Project Number 318772

D7.1 Industrial Evaluation: fortiss Smart Grid

Version 1.0
31 October 2015
Final

Public Distribution

fortiss GmbH

Project Partners: **Fondazione Bruno Kessler**, **fortiss**, **Frequentis**, **Inria**, **LynuxWorks**, **The Open Group**, **RWTH Aachen University**, **TTTech**, **Université Joseph Fourier**, **University of York**

Project Partner Contact Information

Fondazione Bruno Kessler Alessandro Cimatti Via Sommarive 18 38123 Trento, Italy Tel: +39 0461 314320 Fax: +39 0461 314591 E-mail: cimatti@fbk.eu	**fortiss** Harald Ruess Guerickestrasse 25 80805 Munich, Germany Tel: +49 89 36035 22 0 Fax: +49 89 36035 22 50 E-mail: ruess@fortiss.org
Frequentis Wolfgang Kampichler Innovationsstrasse 1 1100 Vienna, Austria Tel: +43 664 60 850 2775 Fax: +43 1 811 50 77 2775 E-mail: wolfgang.kampichler@frequentis.com	**LynuxWorks** Yuri Bakalov Rue Pierre Curie 38 78210 Saint-Cyr-l'Ecole, France Tel: +33 1 30 85 06 00 Fax: +33 1 30 85 06 06 E-mail: ybakalov@lnxw.com
RWTH Aachen University Joost-Pieter Katoen Ahornstrasse 55 D-52074 Aachen, Germany Tel: +49 241 8021200 Fax: +49 241 8022217 E-mail: katoen@cs.rwth-aachen.de	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 894 5845 E-mail: s.hansen@opengroup.org
TTTech Wilfried Steiner Schonbrunner Strasse 7 1040 Vienna, Austria Tel: +43 1 5853434 983 Fax: +43 1 585 65 38 5090 E-mail: wilfried.steiner@tttech.com	**Université Joseph Fourier** Saddek Bensalem Avenue de Vignate 2 38610 Gieres, France Tel: +33 4 56 52 03 71 Fax: +33 4 56 03 44 E-mail: saddek.bensalem@imag.fr
University of York Tim Kelly Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325477 Fax: +44 7976 889 545 E-mail: tim.kelly@cs.york.ac.uk	**Inria** Axel Legay Inria - Campus de Beaulieu 35042 Rennes, France Tel: +33 2 99 84 73 15 Fax: +33 2 99 84 71 71 E-mail: axel.legay@inria.fr

Contents

List of Figures

Document Control

Version	Status	Date
0.1	Initial outline	16 March 2015
0.2	Revised Document Structure	20 April 2015
0.3	Development Process and Functional Description sections added	12 May 2015
0.4	Design Specification, Verification and Assurance Case sections added	1 June
0.5	Deployment, Platform and Requirement sections added	12 July
0.6	Internal Review	3 September
0.7	External Review	6 October
1.0	Final version	31 October

1 Summary

This document summarizes our experience and analysis of the D-MILS architectural approach and the corresponding technology for the design and implementation of dependable and secure information and communication systems. This analysis is intended to provide feedback to the development partners and quantitative results on which to base dissemination messages as the project transitions to the exploitation phase.

The guiding example of our analysis is the smart microgrid living lab at fortiss, which has been developed, in collaboration with Siemens AG, over the last 6 years for demonstrating and show-casing novel architectures, programming concepts, architectures, and novel products and services for smart microgrids. The fortiss smart grid living lab has originally not been designed with respect to dependability and privacy aspects in mind. An adequate and cost-effective design and implementation solution for these pressing issues, however, is a prerequisite for market entry for this technology.

We have therefore modeled the high-level architecture of the fortiss smart microgrid in MILS-AADL and implemented it using D-MILS technology as developed, largely, by the consortium partners. In particular, the communication back-bone of the smart microgrid is replaced with the reliable communication infrastructure of D-MILS, and space partitioning of the resulting *distributed separation kernel* of D-MILS ensures that the impact of faults and of malicious attack is localized. Using parts of this design we analyzed the D-MILS technology:

- MILS-AADL architectural specification language
- Verification of dependability properties
- Generation of assurance cases
- Automated MILS platform configuration compiler
- MILS technical platform

as provided by the D-MILS partners by means of stepping through the D-MILS architectural design and implementation workflow, according to the supplemental evaluation plan.

In particular, MILS-AADL proved to be a suitable architectural specification language for our purposes with a well-defined semantics. In order to promote wide-spread industrial, however, it is recommended to provide tutorial introductions and examples of MILS-AADL language features and their interaction (e.g. on the use of error models).

The range of verification techniques available in the MILS-AADL front-end is impressive indeed, as it includes a number of world-class verification engines such as nuXsmv (model checking, bounded model checking, IC3) and latest component-based verification technology as embodied in OCRA. Temporal specifications and contracts are embedded in the MILS-AADL language. After setting up formal verification conditions for the smart microgrid corresponding to the requirements in D1.1, the verification engines could prove these conditions. However, fairness assumptions need to be made much more explicit, as these constraints are embodied in the transition to back-end verification tools.

Automated construction of assurance cases by means of suitable patterns (e.g. component-based reasoning) and application-specific instantiation based on a MILS-AADL specification is an important innovation, and showed significant potential for certification support in our analysis. In addition, assurance cases could also be used as the central design artefact for driving the design and supporting

design decisions during development. Logical specifications should be an integral part of the MILS-AADL language; this may evolve extending signatures with logical specifications (cmp. Extended ML) and proving, say, refinement relations, by means of the OCRA verification tool of D-MILS.

The D-MILS technical platform consists of Level 0 hypervisors by LynuxWorks for each multi-processor node and a TTEthernet communication network. The main properties used for the smart microgrid are that of space- and time-partitioning in order to avoid uncontrolled spreading of faults and attacks on parts of the distributed system. Even though the (hard) real-time aspects of TTEthernet have not been particularly stressed by our case study, this feature of the MILS technical platform is of utmost importance in more time-critical control applications; e.g. for the tight coordination of machines on one or several production floors.

An initial set-up and configuration of the D-MILS technical platform took more than one person month. The MILS platform configuration compiler automates this task of generating application-specific configurations form the MILS-AADL architectural description. The MILS platform configuration compiler is therefore the centerpiece of technology towards industrial up-take of D-MILS in industrial products and services. Moreover, the MILS platform configuration compiler supports an intermediate format of MILS configurations, which allows to support additional MILS technical components. This intermediate format should be standardized in order to encourage a market place of pre-certified MILS technology components such as separation kernels, MILS network system, and deterministic networking capabilities.

Altogether the D-MILS architectural design and implementation tool chain together with its platform lives up to its promise to be able to automate crucial steps in the development of dependable and secure systems. However, the applicability of D-MILS technological platform is currently restricted to ultra-dependable distributed control systems, as found, for example, in airplanes and other vehicles. Applicability and wide-spread uptake of the D-MILS technology, however, would be greatly increased by also considering dynamic and adaptive systems, in which subjects are added, deleted, or changed at run-time, and security policy architectures are changed dynamically.

Altogether, for wide-spread industrial up-take and increase of applicability - e.g. in the automation, energy, and IoT domain - we recommend as follow-up steps:

- Extension of D-MILS technology to dynamic and adaptable systems in order to considerably increase applicability of this technology;
- Wrapping D-MILS architectural design and implementation technology in terms of a well-defined and well-documented set of services (e.g. simulate this model wrt. the following input, verify this property for this model, generate a suitable configuration); this allows industrial users to seamlessly integrate D-MILS into existing model-based development chains;
- Develop specific tutorials for industrial use of D-MILS technology in various domains;
- Standardize interface of D-MILS technological platform, and start populating a marketplace of pre-certified D-MILS technology platform components and COTS software for applications with varying dependability and security requirements.

2 Introduction

This deliverable includes the assessment of the performance of the *Distributed MILS* for Dependable Information and Communication Infrastructures (D-MILS) project in achieving the established industrial objectives by presenting the results of the evaluation of the technology developed by the D-MILS project used in the context of the fortiss smart microgrid industrial demonstrator. This deliverable is intended to provide feedback to the development partners and quantitative results on which to base dissemination messages as the project transitions to the exploitation phase.

D-MILS provides several technologies components such as extensions to the MILS platform including distributed system configuration features and a network subsystem that employs a hardware-based time-triggered ethernet "backplane". The resulting D-MILS deployment platform will make it possible for an application architecture to seamlessly span multiple computer systems, with scalable deterministic operation. Automated assistance is indispensable for the development and verification of dependable distributed systems. Accordingly, D-MILS provides end-to-end and top-to-bottom automated support for the development and certification of highly-dependable systems. Based on the SAE standard Architecture Analysis and Design Language (AADL), D-MILS defines MILS-AADL, a high-level declarative language, and a chain of machine-processable representations within a powerful verification framework to perform sophisticated analyses of probabilistic and non-probabilistic properties in both finite and infinite-state systems. Using Goal Structuring Notation (GSN), to represent the assurance case for a system, D-MILS enables a concrete and automated linkage between the assurance activities performed at various stages and levels of the specification, design and implementation, and the high-level claims made for the complete D-MILS system. D-MILS culminates with the automated compilation of detailed resource allocation, scheduling, and interaction policy configurations of a collection of single- and multi-processor D-MILS platform nodes.

The scope of this deliverable is to provide the description and results of the evaluation and validation process of these D-MILS technology components by integration of the D-MILS components by means of our industrial case study. The evaluation process has been guided by a set of evaluation measures and criteria identified by the D-MILS consortium. Thus, our evaluation process is in the context of the industrial assessment of D-MILS components with regard to dependability and security, timeliness, integrability, usability, and certifiability. The results of our evaluation are intended to provide needed feedback to the development partners and also industrial based figures that can be used for promotion and dissemination of project results.

In order to demonstrate the feasibility of D-MILS technologies, we evaluated them by applying them to an industrial demonstrator. The fortiss smart microgrid industrial demonstrator for energy-efficient workplaces has been built up over recent years by fortiss with technical support from Siemens and sponsored by the Bavarian Ministry of Economics. The goals were to develop, study, validate, and demonstrate the technical feasibility of distributed control for ultra-dependable self-balancing micro grid nodes in a laboratory environment. Further details on this case study given by the fortiss smart micro demonstrator such as system description, its business context together with expected improvements offered by the D-MILS technologies and related safety and security requirements are presented in deliverable D 1.1. As a D-MILS demonstration, the fortiss smart microgrid case study identified and analyzed safety and security requirements, developed a security architecture, constructed an as-

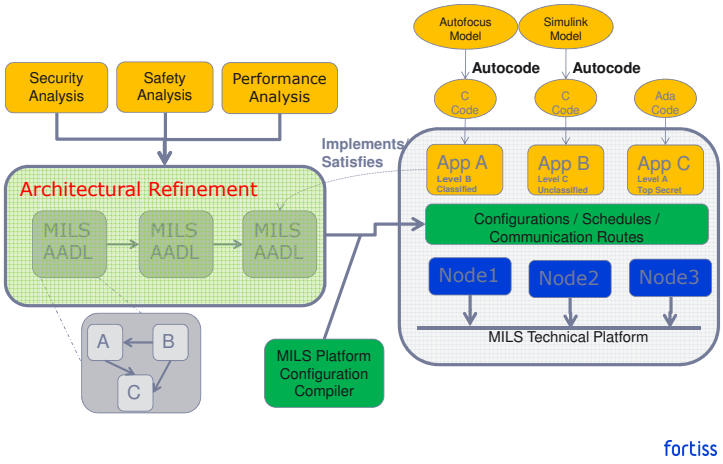

Figure 1: MILS Architectural Design and Deployment

surance case and provided evidence backed by verification results in support of the assurance case. All these have been evaluated and the evaluation results are enclosed in this document.

We first present an overview of the MILS architectural approach to the design and deployment of dependable and secure systems, as depicted in Figure 1. This design flow starts with a *boxes-and-arrows*-like abstract architectural description in MILS-AADL, the so-called *(security) policy architecture*, which has been obtained by an appropriate security analysis of the application at-hand. Using the MILS verification technology, it is possible to state a large number of safety, security, and also performance properties in terms of temporal logic, and to verify these properties with respect to the architectural description in MILS-AADL. Architectural descriptions in MILS-AADL may be refined through property-preserving transformation. In this way, properties can always be established on abstract architectural descriptions, and these properties are preserved by means of architectural transformation and concretization (e.g. by adding behavior or by introducing sub-components). In Figure 1 we assume that concrete implementations of components are being obtained through autocoding in established model-based techniques (such as Simulink or Autofocus), are hand-coded (e.g. in a programming language such as C, Ada), or are commercial component-off-the-shelf (COTS) products. In this way, one needs to demonstrate that the implementation thus obtained indeed implements the corresponding component interface and behavior. In the case of hand-coded component code this amount to a software verification task, whereas model-based techniques might involve the use correctness arguments of the autocoder, and COTS components need to be tested as a black-box component. Each application component may have, among other things, a safety and/or a security attribute (e.g. safety integrity level) as specified, for example, in domain-specific safety standards such as DO 178C for flight-critical aerospace software, the ISO 262626 for safety-relevant automotive functionalities, or a security standard such as the Common Criteria (ISO/IEC 15408).

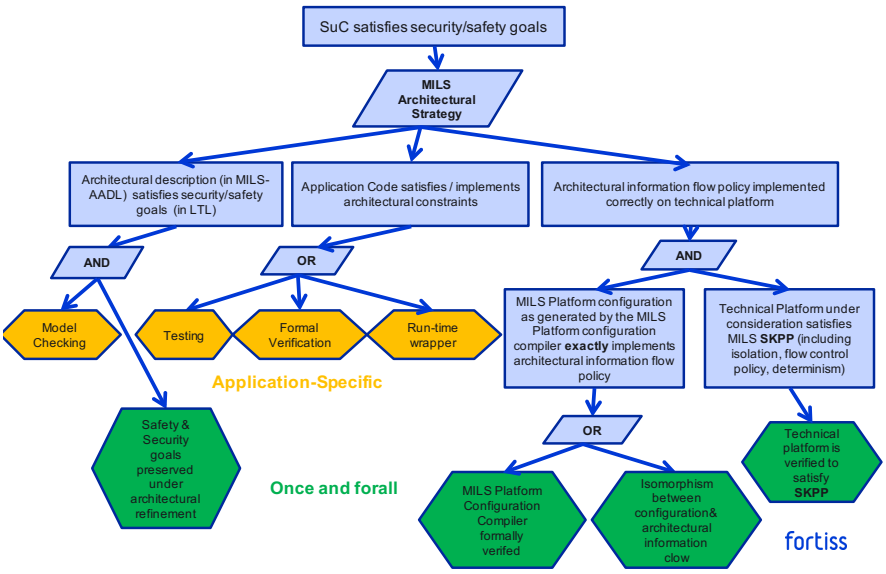

Figure 2: MILS Architectural Strategy

Given an architectural description in MILS-AADL (and some established properties on this model) and a number of application components together with additional constraints and attributes, the *MILS platform configuration compiler* generates an appropriate configuration for the distributed MILS technical platform. In this way, a single *policy architecture* may be span several MILS nodes.

Moreover, the MILS distributed platform provides, similar to a single MILS separation kernel, properties such as *isolation*, *information flow control*, and *determinism*. In particular, no additional flows ("hidden channels") are introduced by the MILS technical platform, and, because of isolation, faults and attacks may be contained locally.

In applying MILS architectural design and implementation workflow, the application-developer may concentrate on application-specific tasks only (colored orange in Fig. 1). This has the potential of leading to much cheaper development of highly dependable and secure systems as well as faster time-to-market. In particular, the MILS architectural strategy in Figure 2 for developing a GSN-like assurance case demonstrates the potential savings of using the MILS technology and infrastructure as developed in this project. The task of demonstrating dependability and security goals (as obtained from an appropriate risk analysis, not shown here) are satisfied, and reduces to:

1. Architectural description in MILS-AADL satisfies dependability and security goals (expressed in LTL, contracts, etc.)

2. Application code implements interface and behavior of corresponding architectural component

3. Architectural (information flow) policy is implemented correctly on the MILS technical platform

1. The user writes manually the system architecture in MILS-AADL using a text editor.
2. The user extends manually the MILS-AADL model with component state machines using a text editor.
3. The user simulates the MILS-AADL model with COMPASS.
4. The user starts the assurance case by instantiating the argument patterns with the MILS-AADL model.
5. The user extends manually the MILS-AADL model with annotations for:
   (a) Deadlock checking with BIP
   (b) Invariant checking with either BIP or nuXmv
   (c) Transitive noninterference properties with BIP
   (d) LTL checking with nuXmv
   (e) Contract-based LTL checking with OCRA and nuXmv
6. The user runs COMPASS to check the annotated properties.
7. The user extends the assurance case by instantiating the argument patterns with the verification activities and results.
8. The user extends manually the MILS-AADL model with error models using a text editor.
9. The user runs COMPASS to extend the MILS-AADL model with failures.
10. The user runs COMPASS to check the annotated properties on the extended model
11. The user extends the assurance case by instantiating the argument patterns with the safety analysis.

Figure 3: MILS Design Workflow

This last task reduces to (1) demonstrating the correctness of the configurations produced by the MILS platform configuration compiler — either by means of a correctness proof or by using the compiler validation techniques as proposed in D5.3, and (2) to indeed demonstrate isolation, information flow control, and determinism of the MILS technical platform; these validations can be done once and forall, and have, at least partially, been performed in the context of the D-MILS project. The first two tasks are application-specific, and the verification technology of the D-MILS project supports in particular architectural analysis of MILS-AADL models. Notice that the "green" boxes in Figure 2 denotes evidence that is provided once-and-forall, whereas the evidence of the "orange" boxes needs to be provided on a case-by-case basis, as these properties are application-specific. Of course, some of the COTS application code could, and indeed should, also be pre-certified.

1. The user extends the MILS-AADL model with hardware allocation.
2. The user extends manually the MILS-AADL model with error models for the new hardware components using a text editor.
3. The user runs COMPASS to extend the MILS-AADL model with failures.
4. The user runs COMPASS to check the annotated properties on the extended model.
5. The user creates additional constraints for the MPCC.
6. The user runs the MPCC to generate a set of MILS platform configurations (set of sets).
7. The user extends the assurance case by instantiating the argument patterns with the configuration details.
8. The user runs Lynx ST tools to configure the separation kernel for the chosen platform configuration.
9. The user runs TTTech tools to automatically configure the TTE network.
10. The user deploys D-MILS configuration on technical platform.
11. The user loads and runs application on D-MILS technical platform.
12. The user extends the assurance case by instantiating the argument patterns for the MILS platform

Figure 4: MILS Deployment Workflow

The first step of our evaluation was to make firsthand use of the D-MILS technologies, by following a certain workflow. The scope of this first evaluation step is to evaluate the D-MILS technologies for designing, verifying and deploying industrial systems. The followed workflow is the representative workflow for the typical way D-MILS technologies are intended to be used within an organisation that is developing and operating critical systems requiring high levels of assurance for security and dependability. By D-MILS technologies we mean the workflow is built of five categories of tasks, which, in fact, represent a software development lifecycle for critical systems, as proposed by the D-MILS technologies:

- Design and specification (using the MILS-AADL declarative language)
- Verification of properties (using COMPASS tool chain)
- Assurance case (using MBAC and the D-MILS assurance case patterns)
- Deployment (using the D-MILS Platform Configuration Compiler)
- Platform operation (on the D-MILS platform)

Figures 3 and 4 state the concrete development steps for the design and the deployment of dependable and secure applications using the MILS technical platform.

The specific measures which were utilised within each category for evaluating the D-MILS technologies are described in the *D-MILS Supplemental Evaluation Plan* document.

After completing the first step of the evaluation, we went to the second step, i.e., checking the degree to which our industrial requirements — specified in deliverable D1.1 — are satisfied for our smart micro grid demonstrator. The evaluation of the appropriateness level of the D-MILS technologies for satisfying the industrial requirements has been performed according to evaluation measures and method for the industrial requirements described in the *D-MILS Supplemental Evaluation Plan* document.

All in all, our assessment demonstrates how D-MILS advances beyond the state-of-the-art, based on the objective measures and methodology defined in WP1.

3 Development Process

The fortiss Smart Microgrid case study has originally been partly modelled in AutoFOCUS3 (http://af3.fortiss.org/). AutoFOCUS3 (AF3) is a model-based tool that allows modeling and validating concurrent, reactive, distributed, timed systems on the basis of formal semantics. It offers several levels of abstraction whereby in our case we used the logical and the technical architecture views. These models form our starting point for the design and implementation according to the D-MILS architectural design and implementation approach.

The Logical Architectural View of a system is defined by means of components communicating via message passing through typed channels, using a clearly defined model of computation. Message exchange is synchronized with respect to a global, discrete time base. Components can directly implement behaviour or consist of other components that do so.

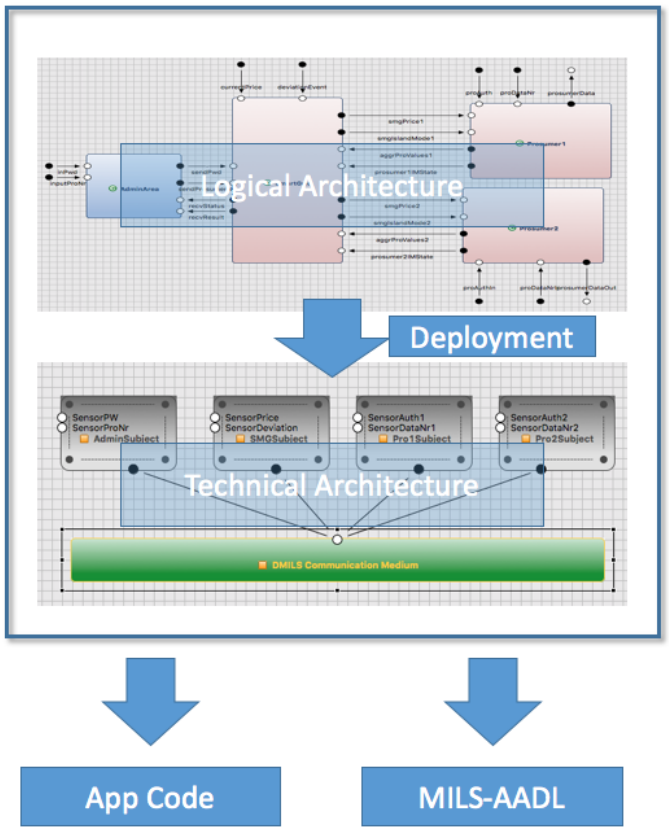

Figure 5: Smartgrid case study development process in AF3

The Technical Architecture View describes a hardware topology that is composed of computation units (e.g. ECUs, cores, etc.), communication units and sensors/actuators.

We modeled the structure and behaviour of our Smart Microgrid demonstrator according to the requirements of D1.1 as a logical architecture in AF3. After the logical architecture is modeled the

components have to be mapped onto hardware. In case of D-MILS we do not map directly onto hardware but onto subjects. Therefore we implemented a D-MILS specific technical architecture which consisted of subjects (instead of ECUs). Those subjects are connected by a communication medium, which in our case is the TTTech switch. Each subject has sensors which represent the interfaces to the data that is received from outside the Smart Microgrid system (i.e., a SmartGrid component receives the current energy price and the deviation event). The subjects can be on the same machine but do not necessarily have to.

Within the scope of this project we adapted the AF3 C Code generator to be compliant with the D-MILS technology. Therefore we could use the AutoFOCUS Smart Microgrid model to generate application code. Furthermore using the architectural information from the model we are able to generate a high-level MILS-AADL file, that can subsequently be used as input for the deployment generator. The high-level MILS-AADL file only describes the system structure. The differences in semantics between MILS-AADL and AutoFOCUS3 are therefore in this case irrelevant.

4 Functional Description

In this section the architecture and functionality of the fortiss Smart Grid System is described. The system was originally modeled in AutoFOCUS3 (http://af3.fortiss.org). From this model, using the approach described in the previous section (3), we synthesized the application code. The Smart Grid System was modeled according to the requirements presented in the D1.1 document. In every sub-section we mention the requirements that are covered by the corresponding sub-component. This section is organized as follows: First the high-level architecture of the Smart Grid System is presented and illustrated using screenshots from AutoFOCUS and the corresponding MILS-AADL description. In the following sub-sections the main sub-components (*SmartGrid*, *Prosumers*, *AdminArea*) are presented in more detail. Finally the deployment of the sub-components on subjects is described.

For our system model we use the following assumptions:

- The time unit used is the hour.
- Each day is decomposed into several time slots. Currently, four time slots (of 6 hours each) are considered.
- The smart grid sub-system sends the current price for electricity and the current grid state once every hour.
- The values for the electricity price and for deviation/power outage come from outside this system.

4.1 High-Level view

The AutoFOCUS3 model of the smart micro grid system is divided in four sub-systems (components): *SmartGrid*, *AdminArea*, *Prosumer1* and *Prosumer2* (cf. fig. 6). Those components exchange information, both with each other and with the outside world (systems which are not part of this model).

The *SmartGrid* component is the central component in this system. It receives the current price for the electricity and the current grid state from outside the system. Based on this information it forwards the price to the prosumers. Additionally, in case of a power grid deviation, the *SmartGrid* sub-system can send the prosumers into *islandMode*. The *SmartGrid* sub-system receives the current state (aggregated values and *islandMode* state) from the prosumers. This information is stored and can be requested from the *AdminArea* sub-system.

The *AdminArea* component allows the system administrator to request information about prosumers remotely. If a correct password is provided the *SmartGrid* sub-system grants the access.

The *Prosumer* components receive the current electricity price and an *islandMode* signal. If this signal is true, then the prosumers switch to *islandMode*. In return the prosumers send aggregated production/consumption values together with their current *islandMode* state back to the *SmartGrid*. Furthermore, each prosumer has an administrator interface. Providing the correct password the prosumer administrator can request certain data from the prosumers. A prosumer administrator can only request data from its own prosumer.

The representation of the system's architecture in MILS-AADL is given in the code listing below:

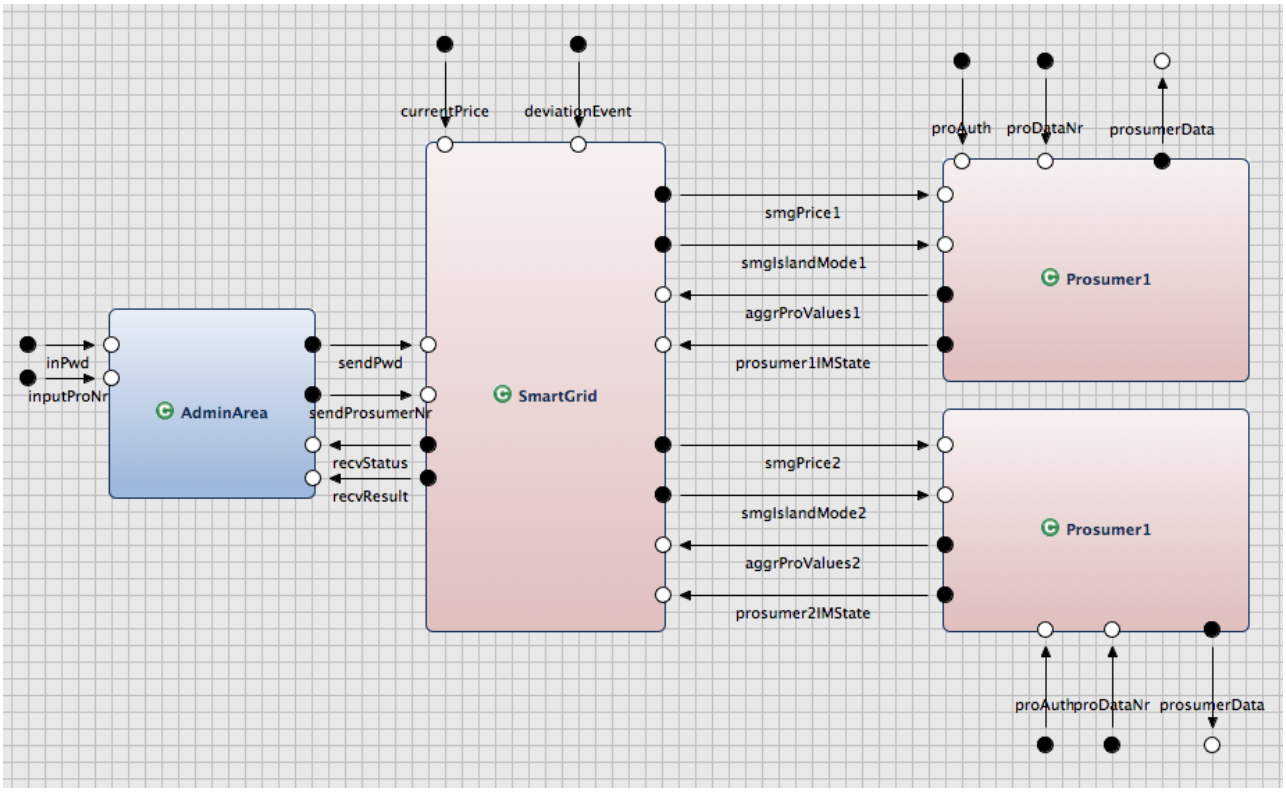

Figure 6: Overview of the SmartGrid AF3 model

Representation in MILS-AADL:

```
system SmartGridSystem
features
      currentPrice: in event port;
      deviationEvent: in event port;
      inCredentials: in data port (int, int);
      inCredentialsPro1: in data port (int, int);
      inCredentialsPro2: in data port (int, int);
      prosumerData1: out data port (int,int) default (0,0);
      prosumerData2: out data port (int,int) default (0,0);

end SmartGridSystem;

system implementation SmartGridSystem.impl

subcomponents
      AdminArea: subject AdminArea.impl;
      SmartGrid: subject SmartGrid.impl;
      Prosumer1: subject Prosumer.impl;
      Prosumer2: subject Prosumer.impl;
connections
      port inCredentials -> AdminArea.inCredentials;
      port currentPrice -> SmartGrid.currentPrice;
      port deviationEvent -> SmartGrid.deviationEvent;
      port inCredentialsPro1 -> Prosumer1.inCredentialsPro;
      port inCredentialsPro2 -> Prosumer2.inCredentialsPro;
      port Prosumer1.aggregatedValues -> SmartGrid.prosumerData1;
      port Prosumer2.aggregatedValues -> SmartGrid.prosumerData2;
      port AdminArea.sendRequest -> SmartGrid.inCredentials;
```

```
      port SmartGrid.sendResponse -> AdminArea.recvResponse;
      port SmartGrid.price1 -> Prosumer1.price;
      port SmartGrid.price2 -> Prosumer2.price;
      port SmartGrid.deviationEvent1 -> Prosumer1.deviationEvent;
      port SmartGrid.deviationEvent2 -> Prosumer2.deviationEvent;
      port Prosumer1.islandModeState -> SmartGrid.islandModeState1;
      port Prosumer2.islandModeState -> SmartGrid.islandModeState2;
      port Prosumer1.prosumerData -> prosumerData1;
      port Prosumer2.prosumerData -> prosumerData2;


end SmartGridSystem.impl;
```

4.2 SmartGrid

The *SmartGrid* sub-system consists of four components (cf. fig. 7): a persistency component, a wrapper component and two prosumer energy agents (one for input and one for output). The following subsections describe those components in more detail.

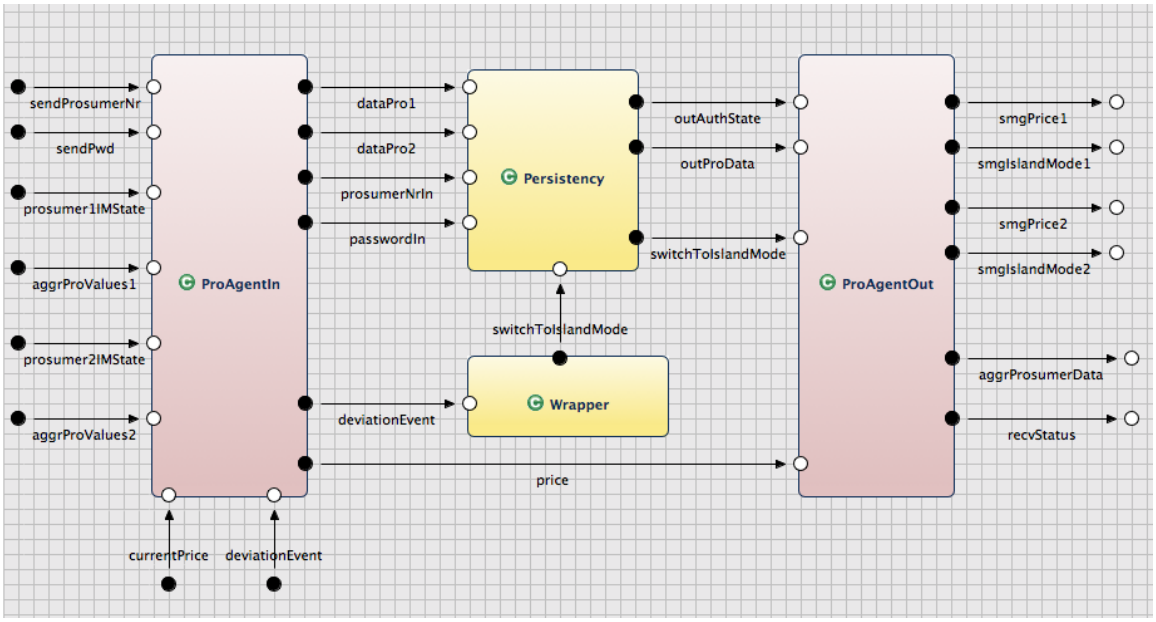

Figure 7: SmartGrid sub-system

Covered Requirements:

[SMG_SO.1] *Shall:* Communication / Information Flow between components shall be according to the policy defined in Deliverable 1.1. No other information flow shall occur.

Representation in MILS-AADL:

```
system SmartGrid
features
      currentPrice: in event port;
      deviationEvent: in event port;
      prosumerIMState1: in event port;
```

```
      prosumerIMState2: in event port;
      aggrProValue1: in data port int;
      aggrProValue2: in data port int;
      smgPrice1: out event port;
      smgPrice2: out event port;
      smgIslandModeState1: out event port;
      smgIslandModeState2: out event port;
      inCredentials: in data port (int, int); -- prosumerNr + password
      sendResponse: out data port (int, int) default (0,0); -- authState + proData

end SmartGrid;

system implementation SmartGrid.impl
subcomponents
      SmgProAgentIn: subject SmgProAgentIn.impl;
      SmgPersistency: subject SmgPersistency.impl;
      SmgWrapper: subject SmgWrapper.impl;
      SmgProAgentOut: subject SmgProAgentOut.impl;

connections
      port currentPrice -> SmgProAgentIn.currentPrice;
      port deviationEvent -> SmgProAgentIn.deviationEvent;

      ...
```

4.2.1 Prosumer Energy Agents

Both prosumer energy agents (*ProAgentIn* and *ProAgentOut*) provide an interface to the outside world. *ProAgentIn* receives the data from outside the system and forwards it to the corresponding sub-components of the *SmartGrid* system. *ProAgentOut* on the other hand receives data from the sub-components of the *SmartGrid* and forwards it to the outside world.

Covered Requirements:

[SMG_SO.4] *Shall:* The prosumer energy agent shall communicate only required data by the micro grid. The micro grid shall not have access to any private data by the prosumer. In particular the sensor values of prosumer systems shall not be transmitted to the micro grid. The only exception is the data of the main smart meter, since this is required for stability control.

4.2.2 Wrapper

The wrapper component receives information about a possible deviation event (e.g., a power outage) in the power grid. In case of a deviation event the wrapper component sends a *switchToIslandMode* request to the *Persistency* component, which forwards it over the *ProAgentOut* component to the prosumers.

Covered Requirements:

[SMG_SA.1] *Shall:* The highest level safety priority is grid stability. Indicators of instability are: deviation from the frequency 50Hz and deviation from the nominal voltage level. In the case that the frequency deviates more than 1Hz, the micro grid shall switch to island mode.

4.2.3 Persistency

The Persistency component collects and forwards data. The task of the Persistency component is fourfold (cf. fig 8):

1. Save the aggregated prosumer data;
2. Forward the power grid data;
3. Check the administrator authorization;
4. Answer prosumer data requests from the AdminArea in case of a successful authorization.

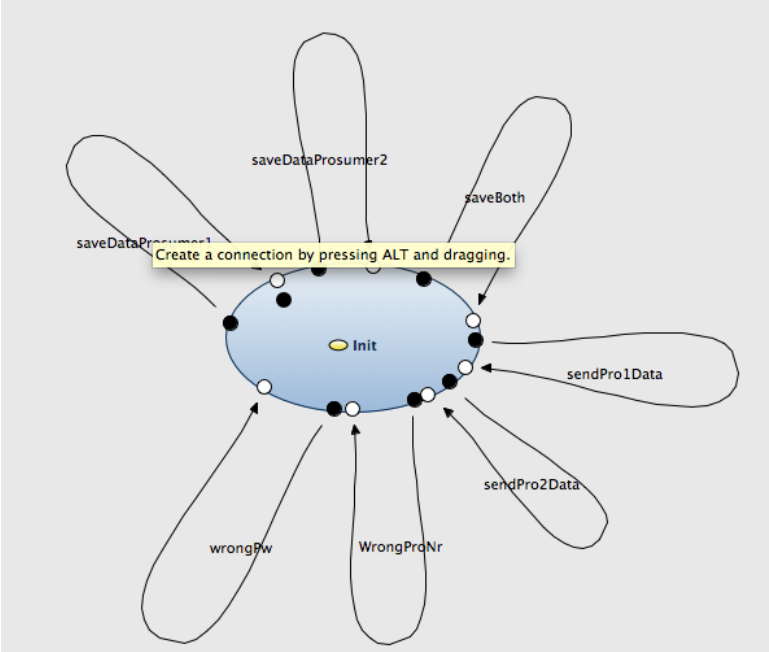

Figure 8: SmartGrid Sub-System: Persistency

Covered Requirements:

[SMG_SO.12] *Shall:* The persistency component shall be (a) available and (b) any other component shall not delete, add or temper with information stored in the persistency component.

4.3 Prosumer

The Prosumer sub-system in our AF3 model consists of Prosumer Energy Agents, Wrapper components, a Persistency component and a Rule component (cf. fig. 9). In the AF3 model there are two Prosumer Energy Agents: *ProAgentIn* and *ProAgentOut*, which provide an interface to the SmartGrid sub-system. The *ProAgentIn* component receives a command whether or not it has switch into

islandMode as well as a price event. This information is forwarded to the *Rule* component, which decides over the possible actuators (e.g. lights) of the system. This information is based on the *islandMode* command and the sensor values, which the Rule component receives from the *WrapperSensors* component.

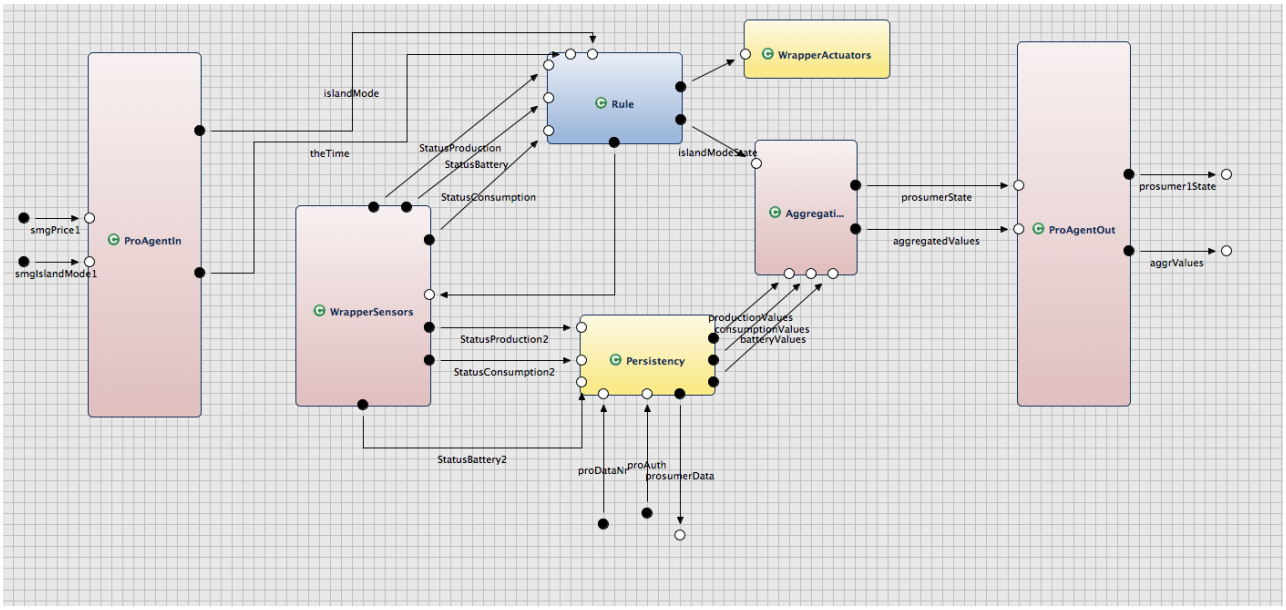

Figure 9: Prosumer Sub-System

The sensor values, which are produced based on the current day time, are also stored in the *Persistency* component. Apart from storing the values, the *Persistency* component forwards the data to the *Aggregation* component, and allows a local administrator to access the stored data using the corresponding password.

The aggregated sensor values and the current *islandMode* state of the prosumer are sent back to the *SmartGrid* component.

Covered Requirements:

[SMG_SO.1] *Shall:* Communication / Information Flow between components shall be according to the policy defined in Figure 9 of D1.1 (Prosumer Information Flow) and Figure 10 of D1.1 (Micro Grid Information Flow). No other information flow shall occur.

[SMG_SO.4] *Shall:* The prosumer energy agent shall communicate only required data by the micro grid. The micro grid shall not have access to any private data by the prosumer. In particular the sensor values of prosumer systems shall not be transmitted to the micro grid. The only exception is the data of the main smart meter, since this is required for stability control.

Representation in MILS-AADL:

```
system Prosumer
features
      inCredentialsPro: in data port (int,int);
      prosumerData: out data port (int,int) default (0,0);
      price: in event port;
```

```
      deviationEvent: in event port;
      aggregatedValues: out data port int default 0;
      islandModeState: out event port;
end Prosumer;


system implementation Prosumer.impl
subcomponents
      ProAgentIn: system ProAgentIn.impl ;
      WrapperSensors: system WrapperSensors.impl ;
      Persistency: system Persistency.impl ;
      Rule: system Rule.impl ;
      WrapperActuators: system WrapperActuators.impl ;
      Aggregation: system Aggregation.impl ;
      ProAgentOut: system ProAgentOut.impl ;
connections
      port price -> ProAgentIn.smgPrice;
      port deviationEvent -> ProAgentIn.smgIslandMode;
      port ProAgentIn.islandMode -> Rule.islandMode;
      port ProAgentIn.currentTime -> Rule.currentTime;
      port WrapperSensors.StatusProduction -> Rule.StatusProduction;
      port WrapperSensors.StatusBattery -> Rule.StatusBattery;
      port WrapperSensors.StatusConsumption -> Rule.StatusConsumption;
      port Rule.currentTimeOut -> WrapperSensors.currentTime;
      port Rule.islandModeState -> Aggregation.islandModeState;
      port Rule.consumptionRule -> WrapperActuators.consumptionRule;
      port WrapperSensors.StatusProduction2 -> Persistency.StatusProduction;
      port WrapperSensors.StatusConsumption2 -> Persistency.StatusConsumption;
      port WrapperSensors.StatusBattery2 -> Persistency.StatusBattery;
      port inCredentialsPro -> Persistency.inCredentials;
      port Persistency.prosumerData -> prosumerData;
      port Persistency.productionValues -> Aggregation.productionValues;
      port Persistency.consumptionValues -> Aggregation.consumptionValues;
      port Persistency.batteryValues -> Aggregation.batteryValues;
      port Aggregation.prosumerState -> ProAgentOut.prosumerState;
      port Aggregation.aggregatedValues -> ProAgentOut.aggrValues;
      port ProAgentOut.prosumerStateOut -> islandModeState;
      port ProAgentOut.aggregatedValues -> aggregatedValues;
end Prosumer.impl;
```

The following sub-sections describe the Prosumer sub-components in more detail.

4.3.1 Prosumer Energy Agents

As already mentioned, the Prosumer Energy Agents basically forward the input data to the corresponding components within the prosumers and vice versa. The only difference is the input value price event. We assume that the *SmartGrid* sends a new price event every hour. For that reason we can use this price event as an internal clock for the prosumers. Each time a time event arrives the internal clock of the prosumer is incremented by one, unless the internal clock is 23. In this case a 24-hours day is over and the clock turns to 0 again (cf. fig. 10).

4.3.2 Wrappers

In our AF3 model there are two different wrappers, one for the sensors and one for the actuators. The wrapper for the actuators is currently a placeholder, since our demonstrator does not possess any actuators. In a real system, actuators could be some energy intensive devices, which can be cut off from the power supply in case of an islandMode.

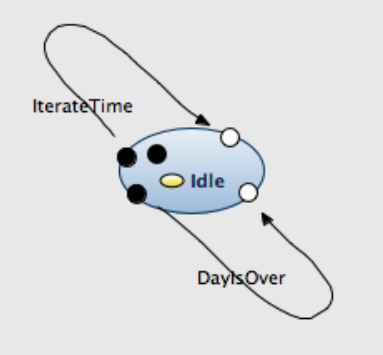

Figure 10: Prosumer Sub-System: Internal Clock

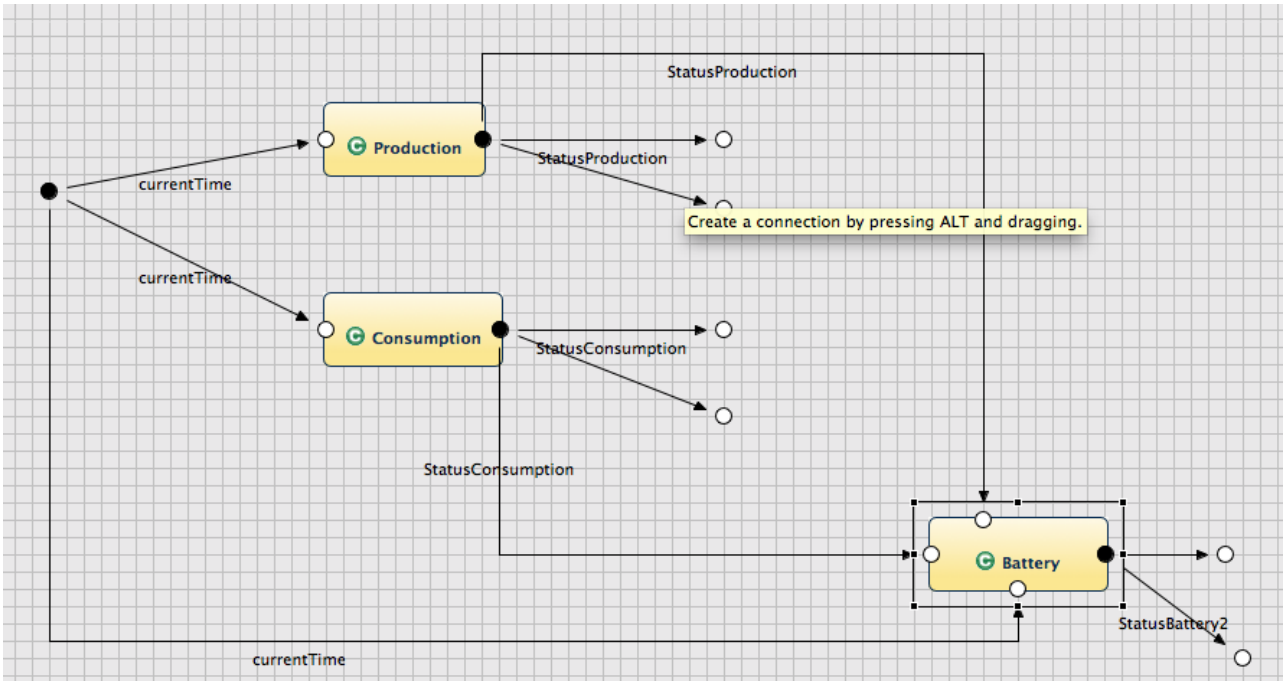

Figure 11: Prosumer Sub-System: Sensor Wrapper

The sensor wrapper on the other hand consists of three sensor components (*Consumption*, *Production*, *Battery*). The *Consumption* and *Production* components generate different values depending on the current time of the day. In our model we assume that, on average, the consumption during the night (lights switched on) is higher than the production (solar panels). Depending on the level of production and consumption, it is decided whether the battery is currently discharged or recharged (cf. fig. 11). We further assume that the battery has a max. capacity.If this value is reached then the battery does not charge any further (cf. fig. 12).

Covered Requirements:

[SMG_SO.1] *Shall:* Communication / Information Flow between components shall be according to the policy defined in Figure 9 of D1.1 (Prosumer Information Flow) and Figure 10 of D1.1 (Micro Grid Information Flow). No other information flow shall occur.

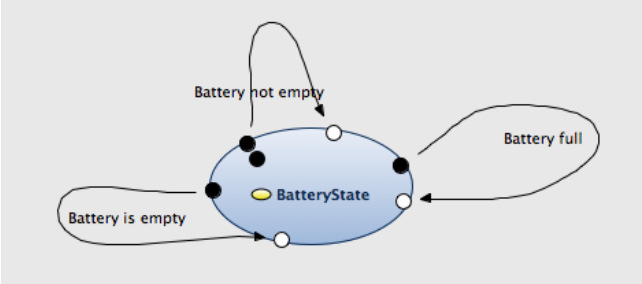

Figure 12: Prosumer Battery

4.3.3 Persistency

The *Persistency* component has two main tasks to fulfill. On the one hand it stores and forwards the sensor data, which it gets from the *WrapperSensors* component, and on the other hand it grants access to the stored data, provided that the password given is correct (cf. fig. 13).

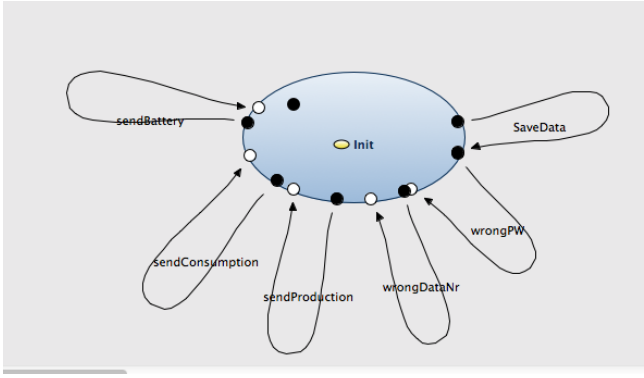

Figure 13: Prosumer Sub-System: Persistency

Covered Requirements:

[SMG_SO.7] *Shall:* Authentication: User shall authenticate himself to the control software in both cases: prosumer and micro grid. Every prosumer system shall be authenticated itself to the micro grid.

[SMG_SO.8] *Shall:* Authorization: Every user shall have a limited set of rights. He shall not be able to obtain more rights than he has.

[SMG_SO.9] *Shall:* The admin of each prosumer and smart grid system shall be able to access only his system. Access to other systems shall not be possible.

4.3.4 Aggregation

The *Aggregation* component aggregates sensor data (consumption, production, battery status) and forwards it alongside the current *islandMode* state to the *ProAgentOut* component.

Covered Requirements:

[SMG_SO.3] *Shall:* The aggregation component shall only communicate aggregated energy data to the prosumer energy agent. No sensor or device specific information shall be communicated for privacy reasons.

4.3.5 Rule Component

The *Rule* component can switch a prosumer to *islandMode* and back to normal depending on its input.

4.4 AdminArea

The *AdminArea* component allows the system administrator remote access to prosumer data stored in *SmartGrid*'s *Persistency* component. The component waits in the Idle mode for an information request. An information request consists of a prosumer number and a password. This information request is forwarded to the *SmartGrid* component, where it is processed. Until the response is received, the *AdminArea* sub-system remains in the *checkAuth* mode. The response is forwarded to the output ports (cf. fig. 14).

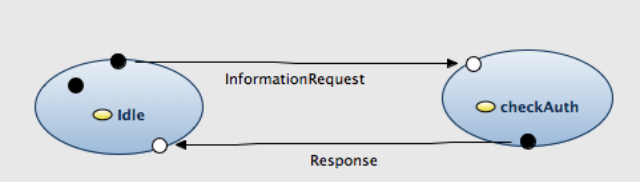

Figure 14: AdminArea Sub-System

Covered Requirements:

[SMG_SO.7] *Shall:* Authentication: User shall authenticate himself to the control software in both cases: prosumer and micro grid. Every prosumer system shall be authenticated itself to the micro grid.

[SMG_SO.8] *Shall:* Authorization: Every user shall have a limited set of rights. He shall not be able to obtain more rights than he has.

4.5 Deployment

After the logical architecture is modeled the components have to be mapped onto hardware. In case of D-MILS we do not map directly onto hardware but onto subjects (cf. fig. 15). Therefore

we implemented a D-MILS specific technical architecture which consisted of subjects (instead of ECUs). Those subjects are connected by a communication medium, which in our case is the TTTech switch. Each subject has sensors which represent the interfaces to the data that is received from outside the Smart Microgrid system (i.e., a SmartGrid component receives the current energy price and the deviation event). The subjects can be on the same machine but do not necessarily have to (cf. fig. 6).

After the deployment of the four main components (AdminArea, SmartGrid, Prosumer1, Prosumer2) on the corresponding subjects is done the application code can be generated.

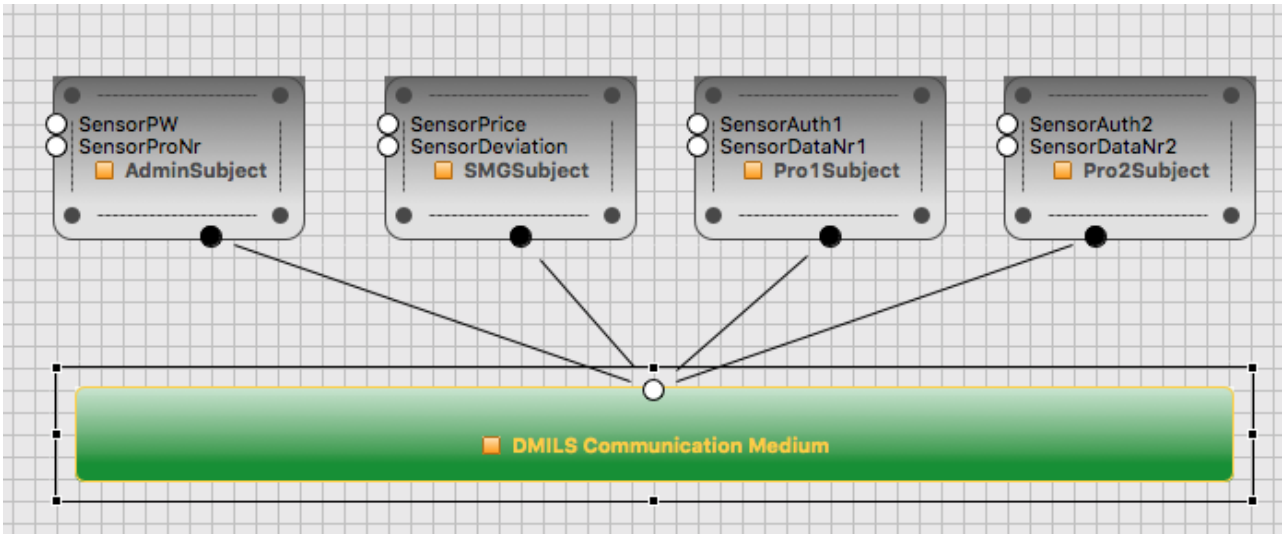

Figure 15: Smart Grid Platform Architecture in AF3

5 Design and specification

This section deals with design and specification capabilities of MILS AADL

5.1 Design Expressivity

Description:

The degree to which MILS-AADL is able to represent and address the functional elements that comprise an industrial demonstrator application.

Experiment:

Evaluate how many components of the SMG case study that can be represented using MILS-AADL.

Verdict:

In scope of the evaluation of MILS-AADL w.r.t. the ability of representing functional elements of the Smart Microgrid case study we focus on two essential aspects: ability of expressing behaviour and the ability of expressing system's structure.

The behaviour of components can be specified in MILS-AADL language through state automatons. A state automaton in MILS-AADL consists of a set of modes and a set of transitions between those modes. A transition between modes has the form as shown in the listing below, where *m1* and *m2* are modes, *e* is a trigger event for the transition, *v* is the event data, *g* is a guard and *f* is an effect. Thereby each of these parts (e, v and g) can be omitted if not needed.

```
m1 -[e(v) when g then f]-> m2
```

Modeling component behaviour in MILS-AADL is relatively straightforward. However, problems may occur if a transition definition exceeds a certain size. A definition can occasionally spread through several lines, as the one in the listing below. Having a clear understanding of such definitions and further working with them is cumbersome and error prone.

```
idle -[when (StatusConsumption < StatusProduction)
              and (StatusProduction - StatusConsumption) > charge_thr
                 then chargingRate := (StatusProduction - StatusConsumption)]
                             -> productionHigherConsumption;
```

The MILS-AADL language allows a hierarchical decomposition of system components. The communication among MILS-AADL system components can be depicted by specification of internal and external communication, and internal and external interfaces. One example of how system structure is expressed in MILS-AADL can be seen in section 4.1.

After performing our evaluation, we reached the conclusion that MILS-AADL is a rich modeling language, which can be used to comprehensively depict systems. However, we also detected two shortcomings of this modeling language. One of them is the missing persistency model (cf. explanation in 5.4 point 3) in MILS-AADL. This affected us while trying to address some of the requirements from the D1.1 deliverable (i.e. requirements that deal with logging events). The other detected shortcoming of the MILS-AADL language is the missing consistency check for the categorization of composite components *Node*, *Subject* or *System*, as described in D2.1. COMPASS seems to ignore those keywords.

Since MILS-AADL is a textual modeling language there are some issues w.r.t. to scalability. Accurately modeling complex systems (i.e. system with numerous components and inter-connections) with MILS-AADL can be problematic. Having an adequate editor with features such as consistency checks, syntax highlighting or auto-completion would significantly contribute to the resolution of this problem.

5.2 Design Usability

The perceived relative effort and ease involved in using MILS-AADL to represent and address the functional elements that comprise an industrial demonstrator application.

Question 1: What is the level of difficulty for representing the functional architecture of the case study?

Rationale:

As already discussed in the previous sub-section we assume that system's structure is meant by functional architecture. We define the system's structure as the ability of expressing the components of the systems (also composed components), their interfaces and their inter-connections with each other or the outside world.

Specifying the system structure in MILS-AADL is straightforward. We suspect that problems would only appear when defining the structure of an industrial-sized system, consisting of a large set of components and inter-connections. Nevertheless is this not a negative aspect of the language itself but rather a motivation to work on tools, that could support developers with state-of-the-art techniques such as code highlighting, auto-completion, consistency checks etc.

In order to argue the facility of specifying system structure in MILS-AADL we next explain how we modeled one of our system components, namely the *prosumer*. An outline of the prosumer's structure is showed in the listing below. We started by defining the interface of the component as the subject's features. Thereby the input and output ports are declared. Furthermore the data type of each port is defined at this point. In our example all the ports are of type *event*. We then defined in the system's implementation the subject's sub-components. In our example we define only two sub-components, called *ProAgentIn* and *ProAgentOut*. At the end of the structure definition one can define connections between sub-components and between sub-components and ports.

```
subject Prosumer
features
      price: in event port;
      deviationEvent: in event port;
      islandModeState: out event port;
      ...
end Prosumer;

system implementation Prosumer.impl
subcomponents
      ProAgentIn: system ProAgentIn.impl ;
      ProAgentOut: system ProAgentOut.impl ;
      ...
connections
      port price -> ProAgentIn.smgPrice;
      port deviationEvent -> ProAgentIn.smgIslandMode;
      port ProAgentOut.prosumerStateOut -> islandModeState;
      ...
```

```
end Prosumer.impl;
```

Verdict:

Very Difficult – Difficult – Easy – **Very Easy**

Question 2:

What is the level of difficulty for modeling the behaviour of a single component in MILS-AADL?

Rationale:

Behaviour can be expressed in MILS-AADL either as *state automatons* or as *flows*.

Automatons provide an abstraction of the concrete behaviour of a component by providing the formalisms for modelling the finite control part of a component. An automaton consists of set of *modes* and a set of *transitions* between them. Every automaton has an *initial* mode (see listing below). After de- and re-activation the execution is resumed in the previous mode. One can attach to modes linear conditions, which act as invariants. These invariants, together with clocks, enable the user to constrain the amount of time that may be spent in one mode.

```
subcomponents
      timer: data clock;
 modes
      idle: initial mode;
      enabled: mode while timer <= timeout min;
      ...
```

A transition in MILS-AADL consists of three parts: a trigger (beginning of the transition), a guard (after the *when* keyword) and an effect (after the *then* keyword), as showed in the listing below. Each of these three parts can be omitted. A transition can be either triggered by incoming events (event data ports) or spontaneously if, for example, a transition does not have a trigger. A guard, similar to a mode invariant, is a logical expression of data ports and variables (defined in the *subcomponents* section of a system implementation). The effect defines the impact of the transition by specifying some update operations on the data (variables or outgoing ports).

```
transitions
      idle -[engage then timer := 0]-> enabled;
      enabled -[when speed < set then control := 1]-> enabled;
```

Another possibility of defining behaviour of a component is the concept of *flows*. Flows allow an immediate reaction of an update of other ports. They provide the possibility to either forward the data from an incoming port to an outgoing port or perform some operations on this data before forwarding it, as it is shown in the small example below.

```
device Adder features
      input1: in data port bool default true;
      input2: in data port bool default true;
      output: out data port (bool, bool);
end Adder;
device implementation Adder.Impl
      flows
            port (input1 and input2, input1 xor input2) -> output;
end Adder.Impl;
```

Having all these language constructs for expressing component behaviour, we managed to successfully define the functional behaviour of the components of our smart grid. Hence, we affirm that MILS-AADL is suitable for expressing functional behaviour.

Verdict:

Very Difficult – Difficult – Easy – **Very Easy**

5.3 Design Feedback

The perceived usefulness of the feedback provided when using MILS-AADL to represent and address the functional elements that comprise an industrial demonstrator application.

Question 1:

How helpful are the error messages from COMPASS when there is an error in definition of functional elements?

Rationale:

In the scope of the evaluation of MILS-AADL using the Smart Microgrid Demonstrator the only tool we used apart from a regular text editor was COMPASS. COMPASS is a tool which offers a set of analysis and verification techniques for MILS-AADL models. As such, in this evaluation step we assess the capability of COMPASS for checking MILS-AADL syntax.

COMPASS allows the loading of MILS-AADL models and performs a syntax check on them. In case the model is syntactically correct the user gets a positive feedback from the tool and is able to use all its verification and simulation functionalities. In case the model comprises syntax errors the tool provides the user with information about what might be wrong and the possible locations of the errors. In most of the cases the feedback provided by the tool was meaningful and it supported us in creating a syntactically correct model. However, in some cases the feedback which was provided by COMPASS was sometimes misleading. An example for such a case is the recommendation of COMPASS to add a bus to a system, even if it was not necessary. The result of following this recommendation was a syntactically, but not semantically correct model of the Smart Microgrid demonstrator.

Verdict:

Not helpful at all – Not that helpful – **Helpful** – Very Helpful

Question 2:

Did the simulation of a MILS-AADL model help you during the design process ?

Rationale:

In scope of the development process of the Smart Grid System case study we used the simulation functionality for testing whether a model behaves as we expected. We only simulated components, whose behaviour was modelled as state automatons.

COMPASS offers three different ways of simulating a model: *random simulation*, *guided by transition* or *guided by values*.

If one chooses *random simulation* the system, as expected, chooses the values for the system automatically. During our case study we did not use this option extensively, but we assume that it is very helpful for stress testing of the system.

If one chooses *guided by transitions*, it is possible to force the system to take certain transitions. It is not possible though to take any transitions but only those which are available in the current state. This option was more adequate for the purpose of checking the model's behaviour as the first one. However, the simulation *guided by values* was the most preferred and used type of simulation during our evaluation.

The simulation *guided by values* allows the user to choose the input values for the model for the next step. In our opinion this is the most useful simulation option, since it gives the user the complete control over what the model should (or is expected to) do in the next step.

In conclusion, we found the simulation functionality of COMPASS very helpful to ensure the correctness of the model for our Smart Grid System case study.

Verdict:

Not at all – Mostly not – Mostly – **A lot**

Question 3:

How clear is the feedback provided by the simulation of a MILS-AADL model?

Rationale:

The result of a simulation run COMPASS provides a trace of the model where the values of all variables and ports are listed (cf. fig. 16). Depending on how complex the model is, the amount of presented steps varies. An invaluable feature of COMPASS is that after the simulation run COMPASS announces if there is any deadlock in the model.

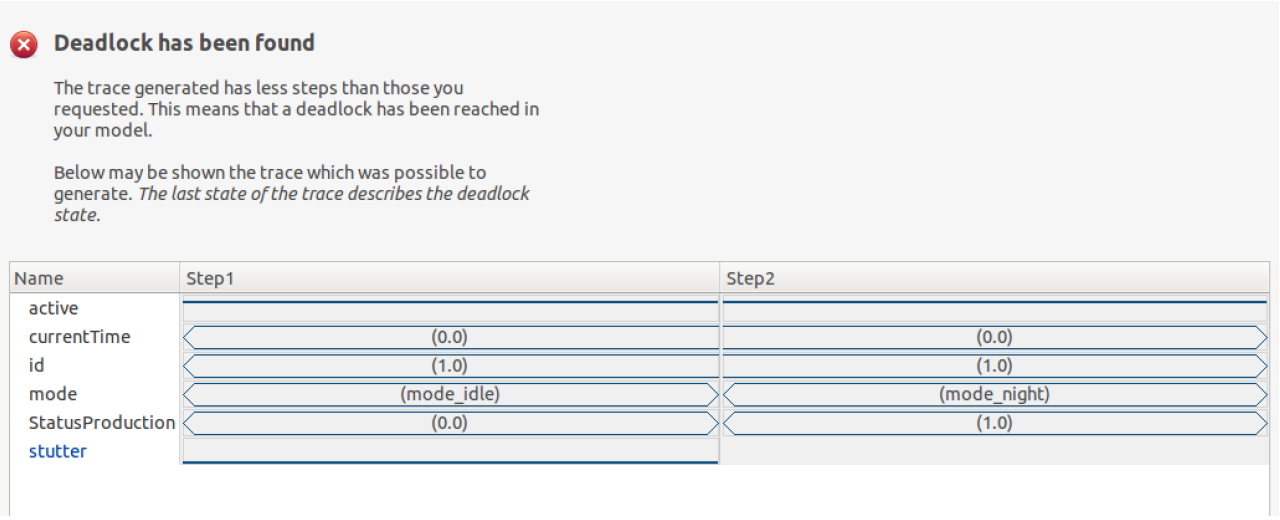

Figure 16: Simulation trace in COMPASS

When traces become very large COMPASS offers a filtering functionality. This enables the user to concentrate only on those parts of the traces, which are relevant for the user.

Verdict:

Very unclear – Mostly unclear – Mostly clear – **Very clear**

5.4 Design Maturity

The perceived maturity of MILS-AADL technologies to represent and address the functional elements that comprise an industrial demonstrator application.

Question 1:

How mature are the MILS-AADL technologies in terms of addressing functional elements of SMG case study?

Rationale: In the scope of our evaluation we confronted ourselves with some difficulties while trying to address some of the SMG requirements. These requirements are enlisted below. However, none of the below requirements were classified as *SHALL*, some of them being classified as *MAY*. Therefore, the fact that these requirements could not be (fully) fulfilled does not considerably impact the maturity of the MILS-AADL technologies.

1. *Requirement* (SMG_SA3): Switching to island mode shall to be accomplished in less than 20 ms.

   Explanation: By using a *clock* in MILS-AADL, it is possible to represent temporal behaviour in MILS-AADL. Therefore the language itself supports this requirement. Nevertheless, temporal behaviour of an embedded system is strongly dependent on the hardware it is running on. MILS-AADL is a high-level modeling language, which does not capture hardware specific properties in a granularity which would make predictions of this kind possible.

2. *Requirement* (SMG_SA7): The rule system of prosumer shall immediately be informed when micro grid island mode is activated. The available time window depends on the hardware components, but typical times are around 100ms.

   Explanation: Same explanation as for the requirement *SMG_SA3*.

3. *Requirement* (SMG_SO10): Logins of the admin shall be logged persistently.

   Explanation Theoretically it is possible to use variables to store the last login (or to use enumerations to store a set of logins). So we can say that semantically it is possible to express persistency but it doesn't seem to be convenient. On the one hand because of a very small set of available data structures and on the other hand because of the lack of any possibilities for file manipulation (that would be needed for writing a log file).

4. *Requirement* (SMG_SO11): Maintenance of the rule system shall be possible for users, which shall be logged. Users shall not be able to modify or delete log files.

   Explanation: We were able to address part of this requirement by having admin users, which have certain rights (access to information). However the possibility of expressing users that modify or delete files is not very convenient in MILS-AADL as already stated for the SMG_SO10.

5. *Requirement* (SMG_SR2): Rule System shall consider constraints specified by the user.

   Explanation: This requirement translates to the fact that it should be possible for a user of the smart grid to specify new constraints. Similar to the persistency requirements, it would be possible to express this in MILS-AADL by using enumerations. This would however would be rather cumbersome.

6. *Requirement* (SMG_SR5): Dynamic addition of new devices (software and hardware) shall be possible.

   Explanation: MILS-AADL does not have any language constructs that could express this kind of plug-and-play behaviour of the system. This requirement is classified as a *MAY* requirement and therefore is not necessarily to be considered.

Verdict:

Very Immature – Immature – **Mature** – Very Mature

6 Verification of Properties

This section presents the evaluation of the D-MILS verification technologies. The objective was to evaluate the capabilities and performance of the D-MILS verification technologies implemented in the COMPASS tool for verifying architectural correctness properties of the MILS-AADL model of the Smart Grid Demonstrator in the process of validating the Smart Grid requirements as defined in D1.1.

COMPASS is a powerful integrated toolset to support various validation and verification activities, including requirements validation, functional verification, safety and dependability analysis, performability analysis, and fault detection, identification and recovery analysis [3]. Within the scope of this evaluation in the context of the Smart Grid demonstrator, however, the full range of the COMPASS V&V support was not employed but focused on the support for functional property verification.

To this end, COMPASS provides a front-end to a state-of-the-art model checking tool NuSMV that allows to model check both finite discrete and infinite hybrid/timed MILS-AADL models with BDD-based (for finite discrete models) and SAT-based techniques. The D-MILS verification technologies support contract-based compositional verification and "monolithic" verification of system properties. The MILS-AADL models developed for the Smart Grid demonstrator are infinite ones; hence the evaluation of the verification was restricted to the SAT-based model checking of properties.

Properties are expressed as linear temporal logic formulae; for the contract-based compositional verification, D-MILS employs the language of the OCRA tool, which is a textual human-readable version of real-time first-order LTL. For monolithic verification tasks, LTL properties in COMPASS can be formulated in two ways: as so-called LTL annotations or by using property patterns. In the first case, properties are written as LTL formula using a textual representation of the logical expressions and operators; such formulas are then directly added to the MILS-AADL model. For the monolithic verification tasks in our analysis of the Smart Grid requirements, we used this style of expressing properties, which, however, requires a certain amount of experience of the user with specifying properties as temporal logic formulas. A more convenient and more intuitive way is to use property patterns. Here, frequently used types of properties are presented as patterns in a natural-language style to the user, who only needs to fill in the atomic propositions specific to the model concerned. The limitation of this approach is that not all conceivable LTL formulas can be covered by such patterns. COMPASS supports patterns for propositional, functional, timed, and probabilistic properties.

For the evaluation of the D-MILS verification support for the Smart Grid demonstrator, we aimed at formulating properties for each of the demonstrator requirements defined in D1.1, with a specific emphasis on the safety and security requirements, and at verifying these properties with the D-MILS verification tool set. Within the Smart Grid verification only functional properties were analysed, while timed or probabilistic ones were beyond the scope of this analysis.

In the following, the results of the evaluation of verifying safety and security requirements are described. In cases where the findings for a certain aspect are common to both safety and security properties, we collectively present them as findings for the functional verification of properties. The detailed account of the models used in the verification process, and the specific properties that have been verified, is given in Sect. 10.

6.1 Verification Expressivity

6.1.1 Security property coverage

Description: The degree to which the specification language is able to represent and address the security properties for verification for an industrial demonstrator application.

Experiment: Evaluate how many security properties of the SMG case study can be represented using MILS-AADL.

Verdict: 24 of 28 properties could be expressed. The properties are described in detail in Sect. 10.3.

6.1.2 Safety property coverage

Description: The degree to which the specification language is able to represent and address the safety properties for verification for an industrial demonstrator application.

Experiment: Evaluate how many safety properties of the SMG case study can be represented using MILS-AADL.

Verdict: 11 of 11 properties could be expressed. The properties are described in detail in Sect. 10.2.

6.2 Verification Usability

6.2.1 Usability for safety and security verification

The perceived relative effort and ease involved in using D-MILS tools to verify safety and security properties for an industrial demonstrator application.

Question 1: How would you rate the effort in using D-MILS tools to verify safety and security properties for the SMG case study?

Rationale:

Carrying out analyses in the COMPASS tool is relatively convenient, as COMPASS provides graphical push-button interface to the underlying model checker. Of course, the MILS-AADL model is automatically translated into the input language of the model checker. Moreover, the user does not have to concern himself with passing parameters to the model checker using various command line options. Instead, the selection of the type of model checking task (such as invariant or LTL checking), of the model checking technique (BDD, BMC), and the setting of bounds for the bounded model checker, can easily be accomplished through radio-buttons and value pickers. Furthermore, the user does not have to concern himself with the various checks necessary to establish the correct refinement of contracts in a compositional verification analysis; these are automatically generated by the COMPASS front-end and all executed in a single run. Moreover, models can be checked for dead-locks; however, automatic dead-lock detection is only supported for finite-state models.

In both the compositional and monolithic verification cases, the model checking analysis can be started by a simple click on a “run” button, and the user is informed about the status of the analysis

while it is running, and eventually the results for the various properties examined are presented in a clearly arranged table.

On the other hand, verifying properties through model checking is generally a challenging task. Typically, for models and properties of a certain complexity, a given property is rarely verified at the first try. In such cases, the model checker usually provides an explanation why the property could not be verified by means of a counterexample. The challenge for the expert is then to analyse the counterexample to reason out whether there is a flaw in the model, or whether the formulation of the LTL formula does not match the intended property.

For this task, verification tools can provide a certain degree of support. COMPASS, for instance, presents a visualization of the counterexample as a trace of states, in the same way as executions of the model behaviour are presented by the model simulation, see Sect. 5.3. However, it is still left to the user to analyse the trace to find out which part of the LTL property to be verified is violated.

This becomes a particular concern in a compositional verification task. While checking an individual contract for a given components is quite similar to checking a LTL formula in a monolithic verification, the user is not exposed to the checks that are carried out by the tool in order to establish the correct refinement of a contract of a composite component into those of the individual constituent components. Consequently, when a verification attempt fails it becomes rather challenging to analyse the counterexamples, because the underlying properties that are checked for such a refinement are only implicitly given. Moreover, our experience with the compositional verification of even a rather simple property such as the one corresponding to the safety requirement presented in Sect. 10.2.8 showed that the contracts can become quite complex. This is due to the fact that the guarantees of the individual components have to capture a sufficiently strong abstraction of the behaviour of those components in order to establish the correct refinement of the overall property. We conjecture that this observation is a direct result of the design approach employed in our Smart Micro Grid study, which has a bottom-up nature: starting from the architectural design of the SMG, properties of the individual components where composed to yield desired properties of the overall system. If, one the other hand, the system design had followed a top-down approach, where complex components and their properties are broken down into individual constituents, it could be expected that the contracts of those subcomponents could have been derived more naturally and systematically. Nevertheless, the successful completion of the compositional verification task, however, was only possible with the direct support of an OCRA expert.

Altogether, the difficulty of getting the analyses right very often outweighs the ease of executing instances of model checking runs.

Verdict:

Very Difficult – **Difficult** – Easy – Very Easy

6.3 Verification Performance

6.3.1 Solution for security verification

Description: The degree to which the D-MILS verification technologies are able to establish the security properties for an industrial demonstrator application.

Experiment: Evaluate how many security properties could be solved and verified for the SMG case study.

Verdict: 24 of 28 properties could be verified and solved.

6.3.2 Solution for safety verification

Description: The degree to which the D-MILS verification technologies are able to establish the safety properties for an industrial demonstrator application.

Experiment: Evaluate how many safety properties could be solved and verified for the SMG case study.

Verdict: 11 of 11 properties could be verified and solved.

6.3.3 Resources for safety and security verification

The perceived response time and resource usage of the D-MILS tools to verify safety and security properties for an industrial demonstrator application.

Question 1: How fast are the D-MILS tools to verify safety and security properties for SMG case study?

Rationale:

COMPASS integrates state-of-the-art model checking technologies. Both the models and the properties that were applied for the verification of the Smart Grid safety requirements were of only modest complexity and did thus not really challenge the capabilities of the model checker. Even in cases when rather high values were chosen for the bounds in bounded model checking, a significant degradation in terms of speed of the analysis could not be perceived; answer times were always within the range of a few seconds. This is not very surprising, as in an architectural setting models usually do not suffer from state explosion problems, and hence the verification technology provided by the D-MILS tools is very suitable for this domain.

Verdict:

Very Slow – Slow – Fast – **Very Fast**

Question 2: How many resources are needed to verify safety properties of the SMG case study with the D-MILS tools?

Rationale:

Resources here refer both to the computer memory needed to establish a model checking analysis and the time required for a user to successfully complete a given property verification. Concerning the first aspect, for the same reasons as in Question 1 above, the memory requirements of the model checkers was negligible. All analyses could easily be carried out on standard laptop computers with a few gigabyte of memory installed.

As for the second aspect, the time it takes for a user to establish a given property through model checking, the same remarks apply as for the usability evaluation above: even for models of limited

complexity such the ones deployed here, it usually takes a number of model checking attempts before a given property can indeed be shown to hold. Nevertheless, due to the small size of the models most of the properties could be established with only a few iterations of refinement.

Verdict:

Too many Resources – Many Resources – **Few Resources** – Very few Resources

6.4 Degree of Integration with Design

6.4.1 Integration of functional verification

The perceived level of integration of the D-MILS tools to verify functional properties for an industrial demonstrator application with the D-MILS design environment.

Question 1: What is the perceived level of integration of the D-MILS tools to verify functional properties for the SMG case study with the D-MILS design environment?

Rationale:

While COMPASS does integrate a number of tools that can be employed for various V&V tasks during the system design, the specific support for the system designer to integrate system modelling and verification is only limited. As described above, the process of verifying given properties typically amounts to successively refining the model to be developed, and sometimes also the property of concern. However, there is no specific design tool for expressing models in the D-MILS AADL language. Models are written in simple text files and then loaded into Compass tool for the purpose of verification; a graphical editor to develop MILS-AADL models is not provided. Furthermore, more elaborate feedback to the designer for detecting flaws in models or property formulas would be helpful, as well as typical book-keeping support such as the tracing of properties which have already been established.

Verdict:

Not integrated at all – **Partially integrated** – Integrated – Very well integrated

6.5 Verification Maturity

6.5.1 Maturity of functional verification

The perceived maturity of the D-MILS technologies for carrying out functional verification for an industrial demonstrator application.

Question 1: What is the perceived maturity of D-MILS technologies for carrying out functional property verification for SMG case study?

Rationale:

The model-checking tools underlying COMPASS for verifying safety and security properties of D-MILS models are world-class and state-of-the-art. However, their integration into the COMPASS

tool as a frontend needs still to be improved in order to be routinely applied for industrial-level development and analysis tasks.

Foremost, there is no integrated documentation available that encompasses all the capabilities of the various tools can be used for developing D-MILS AADL models and verifying related properties. Moreover, in addition to what has been stated in the previous subsections, information provided to the designer during the development process is not always optimal. For example, while syntax errors that are found for a MILS-AADL model are reported in very descriptive way that is helpful for the user, errors that occur during the verification simply result in a message saying that the verification task could not be completed.

Finally, the different capabilities of the underlying model-checking tools may have an influence on the way D-MILS AADL models need to be designed in order to accomplish given verification tasks. For instance, the model checking component for the monolithic verification of LTL properties do not support explicit fairness assumptions which renders verifying LTL properties difficult for many practical cases. On the other hand, the front-end to the OCRA tool for compositional verification offers the choice to include fairness in the verification.

We hence distinguish in our evaluation verdict between the maturity of the verification technology itself and maturity of its integration in the D-MILS tools.

Verdict:

- Verification technology: Very Immature – Immature – **Mature** – Very Mature
- Tool integration: Very Immature – **Immature** – Mature – Very Mature

7 Assurance case

For applications with specific safety and security requirements, an assurance case is essential to demonstrate and provide sufficient arguments and evidence that an industrial application is in compliance with regulations for security and dependability. Traditionally, preparing an assurance case is a very time consuming task for application developers and less widely used for applications where there are no government certification regulations, even though many other types of applications would benefit from such usage.

The evaluation of the assurance case technologies from D-MILS has a two-fold scope and includes both subjective and quantitative evaluation methods. One of the main scopes of the D-MILS assurance technology evaluation is the assessment of all of the claimed capabilities of the D-MILS assurance case tool (MBAC). MBAC claims to have certain capabilities for creating and managing assurance cases in a model-based fashion. MBAC is an Eclipse-based application which allows its users to model assurance cases compliant with an Eclipse Modeling Framework (EMF) meta-model for assurance cases structured according to the Goal Structuring Notation (GSN). The respective meta-model has been proposed by safety engineers working in the D-MILS project. Furthermore, MBAC implements model-based transformation to support construction of assurance cases by automated pattern instantiation. The model-to-model transformation algorithm is implemented as an Epsilon Object Language (EOL) program that runs on the Eclipse platform. The EOL program requires as input GSN argument pattern models, reference information models and a weaving model. The instantiation program first identifies the elements requiring instantiation in the GSN argument pattern models. Second, it determines which information from the reference information model is required to instantiate each GSN element by querying the weaving model. The program then obtains the required information from the relevant information models and finally outputs instantiation information. The second main scope of this evaluation has been asserting the suitability of the D-MILS assurance case patterns. These patterns have been created in order to support the certification of a D-MILS system.

In order to evaluate all of these capabilities, we built during the course of this evaluation a fragment of the interim assurance case for an industrial demonstrator application, namely a smart grid application. This assurance case was built twice - first we built it manually, using traditional development methods and second we built it with the help of D-MILS assurance technologies, which are based on automatic instantiation of an assembly of interrelated assurance case patterns. In both cases we used three of the D-MILS patterns (i.e. the *System properties*, *Composition* and *Process* patterns) in order to create part of the assurance case for our smart grid. We only considered these three patterns because these patterns were provided by the D-MILS assurance tool at the time when we performed the evaluation. We then compared the two alternatives for developing assurance cases. For assessing the quality and relevance of the D-MILS assurance case patterns, we created a meta assurance case for the smart grid system. A meta assurance case is a an assurance case reasoning on the system's assurance at a very high level of abstraction, offering a bird's eye-view on the assurance of D-MILS systems. This meta assurance case contains the system properties offered by the D-MILS technologies which contribute at demonstrating various dependability properties relating to a system or a component such as security and safety. We also analyzed how much the D-MILS patterns cover the D-MILS meta assurance case created by us and how much information is contained by the patterns, but not by our meta assurance case. These analyses and comparison supported us in determining to which extent the

D-MILS assurance case technologies are adequate for certifying a system which is compliant with the D-MILS architecture and platform.

Assurance cases are continually in development, along with system development and operation. The D-MILS assurance technologies support the continuous self-evolution of safety argumentation of a changing D-MILS system and operation environment. Not using the D-MILS assurance patterns impacts significantly the quality of the assurance case.

7.1 Assurance Case Usability

Automation of assurance cases is an important innovation claimed by the D-MILS project and several different aspects related to usability are of particular interest for the tools and methods provided for preparing the assurance case. These could be considered degree of satisfaction measures amongst developers who use the tools in fundamental areas such as satisfaction regarding automation, usefulness of the assurance case outputs, level of integration with analysis and satisfaction that the results support formal certification procedures. In the following we evaluate the D-MILS technology w.r.t assurance cases according to these measures.

7.1.1 Assurance case automation

MBAC's core feature is the manual creation and editing of assurance cases. In addition to this core feature, MBAC provides its users with features such as automated construction and assessment of assurance cases. The automated construction of assurance cases is accomplished by assembling assurance cases out of manually created assurance case patterns self-instantiated with content produced by a formal verification tool or with information provided by system development artifacts, such as system model, requirements analysis documents. During the evaluation we carried out assurance case preparation tasks offered by D-MILS assurance technologies while developing an interim assurance case for our smart grid industrial demonstrator application. In this section we assess the perceived level of automation offered by these tasks, by addressing both subjective and quantitative metrics. We evaluated using a survey instrument with relative questions addressing perceived savings in time and effort in constructing assurance cases and supporting arguments.

Question 1: What is the degree of automation of the D-MILS assurance case tool?

Method of assessment: In the first phase of this evaluation step, we constructed a fragment of the interim assurance case of our smart grid systems in the traditional fashion. By traditional fashion we mean that we used a GSN editor (i.e. AF3 GSN editor) [2] to develop our assurance case. In this editor we integrated the assurance argumentation structures from three of the D-MILS patterns. The D-MILS patterns have been instantiated in this case manually. In the second phase we used for the development of the same system's assurance case the same three D-MILS patterns, whose instantiation benefited from automated support. We only used MBAC to instantiate the D-MILS patterns and export them as Extensible Markup Language (XML) files in order to automatically integrate them in our editor. The rest of the interim assurance case has been created only in the AF3 GSN editor. We then went through all the claims the developers of MBAC wrote down about the tool's performance regarding automation in the D-MILS deliverables. We assessed the validity

of each of those claims according to our own experience gathered while using D-MILS assurance technologies during the development of the assurance case for our smart grid system. As for the quantitative measurement of the perceived level of automation of the assurance case preparation tasks, we use the X formula. The X formula takes the total number of argument elements which have been automatically generated by pattern instantiation and divide it by the total number of argument elements in the assurance case.

Rationale: Using the D-MILS patterns for the creation of the assurance case of a D-MILS system facilitates significant reduction of the time required for creatively building up the argumentation structure. This is due to the fact that the core structure of the assurance case can be reused from the patterns. However, the structure offered by the assurance patterns captures only part of the assurance case. The rest of the assurance case has to be created in the traditional way. This costs significant amounts of time and effort. Manually constructing assurance arguments is time consuming especially due to the creative effort implied by the necessity of building assurance arguments that are valid and syntactical and logical error-free. The traditional approach is error prone, as it relies much on the capabilities of the safety engineer creating the assurance case. Moreover, even just one fragment of an assurance case rapidly evolves into a very large argument structure, which becomes troublesome to manage and evaluate during iterative system development.

In the second phase of the evaluation we instantiated the three selected D-MILS patterns automatically with the help of the MBAC tool. The D-MILS assurance case technologies contribute at the increase of the level of automation for the argument creation process as they propose a method for automated instantiation of patterns. As specified in Deliverable 4.3., in the MBAC tool a weaving model captures the dependencies between the reference information meta-models and the GSN argument patterns. The dependency information explicitly captured in the weaving model enables the automatic instantiation of assurance arguments. On the one hand, automated instantiation of patterns eases the change management process of evolving assurance cases. With the help of the weaving meta-model the MBAC tool manages to scrutinize large volumes of information, from diverse sources, and extracts information needed by argument elements from the assurance patterns. Thus, logical errors in an argument can be avoided by automatically constructing assurance arguments from system artifacts. On the other hand, another benefit of the automated pattern instantiation feature of the MBAC tool is that syntactic errors can be eliminated by simple syntax checking. Furthermore, acyclicity in arguments can be checked by ensuring that there are no loops in the argument structure.

Verdict: On one hand, solely by being able to reuse argument structures from the D-MILS patterns increased the level of automation provided by the D-MILS technologies. These patterns help the safety engineer at identifying the skeleton of the assurance case for a system which is build according to the D-MILS technologies. On the other hand, the automation provided by the D-MILS technologies is supported by the automatic instantiation of assurance case patterns supported by the MBAC tool.

Question 2: Once assurance arguments are automatically generated with the D-MILS assurance case tool how easy it is to integrate them in the system's assurance case?

Method of assessment: We first evaluate the integration of patterns into the system's safety assurance case from a technical point of view. This means that we try to export the instantiated patterns into a safety case graphical editor, where we have developed the rest of the assurance case. We then

evaluate qualitatively the ease of combining manually created arguments with automatically generated fragments of the assurance case resulted from automated pattern instantiation.

Rationale: According to our experience, the user can easily export the instantiated patterns from MBAC in XML format. However, the import of the instantiated pattern into the the user's assurance case editor might sometimes be troublesome as the GSN meta-model implemented in the user's assurance case editor must be compliant with the GSN meta-model implemented in MBAC. We have not been able to import the instantiated patterns into our AF3 GSN editor because of model inconsistencies.

D-MILS patterns offer a clear description of the interface of the module which contains the pattern. In assurance cases constructed without reusing any argument structures the module interfaces are not always explicitly and completely specified. Thus, combining manually created arguments with automatically generated fragments of the assurance case from the D-MILS patterns has a significant advantage. Whether the instantiation of patterns is done automatically or manually has no impact in the integration of the instantiated argument patterns into the rest of the assurance case.

Verdict: Very Difficult – Difficult – **Easy** – Very Easy

7.1.2 Assurance case usefulness

In this section we analyze the perceived usefulness of the D-MILS assurance case tools output for carrying out certification tasks for an industrial demonstrator application.

Question: What is the usefulness of the D-MILS assurance case tool for creating relevant input for the certification of the smart grid case study relative to the effort required for their use?

Method of assessment: On the one hand, assurance cases are the basis of discussion among different stakeholders, such as designers, manufacturers, operators, maintainers and regulators about the level of assurance of the system under certification. This is due to the fact that assurance cases contain both claims and evidence on the safety and security of the system. During these discussions, the stakeholders reason about the assurance arguments and identify potential errors in the arguments or discover contra-arguments. Hence, during this evaluation step we tried to assess to which extent the assurance case created with the help of D-MILS assurance technologies facilitates the communication among stakeholders. The verdict we give is based on the experiences we had on meetings with different stakeholders about the assurance of the smart grid system. We kept scores by following the qualities which an assurance case should demonstrate according to [5].

On the other hand, another important contribution of an assurance case for certification is to enforce design decisions to be in accordance with the safety and security goals of the system. During this evaluation step, we looked for design and verification decisions driven from the D-MILS assurance case patterns.

Rationale: During this evaluation, we acknowledge the advantage of using the D-MILS patterns for building a common understanding among stakeholders of the used terminology by distributing to all stakeholders the D-MILS deliverable documenting the patterns (see Deliverable D 4.3.). This deliverable contains a thorough description of the pattern, explaining all the claims and also the roles to be instantiated in the patterns. After reading the deliverable there were no noticeable misunderstandings among stakeholders concerning the meaning of an argument claim. However, one problem has been

identified, concerning the evidence availability. During the usage of D-MILS assurance technologies we missed a system of identification for a particular version of a particular system artifact. It always takes time to identify the artifact referenced by a certain claim; time which can be significantly reduced by having an identification system. However, such kind of identification systems is out of the scope of this project.

Automatic instantiation of patterns has also its drawbacks. One of these drawbacks is that the safety engineer risks to rely completely on the automatic instantiation of the patterns, without thinking about the extent of the impact a change in the system artifacts might have on the assurance argumentation and the other way around. This might cease in having the construction of the assurance case simply as a paper exercise to get a system certified. There exists literature on the subject *confirmation bias* in safety domain [6]. The concept of confirmation bias affirms that, on the one hand, system engineers always try to build systems compliant with safety requirements and then to verify by themselves that the system is safe. When they execute the verification, the system engineers have already the mind set on the fact that they already did everything possible to develop a safe system. On the other hand, system safety engineers try to demonstrate that the system is unsafe. By simply running the automatic pattern instantiation, safety engineers might overlook important aspects of system safety by not completely reading and analyzing the instantiated assurance argument fragment.

However, patterns encapsulate good practices when it comes to system assurance certification. From the D-MILS patterns one can extract several verification and development choices, which are seen as best practice strategies. These encourages the idea that a system can be developed along with its assurance case, by reasoning about development and verification decisions with the help of the assurance case. A pattern's contribution at making system design and verification decisions is an indicator of the contribution of the assurance pattern at the quality of the system it argues assurance for. On the one hand, if the safety engineer opts for the *Composition pattern*, the pattern obliges the engineer to use apply formal reasoning to software in order to demonstrate that a D-MILS system enforces its required properties. On the other hand, the *D-MILS Platform* pattern suggest that network communication in a D-MILS system has to occur in accordance with Global Information Flow Policy and this is enforced, as suggested by the *TTEthernet* pattern, by the usage of a TTEthernet network as part of a D-MILS platform.Without having patterns there is nothing to guide these design decisions from an assurance perspective. However, the automated pattern instantiation might distract the safety engineer. But, when used rationally, patterns have a valuable contribution to the technologies strategies applied to a certain system.

Verdict: During meetings concerning the assurance of our smart grid system, we used as basis for discussion the assurance case constructed with the D-MILS patterns. Our experience during those meetings revealed absence of vagueness of the text contained in the D-MILS instantiated argument patterns and correct syntax of the GSN structure. Therefore, we can confirm that the assurance case generated with the help of the D-MILS technology provides satisfactory means of communication between different stakeholders.

Also, the D-MILS assurance case patterns provide a means for reasoning about the system in all of the phases of system development because the usage the D-MILS assurance case patterns while arguing the safety and security of a D-MILS system is an aid to making design decisions as it sets out what you will be required to do to make an assurance case, which will then constrain the design decisions.

Not helpful at all – Not that helpful – **Helpful** – Very Helpful

7.1.3 Assurance case design integration

The review of the Nimrod accident [6] indicates the necessity of the ability to explicitly define pattern roles traceable to specific types of information in system artifacts. Hence, we evaluate next the perceived level of integration of the D-MILS assurance case tools with the D-MILS design environment for an industrial demonstrator application. For this evaluation we used a survey instrument with relative questions addressing observed integration of the assurance case tools with the design environment.

Question: How well does MBAC utilize the output of the D-MILS design environment tools in order to create an assurance case for the smart grid case study?

Method of assessment: We answered this question by firstly inspecting the D-MILS design environment and then counting the patterns roles which are to be instantiated with references from the design environment. Moreover, we compared the references to the design environment from the assurance case constructed with the help of the D-MILS assurance technologies against the references to the design environment we expected to have. We were interested in observing if any important elements from the design artifacts were omitted in the D-MILS patterns. As a quantitative measurement, we used the following formula, which computes how many of the references from all of the D-MILS patterns are to the MILS-AADL models:

$$Degree\,of\,design\,integration = \frac{ISdde(Instantiation\ sources\ that\ come\ from\ the\ D{-}MILS\ design\ environment)}{ISa(Instantiation\ source\ that\ come\ from\ whatever\ other\ artifacts\ such\ as\ documents,\ etc.)}$$

Rationale: The tool employs the output of all of the D-MILS design environment tools, as it uses for the instantiation of the D-MILS assurance argument patterns information from the MILS-AADL model (the MILS-AADL file) and from the platform configuration file. Since faults often are caused by the underlying computing platform, it is a very good indicative that the platform is regarded in the assurance case, by being referenced in the *D-MILS Platform* pattern. Moreover, the *Trusted Software Component* pattern ensures that the modular architecture of the assurance case maps the modular design architecture of the system, having one separate assurance argument module for each system component. This contributes at an effective change management.

Verdict:

$$Degree\,of\,design\,integration = \frac{22}{26}$$

7.1.4 Assurance case analysis integration

In this phase of the evaluation we estimate the perceived level of integration of the D-MILS assurance case tools with the D-MILS analysis tools for an industrial demonstrator application.

Question: How well does MBAC utilize the output of the D-MILS analysis tools in order to create the assurance case of the smart case study?

Method of assessment: An assurance case should allow stakeholders to understand assurance claims without having to understand the technical details of the formal verification process. Moreover, an assurance case should not only include the results of the formal verification, but also should

comprise a description of the verification environment. During this evaluation phase we looked for indices of the contribution of a satisfied formal property to the overall safety or security of the system. Last but not least, we expected to also have confidence arguments concerning the tool and the process of the formal verification.

Verdict: The MBAC tool utilizes the output of all of the D-MILS analysis environment tools. MBAC uses as supporting evidence for the *Composition* argument pattern verification results provided by the D-MILS analysis tools after verifying the MILS-AADL model. Also, the fact that inside of the *Composition* argument pattern there is an argumentation leg solely concerning the confidence in the verification process indicates that the assurance case constructed based on the D-MILS patterns meets our expectations about the integration of the D-MILS assurance case tools with the D-MILS analysis tools. Moreover, we appreciate the useful feature of the MBAC tool of enabling automatic report of mismatch between assurance claims and software architecture when a formal property is not satisfied.

7.1.5 Assurance case design maturity

In this section we evaluate the perceived maturity of the D-MILS technologies for carrying out preparation of assurance cases for an industrial demonstrator application, by considering the tool's concepts, the technology requirements, and the demonstrated technology capabilities.

Question: How mature is the D-MILS assurance case tool for generating an assurance case for the smart grid case study?

Method of assessment: The maturity of the tool is determined by assessing the usability and sustainability of the MBAC tool and the readiness of the tool to be used in a real assurance cases development environment. We investigate the maturity of the MBAC tool by using it in a simulated environment, namely the creation of the assurance case for our case study. The assessment involves checking whether the software, and the project that develops it, conforms to various characteristics or exhibits various qualities that are expected of sustainable software. Moreover, we verify in this evaluation step if all the requirements which have been identified for the D-MILS research and development work integrating GSN and MILS-AADL in Deliverable D 1.3. were addressed by the MBAC tool.

Rationale: The MBAC tool is compliant with all the mandatory requirements presented in the D 1.3. deliverable. In the following, we enlist all the mandatory tool requirements of MBAC established in deliverable D 1.3. and explain shortly how each of these requirements was addressed.

Requirement (IGA-WP2.1): Use MILS-AADL to source information about the MILS policy architecture to build and inform the GSN argument.

Verdict: Addressed in the *System Properties* pattern. This pattern argues that a D-MILS system enforces its required properties.

Requirement (IGA-WP2.2): Use MILS-AADL to source information about properties of trusted components to build and inform the GSN argument.

Verdict: Addressed in the *Trusted Software Components* pattern. This pattern argues that a software component implements correctly each of the formal properties as described in the MILS-AADL specification.

Requirement (IGA-WP2.3): Use MILS-AADL to source information about security annotations on subjects to build and inform the GSN argument.

Verdict: Addressed in the *Composition* pattern.

Requirement (IGA-WP2.4): Use MILS-AADL to source information about security annotations on objects to build and inform the GSN argument.

Verdict: Addressed in the *Composition* pattern. Every security annotation on an object is seen as a formal property, hence a source of instantiation for the *formal property* role in the *Composition* pattern.

Requirement (IGA-WP2.5): Use MILS-AADL to source information about security annotations on data flows to build and inform the GSN argument.

Verdict: Addressed in the *Composition* pattern. Every security annotation on a data flow is seen as a formal property, hence a source of instantiation for the *formal property* role in the *Composition* pattern.

Requirement (IGA-WP2.6): Use MILS-AADL to source information about space isolation to build and inform the GSN argument.

Verdict: Addressed in the *D-MILS Platform* pattern. This pattern argues that a D-MILS platform guarantees the required properties. One of the required properties is space isolation and this must be guaranteed by the way the nodes communicate with each other. Inter-nodal communication is controlled by ensuring that network communication only occurs in accordance with Global Information Flow Policy (GIFP). The GIFP is a target specific configuration file and the *D-MILS Platform* pattern has a reference to this configuration file. The compliance with the Global Information Flow Policy is assured by the TTEthernet network.

Requirement (IGA-WP2.7): Use MILS-AADL to source information about time isolation to build and inform the GSN argument.

Verdict: Addressed in *D-MILS Platform* pattern. One of the required properties is time isolation and this must be guaranteed by the way the nodes communicate with each other. Inter-nodal communication is controlled by ensuring that network communication only occurs in accordance with Global Information Flow Policy (GIFP). The GIFP is a target specific configuration file and the *D-MILS Platform* pattern has a reference to this configuration file. The compliance with the Global Information Flow Policy is assured by the TTEthernet network.

Requirement (IGA-WP2.8): Use MILS-AADL to source information about task scheduling to build and inform the GSN argument.

Verdict: Addressed in the *Implementation* pattern. Whether or not the generated MILS Configuration Normal Form (MNCF) file is correct with respect to the MILS-AADL specification and satisfies all the constraints (the system model, the platform model and the additional defined constraints) is argued in the *Implementation* pattern. This pattern has a reference to the MNCF file. Among others, the MNCF file includes the scheduling of the subjects and nodes.

Requirement (IGA-WP2.9): Use MILS-AADL to source information about the effects of error injection to build and inform the GSN argument.

Verdict: Addressed in the *Composition* pattern, as it argues that the MILS-AADL error model is complete and correct.

Requirement (IGA-WP2.10): Incorporate aspects of the modular design indicated within the AADL to inform the structure of the argument.

Verdict: Addressed by the proposed frame of a D-MILS assurance case built by the instantiation of the D-MILS assurance case patterns as individual modules and by creating relationships between these modules.

Requirement (IGA-WP2.11): Accurately record and represent security properties of the D-MILS system. This shall detailing the security policy and details on encryption on all data channels.

Verdict: Addressed in the *Composition* pattern. This pattern argues that formally defined properties of a D-MILS system, including security properties, are satisfied by a MILS-AADL model of that system.

Requirement (IGA-WP2.1): Accurately record and represent dependencies in the D-MILS system.

Verdict: Addressed by the proposed frame of a D-MILS assurance case built by the instantiation of the D-MILS assurance case patterns as individual modules and by creating relationships between these modules.

We then evaluate the usability of the tool by assessing whether or not it is understandable enough, well documented, easy to install and it easy to learn how to be used. The D-MILS assurance case tool can be easily understood without a user guide, assuming that the user is acquaint with the Eclipse Framework, EMF modeling and XML markup language. However, a comprehensive, appropriate and well-structured user documentation is provided especially for those users that are not familiar with the technologies aforementioned. The deliverables, together with the user manual provide description of both what the software does and how it works. The user manual consists of clear, step-by-step instructions and gives examples of what the user can see at each step, by providing screen shots. Moreover, the user guide is based on a running example, example which can be found in the default workspace of the tool. This workspace is also available on the svn. Also, the user guide states command names and syntax and says what menus to use. Installing the tool is straight-forward as the user only needs to unpack a given archive and then run an eclipse instance. An installation manual is provided which presents the installation requirements. Moreover, once the user reads about all the functionalities of the tool and experiments with the embedded getting started *Starlight* example, it is easy to be able to utilize the tool's functionalities in order to instantiate the given patterns.

For assessing the sustainability of the MBAC tool, we checked the accessibility, portability, supportability, interoperability and understandability of the tool. The MBAC tool can be downloaded from the svn repository, however, the tool has be released only for Windows. Due to this users that work under other operation systems cannot work with the tool. Also, there are no source distributions available for download. However, these can be obtained from the developers. The MBAC tool is maintained by its developers, namely the people from *York University*, who also offer user support. As it is tool independent, MBAC offers a very nice feature to be integrated with a graphical assurance cases editor of the user's choice.

Next we describe our experience while using the tool. The user guide also describes very thoroughly how to create a new GSN pattern from scratch. Allowing users create their own patterns enables the increase of automation degree provided by the D-MILS assurance technologies. Once a safety engineer identifies an argumentation structure which encapsulates good practices when it comes to

assurance argumentation for D-MILS systems, she can create a new pattern, accompanied by a weaving model to help with the instantiation. The creation of a new pattern is intuitive as the user can use a graphical editor, where she can create new GSN elements by dragging and dropping from a view which contains all GSN elements. Each newly created GSN element can be annotated with various properties; properties which are taken from the GSN Standard. After creating the pattern, next time the user needs such argument structure, she only needs to create the necessary input files, instantiate the pattern and import the argument structure as an XML file to their own GSN editor.

In order to be able to use MBAC for pattern instantiation the user needs to feed the tool with following:

- An XML file containing the MILS-AADL system model. This file can easily be obtained with the help of the Compass tool.
- An instantiation of the starlight meta-model (ModelElement.xmi), where we point manually to the role bindings that are not to be found in the MILS-AADL model. This is more time consuming as it needs to be done by hand. Also, none of the manually created role bindings appear for more than one time in the argumentation structure. Hence, the instantiations done in this way are basically done in the old fashioned way of manual pattern instantiation.
- An instantiation of the process model. The creation of such model is also time-consuming. However, most of the process models can be reused in several assurance cases for different D-MILS systems (for example the OCRA verification process model).

An instantiated pattern can be visualized in several ways. At first glance, the most convenient way of visualization for an user would be the graphical visualization as a GSN diagram. MBAC enables the user to visualize a pattern, instantiated or not, as a GSN diagram (see Figure 17). However, this editor is quite rudimentary, as it does not depict different GSN elements with the shapes presented in the [1] GSN Standard. Also, one can only see the title of a certain GSN element, not the claim inside it. Therefore, this editor allows the user to visualize the structure of the argument, but not its instantiated or uninstantiated claims. Also there is no depiction of the to be developed or away entities. Another inconvenience of this editor is the fact that the user cannot zoom in or out. However, the intend of the D-MILS assurance technologies was to provide a model-weaving instantiation methodology independent of a GSN editor, so providing a more advanced visualization editor was out of the scope of the project. The visualization of argument structures as XML files allows the user to import the argument in a GSN editor of her choice. A much more helpful visualization of instantiated patterns is an editor which presents an instantiation table (see Figure 18). This table provides a better overview of how the claims in a pattern have been instantiated, as for each role name of every claim the role biding is given. This visualization is by far the one we used the most while developing our interim assurance case. However, for a more detailed description of an instantiated argument structure, we used the GSNmetamodel Model editor (see Figure 19). Even if the table editor provides a more intuitive view of how the pattern has been instantiated, the GSNmetamodel Model editor also contains the full claim, not only its id. Moreover, it offers information related to the status of the claim, namely whether it is undeveloped or not and whether or not some of its sub-claims are undeveloped.

Verdict: MBAC is mature enough (understandable enough, documented enough, easy to install and it easy to learn how to be used) to be used by experienced users. However, it might be cumbersome to be used by users which lack certain technical skills. Another drawback is that it is inopportune

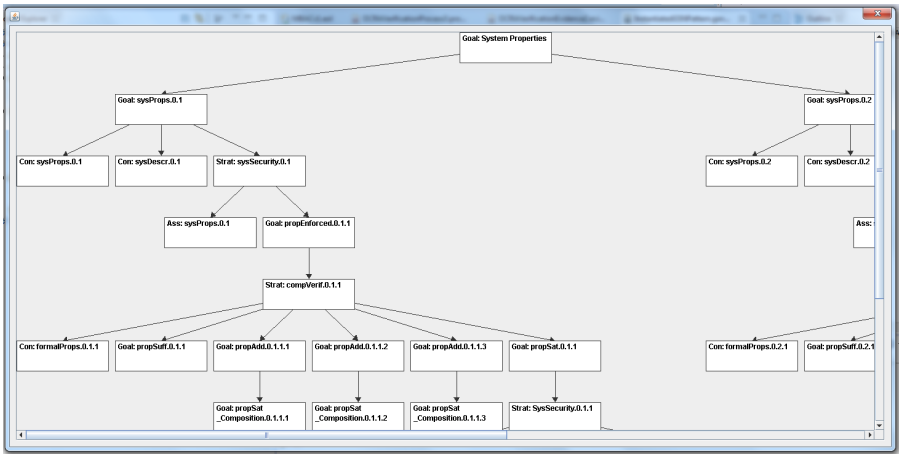

Figure 17: D-MILS GSN Editor

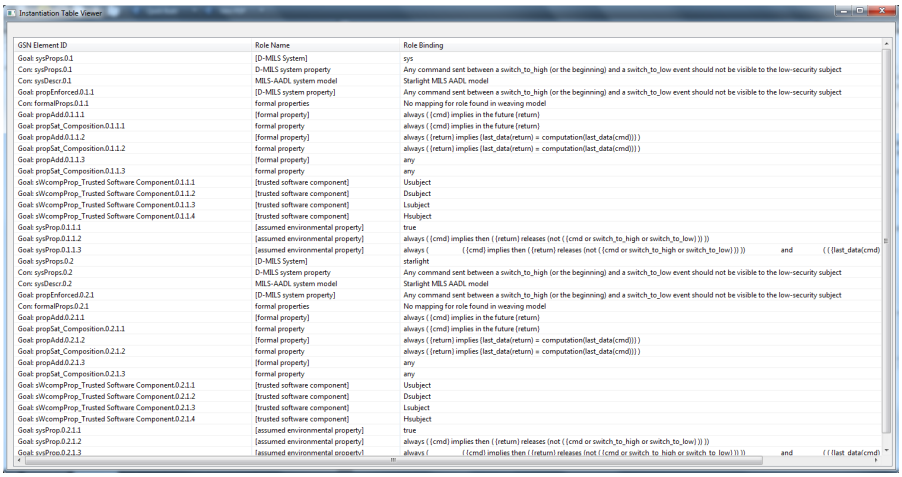

Figure 18: D-MILS GSN Editor

that MBAC is only available for Windows users. Even though the tool is not ready to be used in a real assurance cases development environment, it does satisfy all its imposed requirements. Having an industry ready tool was out of the scope of this project. In the context of the D-MILS project it was only intended that a prototype was created as a proof of concept.

Very Immature – **Immature** – Mature – Very Mature

7.1.6 Assurance case certification suitability

Certification is an important activity during the development of safety-critical systems. For applications with specific safety and security requirements it is a regulatory requirement that a safety or an assurance case is developed and reviewed as part of the certification process. An assurance case is essential to demonstrate and provide sufficient arguments and evidence that an industrial application will operate safely. This is usually done by demonstrating compliance with regulations for safety and security and dependability. In this section we evaluate the perceived suitability of the outputs

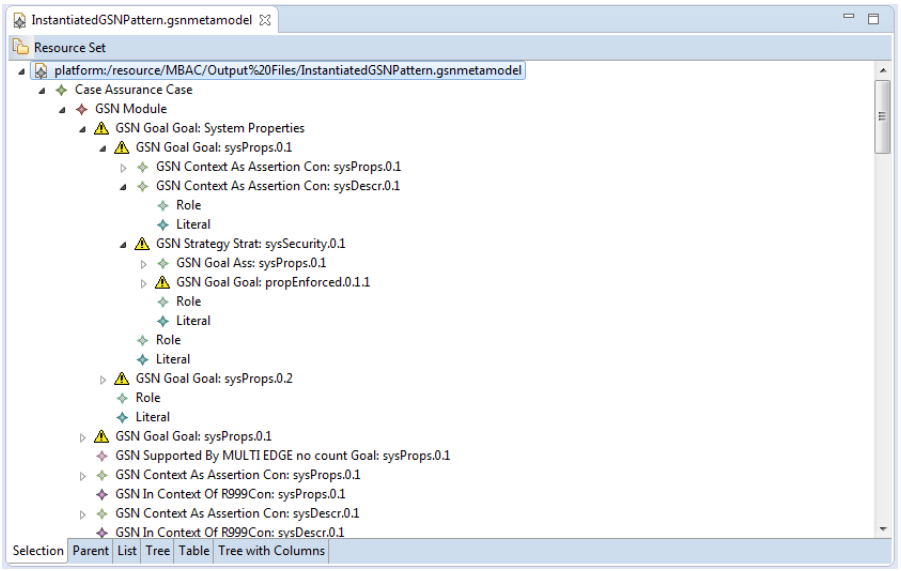

Figure 19: D-MILS GSN Editor

of the D-MILS assurance case technologies in satisfying certification requirements for an industrial demonstrator application.

Question: To which extent do the outputs of the D-MILS assurance case technologies support the certification of the smart grid case study?

Method of assessment: For assessing the suitability of D-MILS assurance technologies for supporting certification we first checked all of the proposed D-MILS argumentation structure against the argumentation structure overview we built (the aforementioned D-MILS meta assurance case). The meta assurance case for D-MILS systems (see Figure 2) is an initial screen of what we unquestionably wanted the assurance case of our grid system to contain. Our meta assurance consists of regular GSN elements. However, we did have some additions to the GSN notation. One special notation is that of the to-be-refined solutions. A to-be-refined solution suggests the type of the evidence item which will ultimately support the high-level claim supported by the to-be-refined solution, such as model checking results. These kind of solutions are to be refined in argumentation structures which connect the claim that the respective solution supports and the actual reference to the evidence item. Another alteration is that we classified these to-be-refined solutions in two types. We have the *once and for all* type, which suggests that these evidence items, once refined into argumentation structures can be taken as such to be used for the assurance of any D-MILS system. Hence, this argumentation structure is an argument template, which can be applied in any assurance case for D-MILS systems. The other type is the *application-specific* type, which indicates that the particular evidence item to be referenced in order to support our argumentation structure needs to be generated for each D-MILS system in particular.

We then did some literature review on important attributes of assurance cases for certification and, based on what there is in the literature, we assessed whether the D-MILS assurance case structure based on the D-MILS patterns has some of those attributes or not.

Rationale: First we assess if and how all of the D-MILS patterns' intent map to our D-MILS meta assurance case. The D-MILS meta assurance case encapsulates the main system properties that need to be demonstrated in order for the system to be certified. Thus, the D-MILS assurance case patterns should argue those properties. It would also be interesting to have the required properties evaluated quantitatively, but this is out of the scope of the D-MILS assurance technologies. This meta assurance case is meant to support us in analyzing whether an assurance case contains the minimum information required to make the assessment process viable or not. Our proposed meta assurance case argues the fact that a D-MILS system satisfies dependability, security and safety goals by referencing the capabilities of distributed MILS (i.e. MILS architectural strategy). The satisfaction of the goals is argued three fold: the policy architecture argument structure, the application code argument structure and the D-MILS platform argument structure. These three main argumentation legs are accompanied by context and assumption GSN elements, which argue about the system's environment. In Figure 20 one can see how the D-MILS assurance case pattern can be mapped upon our meta-assurance case. One the one hand, from Figure 20, one can see that an important assurance argument was left out in the patterns. Namely, information on whether or not the application code implements the architectural constraint is completely disregarded by the patterns. One the other hand, the D-MILS assurance case patterns do cover a significant part of the argumentation structure we expected. First, there is argumentation on the correctness of architectural refinement, as all of the architecture refinements (properties and constraints, error and failure conditions) are discussed in the D-MILS patterns. Moreover, there are arguments about how the system model is transformed into intermediate languages that are processed by D-MILS tools to perform compositional verification and generation of configurations and schedules for D-MILS Platform. Second, the *D-MILS Platform* and the *TTEthernet* patterns argue that the architectural information flow policy is implemented correctly on the technical platform. This is done by specifying that the D-MILS platform includes two or more nodes, each composed of a separation kernel and a MILS Networking System (MNS). The separation kernel and the MNS act together to support the realization of a MILS policy architecture transparently distributed over the nodes. Furthermore, the *Composition* pattern comprises argumentation about how the system's architectural description satisfies safety and security goals.

Second, we evaluate the assumptions from the D-MILS assurance case patterns. Assumptions in assurance cases are a delicate subject because there is always the question *Do we need argumentation and evidence for a certain claim* or *Do we have enough confidence and rely on the* ***assumption*** *that the claim is true?*. Every assumption in an assurance case needs to be checked if it is necessary to be mentioned and if it is reasonable to assume the claim as true and with no need for further argumentation. Usually the certifier tries to construct a plausible argument that the assumption is false. If no such argument is to be found, the assumption is left untouched. Otherwise, the operator is asked to replace the respective assumption with evidence and argumentation. Hence, assumptions are always hot topics during certification. The D-MILS patterns only contain one assumption, namely ***Ass:sysProps*** *The defined system properties must be complete and correct with respect to the threats, vulnerabilities and hazards of the system. It is not within the scope of the D-MILS assurance case to argue about these high-level properties and therefore an assumption is made that this is the case. It is necessary to demonstrate elsewhere that the system properties correctly reflect the system analysis.* This assumption however needs to be transformed into an argumentation fragment. In the assurance case created by us manually, we include this part of argumentation in the assurance case.

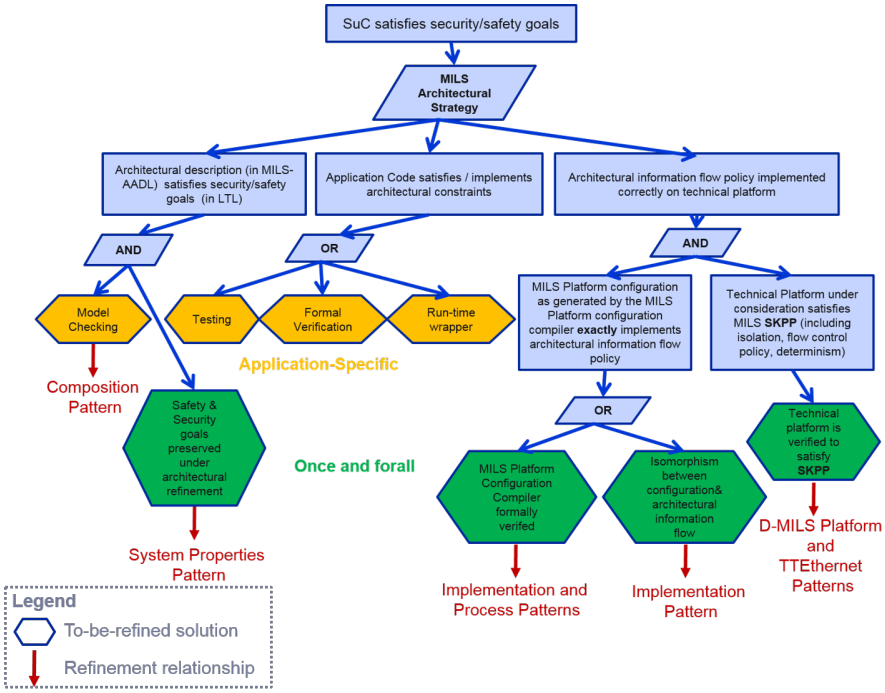

Figure 20: Refinement of the D-MILS meta assurance case by D-MILS assurance case patterns

Also, in order for an assurance case to support certification, it should be checked against well-known anti-patterns. We checked the D-MILS patterns against the anti-patterns presented in [7] and could find, as desired, no matches. Furthermore, it usually helps during certification if the operators document conformance with standards in the assurance case. However, there is nothing about standard requirements in the proposed D-MILS argumentation structure.

Verdict: The D-MILS assurance case patterns cover a significant part of the arguments specified by the D-MILS meta assurance case. This increases the confidence in the relevance of the D-MILS assurance case patterns. Also, despite of the fact that the D-MILS assurance technologies do not consider all the attributes an assurance case should consider in order to be adequate for usage in certification, they surely provide some of those attributes, supporting the operators in building a convincing assurance case.

Very Unsuitable – Unsuitable – **Suitable** – Very Suitable

7.2 Assurance Case Benefit

The use of GSN and the D-MILS tools opens up new opportunities to introduce more robust and automated methods for carrying out assurance tasks for many types of applications. Quantifying the benefits of using these innovative assurance techniques from D-MILS will encourage others who are developing critical applications to exploit D-MILS technologies. Two comparative measures can be utilised to quantify different types of benefits.

7.2.1 Manual assurance case comparison

In this evaluation step we assess the improvement in time required to develop assurance case arguments for a small subset of an industrial demonstrator application using the D-MILS assurance case tools in comparison with existing manual methods.

Experiment: During this evaluation step, a subset of the industrial demonstrator application was be selected as a test case and the associated assurance case arguments were be identified. A side-by-side comparison of instantiating three of the D-MILS assurance case patterns (i.e. the *System properties*, *Composition* and *Process* patterns) with and without the D-MILS MBAC tool will be carried out and the required time will be captured. The measure was the ratio of the two captured times. Moreover, we collected several metric scores for the process of creating an assurance case with the help of the assurance case patterns for a D-MILS system proposed in this project. These scores provide feedback as to the contribution of the proposed assurance case patterns at the quality of the assurance case.

$$TGRC(Time\ \ gain\ \ ratio\ \ for\ \ creating\ \ an\ \ assurance\ \ argument) = \frac{Tcd(Time\ necessary\ for instantiating\ assurance\ arguments\ for\ the\ SMG\ case\ study\ with\ the\ D{-}MILS\ technologies)}{Tcm(Time\ necessary\ for\ manually\ instantiating\ assurance\ arguments\ for\ the\ SMG\ case\ study)}$$

Rationale: On one hand, the instantiation of the three selected D-MILS assurance case patterns for the smart grid with the help of MBAC took approximately 15 minutes, including the creation of the xml file with the MILS-AADL model and the instantiation of the patterns. On the other hand, manually instantiating the arguments suggested by the same three D-MILS patterns took approximately three hours. In Figures 21, 22, 21, one can see one of the manual instantiations of the three selected patterns in the AF3 GSN editor. Whereas in Figures 24, 25 and 26 one can see how the same patterns have been instantiated automatically in MBAC. The instantiated patterns from MBAC in our figures are visualized with the instantiation table editor. There is one substantial difference in the two types of instantiation. In MBAC, if a pattern has to be instantiated multiple times (for example the composition pattern has to be instantiated for each of formal properties), the instantiation output file (i.e. the *InstantiatedGSNPattern.gsnmetamodel* file) contains all the instantiations. Whereas in the AF3 GSN editor for each instantiation of a pattern we have a separate GSN argumentation module. The instantiation depicted in all of the figures is for the *Battery Root* system of our smart grid case study, which contains two sub-components, namely the*Battery* and the *BatteryController*. As the assurance argument structures tend to get large and hence hard to comprehend, we only show in these figures the instantiation where we only argue the satisfaction of one requirements, namely the *Every battery component shall not be overloaded. This means that if the battery status is full, the control system shall not send any further loading signal* requirement.

In Table 2 we write down for each of the D-MILS pattern the claim coverage. The intent of claim coverage is to identify how many claims are supported by evidence. The intent of analyzing the claim coverage of each of the D-MILS patterns is to assess if there is enough overall support for the parent claim. This helps at establishing the effort the safety engineer needs to put into the assurance case development after instantiating the pattern in the assurance case. A high coverage is desirable. As one can see from Table 2, for approximately two thirds of the claims from the patterns there is either already an argumentation and a solution provided by the patterns or they are further developed in other D-MILS patterns. However, almost one third of the claims stated in the D-MILS assurance case patterns need to be further developed without any provided indication on how they should be argued.

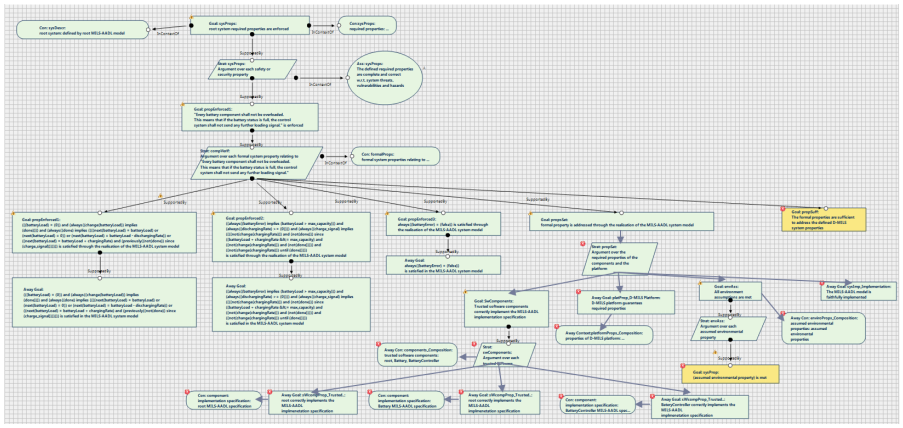

Figure 21: The instantiation of the *System Properties* pattern for the *root* component visualized in AF3 GSN editor.

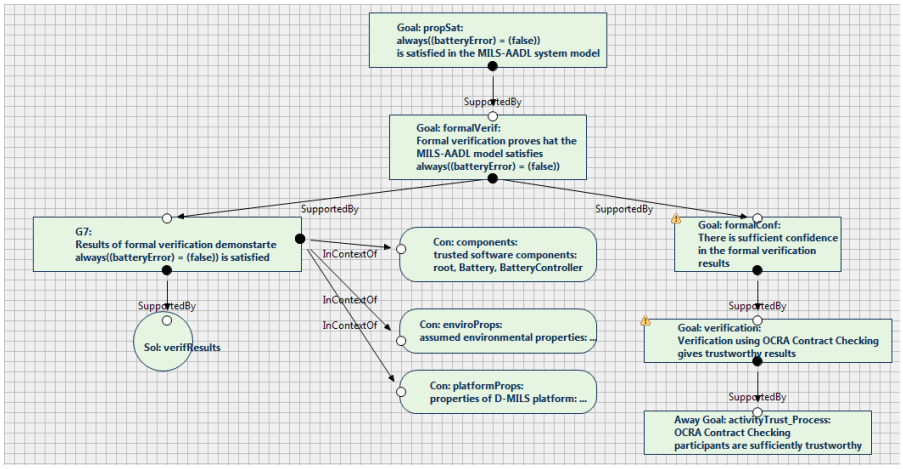

Figure 22: The instantiation of the *Composition* pattern for the *always((batteryError) = (false))* property of the *root* component visualized in AF3 GSN editor.

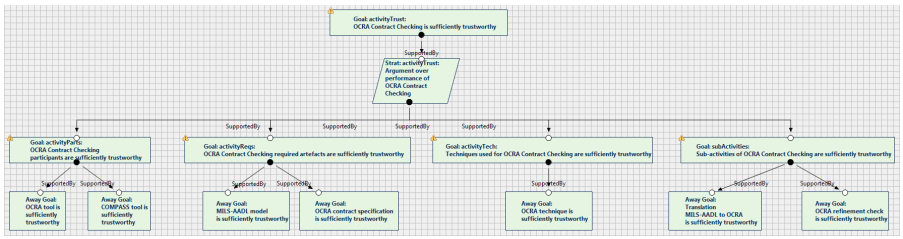

Figure 23: The instantiation of the *Process* pattern for the OCRA contract checking process visualized in AF3 GSN editor.

Hence, we can say that even with the automation provided by the D-MILS assurance technologies the construction of the assurance case remains largely manual.

Assurance patterns comprise the structure of an argumentation, without specific information about the system under certification and therefore can be instantiated in multiple situations. Thus, having a

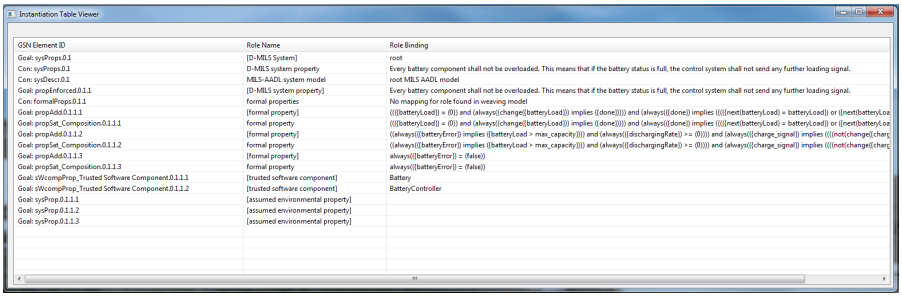

Figure 24: The instantiation of the *System Properties* pattern for the *root* component visualized in MBAC.

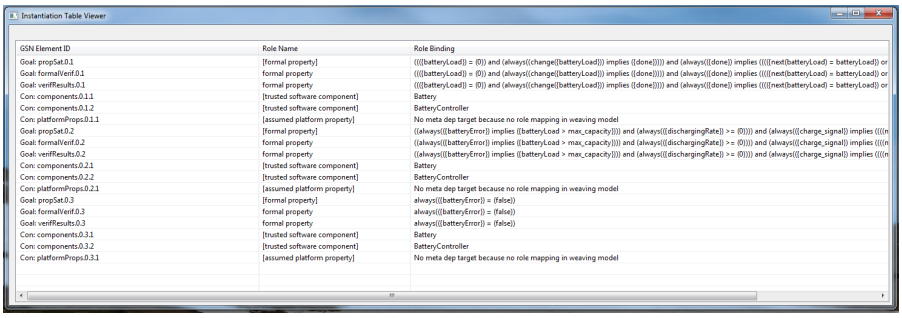

Figure 25: The instantiation of the *Process* pattern for the OCRA contract checking process visualized in MBAC.

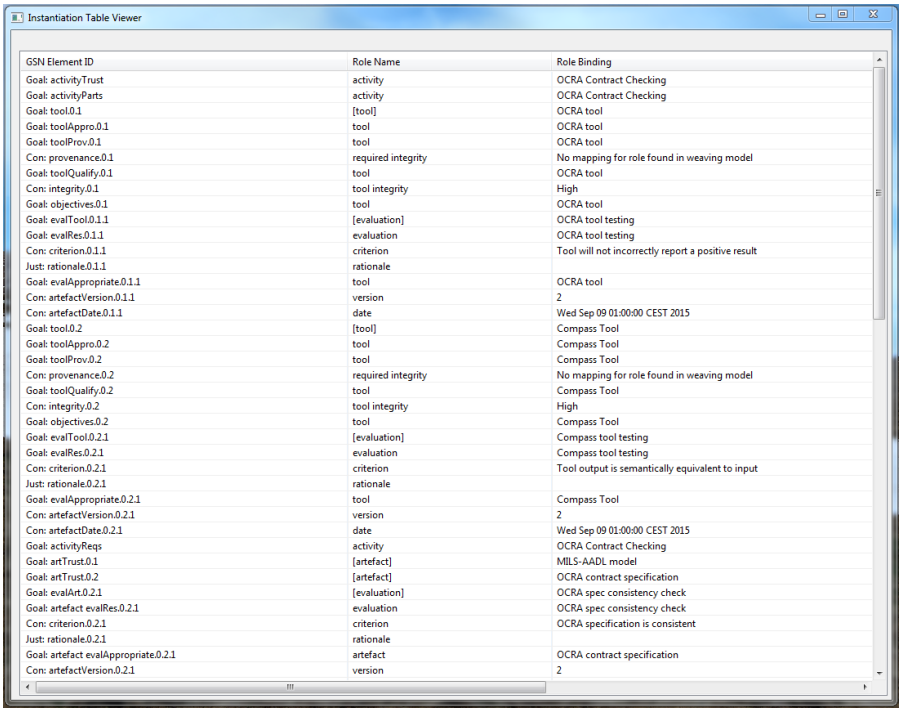

Figure 26: Refinement of the D-MILS meta assurance case by D-MILS assurance case patterns

catalog of patterns which can be applied to any D-MILS systems, eases the process of assurance case creation. Also, the D-MILS assurance case patterns suggest design and process choices that should be made while developing and operating a D-MILS system. Another important feature of the D-MILS assurance technologies is that they facilitate an easy method of assembling assurance argumentation fragments by providing a very detailed documentation about the interfaces of the modules comprising the instantiated patterns.

Pattern Name	**Coverage of Claims = number of developed claims / number of claims**
System Properties	3/5
Composition	3/3
Trusted Software Components	0/1
Implementation	4/5
D-MILS Platform	13/17
Process	5/5
Tool	2/5
Person	2/3
Organisation	1/3
Artefact	1/3
Technique	1/2
TTEthernet	6/9

Table 1: This table depicts for each pattern the claim coverage. The claim coverage identifies how many claims are supported by evidence or are further developed in other patterns.

Traditionally, the instantiation of patterns occurs manually by obtaining the needed information from design or analysis documentation, or directly from an engineer. The D-MILS approach of constructing assurance cases directly instantiates argument patterns with information from system artifacts. This helps at avoid potential humans errors, hence increasing the level of accuracy of the assurance case. The MBAC tool also offers highlights automatically the claims and evidence where the information in the system artifacts is incomplete.

Apart from gaining time by not having to describe the patterns in a tool of her choice, MBAC also offers other features. One of these features is the automatic generation of claims which support a higher-level claim by a multiplicity relationship. This feature frees the user from the time-consuming task of instantiating a single to-be-instantiated claim multiple times for different instantiation items. Also, not having to instantiate a reference to system artifacts whenever it appears into assurance case patterns, but instead creating a weaving-model for automatic instantiation of patterns is time saving. This is an efficient method of avoiding errors caused by user's distraction. Unfortunately, half of the to be instantiated elements from the patterns need to be instantiated manually, as they are not to

be found in the MILS-AADL model. Such elements are instantiated by creating a special mapping meta-model (Ecore file) and an instance of this meta-model (a XMI file). Furthermore, most of these to be automatically instantiated elements appear only once the argumentation. Table 2 depicts for each role form each of the D-MILS assurance case patterns the number of appearances. Also, in the table it is specified if each of those roles appear more than once in the argument structure. The more frequently a role appears in a pattern, the higher the level of automation. If a role only appears only once and it will not be multiple times instantiated, it does not contribute to the level of the automation of the D-MILS assurance technologies. Whether a role will be instantiated with more than one artifact, is suggested by the fact that at some point in the argumentation structure, there is a connection marked with multiplicity on a higher level in the argumentation flow than the claim that contains this role; flow which also contains the claim containing the role. However, the automatic instantiation of a role, even if it happens only once, still helps at avoiding instantiation errors. Also, the idea of automated instantiation has potential for increasing the automation level of the assurance case preparation tasks if individual to be instantiated elements appear more frequently in the patterns.

Pattern Name	**Role**	**Number of appearances**	**Multiple times instantiated**
`System Properties`	D-MILS System	2	No
`System Properties`	MILS-AADL system model	1	No
`System Properties`	D-MILS system properties	1	No
`System Properties`	D-MILS system property	3	Yes
`System Properties`	formal properties	1	Yes
`System Properties`	formal property	3	Yes
`System Properties`	trusted software components	1	Yes
`System Properties`	trusted software component	1	Yes
`System Properties`	component MILS-AADL specification	1	Yes
`System Properties`	assumed D-MILS platform properties	1	Yes
`System Properties`	assumed environmental properties	1	Yes
`System Properties`	assumed environmental property	1	Yes
`Composition`	formal property	4	No
`Composition`	trusted software components	1	No

Composition	assumed environmental properties	1	No
Composition	assumed platform properties	1	No
Composition	formal verification results for formal property	1	No
Composition	technique	1	No
Composition	property type	1	No
Trusted Software Components	trusted software component	1	No
Trusted Software Components	trusted software components	0	No
Trusted Software Components	component MILS-AADL specification	1	No
Implementation	MILS configuration-normal-form (MCNF) config file	0	Yes
Implementation	system model	1	No
Implementation	platform components	1	No
Implementation	platform component	3	Yes
Implementation	types of platform components	1	No
Implementation	component config file	1	Yes
D-MILS Platform	GIFP configuration file	1	No
D-MILS Platform	Lynx Secure software architecture design	5	No
D-MILS Platform	nodes	1	No
D-MILS Platform	node	1	Yes
D-MILS Platform	assumed platform properties	1	No
D-MILS Platform	assumed property	1	Yes
Process	Activity	13	Yes
Process	participant	4	Yes
Process	required artefact	1	Yes
Process	technique	1	Yes
Process	produced artefact	1	Yes
Process	subActivity	1	Yes
Tool	participant	1	No

Tool	tool	5	No
Tool	Activity	1	Yes
Tool	required tool integrity	1	No
Tool	tool integrity	1	No
Tool	objective	1	No
Tool	evaluation	2	Yes
Tool	criterion	1	Yes
Tool	rationale	1	Yes
Tool	version ID	1	Yes
Tool	date	1	Yes
Person	participant	5	Yes
Person	Activity	2	Yes
Person	capability	1	Yes
Person	experience	1	Yes
Organisation	organisation	3	Yes
Organisation	accreditation	1	Yes
Artefact	required artefact	1	No
Artefact	produced artefact	3	No
Artefact	evaluation	2	Yes
Artefact	criterion	1	Yes
Artefact	rationale	1	Yes
Artefact	version ID	1	Yes
Artefact	date	1	Yes
Technique	technique	3	No
Technique	provenance	1	No
TTEthernet	Results of network fault propagation analysis	1	No

Table 2: This table depicts for each role form each of the D-MILS assurance case patterns the number of appearances.

From our own experience, even if the automatic instantiation run terminates almost instantly, we still needed to invest some time in order to configure the *ManualElement.xmi* file. As the patterns roles which are instantiated with the help of the *ManualElement.xmi* file only repeat themselves twice, the time saving was almost nonexistent. The same happens for the roles whose instantiation reference is *unknown*.

Verdict:

$TGRC =_{approx.} \frac{1}{12}$

The construction of a model-based assurance case as offered by the the D-MILS assurance technologies brings the benefits of model-driven engineering, such as automation, transformation and validation, to what is currently a lengthy and informal process.

7.2.2 Assurance case change comparison

In this section we evaluate the improvement in time required to regenerate the assurance case following a component change or replacement for a small subset of an industrial demonstrator application using the D-MILS assurance case tools versus previous manual methods.

Experiment: A component change was introduced (an extra transition between two modes was defined) and assurance case was regenerated manually and using D-MILS technologies as a side-by-side comparison. The time required for regeneration of each assurance case was captured. The measure was the ratio of the two captured times. We evaluated the improvement in time required to regenerate the assurance case using D-MILS technologies both qualitative and quantitative. Our qualitative evaluation consists of the description of our own experience while making changes in our assurance case. We also compare what we had expected and needed with what is actually provided by the features of MBAC. As for the quantitative evaluation we use the following formula, which computes the time gain ratio for modifying an assurance case (TGRM): $TGRM = \frac{Tmd(Time\ necessary\ for\ modifying\ assurance\ arguments\ for\ the\ SMG\ case\ study\ with\ the\ D-MILS\ technologies)}{Tmm(Time\ necessary\ for\ manually\ modifying\ assurance\ arguments\ for\ the\ SMG\ case\ study)}$

Rationale: During system development, the design of the system frequently suffers alterations. Moreover, changes can appear even during the operational life-time phase of the system. Hence, assurance cases should be regarded as constantly evolving artifacts, being means for continually monitoring and assuring safety throughout the life-cycle of a system. Even small changes in the system design have potential of having serious impact on the assurance argumentation. Thus, it is valuable to have a systematic maintenance methodology and the D-MILS assurance technologies propose one. The D-MILS proposed maintenance methodology explores the connection between assurance case patterns and software maintainability, by offering the feature that the user can execute the patterns' instantiation every time one of the system artifacts has been changed. Hence, the argument structures created by pattern instantiation can be automatically regenerated. However, for the rest of the argumentation, manual update needs to be done.

Furthermore, in order to keep the certification document updated the safety engineer must relate the formal verification to the detailed design decisions. When we applied the change to the system design, we determined which assumptions in the formal verification could have been affected by the change and we revisited them. Starting a new verification from scratch is inopportune. A noticeable feature of the D-MILS assurance technologies is that the formal verification is integrated into the assurance case as a pattern which can be instantiated automatically. The pattern also references the software component which needs to satisfy the argued formal properties. Hence, we checked in the interim assurance case the formal properties related to the component where we performed the change, re-run the formal verification for those specific properties and then re-instantiate the *Composition* pattern. Even a better integration of assurance cases with the D-MILS analysis environment, which would support the change management, would be that the re-run of the formal verification and instantiation of patterns would be done automatically.

The D-MILS assurance technologies only track changes from models to the assurance case. However, it would be interesting to investigate how changes in the assurance case impact the other system artifacts. This would most probably increase the time gain ratio for modifying an assurance case while using the D-MILS technologies. However, this was out of the scope of the project.

A crucial aspect of safety case maintenance management is also the challenge of dealing with changes in the regulatory requirements and thus with the impact of these changes in the assurance argumentation. Another important challenge is the appearance of new evidence through the life-time of the assurance case, as it is difficult to identify the knock-on effects of changes in one argumentation fragment. A clear structure of argument and an explicit and complete description of the dependencies among argument fragments would help at identifying these indirect effects. The D-MILS assurance technologies offer a solution to this challenge by enabling the construction of modular assurance cases based on the D-MILS patterns. Each pattern, when instantiated, is seen as a separate module and each pattern is accompanied by a very detailed description of the module interface.

Normally, if information (such as the assumptions and context surrounding safety claims) is not recorded in the assurance case then the recognition of the impact of any changes in the system artifacts becomes even more difficult as the creators of the assurance case have to revise the entire assurance case. Fortunately, context is explicitly considered in the D-MILS assurance patterns.

The maintenance procedure proposed by the D-MILS assurance technologies focuses on how the assurance case might be syntactically affected by a change in the system artifacts. By syntactic we mean that the methodology only is capable to detect changes in the roles of the patterns and instantiate accordingly with the newest version of the system artifacts. However, a systematic method for evaluating the impact of system changes on the safety argumentation would be very valuable. For tracing system changes onto the safety argumentation one needs to reason about whether the safety claims still hold or the changed system violates the previous premises. For example, one valid question could be *For what types of changes in the system architecture do we need re-run the verification of safety and security properties?*. Unfortunately this still remains a question to be answered solely by the developer, without any method of tracking the semantic impact a change in a pattern might have on other patterns. By semantic impact we mean an impact in the argumentation itself. Nonetheless, these facts do not impact the evaluation of the D-MILS assurance technologies, as they are out of the scope of this project.

We applied a change to the X component, namely we defined an extra transition between the X and Y modes. In order to investigate the impact of this design change on our assurance case, we had to formally verify again component X. As our interim assurance case contains an argument structure (described by the *Composition* pattern), which is supported by the results of the formal verification, we needed to ensure that the argument still holds. In order to do this we simply needed to re-instantiate the pattern. Once we re-instantiated it manually, in our GSN editor and once automatically, in MBAC.

Verdict: $TGRC =_{approx.} \frac{1}{2}$

The usage of the usage and automatic instantiation of D-MILS assurance case patterns promotes adaptable and self-evolving assurance cases, hence reducing maintenance effort.

7.3 The evaluation of the D-MILS assurance patterns

Method of assessment: In this section we try to assess the adequacy of all of the D-MILS patterns. The evaluation of the D-MILS assurance case patterns helps at controlling the quality of the assurance arguments and at documenting the benefits and possible withdraws of the instantiation of

these patterns in an assurance case. Assurance case patterns encapsulate valuable knowledge about arguing the compliance of a system with its safety or security requirements and, more importantly, about manners of improving quality of the system. We investigate the effects of the D-MILS patterns on the quality of an interim assurance case for our case study through thorough measurement. On the one hand, we assess the patterns qualitatively, documenting our own experience while using the pattern for the construction of the assurance case of our case study. One the other hand, we perform a quantitative evaluation of the patterns, using a set of metrics adopted, with adjustments, from design patterns evaluations and also a set of metrics based on the syntactic/structural properties of argument structures documented using GSN, taken from [4].

Rationale: Assurance cases are subjective. Therefore, one of the goals of the development of assurance case is to ease the communication between operators and regulators with the scope of mutually accepting the subjective assurance argumentation. Among others, the goal of an assurance case evaluation is to assess if there is a mutual acceptance of the subjective position of the assurance case. Hence, while evaluating the D-MILS patterns we try answering the following question *Are the premises of the argument strong enough to support the conclusions being drawn?*. From our experience, by using the patterns for the development of the assurance case of our smart grid system, we conclude that the D-MILS patterns are sufficient for arguing the properties they intent to argue. We reached to this conclusion because most of the patterns contain separate argument legs dedicated for the confidence. A confidence argument leg documents the confidence in the structure and bases of the safety argument. Having separated confidence argument legs gives arguments greater clarity of purpose, and helps at avoiding the introduction of superfluous arguments and evidence. The D-MILS assurance case patterns present arguments concerning the confidence in both development and verification processes and also in the organization, people and tools which acted as resources during the argued processes. Also, the level of confidence is increased by the appropriateness of the contexts in which the claims in the D-MILS patterns are made. Confidence is also demonstrated by identifying the sources of doubt and removing as many such sources as possible. We could not find any sources of doubt, such as information that contradicts the claim or evidence that an argument is not necessary true, even if premises are true. While further investigating the confidence we have in the argumentation structures encapsulated in the D-MILS patterns and also their sufficiency, we did not find any important assurance statements left unexpressed or argument structures which were developed from false premises. We also looked for redundancies in argumentation, but did not find any.

The D-MILS assurance case patterns are very well documented. Their documentation includes the intent of the argument to be made, the types of evidence that support that argument and a clear outline of the argumentation. Moreover, it contains detailed description of each claim and how the argument structure interacts with other argument structures. What is missing, but it was clearly out of the scope of the project are measures or weighting of the value of particular types of evidence. Also, the context in which a particular pattern can and cannot be applied is under-specified. Moreover, pitfalls in applying the D-MILS pattern would be interesting to be documented.

While conducting the patterns evaluation, we tried to analyze various aspects of the patterns and the results can be seen in the following tables. The purpose of considering assurance case patterns in conjunction with assurance case metrics has been to determine their compatibility in terms of their application in the improvement of assurance case quality. The violation of these metrics indicates

that the application of the pattern which violated the metrics is not appropriate. None of the D-MILS assurance case patterns violated the metrics they were checked against. Next we go through several assurance case metrics and discuss the results of the measurements taken according to these metrics for the D-MILS patterns.

We expected that, after instantiating the patterns in our assurance case for the smart grid, the size of the assurance case would increase, the complexity would be reduced, the coupling of the assurance case would be reduced and the cohesion increased. All of our expectation have been met and this is to be shown next in our evaluation report.

As expected, the size of the argumentation structure increased, as we added the instantiated patterns. We took some measures of the size of the patterns, which can be found in Table 3. The fact that all the D-MILS assurance case patterns that are of significant size contain strategies increases the quality of the patterns.

Pattern Name	**Number of goals**	**Number of strategies**	**Number of solutions**	**Number of argumentation legs**
System Properties	12	5	0	6
Composition	9	0	1	3
Trusted Software Components	1	0	0	1
Implementation	16	8	2	5
D-MILS Platform	35	3	5	17
Process	14	1	0	7
Tool	11	2	1	5
Person	5	1	2	3
Organisation	4	0	1	3
Artefact	7	1	1	3
Technique	3	0	1	2
TTEthernet	21	2	7	9

Table 3: This table depicts the size of each of the pattern. The number of goals in a pattern indicates whether an argumentation structure is rather big or small, as the other argumentation elements are never as abundant as the goals. The number of strategies is an indicative of how explicit the argumentation decisions are made in a pattern. The number of solutions in a pattern reflects on the level of self-sufficiency of the argument enclosed in a pattern. The more solutions in a pattern, the higher the level of self-sufficiency, as the claims it contains do not have to be further developed in other modules. The number of argumentation legs reflects how much the main claim is divided in sub-claims. Having just one leg of argumentation, which is also long, suggests a very complex argumentation structure which can be very hard managed and therefore it is not desirable. However, having too many argumentation legs is also disadvantageous and it reflex that the structure must be reviewed because redundant, irrelevant or unrelated arguments might appear.

During our evaluation we also discovered that the D-MILS patterns are low coupled, as they make use of modularity, because each of the patterns is to be instantiated as an individual module. The fact that the argumentation structures from the D-MILS assurance patterns exhibit low coupling and high cohesion is demonstrated by the coupling and cohesion factors for each of the patterns presented in Table 4. Cohesion increases when patterns are properly built, meaning that one pattern argues the assurance of only one well established aspect of the system. All the D-MILS assurance case patterns follow the single responsibility principle, having only one clear main claim that is afterwards split into sub-claims, which are eventually demonstrated as satisfied by being supported by evidence. The D-MILS assurance case patterns have common references in the argumentation elements. All the references within one pattern are a set of system artifacts which depict only one aspect of the system. Hence, there is high cohesion in the D-MILS patterns. Low coupling is guaranteed by a modular structure of assurance cases. Modularity aims at reducing the dependencies between modules, in our case instantiated patterns.

The complexity of the assurance case built with the D-MILS patterns was reduced due to the low coupling and high cohesion of the argumentation structure, increasing the level of understandability and reusability. Also, another indicator of low complexity of an argumentation structure is the depth of the argumentation. By the depth of the argumentation we mean the maximum distance between the highest claim in the argumentation structure and a solution, a leaf claim or a claim that is to be further developed in another module. On the one hand, if the argumentation structure for one claim is very deep, it is too complex and hard to understand and manage. On the other hand, if the argumentation structure is very shallow, it suggests it has a very high potential of being insufficient. Hence, in a pattern we look for an argumentation structure of moderate length , i.e. of 10 argument elements. Table 5 depicts the length of the argumentation of each D-MILS assurance pattern and from this table we see that all the argumentation structures from the patterns are of a maximum length of 10, hence the patterns which encapsulate them are of moderate complexity.

Verdict: As a general assessment of the D-MILS patterns, we affirm that they are easy to understand due to the detailed documentation one can find in deliverable 4.3, where the patterns are enlisted and the structure, intent, participants, applicability, consequences and related patterns of the each of the patterns are described. Also, the D-MILS patterns are easy to be applied due to the automatic instantiation. This has been demonstrated through the relative ease with which they were applied to the assurance case of the smart grid by people completely unfamiliar with the patterns. Even though our evaluation process does not entitle us to judge whether the D-MILS patterns are flexible enough to be applied to any assurance case of any D-MILS system, we can at least say that they are adequate for constructing the assurance case for our smart grid, containing arguments regarding some of the most important properties of a D-MILS system.

Pattern Name	**Coupling factor = couplings (away entities)/max number of possible couplings in SC**	**Cohesion factor = Number of distinct referenced artifacts**
System Properties	7/11	2
Composition	2/11	1
Trusted Software Components	0/11	1
Implementation	3/11	1
D-MILS Platform	6/11	1
Process	6/11	1
Tool	1/11	1
Person	0/11	1
Organisation	0/11	1
Artefact	0/11	1
Technique	0/11	1
TTEthernet	1/11	1

Table 4: This table depicts the coupling and the cohesion factors of the D-MILS assurance case patterns. The coupling factor indicates how many connections among the D-MILS assurance case patterns are. The fewer dependencies are among patterns, the easiest is to manage changes in the assurance case. The cohesion factor determines whether or not one pattern argues the assurance of only one well established aspect of the system. A high cohesion factor also indicates that when there is a change in the system, the safety engineer knows exactly in which pattern changes might appear in the argumentation.

Pattern Name	**Depth in the argumentation structure**
System Properties	9
Composition	5
Trusted Software Components	1
Implementation	10
D-MILS Platform	9
Process	5
Tool	8
Person	5
Organisation	3
Artefact	6
Technique	3
TTEthernet	8

Table 5: This table depicts the length of the argumentation of each D-MILS assurance pattern in order to assess the complexity of an assurance case pattern.

8 Deployment

In this chapter we evaluate the automated deployment approach developed in D-MILS. The approach is divided in two main steps: 1) Gathering the architecture and platform information from MILS-AADL files together with optional deployment constraints and creating deployment configurations as Prolog terms. 2) Translating these Prolog terms into configuration files (XML). At the time of our evaluation we were able only to evaluate part 1 of this process.

8.1 Deployment Performance

8.1.1 Configuration solution time

Question 1:

How fast is the operation of the configuration compiler ?

Rationale:

In scope of the evaluation of the D-MILS deployment approach we used COTS hardware. On a MacBook Pro the deployment for the Smart Microgrid case study was computed without any delay.

Verdict:

Very Fast – Fast – Slow – Very Slow

Question 2:

How many resources are needed to compute a configuration ?

Rationale:

We evaluated the deployment approach of D-MILS using the Smart Microgrid case study on a COTS laptop. During the execution of the configuration computation there was no noticeable resource usage.

Verdict:

Very Few Resources – Few Resources – Many Resources – Too much resources

8.1.2 Configuration Solution efficiency

Question 1:

Does the generated configuration meet the deployment constraints ?

Rationale:

The D-MILS deployment approach needs the platform and architecture information as an input. This information is formalized to Prolog terms and represents one part of the constraints, such as how much memory or which CPU speed a node has. In addition it is possible to specify other constraints relating usage of hardware resources, deployment neighborhood, memory allocation, scheduling etc (full list of possible deployment constraints is given in D5.3). In listing 1 one can see that we specified

a constraint, which states: *AdminArea* component is not allowed to be on the same node (has to be physically separated) as the *Prosumer* components. The result is presented in listing 2. As one can see *Prosumer1* and *Prosumer2* are mapped onto **node1** and *AdminArea* is mapped onto **node2**.

1)

```
system implementation SmartGridSystem.impl
{MPCC: deployment(not_same(['AdminArea', 'Prosumer1']))}
{MPCC: deployment(not_same(['AdminArea', 'Prosumer2']))}
```

2)

```
% mpcc : generating static deployment constraints
% mpcc : (re)generating dynamic deployment constraints
% mpcc : candidate mapping [Prosumer1-node1,Prosumer2-node1,SmartGrid-node1,AdminArea-node3]
% mpcc : configuration success for node node1
% mpcc : configuration success for node node3
% mpcc : configuration success for node node2
Platform platform_i Configuration
  Node node1
    subjects [Prosumer1,Prosumer2,SmartGrid]
    ss flows [Prosumer1_aggregatedValues__SmartGrid_prosumerData1,Prosumer2_aggregatedValues__SmartGrid_prosumerDa
    Processor cpu2
    Processor cpu3
    Processor cpu0
    Processor cpu1
    Memory ram
    Memory disk
    Device es
  -- End Node node1
  Node node3
    subjects [AdminArea]
    ss flows [AdminArea_sendRequest__SmartGrid_inCredentials,SmartGrid_sendResponse__AdminArea_recvResponse]
    Processor cpu2
    Processor cpu3
    Processor cpu0
    Processor cpu1
    Memory ram
    Memory disk
    Device es
  -- End Node node3
  Node node2
    Processor cpu2
    Processor cpu3
    Processor cpu0
    Processor cpu1
    Memory ram
    Memory disk
    Device es
  -- End Node node2
-- End Platform platform_i
```

Verdict:

Totally – Mostly – Almost not – Not at all

8.2 Deployment Benefit

8.2.1 Configuration automation

Question 1:

How would you describe the degree of automation of the configuration compiler?

Rationale:

The D-MILS deployment approach requires the description of system structure / architecture (cf. 5.2) and the description of the hardware platform (cf. listing below). The deployment approach can be divided in four steps:

- Translation of the information provided in the MILS-AADL files (system's structure and hardware platform) to Prolog terms.
- Computation of deployment configurations in Prolog (as Prolog clauses) using this information. Additionally it is possible to add extra deployment constraints, which will be considered for the configuration generation.
- Generation of actual configuration files (XML) for the platform from the Prolog clauses.
- Deployment of software to hardware using the configuration files from the previous step.

At the time of the evaluation the implementation of the last two steps was still underway. Nevertheless, we think that the translation of Prolog clauses to XML and the actual deployment step are rather technical issues and the important work has already been done in the first two steps. The deployment approach (until the step we were able to evaluate) is highly automated. The user just needs to feed the compiler with the platform and system descriptions as MILS-AADL files and choose additional deployment constraints. After that, the user only need to run the configuration compiler.

Verdict:

Completely Automatic – Mostly Automatic – Mostly Manual – Completely Manual

Question 2:

How much time is required to obtain the same level of guarantee through manual configuration ? . . .

Rationale:

In order to compare the automatic against manual configuration time, we have to compare the time needed to learn how to use the tools and the time needed for actually using them. The manual method requires learning to use the following tools:

- The TTTech tool for configuring the network and the network description language. This took us almost two full working days.
- The LynxSecure tool for configuring the separation kernel of each node. This took us one working day.

To evaluate how much time the automatic method takes to learn depends on whether one counts the MILS-AADL modeling as part of the deployment process or not. As already discussed in the questions before, the D-MILS deployment methods requires a high-level system and hardware descriptions in MILS-AADL as input. If those descriptions are already available then the time needed to generate a deployment is very short. The commands one needs to utilize for the deployment generation are to be simply found in the documentation.

If the MILS-AADL models is not available, the engineer has to first obtain the MILS-AADL knowledge required to create those documents. This implies another two to three hours.

Manual configuration	Automatic configuration
Describe the network and communication between subjects using the network description language, assisted by the TTTech tool. (a few hours)	Complete the MILS-AADL model with annotations indicating deployment constraints.
Plan the deployment of subjects to nodes and make the corresponding mapping (from a few minutes to a few hours)	Run the Configuration Compiler to obtain a configuration of the network and the mapping automatically.
Generate binary configurations for the switch and the TTEthernet cards. Upload the configuration to the switch.	
Create partitions on the hard drives of the nodes and copy the corresponding subject files within them.	
Run the auto-discovery tool on each node. (1min per node)	Run the auto-discovery tool through the DMPCC interface.
Set the parameters and call the LynxSecure configuration tool to generate an xml description of the configuration of each node (a few dozens minutes).	Map nodes and partitions detected on the hardware to the ones declared in the configuration, through the DMPCC interface.
Write and compile a configuration file that binds the configuration of each node with the configuration of the network (a few dozens minutes).	
Generate the images and boot the system.	

Table 6: List of operations needed to obtain a deployed D-MILS system.

The different operations needed to deploy a D-MILS system are presented in Table 6. From this table, one can see that the manual configuration requires to write files in different languages and run various tools. Whereas the other configuration method generates these files automatically and runs the tool, reducing the amount of time and knowledge needed to configure the platform.

In conclusion one can say that the time effort saved using the automatic deployment is between two and three days.

Verdict:

Much more time – A little more time – A little less time – Much less time

8.2.2 Configuration modification

Question 1:

How would characterize the operations needed to update the configuration according to a small change in the model ?

Rationale:

The answer to this question varies according to the type of the change. However, in most cases, according to our experience, it is advisable to perform all the steps of the deployment approach again in case of a model change. As already described in the previous sub-sections, the performance of these steps is straightforward as they are highly automated. Therefore our verdict is that the effort need for updating the configuration after a model change (regardless of the type and size of the change) is very low.

Verdict:

Very Easy – Easy – Difficult – Very Difficult

8.3 Deployment Maturity

8.3.1 Maturity of deployment configuration tools

Question 1:

Is the tool usable without reading the user manual ?

Rationale:

In order to be able to use the D-MILS deployment methods, the user needs to understand two important steps of the deployment generation flow. On the one hand, the user needs to know how to model and annotate the system architecture and the hardware. On the other hand, the user has to learn the commands which are needed to correctly interact with the Python and Prolog scripts. Without this information the user can not know how to use the tool.

Verdict:

Completely – Partially – Almost not – **Not at all**

Question 2:

How would you describe the documentation ?

Rationale:

There are two deliverables which cover the usage of the D-MILS deployment procedure. The D5.1 concentrates on how to model the hardware architecture in MILS-AADL and add deployment constraints to it. The document D5.2 discusses the steps that have to be done (which scripts and commands has to be executed) in order to execute the deployment process itself. At the time of our evaluation the second document (D5.2) was outdated. We therefore required support from the development team to be able to perform the deployment procedure.

Verdict:

Very Complete – Complete – **Incomplete** – Empty

Question 3:

How would you characterize the installation process ?

Rationale:

The installation is very easy to be performed, as, in order to run the tools involved in the D-MILS deployment approach, Python and Prolog need to be installed.

Verdict:

Straightforward – Easy – Complicated – Very Complicated

8.3.2 Skills required for deployment configuration tools

Question 1:

What is the skill level required for generating a valid configuration solution for the SMG case study?

Rationale:

There are two things needed to be known in order to be able to perform the D-MILS deployment approach. On the one hand, one needs to have some knowledge of MILS-AADL, especially how to describe system structure and the platform architecture. On the other hand, one needs to know the set of commands in Prolog, which are needed for configuring the deployment. This information is D-MILS specific and cannot be known by anyone outside of the project. Therefore this information needs to be found in the documentation. With the help of meaningful documentation, any system engineer should be able to perform the D-MILS deployment approach.

Verdict:

Expert researcher or engineer – Specialized engineer – **General Engineer** – Simple user

Question 2:

How likely is it to find a person with such a skill level for the SMG case study ?

Rationale:

As already stated in the last question, the only specific information one needs to know is a set of Prolog Commands and MILS-AADL. This information should be described in the corresponding documentation and should be easy to comprehend for any system engineer.

Verdict:

Very Likely – Likely – Unlikely – Very Unlikely

9 Platform operation

9.1 Platform Adequacy

Question 3:

Is the D-MILS platform adequate for developing the SMG system ?

Rationale:

In the current status of the D-MILS project, each subject is implemented as a separation kernel partition that hosts a fully virtualized operating system. The development of the application code can thus be done on any platform on which the corresponding OS can run. Additionally, the designer can reuse existing software developed for supported operating system and include it in the system through a dedicated subject.

In that sense, the development effort that requires the platform is mainly the inner configuration of each subject. In particular, the subject configuration should exploit the resources allocated to the corresponding separation kernel partition. For instance, the communication between distinct subjects is implemented via virtual Ethernet cards. These virtual cards have to be configured correctly to enable the connections defined in the original MILS-AADL model of the system.

During the development phase, the developer can easily configure the (regular) network to have direct access to each subject from the development host. In that setting, the D-MILS platform allows the developer to run concurrently the different components of the system under development, while having a direct access to each of them. These conditions are adequate for developing the subjects composing the SMG system.

Verdict:

Very Adequate – **Adequate** – Inadequate – Very Inadequate

Question 4:

Is the D-MILS platform adequate for running the SMG system ?

Rationale:

The SMG system is composed of the prosumers and a dedicated smart grid component that coordinates the prosumers. The D-MILS platform is not really adequate to implement this whole system.

First, the studied system typically spans over a geographical area (a few blocks of houses) that is relatively wide. Each node (house) needs a direct cable connection to a TTEthernet switch, which requires to deploy new cables (fiber cables if the cable length is more than a few hundred meters) for the smart grid only. Such a deployment, although technically possible, is not really realistic.

Furthermore, even if dedicated switches and cables are available, the static configuration of the network gives rise to another problem. Whenever adding a new node to the network, reconfiguring the network potentially requires to reprogram all the switches and ethernet cards. In practice, it would mean that connecting a new house to the grid would stop the grid until every house applies the new configuration. Again, such a scenario is not really realistic.

However, the platform can be used to implement each of the prosumers, as well the component coordinating them. Using the D-MiLS platform in that frame is highly beneficial.

First the network becomes deterministic, which ensures that critical communications, such as messages indicating to switch to island mode, are guaranteed to arrive in time. It also ensures that messages sent from the various sensors are not lost.

Moreover, the separation provided by the platform increases the security of the system. A prosumer has typically three communication points with the outside, namely: the sensors and actuators I/O, the communication with the rest of the grid and the users interaction through a local wifi network. If an attacker gets into the systems through one of these points, it cannot access the rest of the system since the platform restricts the communication.

For these reasons, the D-MILS platform is adequate for deploying one of the prosumers.

Verdict:

Very Adequate – **Adequate** – Inadequate – Very Inadequate

9.2 Platform Performance

Description: The observed performance of the D-MILS platform, with respect to the hardware resources available, for execution of an industrial demonstrator application.

Experiment: Compare the response time of the SMG case study running on the D-MILS platform and on a similar platform. The round trip time of a message is measured.

Verdict: The round trip time of a message for both messages was below 1 ms. This means that the D-MILS platform is adequate for using it with real-time applications.

9.3 Platform Maturity

Question 5:

How informative were the error messages returned by the platform tools when developing the SMG case study ?

Rationale:

We did not encounter any error message from the separation kernel configuration tool. We encountered few error messages when using the platform tools, mainly from the network configuration tool.

The error messages indicated an impossibility to configure the network with the imposed constraints. They did not precise what to change in order to have a usable configuration but gave sufficient information to pinpoint the problem and modify the configuration into a usable one.

During the execution of the case study on the platform, we did not encounter any error message.

Verdict:

Very Useful – **Useful** – Not Useful – Not Useful at all

Question 6:

Did you experience bugs from the platform when running the SMG case study ?

Rationale:

To our experience, there was no evidence of serious bugs due to a misbehavior of the platform. We encountered the following issues.

Once a separation kernel partition has been set up, it is necessary to install a operating system on it. When installing Ubuntu, we had to use a PS/2 keyboard as USB keyboards are not supported at that point. This bug was corrected in a version of the separation kernel more recent than the one used in the project.

After a manual configuration of the platform, we were able to exchange small messages between subjects but not large ones. This was due to the fact that the time triggered switch was configured to drop packet incoming more frequently that a given rate. By modifying the configuration, we were able to obtain the intended behavior.

To summarize, we encountered few bugs from the platform. The latter behaved coherently with respect to our expectations.

Verdict:

Not at all – A dozen – A few dozen – A lot

10 Industrial Requirements Measures

In this section we describe the analysis results obtained for each of the requirement of the Smart Micro Grid demonstrator as listed in deliverable D1.1, and provide the evaluation measures collected.

10.1 Description language and modeling

10.1.1 Requirement SMG_DL.1

Description:

Should: Support of compositional modules with interfaces capable representing IEC 61850.

Evaluation:

IEC 61850 defines communication networks and systems for power utility automation, and more specifically the communication architecture for sub-systems such as substation automaton systems. For that reason, the standard defines a specific data model.

As already pointed out in section 5.1 MILS-AADL provides the possibility of expressing the structure of the system, which includes the communication architecture together with the interfaces for each component. Among other things those interfaces define which kind of data is exchanged between the sub-components. Even though the IEC 61850 provides a larger number of different data types they can all be led back to the basic ones, which are supported by MILS-AADL (cf. D2.1).

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.1.2 Requirement SMG_DL.2

Description:

Shall: Support composition mechanisms with asynchronous and synchronous components. In particular event based communication between software components shall be supported as well as channels to represent physical behavior of the system.

Evaluation:

The asynchronous (event-based) communication in MILS-AADL is supported by the event ports. Those ports for example can be used to trigger transitions in state automatons. An example is given by the code snippet below, where an incoming event, called *deviationEvent*, is triggering a state transition.

The synchronous communication in MILS-AADL is supported by the data ports, as *prosumerData1* port in the code snippet below. Those ports are not for transmitting events but data, which can be manipulated by the component.

```
system SmgProAgentIn
features
      deviationEvent: in event port;
      prosumerData1: in data port int;
```

```
      prosumerData1Out: out data port int;
      deviationEventOut: out event port;
end SmgProAgentIn;

subject implementation SmgProAgentIn.impl
flows
      prosumerData1Out := prosumerData1;
modes
      init: initial mode;
      fwdPrice: mode;
      fwdDeviation: mode;

transitions
      init-[deviationEvent]->fwdDeviation;
      fwdDeviation-[deviationEventOut]->init;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.1.3 Requirement SMG_DL.3

Description:

Shall: Modeling of physical behavior (e.g. energy, power, temperature) should be possible with one step difference equations using linear arithmetic.

Evaluation: In MILS-AADL the one step difference equations are represented by the transitions of the state automatons. To change the current state of the system one can use the *effect* part (which comes after the **then** keyword) of the transition. In the example below the *dischargingRate* is changed to the difference of *StatusConsumption* and *StatusProduction*.

```
idle -[when(StatusProduction < StatusConsumption)
     and (StatusConsumption - StatusProduction) > charge_thr
           then dischargingRate := (StatusConsumption - StatusProduction)]
                 -> consumptionHigherProduction;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.1.4 Requirement SMG_DL.4

Description:

Shall: Internal state of the system should be representable with state machines extended with real number attributes, e.g. for describing the physical environment.

Evaluation:

The internal state of the system (or each sub-system) is representable with state machines, as demonstrated in the code snippet below. This listing shows the behavior of the battery in case the consumption of the prosumer is higher the production. At first if that is the case the battery switches to the intermediate mode *consumptionHigherProduction*. From there the battery system has two choices. Either the battery has enough energy to support the prosumer's energy usage. In this case the system

switches into the mode *discharging*. Or the batteries capacity is too low or the temperature of the battery is to high. In this case the battery switches back into the *idling* mode.

```
-- production < consumption
idle -[when(StatusProduction < StatusConsumption)
          and (StatusConsumption - StatusProduction) > charge_thr
           then dischargingRate := (StatusConsumption - StatusProduction)]
            -> consumptionHigherProduction;
--discharging
consumptionHigherProduction-[when(currentLevel - dischargingRate) >= 0
               and dischargingRate <= max_discharging_rate
                           and temperature < max_temperature] -- see above
                       -> discharging;

-- batteries capacity too low or discharging rate is to high
      consumptionHigherProduction
          -[when (currentLevel - dischargingRate) < 0
            and dischargingRate > max_discharging_rate
                  or temperature >= max_temperature]
                  ->idle;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.1.5 Requirement SMG_DL.5

Description:

Should: The description language should allow the representation of component faults or failures in order to evaluate their effect on the system.

Evaluation:

MILS-AADL allows the specification of error models. These models specify the behaviour of the components and the whole system in case of errors. The behaviour of an error model is given by state automaton operating on error states. Each state is representing an error mode and the transitions between stated are triggered by error events or error propagations. An error event thereby is internal to a component and reflects changes of the error state caused by local faults and repair operations. The outgoing error propagation on the other hand report an error state to other components.

For our Smart Grid case study we used the concept of error models to represent possible deviation or total black out events of the power grid.

```
error model VoltageError
  features
    stable: initial state;
    unstable: error state;
    down: error state;
end VoltageError;

error model implementation VoltageError.impl
  events
    variation: error event;
    blackout: error event;
  transitions
    stable -[ variation] -> unstable;
```

```
   stable -[ blackout] -> down;
   unstable -[ blackout] -> down;
   unstable -[reset]-> stable;
   down -[reset] -> stable ;
end VoltageError.impl;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.1.6 Requirement SMG_DL.6

Description:

Should: The description language should support discrete time for modeling of the system behavior.

Evaluation:

MILS-AADL supports two analogue datatypes (**clock** and **continuous**), and a set of discrete data types. In our system (as one can see on the previous code snippets) we use discrete data types (mostly **int**) and are able to model the behaviour of our system. Analogue data types were not used in our system models.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2 System Safety

The safety requirements basically concern two types of properties: the requirements SA.1 – SA.7 and SA.11 deal with the *island mode* behaviour, and affect the models of the Smart Grid Wrapper and the Prosumers, while SA.8, SA.9, and SA.10 concern the correct control of the battery.

10.2.1 Requirement SMG_SA.1

Description:

Shall: The highest-level safety priority shall be grid stability. Indicators of instability are: deviation from the frequency 50 Hz and deviation from the nominal voltage level. In the case that the frequency deviates more than 1 Hz, the micro grid shall switch to island mode.

Evaluation:

In the model a deviation of frequency or voltage from their nominal value is signalled to the SMG wrapper via an incoming event port `deviationEvent`. When this event is received, the SMG switches to island mode, which is described by setting the outgoing Boolean data port `smgInIslandMode` to `true`. In addition, an (outgoing) event `switchToIslandMode` is used to inform the Prosumer components about the island mode state.

```
 subject SmgWrapper
   features
       deviationEvent: in event port;
       smgInIslandMode : out data port bool default false;
       switchToIslandMode: out event port;
   [...]

end SmgWrapper;

subject implementation SmgWrapper.impl
 modes
   idle: initial mode;
   fwdIM : mode;
   islandmode : mode;
   [...]

 transitions
   idle-[deviationEvent]->fwdIM;
   fwdIM-[switchToIslandMode then smgInIslandMode := true]->islandmode;
   [...]

end SmgWrapper.impl;
```

The LTL property covering this requirement states that whenever the `deviationEvent` is received the SMG switches to island mode by setting `smgInIslandMode` to `true`.

```
 {nuXmv:
  LTLSPEC
    (G ({deviationEvent} -> (F {smgInIslandMode}))) ;
 }
```

This property is easily established with the D-MILS model checking tools.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.2 Requirement SMG_SA.2

Description:

Shall: In case of a power outage, the micro grid shall switch to an island mode.

Evaluation:

This requirement is modelled in the same way as SA.1. We use an additional incoming event port `poweroutageEvent` to signal a power outage to the SMG component.

```
 subject SmgWrapper
   features
    poweroutageEvent : in event port;
    [...]

end SmgWrapper;

subject implementation SmgWrapper.impl
  [...]

 transitions
   idle-[poweroutageEvent]->fwdIM;

   [...]
```

```
  {nuXmv:
   LTLSPEC
     (G ({poweroutageEvent} -> (F {smgInIslandMode}))) ;
  }

end SmgWrapper.impl;
```

Again, this property can easily be established with the D-MILS model checker.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.3 Requirement SMG_SA.3

Description:

Should: Switching to island mode shall to be accomplished in less than 20 ms.

Evaluation:

While MILS-AADL supports analogue data types that allow to model physical behaviour such as time and clocks, we have chosen a discretized model, where time is captured on a more abstract level as a sequence of steps. On this level of abstraction, it takes two steps for the SMG Wrapper component to switch to island mode (as expressed by setting the variable `smgInIslandMode` to `true`) after the occurrence of a `deviationEvent`.

```
  {nuXmv:
   LTLSPEC
     (G ({deviationEvent} -> (X X {smgInIslandMode}))) ;
  }
```

Adequacy of this property with respect to the requirement would have to be substantiated by measurements on the real demonstrator hardware. Initial results of such measurements that we have performed indicate that corresponding execution times are very low. As the requirement is not fully reflected in our model, but suitably close, we consider it *largely fulfilled*.

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.2.4 Requirement SMG_SA.4

Description:

Should: If the voltage of the connection between the micro grid and smart grid becomes higher than 10% of the nominal value (e.g. 400V), the smart micro grid island mode shall be activated to protect consumer electronics.

Evaluation:

This requirement is similar to SA.1 and SA.2. We use to constants `nominal_voltage` and `voltage_threshold` to model the nominal voltage value and a 10% deviation threshold, respectively. The voltage level is modelled by an incoming data port `voltage`. A transition is added

to the SMG Wrapper so that whenever the voltage level deviate by more than 10% the SMG proceeds to the `fwdIM` mode, from which it transitions to island mode, cf. SA.1.

```
 constants
   nominal_voltage : int := 400;
   voltage_threshold : int := 40; -- 10% of nominal value

subject SmgWrapper
   features
     voltage : in data port int;

   [...]

end SmgWrapper;

subject implementation SmgWrapper.impl
  [...]
  transitions
    idle-[when (voltage > nominal_voltage + voltage_threshold)]->fwdIM;
    [...]

  {nuXmv:
     LTLSPEC
       (G (({voltage} > {nominal_voltage} + {voltage_threshold})
          -> (F {smgInIslandMode}))) ;
  }

end SmgWrapper.impl;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.5 Requirement SMG_SA.5

Description:

Should: If the voltage in the micro grid is above a certain threshold (e.g. 10% above 400V), the switch box of the prosumer shall activate the prosumer island mode, in order to keep the battery in safe operation.

Evaluation:

This requirement is analogous to the previous one, but now considers the prosumers. As is in the case of the Smart Grid, the voltage level is modelled by an incoming data port `voltage`, and the island mode status of the Prosumer is modelled through an outgoing Boolean data port `isInIslandMode`. When the Prosumer has switched into island mode, it signals a `prosumerIslandMode` event.

```
subject Prosumer
  features
    voltage : in data port int;
    prosumerIslandMode : out event port;
    isInIslandMode : out data port bool default false;
    [...]

end Prosumer;

subject implementation Prosumer.impl
  modes
    normal : initial mode;
```

```
    switchToIsland : mode;
    island : mode;

  transitions
    normal-[when (voltage > nominal_voltage + voltage_threshold)]->switchToIsland;
    switchToIsland-[then isInIslandMode := true]->island;
    [...]

end Prosumer.impl;
```

The requirement is directly translated into a LTL formula just as SA.4 and can easily be established in the D-MILS verification tool.

```
 {nuXmv:
  LTLSPEC
    (G ((({voltage} > {nominal_voltage} + {voltage_threshold})
         -> (F {isInIslandMode})))) ;
 }
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.6 Requirement SMG_SA.6

Description:

Should: If prosumer island mode is active and a high consuming device is turned on (e.g. vacuum cleaner or oven), the power of the corresponding power socket shall be turned off in the switch box.

Evaluation:

To express this requirement we have added another Boolean incoming port `high_consuming_device_status` to the Prosumer model, which models whether or not the respective device is turned on. Since our model does not contain explicit models of switch boxes or power socket, we describe the status of the power socket of the high consuming device with an outgoing Boolean data port `power_socket_hcd`. The default value of this port is `true` to express that the power socket is usually switched on.

We extend the Prosumer transitions used for the previous requirement SA.5 with an intermediate step: an additional mode `islandSignal` is introduced which is visited when the Prosumer is switched to island mode, and where the power socket can be switched of in case the high consuming device is on.

```
subject Prosumer
  features
    [...]
    high_consuming_device_status : in data port bool;
    power_socket_hcd : out data port bool default true;

end Prosumer;

subject implementation Prosumer.impl
  modes
    [...]
    islandSignal: mode;

  transitions
    switchToIsland-[then isInIslandMode := true]->islandSignal;
```

```
   islandSignal-[when high_consuming_device_status
              then power_socket_hcd := false]->island;
   [...]

end Prosumer.impl;
```

The requirement can most directly be translated into LTL and easily be verified with the model checker.

```
{nuXmv:
 LTLSPEC
   (G ({isInIslandMode} & {high_consuming_device_status}
      -> (F !{power_socket_hcd}))) ;
}
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.7 Requirement SMG_SA.7

Description:

Should: The rule system of prosumer shall immediately be informed when micro grid island mode is activated. The available time window depends on the hardware components, but typical times are around 100ms.

Evaluation:

Informing the rule system is accomplished by emitting a dedicated event `prosumerIslandMode`, which is added as another outgoing event port to the Prosumer. The signalling of the event can be accomplished by extending the transition mentioned above for SA.6. The first sentence of the requirement can be directly expressed as LTL property. For analogous reasons as in the case of SA.3, the constraint on the available time for the rule system to be informed has not been modelled here. We hence consider the requirement *largely fulfilled*.

```
prosumerIslandMode : out event port;

islandSignal-[prosumerIslandMode
           when high_consuming_device_status
           then power_socket_hcd := false]->island;

{nuXmv:
 LTLSPEC
   (G ({isInIslandMode} -> (F {prosumerIslandMode}))) ;
}
```

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.2.8 Requirement SMG_SA.8

Description:

Shall: Every battery component shall not be overloaded. This means that if the battery status is full, the control system shall not send any further loading signal.

Evaluation:

The requirements SA.8 to SA.10 concern the Battery component of the Smart Grid together with its control system. Accordingly, two component specifications have been created, one for the battery and one for the battery control system.

A number of constant definitions are used to describe the various constraints and bounds on the battery capacity, temperature, charging rates, etc.

```
constants
  max_capacity: int := 30;
  max_temperature: int := 20;
  min_temperature: int := 0;
  charge_thr: int := 5;
  max_charging_rate: int := 10;
  max_discharging_rate: int := 10;
  coolingRate: int := 2;
  heatupRate: int := 2;
```

The battery provides information in its internal state including load level and temperature via respective outgoing data ports. The battery receives commands from the control system via its incoming ports for charging and discharging command together with the respective rates.

```
subject Battery

  features
    batteryLoad: out data port int;
    temperature: out data port int default 10;
    chargingRate: in data port int;
    dischargingRate: in data port int;
    charge_signal: in event port;
    discharge_signal: in event port;
    idle_signal: in event port;
    done: out event port;

end Battery;
```

The battery waits for charging or discharging commands from the control system and proceeds to the respective modes. In the case of charging, the `batteryLoad` is increased by the current `chargingRate` and the status of the battery is modified accordingly. Note that there is no prevention of overloading the battery in this part of the model.

```
subject implementation Battery.impl
  modes
    wait: initial mode;
    idle: mode;
    charging: mode;
    discharging: mode;

  transitions
    wait-[idle_signal]->idle;
    wait-[charge_signal]->charging;
    wait-[discharge_signal]->discharging;

    charging-[done
           then batteryLoad := batteryLoad + chargingRate;
               temperature := temperature + heatupRate
          ]->wait;
```

```
    [...]

end Battery.impl;
```

The desired property to be established is that the battery load never exceeds its `max_capacity`.

```
{nuXmv:
 LTLSPEC
   (G (!({batteryLoad} > {max_capacity}))) ;
}
```

Obviously, this property can only hold if the control system prevents the battery from overloading.

The battery controller has ports symmetric to the ones of the battery, plus two more incoming ports `productionLevel` and `consumptionLevel` used for communicating the amount of energy produced and consumed in the system, respectively.

```
subject BatteryController
 features
  batteryLoad: in data port int default 0;
  batteryError: out data port bool default false;
  temperature: in data port int;
  productionLevel: in data port int;
  consumptionLevel: in data port int;
  chargingRate: out data port int default 0;
  dischargingRate: out data port int default 0;
  charge_signal: out event port;
  discharge_signal: out event port;
  idle_signal: out event port;
  done: in event port;

end BatteryController;
```

We consider the transitions of the battery controller for the case of charging the battery. When the production is significantly higher than the consumption (above a certain threshold `charge_thr`), the `chargingRate` is calculated. Only if the `chargingRate` does not increase the `batteryLoad` above the maximum capacity a `charge_signal` is sent to the battery.

```
subject implementation BatteryController.impl
 flows
  port (batteryLoad > max_capacity) -> batteryError;

 modes
  control: initial mode;
  wait: mode;
  idle: mode;
  charge: mode;
  charge_safety_check: mode;
  [...]

 transitions
  wait-[done]->control;
  [...]

  control-[when (consumptionLevel <= productionLevel)
           and (productionLevel - consumptionLevel) > charge_thr
       then chargingRate := (productionLevel - consumptionLevel)
      ]->charge_safety_check;

  charge_safety_check
      -[when (batteryLoad + chargingRate) > max_capacity
           or chargingRate > max_charging_rate
           or temperature >= max_temperature
       ]->idle;
```

```
   charge_safety_check
         -[when (batteryLoad + chargingRate) <= max_capacity
              and chargingRate <= max_charging_rate
              and temperature < max_temperature
          ]->charge;

   charge-[charge_signal]->wait;

end BatteryController.impl;
```

Now, the property that the battery is never out-of-bounds can be established. Furthermore, it can be shown that a `charge_signal` with a positive `chargingRate` is not raised as long as the `batteryLoad` is at `max_capacity`. In the LTL formula below, "`V`" denotes the *"release"* operator. The formula p `V` q is true at time t, if q holds at all time steps $t' \geq t$ up to and including t'' where p also holds. Alternatively, it may be the case that p never holds, in which case q must hold in all time steps $t' \geq t$.

Remark: using the *until* operator `U` instead of `V` would be too strong, as q `U` p requires that p eventually holds.

```
{nuXmv:
 LTLSPEC
   (G (({batteryLoad} = {max_capacity}) ->
       (({batteryLoad} != {max_capacity})
        V
        (!({charge_signal} & ({chargingRate} > 0)))
       )));
}
```

The two properties together establish the requirement SA.8.

To contrast the monolithic verification of this requirement, we also present an analysis following the compositional verification approach as an example.

To this end, we define the a composite component `BatterySystem`, which contains the two components described above, `Battery` and `BatteryController`, as subcomponents.

```
system BatterySystem
 features
   productionLevel: in data port int;
   consumptionLevel: in data port int;
   batteryError: out data port bool;

end BatterySystem;

system implementation BatterySystem.impl
 subcomponents
   bttry: subject Battery;
   ctrl: subject BatteryController;

 connections
   data port productionLevel -> ctrl.productionLevel;
   data port consumptionLevel -> ctrl.consumptionLevel;
   event port ctrl.charge_signal -> bttry.charge_signal;
   event port ctrl.discharge_signal -> bttry.discharge_signal;
   event port ctrl.idle_signal -> bttry.idle_signal;
   event port bttry.done -> ctrl.done;
   data port ctrl.chargingRate -> bttry.chargingRate;
   data port ctrl.dischargingRate -> bttry.dischargingRate;
   data port bttry.batteryLoad -> ctrl.batteryLoad;
   data port ctrl.batteryError -> batteryError;
   data port bttry.temperature -> ctrl.temperature;
```

```
end BatterySystem.impl;
```

For this composition, we would like to establish the property that the battery load will never exceed the maximum capacity of the battery. This is expressed in the following OCRA guarantee:

```
{OCRA:
  CONTRACT no_overload
  assume:
    always ({productionLevel} >= 0 and {consumptionLevel} >= 0 and
          {productionLevel} - {consumptionLevel} <= {max_charging_rate} and
          {consumptionLevel} - {productionLevel} <= {max_discharging_rate}
         );
  guarantee: (always ({batteryError} = false));
}
```

The goal is to find contracts for each of the two components that together ensure the overall property. For the battery controller, we must basically ensure that whenever it sends a `charge_signal`, the corresponding `chargingRate` is such that it will not increase the `batteryLoad` beyond its limits, and that its value does not change before the battery has applied the change.

```
{OCRA:
 CONTRACT no_overload
   assume: true;
   guarantee:
     (always ({batteryError} implies {batteryLoad > max_capacity})) and
     (always ({dischargingRate}>=0)) and
     (always ({charge_signal} implies
           (((not change({chargingRate}) and (not {done})) since
                ({batteryLoad + chargingRate <= max_capacity} and
                 (not change({chargingRate}) and (not {done}))))) and
            ((not change({chargingRate})) until {done})));
}
```

Likewise, it has to be guaranteed that the `Battery` component applies the changes to the `batteryLoad` according to the `chargingRate` commands received from the controller, and that the `batteryLoad` does not spontaneously change between two such commands.

```
{OCRA:
   CONTRACT no_overload
   assume: true;
   guarantee:
    {batteryLoad} = 0 and
    (always (change({batteryLoad}) implies {done})) and
    (always ({done} implies
          ({next(batteryLoad) = batteryLoad} or
           {next(batteryLoad) = 0} or
           {next(batteryLoad) = batteryLoad - dischargingRate} or
           ({next(batteryLoad) = batteryLoad + chargingRate} and
            previously ((not {done}) since ({charge_signal})))))));
}
```

Together, these two contracts of the components establish a refinement of the overall safety property for the `BatterySystem` corresponding to the requirement SA.8.

```
 {OCRA:
   CONTRACT no_overload REFINEDBY bttry.no_overload, ctrl.no_overload;
 }
```

We note that the OCRA contracts for the two subcomponents are significantly more complex than the corresponding LTL formulae used in the monolithic verification approach. This is due to the fact

that the guarantees of the individual components have to capture a sufficiently strong abstraction of the behaviour of those components in order to establish the correct refinement of the overall property.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.9 Requirement SMG_SA.9

Description:

Should: The charging or discharging rate of the batteries shall be within the bound specified in the documents. The control system shall take care that the rate is within the limits.

Evaluation:

In the battery model, we would like to establish the following property corresponding to the first part of the requirement:

```
{nuXmv:
 LTLSPEC
   (G (({charging} ->
         (({chargingRate} >= 0) & ({chargingRate} <= {max_charging_rate}))
       )
       &
       ({discharging} ->
         (({dischargingRate} >= 0) &
          ({dischargingRate} <= {max_discharging_rate}))
       )
      ));
}
```

Again, it is the responsibility of the batter controller to achieve this. The transitions shown above for requirement SA.8 already ensure this for the charging rate: the charge signal is only emitted if the `chargingRate` does not exceed the `max_charging_rate`.

We reproduce the analogous transitions for the case of discharging below:

```
subject implementation BatteryController.impl
 modes
   discharge: mode;
   discharge_safety_check: mode;
   [...]

 transitions
   [...]

   control-[when (productionLevel <= consumptionLevel)
             and (consumptionLevel - productionLevel) > charge_thr
         then dischargingRate := (consumptionLevel - productionLevel)
        ]->discharge_safety_check;

   discharge_safety_check
        -[when dischargingRate > max_discharging_rate
             or temperature >= max_temperature
         ]-> idle;

   discharge_safety_check
        -[when dischargingRate <= max_discharging_rate
             and temperature < max_temperature
         ]-> discharge;
```

```
   discharge-[discharge_signal]->wait;

end BatteryController.impl;
```

Now the second part of the requirement can be expressed as follows and is easily checked with the D-MILS verification tool.

```
{nuXmv:
 LTLSPEC
   (G (({charge_signal} ->
         (({chargingRate} >= 0) & ({chargingRate} <= {max_charging_rate}))
       )
       &
       ({discharge_signal} ->
          (({dischargingRate} >= 0) &
           ({dischargingRate} <= {max_discharging_rate}))
       )
      ));
}
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.10 Requirement SMG_SA.10

Description:

Should: The battery temperature shall be within the bounds specified in the documents. The control system shall reduce charge or discharge rate to zero, in case the battery temperature is above the temperature limit.

Evaluation:

The requirement corresponds to two properties. The first part concerns the battery itself:

```
{nuXmv:
 LTLSPEC
   (G (({temperature} >= {min_temperature}) &
       ({temperature} <= {max_temperature})));
}
```

The second part is ensured by the battery controller. It states that whenever the battery temperature has reached its maximum allowed value then the charging or discharging rates will remain zero until the temperature has decreased below the maximum. Note that "U" in the LTL formula below is the *until* operator.

```
-- SA.10b_charge
{nuXmv:
 LTLSPEC
   (G (({temperature} = {max_temperature}) ->
         (({charge_signal} -> ({chargingRate} = 0))
          U ({temperature} < {max_temperature}))
     ));
}

-- SA.10b_discharge
{nuXmv:
 LTLSPEC
   (G (({temperature} = {max_temperature}) ->
```

```
        (({discharge_signal} -> ({dischargingRate} = 0))
         U ({temperature} < {max_temperature}))
      ));
}
```

The two properties are fully established with the model checker.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.2.11 Requirement SMG_SA.11

Description:

May: In case of island mode, full batteries and a higher production than consumption - the production units shall reduce their production.

Evaluation:

This requirement again concerns the Prosumer model, to which three more incoming data ports are added in order to transmit information about the current load level of the battery, and the level of production and consumption, respectively.

```
subject Prosumer
 features
  batteryStatus: in data port enum (empty,charged,full,outofbounds);
  productionLevel: in data port int;
  consumptionLevel: in data port int;

  reduceProduction : out event port;
  [...]

end Prosumer;
```

In order to ensure the property, an additional mode `reduce_production` is introduced for the Prosumer together with a new transition: when the Prosumer has switched to island mode and the conditions described in the requirement are fulfilled, the Prosumer transitions to the `reduce_production` from which it emits the `reduceProduction` event to inform the rule system to reduce the production of the production units.

```
subject implementation Prosumer.impl
 modes
  [...]
  reduce_production : mode;

 transitions
  [...]

  island-[when (batteryStatus = full)
           and (productionLevel > consumptionLevel)
       ]->reduce_production;

  reduce_production-[reduceProduction]->island;

end Prosumer.impl;
```

The property corresponding to the requirement can directly be translated into an LTL formula and checked with the D-MILS model checker.

```
{nuXmv:
 LTLSPEC
   (G ( ({isInIslandMode})
       & ({batteryStatus} = full)
       & ({productionLevel} > {consumptionLevel})
       -> (F {reduceProduction})));
}
```

It has to be noted that we do not cover fully the intentions of the requirement, namely that indeed the production units do reduce their production; it is only considered that the command to do so is sent. We hence consider the property only *largely fulfilled*.

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.3 System Security

This section lists the security requirements for the SMG demonstrator. All security requirements are specified according the ISO 15408/18045 (Common Criteria). The requirements elicitation is based on a comprehensive smart grid scenario including system, user and administration considerations.

10.3.1 SMG_SO.1 requirement

SHALL: Communication / Information Flow between components shall be according to the policy defined for Prosumer and Smart Grid. No other information flow shall occur.

Evaluation: Information flow among components can be modelled using MILS-AADL language. For example, the requirement that the information flow between aggregation and persistency components must be unidirectional, i.e. from persistency to aggregation. This is depicted in the following model of the prosumer in which all connections of ports are directed from persistency to aggregation component and not the other way around.

```
system Prosumer
[...]
end Prosumer;
system implementation Prosumer.impl

subcomponents
      Persistency: subject Persistency.impl ;
      Aggregation: subject Aggregation.impl ;
      [...]

connections
      port Persistency.prosumerData -> prosumerData;
      port Persistency.productionValues -> Aggregation.productionValues;
      port Persistency.consumptionValues -> Aggregation.consumptionValues;
      port Persistency.batteryValues -> Aggregation.batteryValues;

[...]
end Prosumer.impl
```

For verification purpose, one can extract the information flow graph of the implementation of the smart grid system and check whether it coincides with the one defined for prosumer and smart grid components. We refer to Deliverable 5.3 for more details.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.2 SMG_SO.2 requirement

SHOULD: All wrapper events as well as device status data shall be logged by the persistency component. This means in particular that no wrapper component can hide events or change of states.

Evaluation: In the below model, there exists a communication link between the wrapper component and the persistency component so that the information about the devices can be transferred to the persistency component. Note that as the status of the energy production and consumption as well as that of the battery is changed, the corresponding events `ConsumptionEvent`, `ProductionEvent` and `BatteryEvent` are sent to the persistency component. The persistency component, as a result, stores the new information in local data components `saveConsumption`, `saveProduction` and `saveBattery` respectively.

```
system Prosumer
features
aggregatedValues: out data port int default 0;
meterReading: out data port int default 0;
end Prosumer;


system implementation Prosumer.impl


subcomponents



      WrapperSensors: system WrapperSensors.impl ;
      Persistency: subject Persistency.impl ;
      Aggregation: subject Aggregation.impl;
      ProAgentOut: subject ProAgentOut.impl;


connections
      port WrapperSensors.StatusProduction -> Persistency.StatusProduction;
      port WrapperSensors.StatusConsumption -> Persistency.StatusConsumption;
      port WrapperSensors.StatusBattery -> Persistency.StatusBattery;

      port WrapperSensors.ProductionEvent -> Persistency.ProductionEvent;
      port WrapperSensors.ConsumptionEvent -> Persistency.ConsumptionEvent;
      port WrapperSensors.BatteryEvent -> Persistency.BatteryEvent;

      port Persistency.OutProduction -> Aggregation.productionValues;
      port Persistency.OutConsumption -> Aggregation.consumptionValues;
      port Persistency.OutBattery -> Aggregation.batteryValues;

      port Aggregation.aggregatedValues -> ProAgentOut.InaggregatedValues;
      port ProAgentOut.OutaggregatedValues -> aggregatedValues;
      port ProAgentOut.meterReading -> meterReading;
```

```
end Prosumer.impl;


subject Persistency
features
      StatusProduction: in data port int;
      StatusConsumption: in data port int;
      StatusBattery: in data port int;

      ProductionEvent: in event port;
      ConsumptionEvent: in event port;
      BatteryEvent: in event port;

      OutProduction: out data port int;
      OutConsumption: out data port int;
      OutBattery: out data port int;

end Persistency;

subject implementation Persistency.impl


subcomponents
      saveProduction: data int default 0;
      saveConsumption: data int default 0;
      saveBattery: data int default 0;

modes
      idle: initial mode;
transitions

      idle-[ProductionEvent
           then saveProduction := StatusProduction;
           OutProduction := StatusProduction]->idle;

      idle-[ConsumptionEvent
           then saveConsumption := StatusConsumption;
           OutConsumption := StatusConsumption]->idle;

      idle-[BatteryEvent
           then saveBattery := StatusBattery;
           OutBattery := StatusBattery]->idle;

end Persistency.impl;

subject Aggregation
features
      productionValues: in data port int;
      consumptionValues: in data port int;
      batteryValues: in data port int;
      aggregatedValues: out data port int default 0;
end Aggregation;

subject implementation Aggregation.impl

flows

  (consumptionValues + productionValues + batteryValues) -> aggregatedValues;

end Aggregation.impl;

subject ProAgentOut
features
      InaggregatedValues: in data port int default 0;
      OutaggregatedValues: out data port int default 0;
      meterReading: out data port int default 0;
```

```
end ProAgentOut;

subject implementation ProAgentOut.impl
flows
      port InaggregatedValues -> OutaggregatedValues;
      port 100 -> meterReading;

end ProAgentOut.impl;

system WrapperSensors
features

      StatusProduction: out data port int default 0;
      StatusBattery: out data port int default 0;
      StatusConsumption: out data port int default 0;

      ProductionEvent: out event port;
      BatteryEvent: out event port;
      ConsumptionEvent: out event port;

end WrapperSensors;

system implementation WrapperSensors.impl

modes
idle: initial mode;
consumptionChanged: mode;
productionChanged: mode;
batteryStatusChanged: mode;


transitions

idle - [then StatusProduction := 1]
     -> productionChanged;

productionChanged - [ProductionEvent] -> idle;

idle - [then StatusConsumption := 2] -> consumptionChanged;

consumptionChanged - [ConsumptionEvent] -> idle;

idle - [then StatusBattery := 3] -> batteryStatusChanged;

batteryStatusChanged - [BatteryEvent] -> idle;

end WrapperSensors.impl;
```

An example property that can be checked for the verification of the above requirement is given as:

```
  {nuXmv:
  LTLSPEC
    (G ({WrapperSensors.StatusBattery} = 3 -> F ({Persistency.saveBattery} = 3)));
 }
```

which assures that as the status of the battery is updated to 3, this information is also updated in the persistency component.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.3 SMG_SO.3 requirement

SHALL: The aggregation component shall only communicate aggregated energy data to the prosumer energy agent. No sensor or device specific information shall be communicated for privacy reasons.

Evaluation: In the model given in SMG_SO.2, this requirement is handled by passing only aggregated data `aggregatedValues` from the aggregation to the prosumer energy agent component. For the verification of this requirement, one can check the following property:

```
    {nuXmv:
LTLSPEC
  (G ({Persistency.OutProduction}+{Persistency.OutConsumption}+{Persistency.OutBattery})
   = {ProAgentOut.InaggregatedValues} );
}
```

which asserts that the value of the in-port `InaggregatedValues` of the energy agent component is the sum of the values that persistency component passes to the aggregation component.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.4 SMG_SO.4 requirement

SHALL: The prosumer energy agent shall communicate only required data by the micro grid. The micro grid shall not have access to any private data by the prosumer. In particular the sensor values of prosumer systems shall not be transmitted to the micro grid. The only exception is the data of the main smart meter, since this is required for stability control.

Evaluation: In the model (given in SMG_SO.3), the prosumer energy agent `ProAgentOut` only passes aggregated data of the devices `OutaggregatedValues` and the reading of the main meter `meterReading` to the smart grid. This shows that the smart grid does not have access to the internal data of the prosumer. For the verification of this requirement, one can check the following property:

```
    {nuXmv:
LTLSPEC
  (G ({Persistency.OutProduction}+{Persistency.OutConsumption}+{Persistency.OutBattery})
   = {ProAgentOut.OutaggregatedValues} );
}
```

which asserts that the value of the out-port `OutaggregatedValues` of the energy agent component is the sum of the values that persistency component passes to the aggregation component.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.5 SMG_SO.5 requirement

SHOULD: A prosumer shall not be able to access the data from any other prosumer.

Evaluation: The above requirement is modeled by not having any connection between two prosumers of the smart grid system, as shown in the below model. This requirement can be verified as SMG_SO.1.

```
system SmartGridSystem
[...]

end SmartGridSystem;

system implementation SmartGridSystem.impl


subcomponents
      Prosumer1: system Prosumer.impl1 accesses Bus;
      Prosumer2: system Prosumer.impl2 accesses Bus;
      Bus: bus Bus.impl;
[...]

connections
[...]

end SmartGridSystem.impl;
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.6 SMG_SO.6 requirement

SHOULD: Components (e.g. Rule System) shall receive information events according to the information flow policy for prosumer and smart grid. However, subcomponents of a components (e.g. a certain rule) shall be able to subscribe to a subset of events (e.g. subset of sensors in the room). No other events shall be communicated to this subcomponent. Every new subscription or subscription change shall require a confirmation from the system administrator.

Evaluation: The information flow policy can be ensured as discussed in SMG_SO.1. However, the dynamic behaviour of a system, like subscription for events and addition of new components or events, cannot be modeled using MILS-ADDL.

Verdict:

Not Fulfilled – **Partially Fulfilled** – Largely Fulfilled – Fully Fulfilled

10.3.7 SMG_SO.7 requirement

SHALL: Authentication: A user shall authenticate himself to the control software in both cases: prosumer and micro grid. Every prosumer system shall be authenticated to the micro grid.

Evaluation: In the below model, the smart grid administrator authenticates a prosumer component. The prosumer first sets its ID on out-port `ProAuthCode` and then gives a signal on out-port `ProAuthenticateReq`. The administrator component gets this data, authenticates the prosumer and sends the result back to the prosumer.

```
constants
      Auth_Code_Pro1: int := 222;
[...]
system Prosumer
features

      ProAuthenticateReq: out event port;
      ProAuthCode: out data port int;

      ProAuthenticateRec: in event port;
      ProAuthReply: in data port bool;
end Prosumer;


system implementation Prosumer.impl

   {nuXmv:
   LTLSPEC
     (F (mode = mode_Authenticated));
   }
subcomponents

      Is_Authenticated: data bool default false;
[...]
modes
      idle: initial mode;
      Auth_Start: mode;
      Authentication_Waiting: mode;
      Authenticated: mode;
      Not_Authenticated: mode;
[...]

transitions

      idle -[then ProAuthCode := Auth_Code_Pro1] -> Auth_Start;

      Auth_Start -[ProAuthenticateReq] -> Authentication_Waiting;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = true
                        then Is_Authenticated := true]
                      -> Authenticated;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = false
                        then Is_Authenticated := false]
                      -> Not_Authenticated;

      Authenticated - [] -> Authenticated;
      Not_Authenticated - [] -> Not_Authenticated;
[...]
end Prosumer.impl;


system SmartGrid
features

      Pro1AuthenticateReq: in event port;
      Pro1AuthCode: in data port int;
      Pro1AuthenticateReply: out event port;
      Pro1AuthReply: out data port bool default false;

[...]

end SmartGrid;

system implementation SmartGrid.impl
subcomponents
```

```
      IsPro1_Authenticated: data bool default false;
      [...]


modes
      idle: initial mode;
      Pro1_Authenticated: mode;
      Pro1_Not_Authenticated: mode;
[...]
transitions



      idle -[Pro1AuthenticateReq when Pro1AuthCode = Prosumer1_Auth_Code then
            IsPro1_Authenticated := true; Pro1AuthReply := true ]
            -> Pro1_Authenticated;

      idle -[Pro1AuthenticateReq when Pro1AuthCode != Prosumer1_Auth_Code then
            IsPro1_Authenticated := false; Pro1AuthReply := false ]
            -> Pro1_Not_Authenticated;


      Pro1_Authenticated - [Pro1AuthenticateReply] -> idle;

      Pro1_Not_Authenticated - [Pro1AuthenticateReply] -> idle;
[...]

end SmartGrid.impl;
```

An example property that can be checked for the verification of the above requirement is given as:

```
   {nuXmv:
   LTLSPEC
     (G ({ProAuthCode} = 222 -> (F (mode = mode_Authenticated))));
  }
```

which asserts that the prosumer is authenticated by the smart grid system component with the correct authentication code `222`.

Similarly, the smart grid administrator and the prosumer administrator components authenticate their respective users. In the model given in the requirement SMG_SO.8, prosumer administrator component authenticates the prosumer user component.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.8 SMG_SO.8 requirement

SHALL: Authorization: Every user shall have a limited set of rights. He shall not be able to obtain more rights than he has been assigned.

Evaluation: This requirement is modeled in the following model by showing that the prosumer user component can only update consumption-rules if it is authorized to do so – depicted by a local boolean flag `Can_Update_Consumption_Level`.

```
constants
      ProUser_Auth_Code : int := 444;

      ConsumptionLevel1: int := 1;
      ConsumptionLevel2: int := 2;
[...]


system Prosumer

end Prosumer;


system implementation Prosumer.impl1

subcomponents


      ProUserAdmin: subject ProUserAdmin.impl;
      ProUser: subject ProUser.impl;
      Rule: subject Rule.impl;

connections


      port ProUser.ProAuthenticateReq -> ProUserAdmin.ProAuthenticateReq;

      port ProUser.ProAuthCode -> ProUserAdmin.ProAuthCode;

      port ProUserAdmin.ProAuthenticateReply -> ProUser.ProAuthenticateRec;

      port ProUserAdmin.ProAuthReply -> ProUser.ProAuthReply;

      port ProUser.setConsumptionLevel -> Rule.setConsumptionLevel;

end Prosumer.impl1;


subject ProUser
features

      ProAuthenticateReq: out event port;
      ProAuthCode: out data port int;

      ProAuthenticateRec: in event port;
      ProAuthReply: in data port bool;

      setConsumptionLevel: out data port int default ConsumptionLevel1;

end ProUser;


subject implementation ProUser.impl


subcomponents

      Is_Authenticated: data bool default false;
      Can_Update_Consumption_Level: data bool default false;


modes
      idle: initial mode;
      Auth_Start: mode;
      Authentication_Waiting: mode;
      Authenticated: mode;
```

```
transitions

      idle -[then ProAuthCode := ProUser1_Auth_Code ] -> Auth_Start;

      Auth_Start -[ProAuthenticateReq] -> Authentication_Waiting;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = true
                              then Is_Authenticated := true] -> Authenticated;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = false
                              then Is_Authenticated := false] -> idle;

      Authenticated -[when Can_Update_Consumption_Level = false ]
                -> Authenticated;

      Authenticated -[when Can_Update_Consumption_Level = true
      then setConsumptionLevel := ConsumptionLevel2 ] -> Authenticated;

end ProUser.impl;


subject ProUserAdmin
features

      ProAuthenticateReq: in event port;
      ProAuthCode: in data port int;
      ProAuthenticateReply: out event port;
      ProAuthReply: out data port bool default false;



end ProUserAdmin;

subject implementation ProUserAdmin.impl
subcomponents

      IsProUser_Authenticated: data bool default false;
      ProUser_Ath_Attempt: data int default 0;



modes
      idle: initial mode;
      ProUser_Authenticated: mode;
      ProUser_Not_Authenticated: mode;
      ProUser_Blocked: mode;

transitions

      idle -[ProAuthenticateReq when ProAuthCode = ProUser_Auth_Code then
            IsProUser_Authenticated := true; ProAuthReply := true ] -> ProUser_Authenticated;

      idle -[ProAuthenticateReq when ProAuthCode != ProUser_Auth_Code then
            IsProUser_Authenticated := false; ProUser_Ath_Attempt := ProUser_Ath_Attempt + 1;
            ProAuthReply := false ] -> ProUser_Not_Authenticated;


      ProUser_Authenticated - [ProAuthenticateReply] -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt<3]
                        -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt>2 ]
                        -> ProUser_Blocked;

      ProUser_Blocked - [] -> ProUser_Blocked;
```

```
end ProUserAdmin.impl;
```

For the verification of the above requirement, we can check that the following property will not hold as the default value of `Can_Update_Consumption_Level` is `false`, i.e, the user is not authorized to update consumption rules.

```
{nuXmv:
LTLSPEC
  (G ((mode = mode_Authenticated) -> F ({setConsumptionLevel} = 2)));
}
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.9 SMG_SO.9 requirement

SHALL: The admin of each prosumer and smart grid system shall be able to access only his system. Access to other systems shall not be possible.

Evaluation: This requirement is analogous to the second half of the requirement SMG_SO.7. If a user tries to access another prosumer (not assigned to him), he is not authenticated.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.10 SMG_SO.10 requirement

SHOULD: Logins of the admin shall be logged persistently.

Evaluation: This requirement is analogous to the second half of the requirement SMG_SO.7. Note that in the model (given in SMG_SO.8) when the prosumer administrator authenticates the prosumer user, it sets its local boolean flag `Is_ProUser_Authenticated` to `true`. This requirement can be verified by checking the following property:

```
{nuXmv:
LTLSPEC
  ( F ({IsProUser_Authenticated}));
}
```

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.11 SMG_SO.11 requirement

SHOULD: Maintenance of the rule system shall be possible for users, which shall be logged. Users shall not be able to modify or delete log files.

Evaluation: In the below model, the user component of the prosumer can change the consumption-level rule (as `Can_Update_Consumption_Level` is `true`) of the rule component. Moreover, the updated consumption value is saved in the local variable `saveConsumptionLevel` of the rule component. However, MILS AADL does not support modeling the modification and deletion of log files.

```
constants

      ProUser_Auth_Code: int := 444;
      ConsumptionLevel1: int := 1;
      ConsumptionLevel2: int := 2;


system Prosumer

end Prosumer;


system implementation Prosumer.impl


subcomponents


      ProUserAdmin: subject ProUserAdmin.impl;
      ProUser: subject ProUser.impl;
      Rule: subject Rule.impl;

connections


      port ProUser.ProAuthenticateReq -> ProUserAdmin.ProAuthenticateReq;

      port ProUser.ProAuthCode -> ProUserAdmin.ProAuthCode;

      port ProUserAdmin.ProAuthenticateReply -> ProUser.ProAuthenticateRec;

      port ProUserAdmin.ProAuthReply -> ProUser.ProAuthReply;

      port ProUser.setConsumptionLevel -> Rule.setConsumptionLevel;

end Prosumer.impl;



subject ProUser
features

      ProAuthenticateReq: out event port;
      ProAuthCode: out data port int;

      ProAuthenticateRec: in event port;
      ProAuthReply: in data port bool;

      setConsumptionLevel: out data port int default ConsumptionLevel1;
```

```
end ProUser;


subject implementation ProUser.impl

subcomponents

      Is_Authenticated: data bool default false;
      Can_Update_Consumption_Level: data bool default true;


modes
      idle: initial mode;
      Auth_Start: mode;
      Authentication_Waiting: mode;
      Authenticated: mode;


transitions

      idle -[then ProAuthCode := ProUser1_Auth_Code ] -> Auth_Start;

      Auth_Start -[ProAuthenticateReq] -> Authentication_Waiting;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = true
                  then Is_Authenticated := true]
                  -> Authenticated;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = false
                  then Is_Authenticated := false]
                  -> idle;

      Authenticated -[when Can_Update_Consumption_Level = false ]
                -> Authenticated;

      Authenticated -[when Can_Update_Consumption_Level = true
      then setConsumptionLevel := ConsumptionLevel2 ] -> Authenticated;

end ProUser.impl;


subject ProUserAdmin
features

      ProAuthenticateReq: in event port;
      ProAuthCode: in data port int;
      ProAuthenticateReply: out event port;
      ProAuthReply: out data port bool default false;



end ProUserAdmin;

subject implementation ProUserAdmin.impl

subcomponents

      IsProUser_Authenticated: data bool default false;
      ProUser_Ath_Attempt: data int default 0;



modes
      idle: initial mode;
      ProUser_Authenticated: mode;
      ProUser_Not_Authenticated: mode;
      ProUser_Blocked: mode;
```

```
transitions


      idle -[ProAuthenticateReq when ProAuthCode = ProUser_Auth_Code then
            IsProUser_Authenticated := true; ProAuthReply := true ]
            -> ProUser_Authenticated;

      idle -[ProAuthenticateReq when ProAuthCode != ProUser_Auth_Code then
            IsProUser_Authenticated := false;
            ProUser_Ath_Attempt := ProUser_Ath_Attempt + 1;
            ProAuthReply := false ] -> ProUser_Not_Authenticated;


      ProUser_Authenticated - [ProAuthenticateReply] -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt<3]
                          -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt>2]
                          -> ProUser_Blocked;

      ProUser_Blocked - [] -> ProUser_Blocked;




end ProUserAdmin.impl;


subject Rule
features
      setConsumptionLevel: in data port int;
      consumptionLevel: out data port int default ConsumptionLevel1;
end Rule;

subject implementation Rule.impl

subcomponents
   saveConsumptionLevel: data int default 0;

flows
      port setConsumptionLevel -> consumptionLevel;

modes

   idle: initial mode;
   Ruleupdated: mode;

transitions

     idle - [when setConsumptionLevel != saveConsumptionLevel]
        -> Ruleupdated;

     Ruleupdated - [then saveConsumptionLevel := setConsumptionLevel]
       -> idle;

end Rule.impl;
```

For the verification of the above requirement, one can check the following property:

```
  {nuXmv:
```

```
LTLSPEC
  (G (({ProUser.setConsumptionLevel}=2) -> F ({Rule.consumptionLevel} = 2 and
  {Rule.saveConsumptionLevel} = 2)));
}
```

which asserts that as the data on the out-port `setConsumptionlevel` of the user component updates, it also reflects on the out-port `consumptionlevel` of the rule component and is saved in the local variable `saveConsumptionLevel`.

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.3.12 SMG_SO.12 requirement

SHALL: The persistency component (a) shall always be invoked and (b) shall be tamper-proof, meaning that no other component can delete, add or tamper with information stored in the Persistency component.

Evaluation: The first part of this requirement is analogous to SMG_SO.2. However, it is not possible to model addition, deletion or tempering of information stored in Persistency component.

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.3.13 SMG_SO.13 requirement

SHOULD: Each event shall have a trustworthy time stamp to allow safety and security analysis.

Evaluation: MILS-AADL provide a clock data type, which allows for time stamping events. However, we are not modeling the timing behaviour of our smart grid demonstrator, therefore, we do not verify this requirement.

Verdict:

Not Fulfilled – **Partially Fulfilled** – Largely Fulfilled – Fully Fulfilled

10.3.14 SMG_SO.14 requirement

SHOULD: Command events, which trigger certain actions of devices, shall not be lost. A confirmation of transmission is required.

Evaluation: D-MILS AADL provides constructs to give error models of components. Therefore, its possible to model a network in which messages are lost with certain probabilities. However, for our smart grid demonstrator, we are not modeling probabilistic behaviour of components, this requirement, therefore, is not verifiable.

Verdict:

Not Fulfilled – **Partially Fulfilled** – Largely Fulfilled – Fully Fulfilled

10.3.15 SMG_SO.15 requirement

SHOULD: The system shall provide a subscription mechanism so that subcomponents can subscribe to events. The subscription mechanism shall consider the information flow policy.

Evaluation: MILS AADL does not provide constructs to model dynamically changing systems. Therefore, modeling the subscription mechanism for events is not possible.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – Fully Fulfilled

10.3.16 SMG_SO.16 requirement

SHALL: The system shall provide a mechanism for authentication and authorization for users.

Evaluation: This requirement is analogous to SMG_SO.7 and SMG_SO.8

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.17 SMG_SO.17 requirement

SHALL: The system shall provide a mechanism to check the rights of a user to enable the control of the user interaction.

Evaluation: This requirement is analogous to SMG_SO.8

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.18 SMG_SF_DP.1 requirement

SHALL: The TSF shall enforce the information flow control expressed as an annotated information flow graph.

Evaluation: This requirement is analogous to SMG_SO.7

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.19 SMG_SF_DP.2 requirement

SHOULD: The TSF shall enforce the information flow control policy on all subjects and objects represented in the flow graph and all operations that cause that information to flow to and from subjects covered by the SFP[1].

Evaluation: This requirement is analogous to SMG_SO.7

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.20 SMG_SF_DP.3 requirement

SHOULD: The TSF shall ensure that all operations that cause any information in the TOE to flow to and from any subject in the TOE are covered by an information flow control SFP.

Evaluation: This requirement is analogous to SMG_SO.7

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.21 SMG_SF_IA.4 requirement

SHOULD: The TSF shall detect when three unsuccessful authentication attempts occur related to user authentication events.

Evaluation: In the model given in SMG_SO.8, when there are three unsuccessful attempts by the prosumer user component, the prosumer user administrator component enters into `ProUser_Blocked` mode. This requirement can be verified by checking the following property:

```
{nuXmv:
LTLSPEC
  (G (({ProUser_Ath_Attempt}=3) -> F (mode = mode_ProUser_Blocked)));
}
```

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.22 SMG_SF_IA.5 requirement

SHOULD: When the defined number of unsuccessful events has been detected, the TSF shall prohibit new authentication attempts for the next 30 seconds.

[1] Security Function Policies

Evaluation: The first half of this requirement is analogous to SMG_SF_IA.4. As MILS AADL provides clock data type, it allows for modeling authentication-blocking for 30 seconds. However, as our smart grid demonstrator does not have timing behaviour, we are not verifying the timing aspects of this requirement.

Verdict: Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.3.23 SMG_SF_IA.6 requirement

SHOULD: The TSF shall maintain the following list of security attributes belonging to individual users:

a) The user authentication attributes (name, password)

b) The internal userID

c) The user dedicated roles

d) The active user roles

e) The room numbers related to the user

Evaluation: D-MILS AADL allows to arrange different data types in tuples; this allows us to store the above attributes for each user.

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.24 SMG_SF_IA.7 requirement

SHOULD: The TSF shall allow access to the login screen on behalf of the user to be performed before the user is authenticated.

Evaluation: D-MILS AADL does not provide constructs for modeling interaction of a user with a login screen.

Verdict: **Not Fulfilled** – Partially Fulfilled – Largely Fulfilled – Fully Fulfilled

10.3.25 SMG_SF_IA.8 requirement

SHOULD: The TSF shall require each user to be successfully authenticated before allowing any other TSF-mediated actions on behalf of the user.

Evaluation: In the model given in SMG_SO.11, a user can only update the `consumptionlevel` in the rule component if he is successfully authenticated; otherwise, he is not allowed to make any changes in the rule component. For the verification of the above requirement, the following property can be checked which asserts that the consumption rule is changed if the user is in `Authenticated` mode.

```
{nuXmv:
 LTLSPEC
   (G (({setConsumptionLevel} = 2) -> (mode = mode_Authenticated)));
}
```

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.26 SMG_SF_IA.11 requirement

SHOULD: The TSF shall associate the following user security attributes with subjects acting on behalf of that user:

a) User name, internal userID, active user role.

Evaluation: This requirement is analogous to SMG_SF_IA.6

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.3.27 SMG_SF_TA.12 requirement

SHOULD: The TSF shall terminate an interactive session after 24 hours of user's inactivity.

Evaluation: MILS AADL provides clock data type that allows for modeling timing behaviour. However, as our smart grid demonstrator does not have timing behaviour, we are not verifying this requirement.

Verdict: Not Fulfilled – **Partially Fulfilled** – Largely Fulfilled – Fully Fulfilled

10.3.28 SMG_SF_TA.13 requirement

SHOULD: The TSF shall allow user-initiated termination of the user's own interactive session.

Evaluation: In the model given below, a user in `Authenticated` mode generates an event `ProCloseSession` to close her session. The event is handled by the user administrator component which closes the session of the user by setting the flag `IsProUser_Authenticated` to `false`.

```
constants
      ProUser_Auth_Code : int := 444;


subject ProUser
features

      ProAuthenticateReq: out event port;
      ProAuthCode: out data port int;

      ProAuthenticateRec: in event port;
      ProAuthReply: in data port bool;
      ProCloseSession: out event port;

end ProUser;


subject implementation ProUser.impl


subcomponents

      Is_Authenticated: data bool default false;

modes
      idle: initial mode;
      Auth_Start: mode;
      Authentication_Waiting: mode;
      Authenticated: mode;


transitions

      idle -[then ProAuthCode := ProUser_Auth_Code ] -> Auth_Start;

      Auth_Start -[ProAuthenticateReq] -> Authentication_Waiting;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = true
                      then Is_Authenticated := true]
                     -> Authenticated;

      Authentication_Waiting - [ProAuthenticateRec when ProAuthReply = false
                     then Is_Authenticated := false]
                     -> idle;
      Authenticated - [ProCloseSession then Is_Authenticated:= false] -> idle;


end ProUser.impl;


subject ProUserAdmin
features

      ProAuthenticateReq: in event port;
      ProAuthCode: in data port int;
      ProAuthenticateReply: out event port;
      ProAuthReply: out data port bool default false;
      ProCloseSession: in event port;



end ProUserAdmin;

subject implementation ProUserAdmin.impl
```

```
subcomponents

      IsProUser_Authenticated: data bool default false;
      ProUser_Ath_Attempt: data int default 0;

modes
      idle: initial mode;
      ProUser_Authenticated: mode;
      ProUser_Not_Authenticated: mode;
      ProUser_Blocked: mode;

transitions


      idle -[ProAuthenticateReq when ProAuthCode = ProUser_Auth_Code then
            IsProUser_Authenticated := true; ProAuthReply := true ]
            -> ProUser_Authenticated;

      idle -[ProAuthenticateReq when ProAuthCode != ProUser_Auth_Code then
            IsProUser_Authenticated := false;
            ProUser_Ath_Attempt := ProUser_Ath_Attempt + 1;
            ProAuthReply := false ] -> ProUser_Not_Authenticated;


      ProUser_Authenticated - [ProAuthenticateReply] -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt<3]
                        -> idle;

      ProUser_Not_Authenticated - [ProAuthenticateReply when ProUser_Ath_Attempt>2 ]
                        -> ProUser_Blocked;

      ProUser_Blocked - [] -> ProUser_Blocked;

      idle - [ProCloseSession when IsProUser_Authenticated = true
            then ProUser_Ath_Attempt := 0; IsProUser_Authenticated := false]
            -> idle;


end ProUserAdmin.impl;


system Prosumer

end Prosumer;


system implementation Prosumer.impl

subcomponents


      ProUserAdmin: subject ProUserAdmin.impl;
      ProUser: subject ProUser.impl;


connections


      port ProUser.ProAuthenticateReq -> ProUserAdmin.ProAuthenticateReq;

      port ProUser.ProAuthCode -> ProUserAdmin.ProAuthCode;

      port ProUserAdmin.ProAuthenticateReply -> ProUser.ProAuthenticateRec;
```

```
      port ProUserAdmin.ProAuthReply -> ProUser.ProAuthReply;

      port ProUser.ProCloseSession -> ProUserAdmin.ProCloseSession;

end Prosumer.impl;
```

For the verification of the above requirement, the following property can be verified which asserts that as the user generates an event to close its session, the corresponding flag in the administrator component is set to `false`.

```
  {nuXmv:
  LTLSPEC
    (G (event_ProCloseSession) ->
       F (!{IsProUser_Authenticated}));
 }
```

Verdict: Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.4 System Requirements

10.4.1 SMG_SR.1 requirement

Description:

Should: Rule System shall monitor the system and trigger rules specified by the users.

Evaluation: Modeling of dynamic behavior in MILS-AADL is not possible.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – Fully Fulfilled

10.4.2 SMG_SR.2 requirement

Description:

Should: Rule System shall consider constraints specified by the user.

Evaluation: Modeling of dynamic behavior in MILS-AADL is not possible.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – Fully Fulfilled

10.4.3 SMG_SR.3 requirement

Description:

Shall: Rule System shall consider safety constraints specified according the safety constraints of the devices.

Evaluation: MILS-AADL supports modeling any kind of functional behavior using state automatons. Any safety rules could be encoded using state automatons and therefore would be then considered by the devices. A possible rule, which we considered in our system, is that when a deviation event arrives the unnecessary devices are sent into sleep mode.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

10.4.4 SMG_SR.4 requirement

Description:

May: Wrappers shall send data of sensors to the system as events in real time.

Evaluation: The real time properties of the system was not possible to prove in MILS-AADL. Therefore we performed an experiment on the platform: We measured the Round Trip Time of a signal in our demonstrator. The signal took less than 1 ms. Therefore we conclude that using D-MILS technology it is possible to satisfy real-time requirements.

Verdict:

Not Fulfilled – Partially Fulfilled – **Largely Fulfilled** – Fully Fulfilled

10.4.5 SMG_SR.5 requirement

Description:

May: Dynamic addition of new devices (software and hardware) shall be possible.

Evaluation: Expressing a dynamic addition of new devices (software and hardware) is not possible in MILS-AADL.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – Fully Fulfilled

10.5 Hardware / Software Platform

Description:

Shall: The software shall run on a platform as described in section 9.

Evaluation: Our demonstrator runs on the platform and therefore this requirement is satisfied.

Verdict:

Not Fulfilled – Partially Fulfilled – Largely Fulfilled – **Fully Fulfilled**

11 Compliance Matrix

This chapter presents the compliance matrix for Smart Microgrid case study. The compliance matrix table lists the requirements for the Smart Microgrid case study, defined in the deliverable D1.1 and shows whether the requirements are addressed by the D-MILS Platform. The verdicts can go from *Not Fulfilled* over *Partially Fulfilled* and *Largely Fulfilled* to *Fully Fulfilled*. A detailed justification for each verdict can be found in section 10.

Table 7: Requirements Compliance Matrix

Req ID	**Priority**	**Requirement text (may be abbreviated)**	**Verdict**
SMG_DL.1	Should	Support of compositional modules with interfaces capable representing IEC 61850.	Largely Fulfilled
SMG_DL.2	Shall	Support composition mechanisms with asynchronous and synchronous components. In particular event based communication between software components shall be supported as well as channels to represent physical behavior of the system.	Fully Fulfilled
SMG_DL.3	Shall	Modeling of physical behavior (e.g. energy, power, temperature) should be possible with one step difference equations using linear arithmetic.	Fully Fulfilled
SMG_DL.4	Shall	Internal state of the system should be representable with state machines extended with real number attributes, e.g. for describing the physical environment.	Fully Fulfilled
SMG_DL.5	Should	The description language should allow the representation of component faults or failures in order to evaluate their effect on the system.	Fully Fulfilled
SMG_DL.6	Should	The description language should support discrete time for modeling of the system behavior.	Fully Fulfilled
SMG_SA.1	Shall	The highest-level safety priority shall be grid stability. Indicators of instability are: deviation from the frequency 50 Hz and deviation from the nominal voltage level. In the case that the frequency deviates more than 1 Hz, the micro grid shall switch to island mode.	Fully Fulfilled
SMG_SA.2	Shall	In case of a power outage, the micro grid shall switch to an island mode.	Fully Fulfilled
SMG_SA.3	Should	Switching to island mode shall to be accomplished in less than 20 ms.	Largely Fulfilled
SMG_SA.4	Should	If the voltage of the connection between the micro grid and smart grid becomes higher than 10% of the nominal value (e.g. 400V), the smart micro grid island mode shall be activated to protect consumer electronics.	Fully Fulfilled
SMG_SA.5	Should	If the voltage in the micro grid is above a certain threshold (e.g. 10% above 400V), the switch box of the prosumer shall activate the prosumer island mode, in order to keep the battery in safe operation.	Fully Fulfilled

continued on next page

continuing from previous page

SMG_SA.6	Should	If prosumer island mode is active and a high consuming device is turned on (e.g. vacuum cleaner or oven), the power of the corresponding power socket shall be turned off in the switch box.	Fully Fulfilled
SMG_SA.7	Should	The rule system of prosumer shall immediately be informed when micro grid island mode is activated. The available time window depends on the hardware components, but typical times are around 100ms.	Largely Fulfilled
SMG_SA.8	Shall	Every battery component shall not be overloaded. This means that if the battery status is full, the control system shall not send any further loading signal.	Fully Fulfilled
SMG_SA.9	Should	The charging or discharging rate of the batteries shall be within the bound specified in the documents. The control system shall take care that the rate is within the limits.	Fully Fulfilled
SMG_SA.10	Should	The battery temperature shall be within the bounds specified in the documents. The control system shall reduce charge or discharge rate to zero, in case the battery temperature is above the temperature limit.	Fully Fulfilled
SMG_SA.11	May	In case of island mode, full batteries and a higher production than consumption - the production units shall reduce their production.	Largely Fulfilled
SMG_SO.1	Shall	Communication / Information Flow between components shall be according to the policy defined for Prosumer and Smart Grid. No other information flow shall occur.	Fully Fulfilled
SMG_SO.2	Should	All wrapper events as well as device status data shall be logged by the persistency component. This means in particular that no wrapper component can hide events or change of states.	Fully Fulfilled
SMG_SO.3	Shall	The aggregation component shall only communicate aggregated energy data to the prosumer energy agent. No sensor or device specific information shall be communicated for privacy reasons.	Fully Fulfilled
SMG_SO.4	Shall	The prosumer energy agent shall communicate only required data by the micro grid. The micro grid shall not have access to any private data by the prosumer. In particular the sensor values of prosumer systems shall not be transmitted to the micro grid. The only exception is the data of the main smart meter, since this is required for stability control.	Fully Fulfilled
SMG_SO.5	Should	A prosumer shall not be able to access the data from any other prosumer.	Fully Fulfilled

continued on next page

continuing from previous page

SMG_SO.6	Should	Components (e.g. Rule System) shall receive information events according to the information flow policy for prosumer and smart grid. However, subcomponents of a components (e.g. a certain rule) shall be able to subscribe to a subset of events (e.g. subset of sensors in the room). No other events shall be communicated to this subcomponent. Every new subscription or subscription change shall require a confirmation from the system administrator.	Partially Fulfilled
SMG_SO.7	Shall	Authentication: A user shall authenticate himself to the control software in both cases: prosumer and micro grid. Every prosumer system shall be authenticated to the micro grid.	Fully Fulfilled
SMG_SO.8	Shall	Authorization: Every user shall have a limited set of rights. He shall not be able to obtain more rights than he has been assigned.	Fully Fulfilled
SMG_SO.9	Shall	The admin of each prosumer and smart grid system shall be able to access only his system. Access to other systems shall not be possible.	Fully Fulfilled
SMG_SO.10	Should	Logins of the admin shall be logged persistently.	Fully Fulfilled
SMG_SO.11	Should	Maintenance of the rule system shall be possible for users, which shall be logged. Users shall not be able to modify or delete log files.	Largely Fulfilled
SMG_SO.12	Shall	The persistency component (a) shall always be invoked and (b) shall be tamper-proof, meaning that no other component can delete, add or tamper with information stored in the Persistency component.	Largely Fulfilled
SMG_SO.13	Should	Each event shall have a trustworthy time stamp to allow safety and security analysis.	Partially Fulfilled
SMG_SO.14	Should	Command events, which trigger certain actions of devices, shall not be lost. A confirmation of transmission is required.	Partially Fulfilled
SMG_SO.15	Should	The system shall provide a subscription mechanism so that subcomponents can subscribe to events. The subscription mechanism shall consider the information flow policy.	Not Fulfilled
SMG_SO.16	Shall	The system shall provide a mechanism for authentication and authorization for users.	Fully Fulfilled
SMG_SO.17	Shall	The system shall provide a mechanism to check the rights of a user to enable the control of the user interaction.	Fully Fulfilled
SMG_SF_DP.1	Shall	The TSF shall enforce the information flow control expressed as an annotated information flow graph.	Fully Fulfilled
SMG_SF_DP.2	Should	The TSF shall enforce the information flow control policy on all subjects and objects represented in the flow graph and all operations that cause that information to flow to and from subjects covered by the SFP.	Fully Fulfilled

continued on next page

			continuing from previous page
SMG_SF_DP.3	Should	The TSF shall ensure that all operations that cause any information in the TOE to flow to and from any subject in the TOE are covered by an information flow control SFP.	Fully Fulfilled
SMG_SF_IA.4	Should	The TSF shall detect when three unsuccessful authentication attempts occur related to user authentication events.	Fully Fulfilled
SMG_SF_IA.5	Should	When the defined number of unsuccessful events has been detected, the TSF shall prohibit new authentication attempts for the next 30 seconds.	Largely Fulfilled
SMG_SF_IA.6	Should	The TSF shall maintain the following list of security attributes belonging to individual users: (authentication attributes, userID, roles, room numbers)	Fully Fulfilled
SMG_SF_IA.7	Should	The TSF shall allow access to the login screen on behalf of the user to be performed before the user is authenticated.	Not Fulfilled
SMG_SF_IA.8	Should	The TSF shall require each user to be successfully authenticated before allowing any other TSF-mediated actions on behalf of the user.	Fully Fulfilled
SMG_SF_IA.11	Should	The TSF shall associate the following user security attributes with subjects acting on behalf of that user: user name, internal userID, active user role.	Fully Fulfilled
SMG_SF_TA.12	Should	The TSF shall terminate an interactive session after 24 hours of user's inactivity.	Partially Fulfilled
SMG_SF_TA.13	Should	The TSF shall allow user-initiated termination of the user's own interactive session.	Fully Fulfilled
SMG_SR.1	Should	Rule System shall monitor the system and trigger rules specified by the users.	Not Fulfilled
SMG_SR.2	Should	Rule System shall consider constraints specified by the user.	Not Fulfilled
SMG_SR.3	Shall	Rule System shall consider safety constraints specified according the safety constraints of the devices.	Fully Fulfilled
SMG_SR.4	May	Wrappers shall send data of sensors to the system as events in real time.	Largely Fulfilled
SMG_SR.5	May	Dynamic addition of new devices (software and hardware) shall be possible.	Not Fulfilled
SMG_HS.1	Shall	The software shall run on a platform as described in section 9.	Fully Fulfilled

References

[1] Gsn community standard version 1. Technical report, Origin Consulting (York) Limited, November 2011. 44

[2] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hoelzl, and Bernhard Schaetz. Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. In *Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-Physical and Embedded Systems (MODELS 2015)*, September 2015. 36

[3] The COMPASS project web site. http://compass.informatik.rwth-aachen.de/. 29

[4] Ewen Denney, Ganesh Pai, and Josef Pohl. Advocate: An assurance case automation toolset. In *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 8–21, Berlin, Heidelberg, 2012. Springer-Verlag. 58

[5] Patrick Graydon, John Knight, and Mitchell Green. Certification and safety cases. In *28th International System Safety Conference*, August 2010. 38

[6] Prof Nancy Leveson. White paper on the use of safety cases in certification and regulation, 2011. 39, 40

[7] R. Weaver. *The Safety of Software - Constructing and Assuring Argument*. PhD thesis, 2004. 48