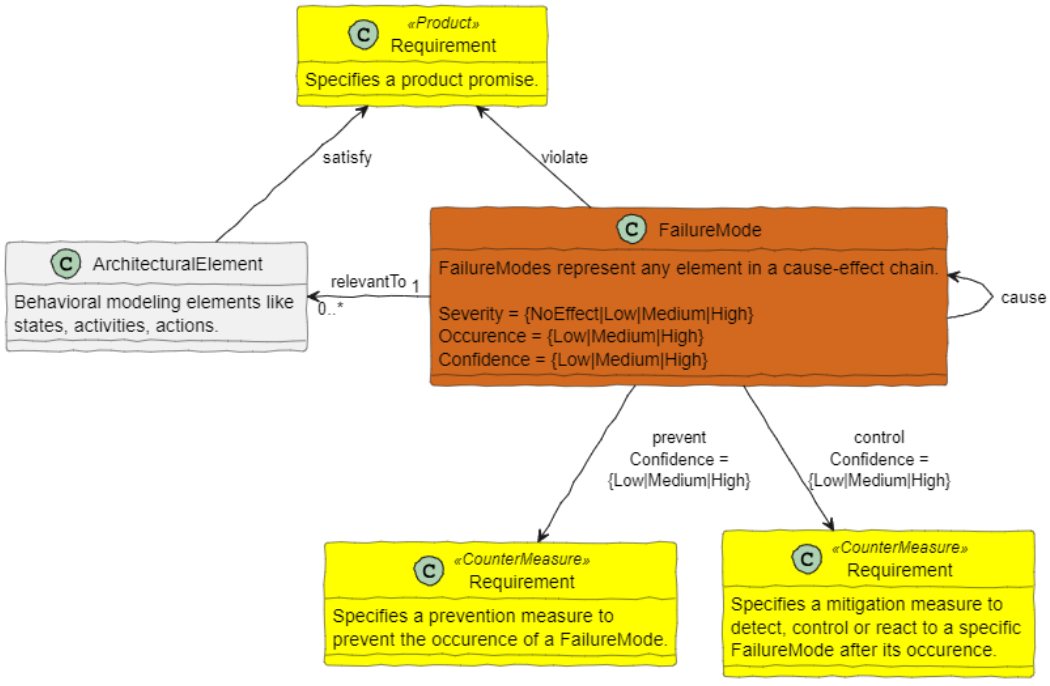

Fig. 1 Simplified meta-model for AbRA

AbRA as UML Meta-Model extension

To provide AbRA to architects, the simplified meta-model must be implemented in an UML conformant way as shown in Fig. 2, Fig. 3, and Fig. 4. This meta-model can be applied to any UML-compliant modeling tool.

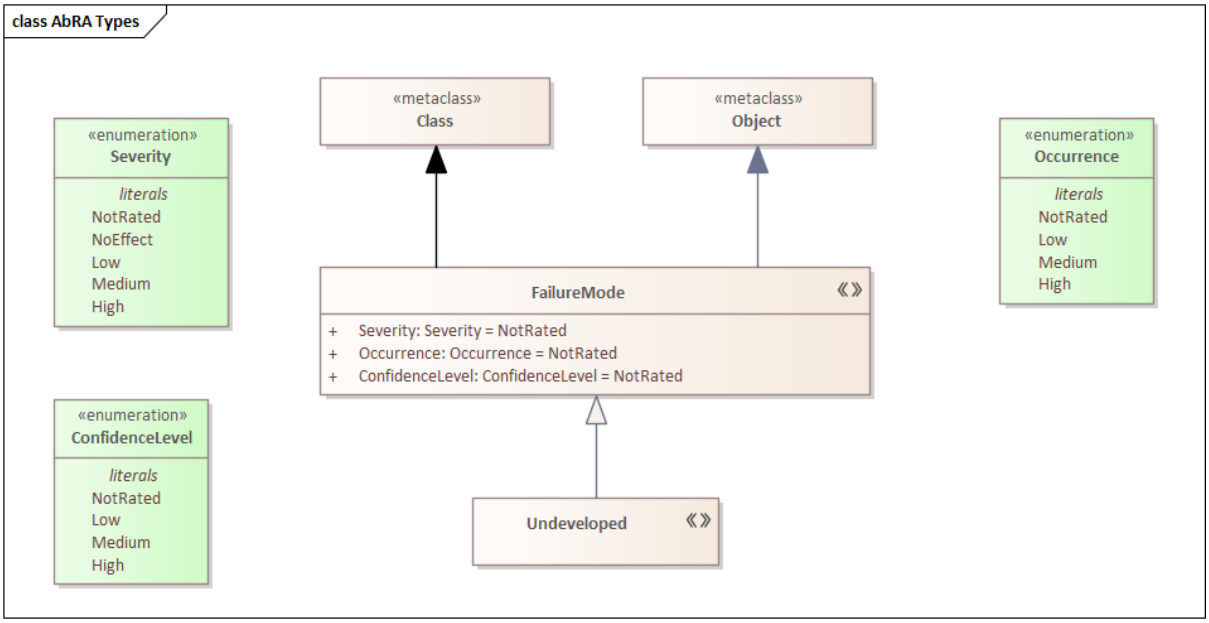

Fig. 2 AbRA meta-model elements

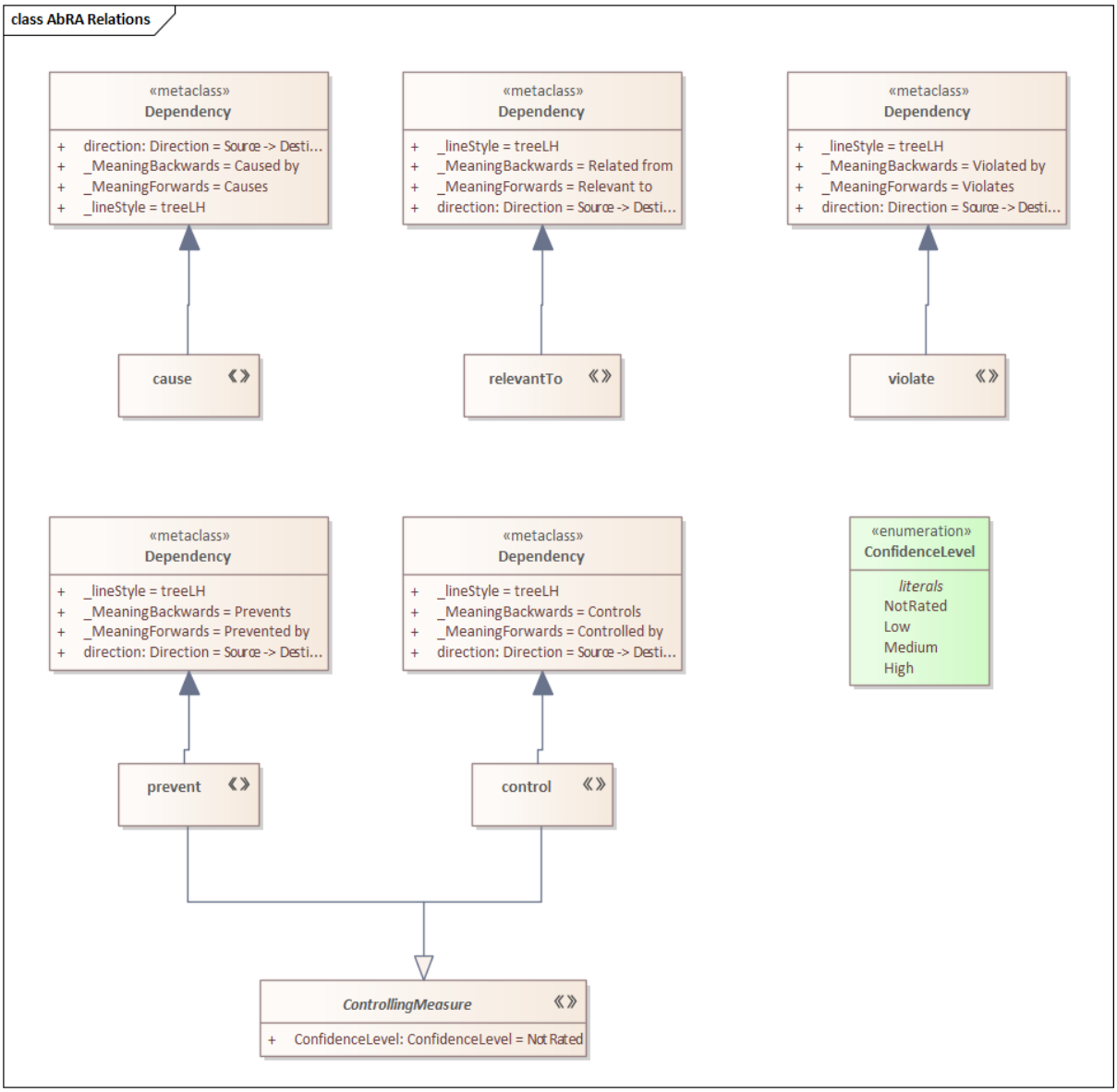

Fig. 3 AbRA meta-model relations

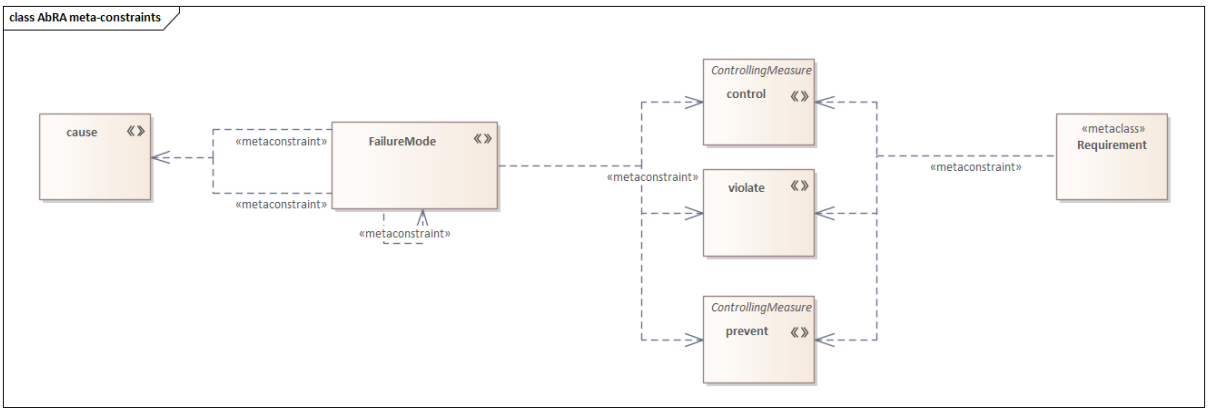

Fig. 4 Meta-constraints for AbRA relations

Tool-Box implementation in Enterprise Architect

To increase the usability, the best way is to create a toolbox and define instantiation rules to be applied by the software automatically. By defining diagram types (Fig. 5), quick-linker information (Fig. 6) and a toolbox (Fig. 7), the elements can be used directly in the working flow.

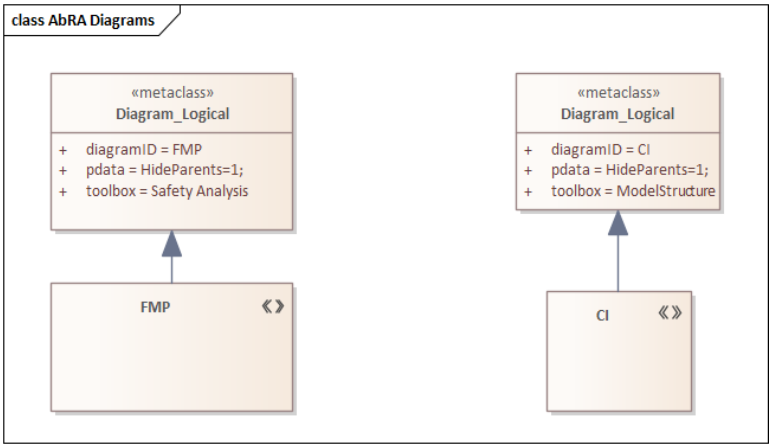

Fig. 5 Diagram type definitions

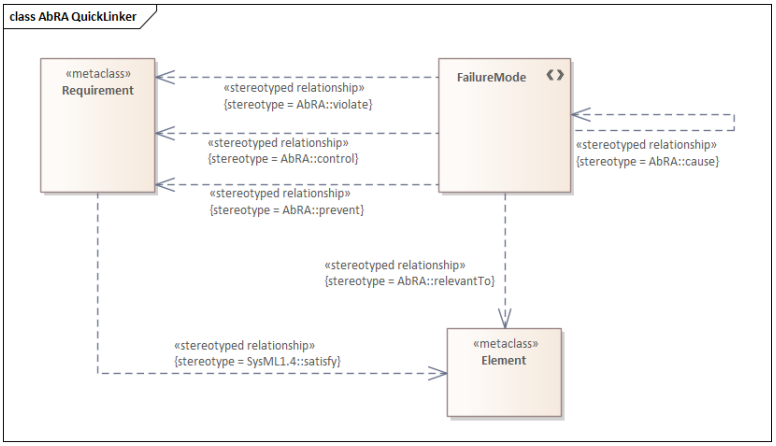

Fig. 6 Quick-Linker relations for AbRA

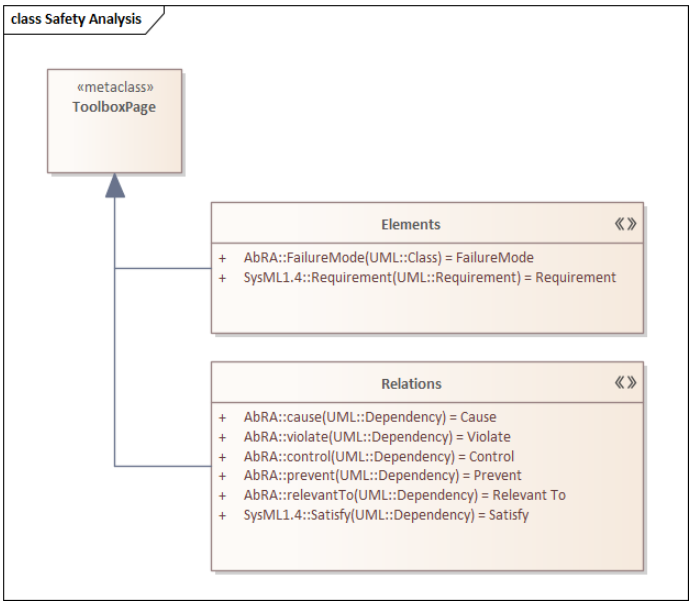

Fig. 7 AbRA toolbox definition

Modelling Concept

As shown in Fig. 8, failure modes have different roles. Failure modes, which are relevant to the user of an element are referred as public failures. Failure modes originating from single steps are referred as internal causes. When instantiating an element, its public failures become internal failures as shown in Fig. 9. To make the cause-effect-chain more readable, internal failures can be introduced as intermediate steps as shown in Fig. 10.

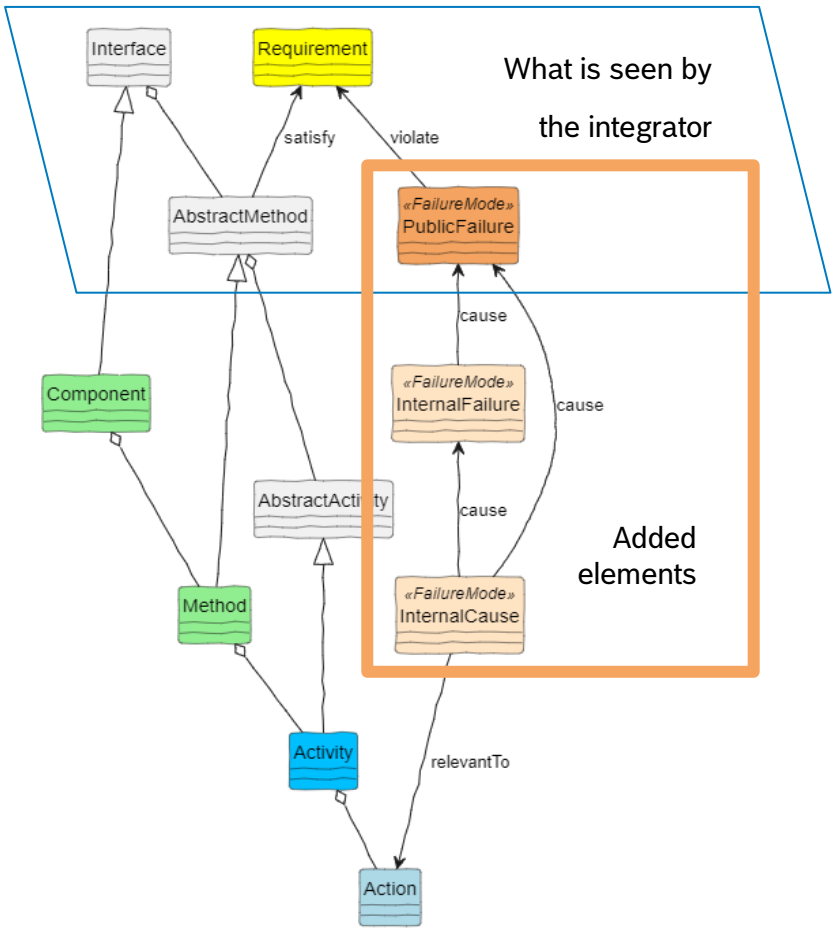

Fig. 8 Concept of FailureModes

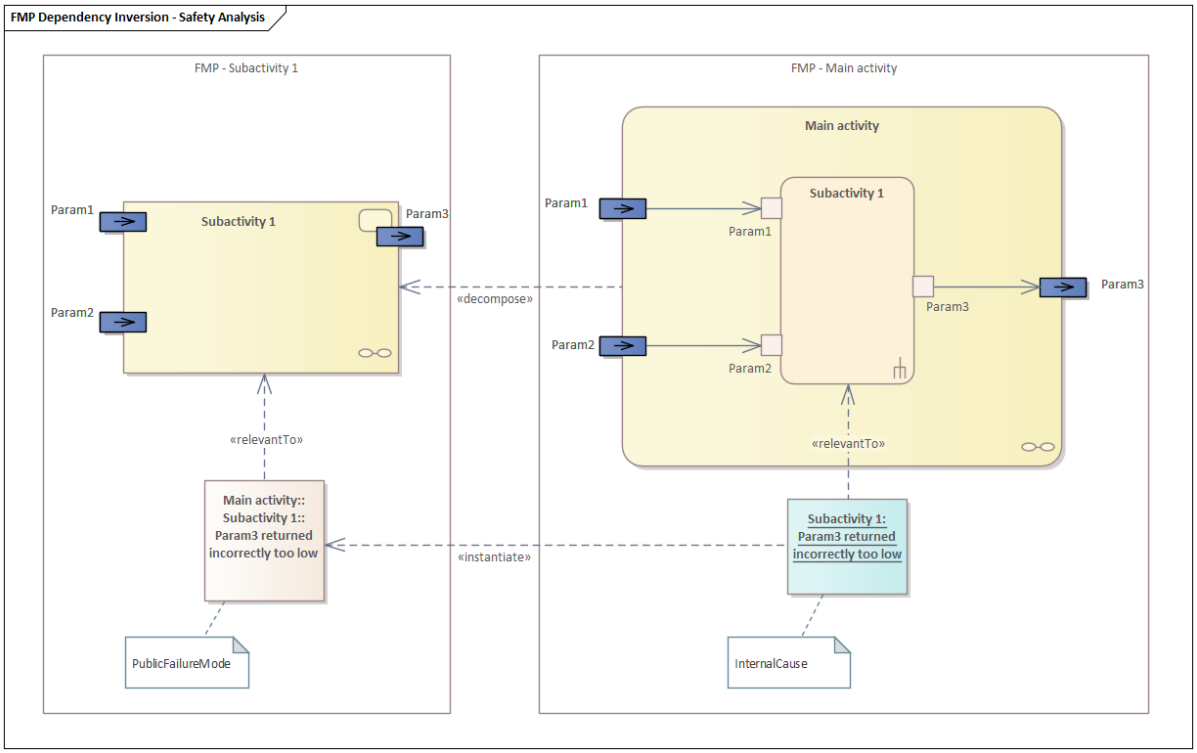

Fig. 9 Instantiation of <<FailureMode>>

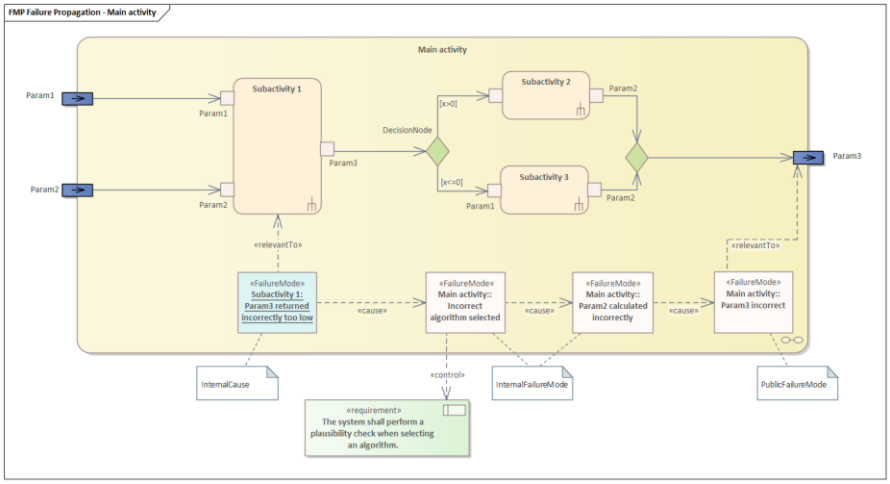

Fig. 10 Effect chain from internal cause to public failure mode.

SysMLv2 definition

The following snippet shows, how AbRA can be implemented in SysMLv2.

```
package AbRA_simple_demo {
    import Metaobjects::SemanticMetadata;
    import ScalarValues::*;

    enum def Rating {
      low; medium; high; notRated; noEffect;
    }

    occurrence def FailureMode {
            attribute sev: Rating; //sev = severity
            attribute occ: Rating; //occ = occurrence
            attribute cl[0..1]: Rating; //cl = confidenceLevel
        }

     abstract occurrence failureModes : FailureMode[*] nonunique;

     metadata def <FM> FailureModeMetadata :> SemanticMetadata {
            :>> baseType = failureModes meta SysML::Usage;
        }

     connection def Cause :> Occurrences::HappensBefore {
            end fm_start[*]: FailureMode;
            end fm_end[*]: FailureMode;
        }

     abstract connection causes : Cause[*] nonunique;

     metadata def <cause> CausationMetadata :> SemanticMetadata {
            :>> annotatedElement : SysML::SuccessionAsUsage;
            :>> baseType = causes meta SysML::Usage;
        }

   //Include Failure Modes and references to the actions
   #FM occurrence 'Delayed Request' {
       :>> sev = Rating::high;
       :>> occ = Rating::low;
       :>> cl = Rating::medium;
        ref relevantTo:RequestDrivingGearChange;
     }

    //Include Failure Modes and references to the actions
   #FM occurrence 'Delayed Engagement' {
       :>> sev = Rating::high;
       :>> occ = Rating::low;
       :>> cl = Rating::medium;
        ref relevantTo:EngageDrivingGearRND;
     }

    // now model the failure propagation ... first ... then
    #cause first 'Delayed Request' then 'Delayed Engagement';
}
```

Open-Source repository for AbRA

To facilitate adoption of AbRA by the engineering community, we created a repository under the umbrella of Open-MBEE. We want this repository become a living exchange for AbRA and encourage

everyone to try it out and contribute.
https://github.com/Open-MBEE/architecture-based-risk-analysis

Package-based Collaboration

To setup a package-based exchange with Github, for all provided components and the consuming architectures, according conan recipes and github workflows have to be created. The examples assume, that the workflows are performed on a runner, which has Python 3.x, Conan 2.0 and LemonTree.Automation installed.

With this configuration, each component provider can publish a new version of its package by starting the workflow manually. Every architect with access to the repository can retrieve these updates with starting the workflow. No local installation of python, conan or maintenance of scripts by the architects is required. With slight modifications, the workflow can be executed on public, ubuntu-based runners in the cloud. In that case, even no local runner is required.

Conanfile for packages

The architect of the component has to maintain name and version. LemonTree should be configured to use the relative folder *modelCache* as repository.

```
from conan import ConanFile
from conan.tools.files import copy

class ArchitectureBuildingBlockRecipe(ConanFile):
    name = "some-component"
    version = "1.0"

    exports_sources = "modelCache/*"

    def package(self):
        copy(self, "modelCache/*", self.source_folder,
self.package_folder)
```

Conanfile for product architecture

The product architect has to maintain the list of required packages. As long as the directory *modelCache* exists, the MPMS files will be provided to this location.

```
import os
import shutil
from conan import ConanFile
from conan.tools.files import copy

class SampleProjectConan(ConanFile):
    name = "sample_project"
    version = "1.0"

    def requirements(self):
        self.requires("some-component/1.0")

    def generate(self):
        shutil.rmtree("modelCache", ignore_errors=True)
        for require, dependency in self.dependencies.items():
            self.copy_model_files(dependency.package_folder)

    def copy_model_files(self, src_folder):
        src_folder = os.path.join(src_folder, "modelCache")
        dest_folder = os.path.join(self.source_folder, "modelCache")
        shutil.copytree(src_folder, dest_folder, dirs_exist_ok=True)
```

Workflow for delivery

To use the workflow, the action for LTA has to be correctly set and the conan repository *model_repository* has to be setup correctly. To prevent leaking of IP, we strongly recommend using a non-public repository with authentication.

```yaml
name: Create Conan Package

on:
  workflow_dispatch:

jobs:
  build:
    runs-on: [conan-lta]

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
      with:
        lfs: true
        fetch-depth: 0

    - name: Export MPMS package
      uses: LemonTree.Automation@main
      with:
          Task: MpmsExport
          Mine: Model.qeax
          OutputFile: modelCache
          License: ${{secrets.LTALICENSE}}

    - name: Create package for some-component
      run: |
        conan create ./
        conan upload some-component -r=model_repository
```

Workflow for composition

The following workflow installs the packages defined by the recipe into the modelCache. The *update_packages.sh* script adds/removes changed files in the modelCache to the git repository, creates a new commit and pushes the changes to the repository.

```yaml
name: Install Conan Packages

on:
  workflow_dispatch:

jobs:
  build:
    runs-on: [conan]

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
      with:
        lfs: true
        fetch-depth: 0
        token: ${{ secrets.GITHUBTOKEN }}

    - name: Install packages
      run: |
        conan install . -r=repo_cache

    - name: Add changed files to git and push commit
      run: |
        ./tools/update_packages.sh
```