A Model-Based Approach for Product Testing and Certification in Digital Ecosystems

Bruno Lima*† and João Pascoal Faria*†
*INESC TEC,
FEUP campus, Rua Dr. Roberto Frias, s/n4200-465 Porto, Portugal
†Faculty of Engineering, University of Porto,
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal

{bruno.lima, jpf}@fe.up.pt

Abstract—**In a growing number of domains, such as ambient-assisted living (AAL) and e-health, the provisioning of end-to-end services to the users depends on the proper interoperation of multiple products from different vendors, forming a digital ecosystem. To ensure interoperability and the integrity of the ecosystem, it is important that candidate products are independently tested and certified against applicable interoperability requirements. Based on the experience acquired in the AAL4ALL project, we propose in this paper a model-based approach to systematize, automate and increase the assurance of such testing and certification activities. The approach encompasses the construction of several models: a feature model, an interface model, a product model, and unit and integration test models. The abstract syntax and consistency rules of these models are specified by means of metamodels written in UML and Alloy and automatically checked with Alloy Analyzer. Using the model finding capabilities of Alloy Analyzer, integration tests can be automatically generated from the remaining models, through the composition and instantiation of unit tests. Examples of concrete models from the AAL4ALL project are also presented.**

Index Terms—**Certification; Metamodel; Test Models; Test Generation; Integration Testing; Ambient-Assisted Living**

I. Introduction

In a growing number of domains, the provisioning of end-to-end services to the users depends on the proper interoperation of multiple products (devices, applications, etc.) from different vendors, forming a digital ecosystem. To ensure interoperability and the integrity of the ecosystem, it is important that candidate products are independently tested and certified against applicable interoperability requirements, usually involving integration test scenarios.

An example is the e-health domain, where Ambient-Assisted Living (AAL) technologies [1] are being increasingly used in response to problems caused by the increasing age of the population. Some of the first AAL solutions (e.g. [2]) were monolithic, incompatible and thus expensive and potentially not sustainable. The AAL4ALL project [3] tried to answer those problems through the development of an ecosystem of interoperable products for AAL, associated to a business model and validated through a large scale trial. One goal of this project was to ensure that any supplier of AAL products, whether they are physical devices or software, can enter the ecosystem easily and independently, whilst assuring their interoperability with the rest of the ecosystem. To that end, the AAL4ALL project encompassed the specification of a set of reference models and requirements or standards, against which candidate products can be certified and subsequently integrated as components of the ecosystem. The project also encompassed the definition of a testing and certification methodology for candidate components [4]. However, there was a lack of formalization of reference models, requirements and test cases, and a lack of tool support for test generation, test execution and overall traceability and consistency checking between artifacts.

Recently the use of Model-Driven Engineering (MDE) techniques as a response to deal with complex problems (like the previously described), has been increasing [5]. One of the MDE technologies are the Domain-Specific Modeling Languages (DSMLs) [5]. These languages allow the formalization of the application structure, behavior, and requirements within particular domains, such as software-defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of middleware platforms. The abstract syntax of DSMLs may be described through metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Another key MDE technology are model transformation languages (MTLs) and engines [6]. Model-to-model transformations can be specified declaratively as relations or mappings [7] between source and target metamodels. Both DSMLs and MTLs are enabling technologies for model-based testing purposes (MBT) [8], namely for automatic test generation from models.

In this article, we take advantage of the aforementioned techniques, to formalize, improve and generalize for digital ecosystems the testing and certification approach that was implemented in the AAL4ALL project, and enable automatic test generation. In summary, the main contributions of this paper are:

- an overall modeling and testing approach for organizing and deriving (whenever possible) the relevant models needed for product testing and certification in digital ecosystems;
- detailed metamodels, formally specified in Alloy [9] and documented with UML [10], defining the abstract syntax,

consistency rules and semantics of DSMLs that can be used for constructing and checking the previous models;
- model transformation rules, embedded in the previous metamodels, that allow the automatic derivation of integration test models for candidate products, through the instantiation and composition of generic unit test models defined at the domain level, taking advantage of the model finding capabilities of Alloy Analyzer;
- application examples (models) from the AAL4ALL ecosystem, formally specified in Alloy and represented visually in a concrete notation, for illustrating and partially validating the approach.

The rest of the paper is organized as follows: section II presents the overall modeling and test generation approach; section III presents the proposed metamodels; in section IV are presented application examples; related work is presented in section V; conclusions an future work are presented in section VI.

II. Modeling and Test Generation Approach

A. Modeling Approach

Our modeling approach is based on the traditional OMG Modeling Infrastructure [11] that consists of a hierarchy of model levels, each (except the top) being characterized as "an instance" of the level above. The bottom level, also referred to as M0 is said to hold the "user data", i.e., the actual data objects the software is designed to manipulate. The next level, M1, is said to hold a "model" of the M0 user data. This is the level at which user models reside. Level M2 is said to hold a "model" of the information at M1. Since it is a model of a (user) model, it is often referred to as a metamodel. Finally, level M3 is said to hold a model of the information at M2, and hence is often characterized as the meta-metamodel. For historical reasons it is also referred to as the Meta Object Facility (MOF) [12]. Figure 1 shows the global perspective of the models proposed.

Our modeling approach is also sought to support an incremental process, starting with domain modeling, followed by the modeling of specific products (certified or candidate for certification), and the manual or automatic generation of integration tests for candidate products.

Hence, in M2 we propose a set of metamodels divided in two different levels, *Domain Level* and *Product Level*. The *Domain Level* contains generic definitions related with a given domain, needed to support the testing and certification of candidate products, but without reference to actual products: product categories allowed in the domain (for certification purposes); product features permitted in each category; generic interfaces and allowed message types associated with the product categories; and generic unit tests associated with the product features, defining required behaviors associated with those features. Hence, inside this level we propose three different metamodels, `FeatureMetamodel` (related with product categories and features), `InterfaceMetamodel` and `UnitTestMetamodel`. Whenever possible, concepts from UML are reused.

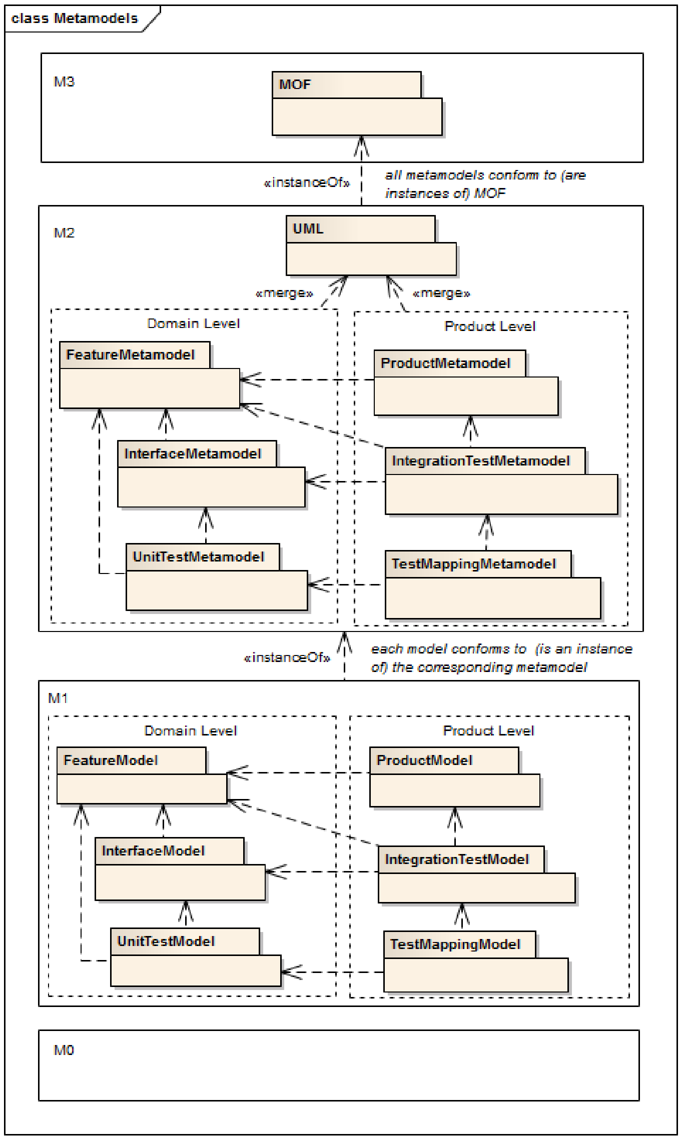


Fig. 1. Proposed Models and Metamodels

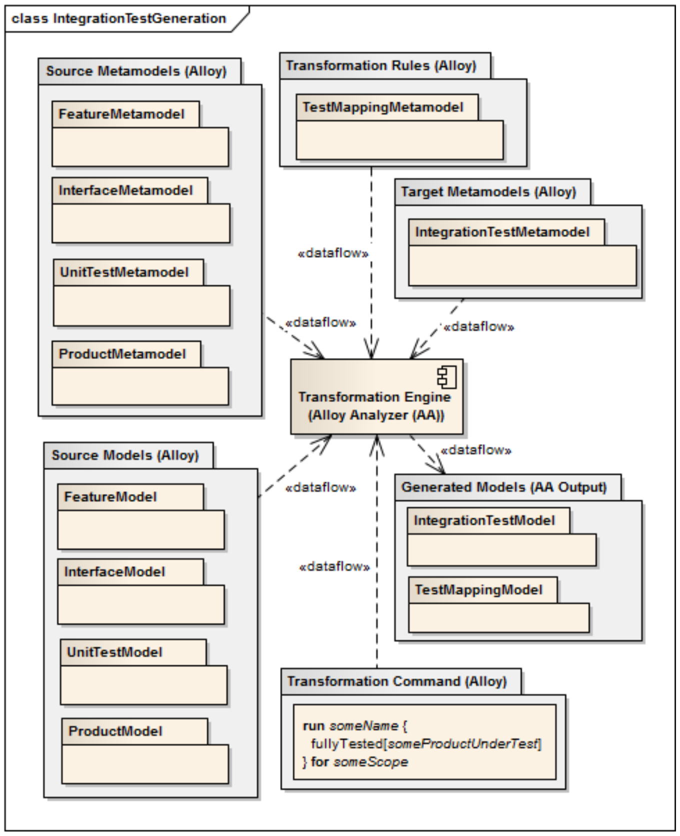
The *Product Level* contains specific definitions related with actual products: candidate and certified products (indicating their categories, supported features and possible message specializations); integration test scenarios used for the certification of candidate products, possibly involving other already certified products; and mappings (composition and instantiation) between integration tests and unit tests. Hence, inside this level we propose three metamodels, `ProductMetamodel`, `IntegrationTestMetamodel`, and `TestMappingMetamodel`. All the metamodels are presented in section III by means of UML class diagrams and associated constraints. The constraints and data types are written in Alloy to support the automated analysis and validation of the metamodels with Alloy Analyzer, as well as the automatic derivation of integration test models.

Level M1 contains models that are instances of the metamodels above, also divided in two different levels. The Domain

Level contains `FeatureModel`, `InterfaceModel` and `UnitTestModel`. The Product Level contains `ProductModel`, `IntegrationTestModel`, and `TestMappingModel`. Example models for the AAL domain are presented in section IV in diagrammatic notation. Those models are also formally specified in Alloy, using singleton signatures that extend the metamodel signatures.

B. Test Generation Approach

Figure 2 shows a data flow view of the integration test generation process.



```
run someName {
  fullyTested[someProductUnderTest]
} for someScope
```

Fig. 2. Integration Test Generation as a Model Transformation Process

The outer packages and components represent the usual participants in a model transformation process: source metamodels, source models (conforming to the source metamodels), target metamodels, transformation rules (from source models to target models), a transformation engine, a transformation command, and generated models (conforming to the target metamodels). In our approach, integration tests for specific products are created by instantiating and composing generic unit tests previously defined for the domain features, according to a set of mappings and constraints formalized in the `TestMappingMetamodel`. Hence, this metamodel defines the transformation rules. Besides the `IntegrationTestModel`, the generation process also generates the corresponding `TestMappingModel` with the actual mappings. Alloy Analyzer plays the role of a transformation engine, thanks to its model finding capabilities. The transformation command is a model finding (`run`) command written in Alloy, specifying the test generation criteria and the search scope (maximum number of instances to explore of each signature). The relevant outputs (`IntegrationTestModel` and `TestMappingModel`) are displayed in different formats by Alloy Analyzer (tree, graph, XML), but can easily be converted to Alloy.

III. Metamodels

A. Feature Metamodel

In some ecosystems, such as in the AAL field, there is a wide variety of products with a great features variability. It is therefore very important to limit the scope of the certification of such product types within a ecosystem. For this, based on the formalism of feature models [13], we propose a metamodel that allows representing the variability of products and their features within an ecosystem.

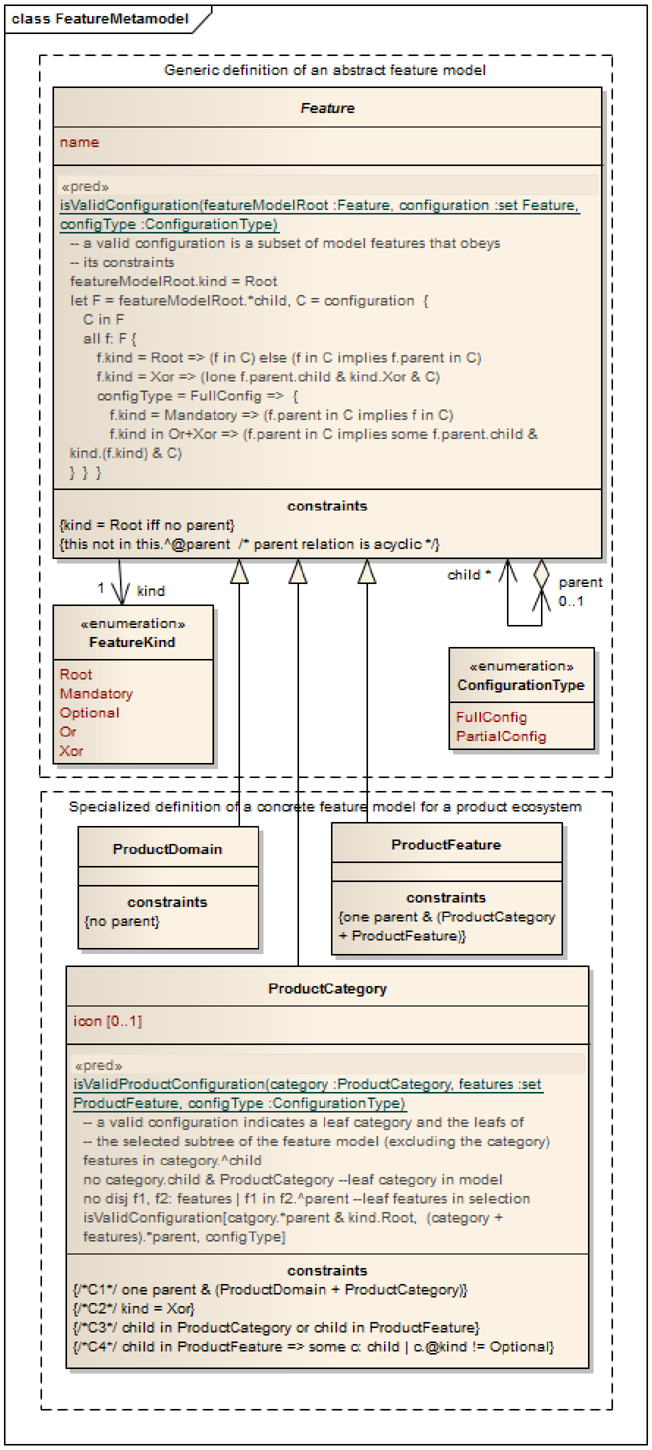
Firstly, in the upper layer of Figure 3, we define the structure and semantics of general purpose feature models. A feature model is a hierarchically arranged set of features where relationships between a parent (or compound) feature and its child features (or subfeatures) are categorized as:

- And – all subfeatures must be selected
- Xor – only one subfeature can be selected
- Or – one or more subfeature can be selected
- Mandatory – subfeatures that are required
- Optional – subfeatures that are optional

The semantics of these constructs is formalized by the auxiliary predicate `isValidConfiguration` in Figure 3, which checks if a configuration (a particular selection of features in the feature model) is valid. Some well formedeness rules of feature models are specified in Figure 3 by constraints (facts) written in Alloy.

An excerpt of the corresponding definition in Alloy is shown in Figure 4. A class in UML is represented by a signature in Alloy, with a number of fields (representing attributes or associations in UML) and constraints.

The lower layer of Figure 3 contains specialized definitions for our purpose. The feature model is partitioned intro three layers, with the root node representing a domain (`ProductDomain`) and child nodes representing allowed product (sub)categories within the domain (`ProductCategory`) and allowed product (sub)features within each category (`ProductFeature`). This layering is ensured by several constraints on the values of the `parent` and `child` fields. Categories represent different types of products within the domain, sharing similar characteristics (namely, the interfaces used or implemented). Each product is certified in a specific category, so categories are mutually exclusive (constraint `C2` in `ProductCategory`). The actual functionalities and properties that are the subject of testing and certification are the (sub)features defined within each (sub)category. Constraint `C4` in `ProductCategory` ensures that at least one (sub)feature must be selected within each (sub)category.



```
«pred»
isValidConfiguration(featureModelRoot :Feature, configuration :set Feature,
configType :ConfigurationType)
  -- a valid configuration is a subset of model features that obeys
  -- its constraints
  featureModelRoot.kind = Root
  let F = featureModelRoot.*child, C = configuration  {
    C in F
    all f: F {
      f.kind = Root => (f in C) else (f in C implies f.parent in C)
      f.kind = Xor => (lone f.parent.child & kind.Xor & C)
      configType = FullConfig =>  {
        f.kind = Mandatory => (f.parent in C implies f in C)
        f.kind in Or+Xor => (f.parent in C implies some f.parent.child &
  kind.(f.kind) & C)
  } } }

constraints
{kind = Root iff no parent}
{this not in this.^@parent  /* parent relation is acyclic */}
```

```
ProductDomain
constraints
{no parent}
```

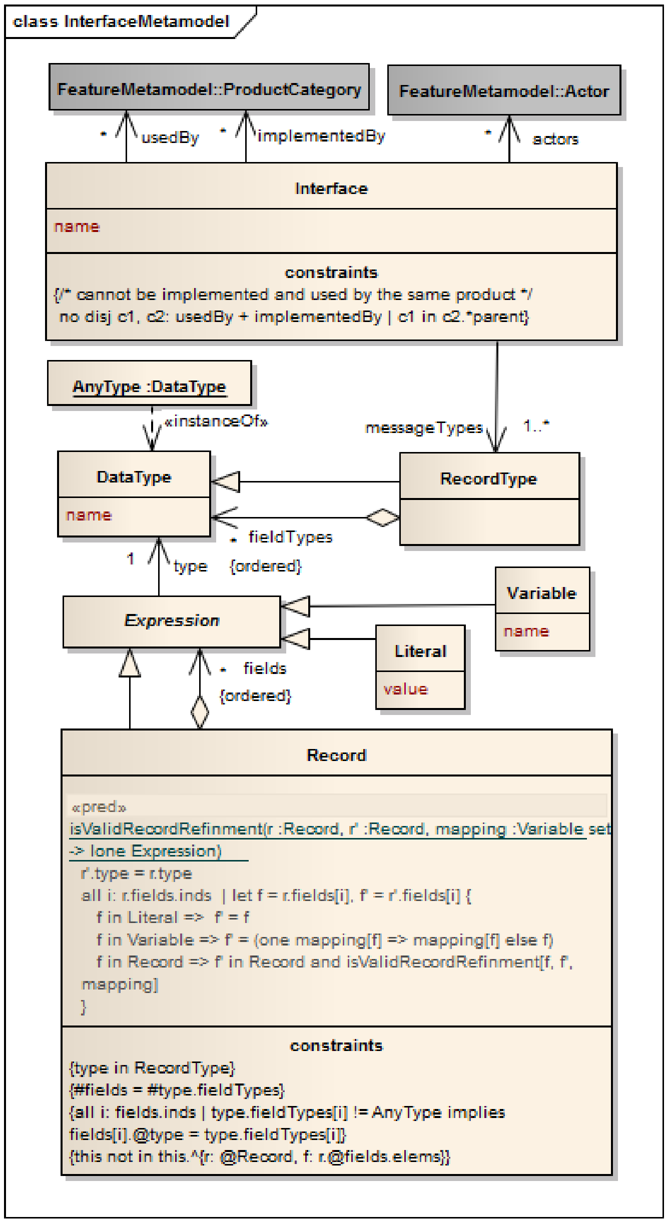
```
ProductFeature
constraints
{one parent & (ProductCategory
+ ProductFeature)}
```

```
ProductCategory
«pred»
isValidProductConfiguration(category :ProductCategory, features :set
ProductFeature, configType :ConfigurationType)
  -- a valid configuration indicates a leaf category and the leafs of
  -- the selected subtree of the feature model (excluding the category)
  features in category.^child
  no category.child & ProductCategory --leaf category in model
  no disj f1, f2: features | f1 in f2.^parent --leaf features in selection
  isValidConfiguration[catgory.*parent & kind.Root,  (category +
  features).*parent, configType]

constraints
{/*C1*/ one parent & (ProductDomain + ProductCategory)}
{/*C2*/ kind = Xor}
{/*C3*/ child in ProductCategory or child in ProductFeature}
{/*C4*/ child in ProductFeature => some c: child | c.@kind != Optional}
```

Fig. 3. Feature Metamodel

```
enum FeatureKind {Root, Mandatory, Optional, Or, Xor}

abstract sig Feature {
    kind: FeatureKind,
    parent: lone Feature,
    child: set Feature
}{
    kind = Root iff no parent
    -- parent relation must be acyclic
    this not in this.^@parent
    -- consistency of bidirectional association (implicit in UML)
    @child = ~@parent
}
```

Fig. 4. Excerpt of the Feature Metamodel in Alloy



```
Interface
constraints
{/* cannot be implemented and used by the same product */
 no disj c1, c2: usedBy + implementedBy | c1 in c2.*parent}
```

```
Record
«pred»
isValidRecordRefinment(r :Record, r' :Record, mapping :Variable set
-> lone Expression)
  r'.type = r.type
  all i: r.fields.inds  | let f = r.fields[i], f' = r'.fields[i] {
    f in Literal =>  f' = f
    f in Variable => f' = (one mapping[f] => mapping[f] else f)
    f in Record => f' in Record and isValidRecordRefinment[f, f',
  mapping]
  }

constraints
{type in RecordType}
{#fields = #type.fieldTypes}
{all i: fields.inds | type.fieldTypes[i] != AnyType implies
fields[i].@type = type.fieldTypes[i]}
{this not in this.^{r: @Record, f: r.@fields.elems}}
```

Fig. 5. Interface Metamodel

B. Interface Metamodel

To ensure interoperability, the ecosystem products have to communicate by exchanging standardized messages via standardized interfaces. To that end, we propose the `InterfaceMetamodel` shown in Figure 5, containing the set of rules necessary to message specification for a particular domain. As represented in Figure 5, an `Interface` defines the types of messages that can be sent or received through it, by product categories that use or implement the interface, respectively. In the case of user interfaces, the associated actors are also represented in the metamodel.

A message body is a composite data structure (`Record`) containing a number of fields. Hence, a message type is defined by a `RecordType`. Each field may in turn hold another `Record`, a `Literal` or a `Variable`. Variables are used in message specifications (namely in test specifications) to represent placeholders for more concrete expressions to be defined later by a refinement process, according to the constraints set by the predicate `isValidRecordRefinment`.

C. Unit Test Metamodel

Our proposal is based on the principle that the features of the Feature Model (see subsection III-A) have one or more associated unit tests; these tests allow the verification if a product implements this feature according to the ecosystem rules. To describe the unit tests we propose the Unit Test Metamodel shown in Figure 6.

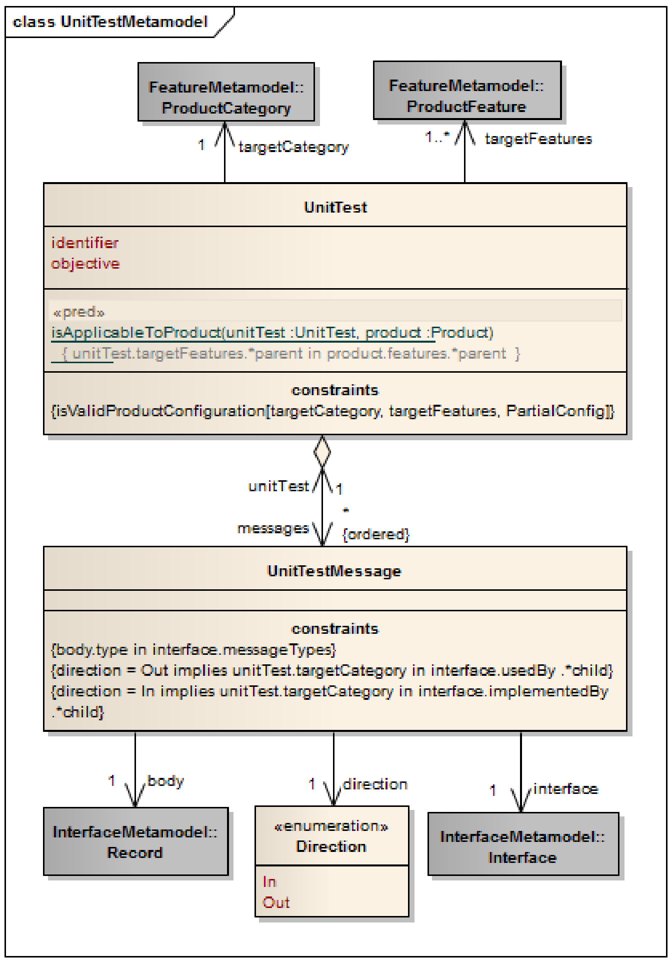

class UnitTestMetamodel
FeatureMetamodel::
ProductCategory
FeatureMetamodel::
ProductFeature
1
targetCategory
1..*
targetFeatures
UnitTest
identifier
objective
«pred»
isApplicableToProduct(unitTest :UnitTest, product :Product)
{ unitTest.targetFeatures.*parent in product.features.*parent }
constraints
{isValidProductConfiguration[targetCategory, targetFeatures, PartialConfig]}
unitTest
1
messages
*
{ordered}
UnitTestMessage
constraints
{body.type in interface.messageTypes}
{direction = Out implies unitTest.targetCategory in interface.usedBy .*child}
{direction = In implies unitTest.targetCategory in interface.implementedBy .*child}
1
body
1
direction
1
interface
InterfaceMetamodel::
Record
«enumeration»
Direction
In
Out
InterfaceMetamodel::
Interface

Fig. 6. Unit Test Metamodel

A unit test specifies a required behavior related with a target product category and a target feature or group of interdependent features (within a specific domain), as observed through the product interfaces (without knowledge of internal state). A unit test refers to a sequence of messages sent (output) or received (input) by a product that matches the target of the unit test (in the sense formalized by the predicate `isApplicableToProduct` in Figure 6), through interfaces it uses or implements, respectively. At this level, messages are usually specified in a generic way, using variables for the message fields. The metamodel includes some self-explanatory consistency constraints.

D. Product Metamodel

The Product Metamodel proposed (see Figure 7) allows the description of concrete products, already certified or candidate for certification. In this context a product belongs to a domain and a specific (sub)category and can have multiple features. The product also has a name and a flag indicating if it is a certified product or not. For each product, it is possible to restrict the admissible values for message parameters' data types (`productSpecializations`).

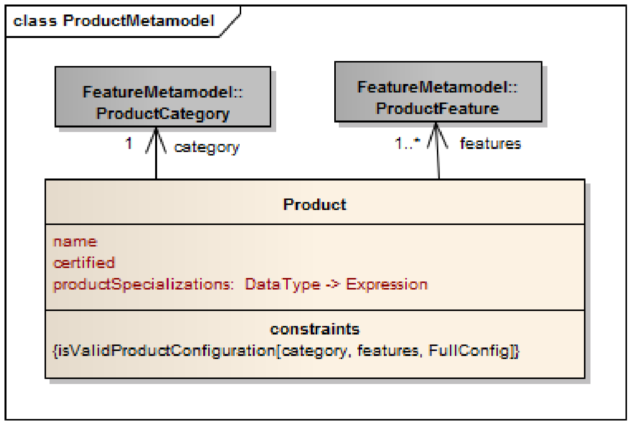


Fig. 7. Product Metamodel

E. Integration Test Metamodel

To obtain certification, candidate products are submitted to end-to-end integration tests in scenarios that may include other products (already certified), in order to ensure that they are able to communicate according to the ecosystem specifications.

To describe integration tests, we propose the metamodel shown in Figure 8. An `IntegrationTest` refers to a sequence of messages exchanged between a set of participants, including the product under test (not yet certified), zero or more products already certified, and actors. Each `Message` flows from a source product or actor to a destination product or actor through a well defined interface. The body of a message is a composite data structure, i.e., a record. The metamodel includes some self-explanatory consistency constraints.

The actual mechanisms of test execution are not specified in the models, but it is assumed that the actors' behavior (send messages to the system and monitor and check messages

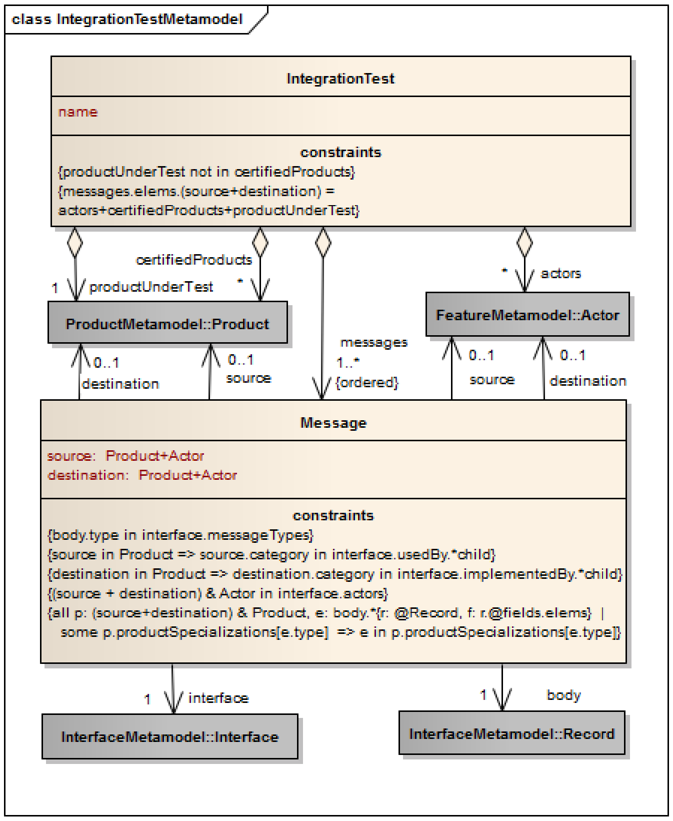

class IntegrationTestMetamodel
IntegrationTest
name
constraints
{productUnderTest not in certifiedProducts}
{messages.elems.(source+destination) =
actors+certifiedProducts+productUnderTest}
certifiedProducts
1
productUnderTest
*
*
actors
ProductMetamodel::Product
FeatureMetamodel::Actor
messages
1..*
{ordered}
0..1
destination
0..1
source
0..1
source
0..1
destination
Message
source: Product+Actor
destination: Product+Actor
constraints
{body.type in interface.messageTypes}
{source in Product => source.category in interface.usedBy.*child}
{destination in Product => destination.category in interface.implementedBy.*child}
{(source + destination) & Actor in interface.actors}
{all p: (source+destination) & Product, e: body.*{r: @Record, f: r.@fields.elems} |
some p.productSpecializations[e.type] => e in p.productSpecializations[e.type]}
1
interface
1
body
InterfaceMetamodel::Interface
InterfaceMetamodel::Record

Fig. 8. Integration Test Metamodel

received from the system) will be simulated by a tester or a test script. Messages exchanged between products in the system may or may not be monitored and checked. In case of variables used in the specification of message bodies (as placeholders for actual values), it is assumed that each variable must take the same value in all occurrences of the variable.

F. Test Mapping Metamodel

Integration tests for concrete products are generated by instantiating and composing generic unit tests associated with features of the feature model (supported by the products under test). The instantiation and composition are defined by three mappings: a mandatory mapping from unit tests (targeting a feature or group of features) to actual target products in the integration test (`targetMapping`); a mandatory mapping from unit test messages to integration test messages (`messageMapping`); and an optional mapping from variables used in the specification of the unit test message fields to more specific expressions (`variableMapping`). To ensure proper composition and instantiation, several consistency constraints apply for these mappings, as described in Figure 9. Constraints C1 to C4 restrict the domains and ranges of the three mappings. Constraints C7 and C8 check the consistency between different mappings. Of particular importance are constraints C10 and C11, because they control how the unit test messages are composed. Constraint C10 ensures that the ordering of messages of each unit test is preserved. Coupling

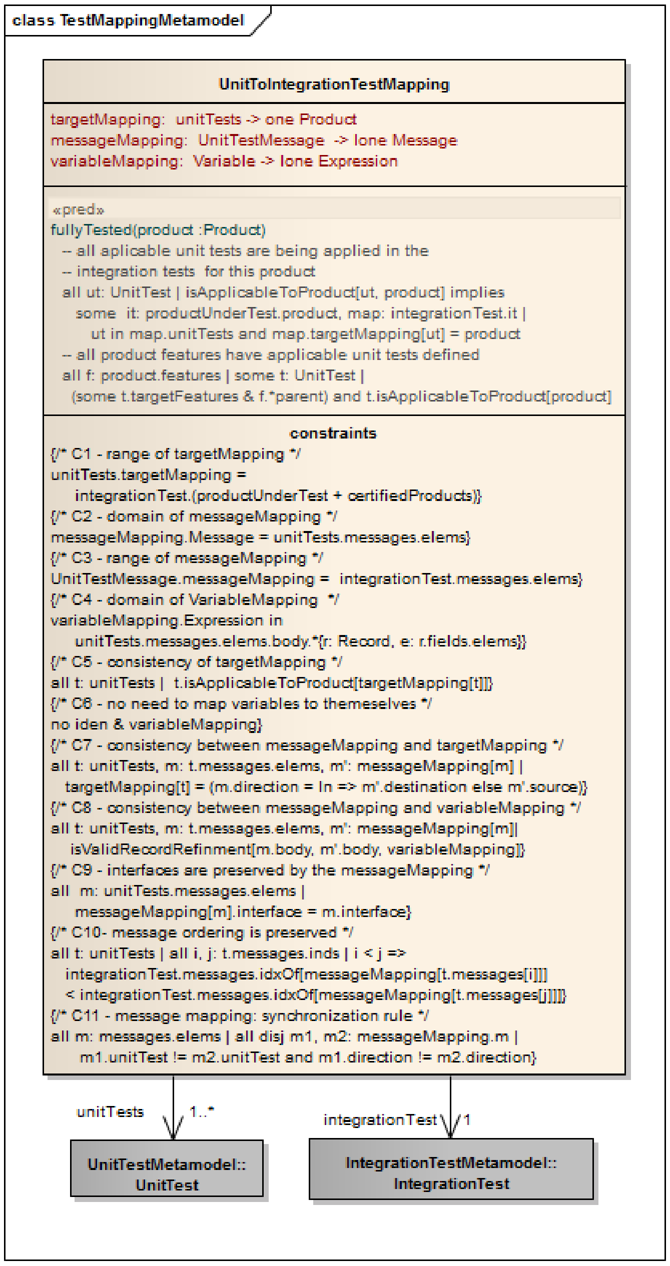

class TestMappingMetamodel
UnitToIntegrationTestMapping
targetMapping: unitTests -> one Product
messageMapping: UnitTestMessage -> lone Message
variableMapping: Variable -> lone Expression
«pred»
fullyTested(product :Product)
-- all aplicable unit tests are being applied in the
-- integration tests for this product
all ut: UnitTest | isApplicableToProduct[ut, product] implies
some it: productUnderTest.product, map: integrationTest.it |
ut in map.unitTests and map.targetMapping[ut] = product
-- all product features have applicable unit tests defined
all f: product.features | some t: UnitTest |
(some t.targetFeatures & f.*parent) and t.isApplicableToProduct[product]
constraints
{/* C1 - range of targetMapping */
unitTests.targetMapping =
integrationTest.(productUnderTest + certifiedProducts)}
{/* C2 - domain of messageMapping */
messageMapping.Message = unitTests.messages.elems}
{/* C3 - range of messageMapping */
UnitTestMessage.messageMapping = integrationTest.messages.elems}
{/* C4 - domain of VariableMapping */
variableMapping.Expression in
unitTests.messages.elems.body.*{r: Record, e: r.fields.elems}}
{/* C5 - consistency of targetMapping */
all t: unitTests | t.isApplicableToProduct[targetMapping[t]]}
{/* C6 - no need to map variables to themeselves */
no iden & variableMapping}
{/* C7 - consistency between messageMapping and targetMapping */
all t: unitTests, m: t.messages.elems, m': messageMapping[m] |
targetMapping[t] = (m.direction = In => m'.destination else m'.source)}
{/* C8 - consistency between messageMapping and variableMapping */
all t: unitTests, m: t.messages.elems, m': messageMapping[m]|
isValidRecordRefinment[m.body, m'.body, variableMapping]}
{/* C9 - interfaces are preserved by the messageMapping */
all m: unitTests.messages.elems |
messageMapping[m].interface = m.interface}
{/* C10- message ordering is preserved */
all t: unitTests | all i, j: t.messages.inds | i < j =>
integrationTest.messages.idxOf[messageMapping[t.messages[i]]]
< integrationTest.messages.idxOf[messageMapping[t.messages[j]]]}
{/* C11 - message mapping: synchronization rule */
all m: messages.elems | all disj m1, m2: messageMapping.m |
m1.unitTest != m2.unitTest and m1.direction != m2.direction}
unitTests
1..*
integrationTest
1
UnitTestMetamodel::
UnitTest
IntegrationTestMetamodel::
IntegrationTest

Fig. 9. Test Mapping Metamodel

between unit tests is achieved by mapping a pair of unit test messages - a message sent in the scope of one unit test and a message received in the scope of another unit test - to the same integration test message (constraint C11).

The `fullyTested` predicate is useful to define the goal of model finding commands, namely, for integration test generation purposes. It defines a test coverage criteria for any candidate product, by requiring two conditions: (i) all defined unit tests that are applicable for the product at hand (i.e., that refer to features supported by the product at hand) should be applied to that product in integration tests; (ii) there exist at least one unit test defined for each feature supported by the product at hand. Test minimization is achieved by controlling

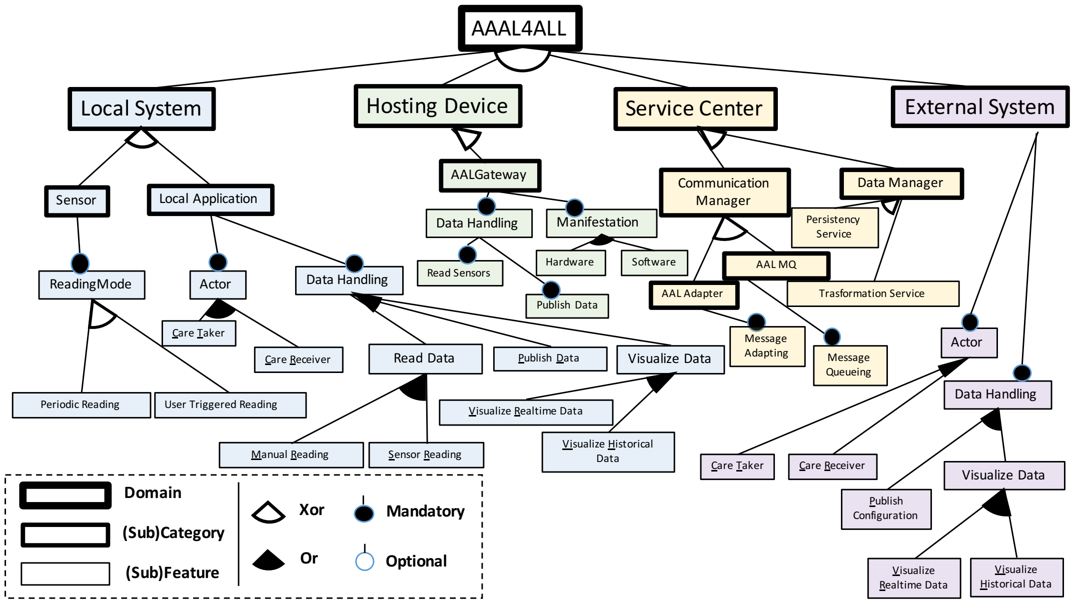


Fig. 10. Feature Model of the AAL4ALL Ecosystem

the scope (maximum number of instances to generate of each signature) of the model finding commands. The latter condition implicitly defines a necessary coverage criteria for unit tests themselves.

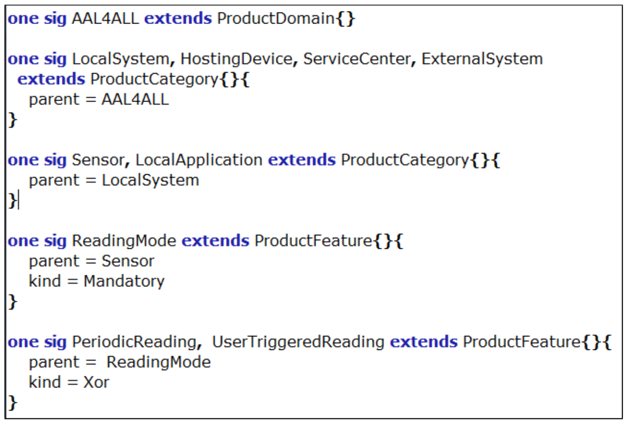
Thanks to the model finding capabilities of Alloy Analyzer, the consistency constraints defined in the `TestMappingMetamodel` are useful in three different ways: given user-defined unit test models, they can be used to generate integration test models (and associated test mapping models), as explained in Section II and illustrated in the next section; given user-defined unit test models and integration test models, they can be used to generate corresponding test mapping models and hence check the consistency between unit and integration test models (in case test mapping models cannot be generated, the unit and integration test models are inconsistent); given user-defined unit test models, integration test models, and test mapping models, they can be used to check the consistency of the overall specification.

IV. Example Models for the AAL Domain and Concrete Notations

A. *Feature Model*

The certification process defined in the AAL4ALL project comprises four main categories of products for which candidate products could be certified. Within some of these main categories it were defined product subcategories. Taking into account the rules described in the Feature Metamodel proposed in section III-A the feature model for the AAL4ALL domain containing the categories, subcategories and features can be observed in Figure 10.

The `FeatureMetamodel` proposed in section III-A defines the abstract syntax and semantics of a DSML for modeling the categories and features of products within a domain; such metamodel is formally specified in Alloy and is documented visually in UML. Regarding the concrete syntax (concrete notation) for the DSML, we propose to use a 'front-end' visual notation based on general purpose feature models, as illustrated in Figure 10, and a corresponding 'back-end' formal notation in Alloy. It will the subject of future work to provide tool support for the construction of the visual model and translation to the formal notation. An excerpt of the Alloy formal specification for this example is shown in Figure 11. Each model element is defined in Alloy as a singleton signature that extends the corresponding metamodel signature, with appropriate constraints on the field values. Constraints need not be defined for fields that can be derived from other information (such as the `kind` of product categories, because it is already constrained to `Xor` in the metamodel).



```
one sig AAL4ALL extends ProductDomain{}

one sig LocalSystem, HostingDevice, ServiceCenter, ExternalSystem
 extends ProductCategory{}{
   parent = AAL4ALL
}

one sig Sensor, LocalApplication extends ProductCategory{}{
   parent = LocalSystem
}

one sig ReadingMode extends ProductFeature{}{
   parent = Sensor
   kind = Mandatory
}

one sig PeriodicReading,  UserTriggeredReading extends ProductFeature{}{
   parent =  ReadingMode
   kind = Xor
}
```

Fig. 11. Excerpt of the AAL4ALL Feature Model in Alloy

B. Interface Model and Unit Test Model

The categorization process previously described allowed the definition of generic unit tests associated with the product categories and features, independently of actual products. These unit tests are later instantiated for the candidate products.

Some of the unit tests defined for some of the categories and features shown in Figure 10 are presented in Figure 12. In this case, we propose to use a concrete 'front-end' notation based on UML communication diagrams. Each box represents a product category with an icon and a name, as indicated by the stereotype ≪`ProductCategory`≫. The target features for the unit test being represented are indicated by a property string, such as `{PeriodicReading}`. The used and implemented interfaces relevant for the unit test at hand are represented using the usual UML notation. The sequence of messages sent or received through such interfaces are represented using the usual UML notation. Variables are normally used for the message fields.

We omit the presentation of the underlying interface model because it is not essential to understand the unit test model.

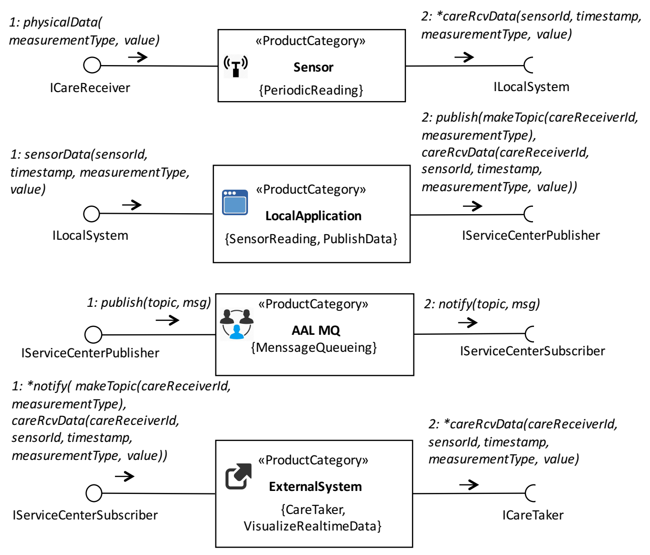
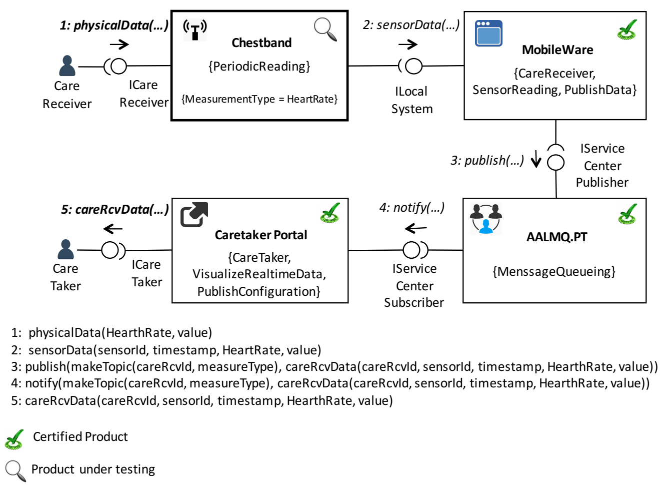

1: physicalData(measurementType, value)
ICareReceiver
«ProductCategory»
Sensor
{PeriodicReading}
2: *careRcvData(sensorId, timestamp, measurementType, value)
ILocalSystem
1: sensorData(sensorId, timestamp, measurementType, value)
ILocalSystem
«ProductCategory»
LocalApplication
{SensorReading, PublishData}
2: publish(makeTopic(careReceiverId, measurementType), careRcvData(careReceiverId, sensorId, timestamp, measurementType, value))
IServiceCenterPublisher
1: publish(topic, msg)
IServiceCenterPublisher
«ProductCategory»
AAL MQ
{MenssageQueueing}
2: notify(topic, msg)
IServiceCenterSubscriber
1: *notify(makeTopic(careReceiverId, measurementType), careRcvData(careReceiverId, sensorId, timestamp, measurementType, value))
IServiceCenterSubscriber
«ProductCategory»
ExternalSystem
{CareTaker, VisualizeRealtimeData}
2: *careRcvData(careReceiverId, sensorId, timestamp, measurementType, value)
ICareTaker

Fig. 12. Example of Unit Test Model for some Categories and Features

C. Product Model and Generated Integration Test Model

Suppose that a manufacturer is interested in getting AAL4ALL certification for a new sensor (Chestband). To obtain certification, the sensor will be submitted to a series of integration tests in end-to-end scenarios that must includes other products (already certified), in order to ensure that it is able to communicate according to the message specification defined by the ecosystem. For this specific example, assume that there are available and already certified, among other products, a Local Application (MobileWare), a AALMQ application (AALMQ.PT), and an External System (CaretakerPortal), as described by the boxes in Figure 13. Each box models a concrete product of a specific category



1: physicalData(HearthRate, value)
2: sensorData(sensorId, timestamp, HeartRate, value)
3: publish(makeTopic(careRcvId, measureType), careRcvData(careRcvId, sensorId, timestamp, HearthRate, value))
4: notify(makeTopic(careRcvId, measureType), careRcvData(careRcvId, sensorId, timestamp, HearthRate, value))
5: careRcvData(careRcvId, sensorId, timestamp, HearthRate, value)

Certified Product

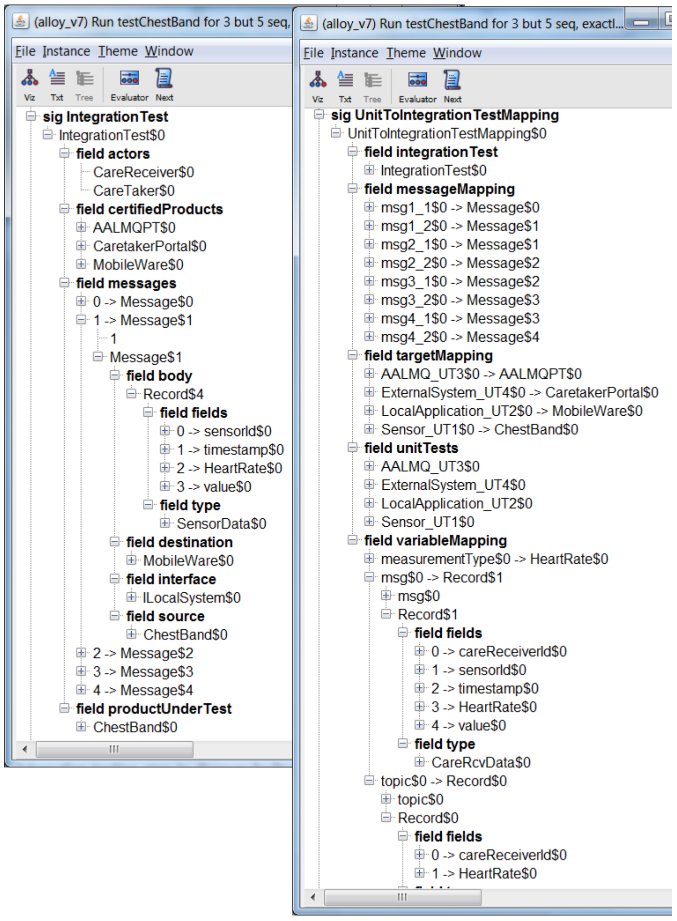
Product under testing

Fig. 13. Example of Integration Test Model Generated Automatically

(indicated by the icon), with a specific name, set of features, product specialization, and status (certified or candidate). In this example, the Chestband sensor has the product specialization `MeasurementType = HeartRate`, where `MeasurementType` is a `DataType` and `HeartRate` is a literal, meaning that the Chestband measures heart rate values.

From the description of the existent products in Alloy (i.e., from the product model), an integration test model can be automatically generated with Alloy Analyzer and an appropriate model finding command (`run { fullyTested[Chestband] } for` *`someSearchScope`*). An excerpt of the output obtained with Alloy Analyzer, in tree view mode, showing parts of the `IntegrationTest` instance generated, is displayed in Figure 14 (left). A more user friendly representation of the integration test model generated is shown in Figure 13. It will be the subject of future work to generate the friendly representation from the representation obtained with Alloy Analyzer. After fine tunning the search scope (maximum or exact number of instances to explore for each signature), the execution time of the model finding command was approximately 64 seconds in a computer with an Intel(R) Core(TM) i5-4210U CPU @ 1.7 GHz - 2.4 GHz and 8 GB RAM, running the 64 bits Windows 7 operating system. It will be the subject of future work to automatically tune the search bounds; to cope with the scalability issues inherent to Alloy Analyzer, more specialized search algorithms and engines may also be needed to handle very complex systems.

Another excerpt of the output obtained with Alloy Analyzer, in tree view mode, showing parts of the `UnitToIntegrationTestMapping` instance generated, is displayed in Figure 14 (right). Of particular relevance are the mappings of some variables used in the specification of unit test messages (`measurementType`, `msg` and `topic`) to more specialized expressions. It can also be seen that, except for user interaction messages, pairs of unit test messages are mapped to the same integration test message, thus coupling


(alloy_v7) Run testChestBand for 3 but 5 seq,
File Instance Theme Window
Viz Txt Tree Evaluator Next
(alloy_v7) Run testChestBand for 3 but 5 seq, exactl...
File Instance Theme Window
Viz Txt Tree Evaluator Next

```
sig IntegrationTest
  IntegrationTest$0
    field actors
      CareReceiver$0
      CareTaker$0
    field certifiedProducts
      AALMQPT$0
      CaretakerPortal$0
      MobileWare$0
    field messages
      0 -> Message$0
      1 -> Message$1
        1
        Message$1
          field body
            Record$4
              field fields
                0 -> sensorId$0
                1 -> timestamp$0
                2 -> HeartRate$0
                3 -> value$0
              field type
                SensorData$0
          field destination
            MobileWare$0
          field interface
            ILocalSystem$0
          field source
            ChestBand$0
      2 -> Message$2
      3 -> Message$3
      4 -> Message$4
    field productUnderTest
      ChestBand$0
```

```
sig UnitToIntegrationTestMapping
  UnitToIntegrationTestMapping$0
    field integrationTest
      IntegrationTest$0
    field messageMapping
      msg1_1$0 -> Message$0
      msg1_2$0 -> Message$1
      msg2_1$0 -> Message$1
      msg2_2$0 -> Message$2
      msg3_1$0 -> Message$2
      msg3_2$0 -> Message$3
      msg4_1$0 -> Message$3
      msg4_2$0 -> Message$4
    field targetMapping
      AALMQ_UT3$0 -> AALMQPT$0
      ExternalSystem_UT4$0 -> CaretakerPortal$0
      LocalApplication_UT2$0 -> MobileWare$0
      Sensor_UT1$0 -> ChestBand$0
    field unitTests
      AALMQ_UT3$0
      ExternalSystem_UT4$0
      LocalApplication_UT2$0
      Sensor_UT1$0
    field variableMapping
      measurementType$0 -> HeartRate$0
      msg$0 -> Record$1
        msg$0
        Record$1
          field fields
            0 -> careReceiverId$0
            1 -> sensorId$0
            2 -> timestamp$0
            3 -> HeartRate$0
            4 -> value$0
          field type
            CareRcvData$0
      topic$0 -> Record$0
        topic$0
        Record$0
          field fields
            0 -> careReceiverId$0
            1 -> HeartRate$0
```

Fig. 14. Excerpt of Integration Test and Test Mapping Instances Generated with Alloy Analyzer

the unit tests together as explained previously.

The complete Alloy specification of the metamodels and models for our example and some Alloy Themes for better visualization in Alloy Analyzer are available at https://goo.gl/Y9Vzd7.

V. Related Work

Integration testing aims to discover faults that are due to incorrect interactions between different software based products. Even if each product is correct and delivers the specified functionality, when integrated into a larger system, the interactions between products can lead to incorrect results, for example because different products interpret data processed by other products incorrectly, or do not follow the same interaction protocol [14]. The purpose of integration testing is to find such faults that are difficult if not impossible to find when testing products independently. However, and despite unit test case generation been explored extensively in the literature, there is still little work on the generation of integration test cases [15]. One approach to generate integration test cases from simple unit test cases can be found in [15]. The approach requires the system source code and a set of test cases as input, and works in three main phases: (1) identify class dependencies within the system in the form of an object relation diagram, (2) compute the data flow information within the input test cases, and use this information to segment the test cases into useful blocks (initialization and execution), and (3) generate new, more complex test cases from the blocks extracted from unit test cases using the class dependence and data flow information. Although there are some similarities with our work (unit tests composition for the generation of integration tests), there are significant differences: they work at the level of objects and source code, whilst we work at the level of products (or components) and models; in our approach, we have not only the composition but also the instantiation of generic unit tests.

Testing for feature models and SPLs is widely studied in literature. Surveys [16] and [17] present a large selection of papers about feature models testing and how tests are generated respectively. However, the problem of test selection is considered still an open problem [18]. These works are complementary to our work, providing strategies that might be used in our work, namely for selecting and minimizing the tests needed for adequately testing products supporting multiple interrelated features.

Regarding the usage of metamodels for describing test related concepts, the UML Testing Profile (UTP) [19] is a prominent example. Although some concepts in our approach can be mapped to UTP concepts (such as SUT, TestComponent and TestCase), the UTP specifies more detailed concepts (at a lower level of abstraction) relavant for test implementation (such as Verdict, Arbiter and TestLog).

With regard to the use of Alloy for software testing there are some approaches [20] where Alloy is used to enable writing specifications at an intuitive abstract level (such as method pre/post conditions and class invariants), and the automatic generation of test cases (including test inputs, and expected outputs) by constraint solving with Alloy Analyzer. There exist also some approaches where Alloy is used for composing UML class diagrams [21] or UML sequence diagrams [22], using metamodels and composition constraints written in Alloy. There are some similarities between the latter work and ours, because both communication diagrams (used in our work as a basis for representing unit and integration tests) and sequence diagrams represent interactions. However, we don't have complete interactions in the unit tests and we have the additional problem of instantiation, besides composition.

In the area of health, models are often used in standardization initiatives. For example, Eggebraaten [23] proposed a health-care data model based on the HL7 Reference Information Model [24] to ensure the integration of medical information from various sources, and enable the health-care industry to perform analytical studies that can help discover new treatments and improve patient care. In our work these medical data structures models could be harnessed and used as mandatory message structures exchanged between the various components of the ecosystem. Other similar work for ensuring the proper integration between medical devices is based on ISO/IEEE 11073 standard [25]. In that standard, an object oriented data model, the domain information model (DIM), defined in ISO 1173-10201, is used to specify objects, attributes, attribute groups, event reports, and communication services, that may be used to communicate device data and to configure medical devices and functionalities. The standardized nomenclature (ISO 11073-10101) comprises a set of numeric codes that identify every item that is communicated between systems. Related to the general DIM, there exist device specializations for several medical devices, which provide guidelines for how the DIM should be constrained for application to specific devices. These standards were adopted by the Continua Health Alliance [26], a industry consortium that aims at enabling end-to-end system interoperability in the personal telehealth market by leveraging and integrating existing standards for all layers of the communications stack and for all parts of the overall system ranging from the patients-end to the service providers-end.

VI. Conclusions and Future Directions

In this article we presented a set of metamodels and associated consistency rules for the description of certification requirements and products within a specific domain, so that integration tests can be automatically generated for the certification of candidate products. For a better understanding of the models, it were presented application examples based on the nationwide AAL4ALL project. The proposed (meta)model-based approach, allows systematizing, partially automate and increase the assurance on testing and certification activities in digital ecosystems.

As future work we intend to develop a tool set, applicable in different application domains, to facilitate: (i) the construction of the relevant visual models, (ii) their automatic translation to the formal notation (Alloy), (iii) the automatic determination of search bounds to be used in the derivation of integration tests with Alloy Analyzer (or more specialized algorithms and engines to cope with scalability issues), (iv) the automatic translation of the generated integration test models to the visual notation, and (v) the integration with frameworks for the test concretization and execution in distributed and heterogeneous environments.

Acknowledgment

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013.

References

[1] C. Hill, R. Grant, and I. Yeung, "Ambient assisted living technology," *An interactive qualifying project report submitted to the Faculty of Worcester Polytechnic Institute*, 2013.

[2] tcare. (2015, November) tcare - Conhecimento e saÃžde SA. [Online]. Available: http://www.tcare.pt/

[3] AAL4ALL. (2015) AAL4ALL - Official Website. [Online]. Available: http://www.aal4all.org/

[4] J. P. Faria, B. Lima, T. B. Sousa, and A. Martins, "A testing and certification methodology for an open ambient-assisted living ecosystem," *International Journal of E-Health and Medical Communications*, vol. 5, no. 4, pp. 90–107, 2014.

[5] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 0025–31, 2006.

[6] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed. Springer Berlin Heidelberg, 2006, vol. 3844, pp. 128–138. [Online]. Available: http://dx.doi.org/10.1007/11663430_14

[7] L. Gammaitoni and P. Kelsen, "F-alloy: An alloy based model transformation language," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, D. Kolovos and M. Wimmer, Eds. Springer International Publishing, 2015, vol. 9152, pp. 166–180. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21155-8_13

[8] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[9] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[10] OMG, "OMG Unified Modeling Language TM (OMG UML) Superstructure," Object Management Group, Tech. Rep., 2011.

[11] C. Atkinson and T. Kühne, "Model-driven development: a metamodeling foundation," *Software, IEEE*, vol. 20, no. 5, pp. 36–41, 2003.

[12] OMG, "Meta Object Facility (MOF) Specification," Object Management Group, Tech. Rep., 2005.

[13] D. Batory, *Feature models, grammars, and propositional formulas*. Springer, 2005.

[14] M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[15] M. Pezze, K. Rubinov, and J. Wuttke, "Generating effective integration test cases from unit ones," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 11–20.

[16] P. A. D. M. S. Neto, I. do Carmo Machado, J. D. McGregor, E. S. De Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[17] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012.

[18] J. Lee, S. Kang, and D. Lee, "A survey on software product line testing," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 31–40.

[19] OMG, "UML Testing Profile (UTP), v1.2," Object Management Group, Tech. Rep., 2013.

[20] S. A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid, "Testera: A tool for testing java programs using alloy specifications," in *Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 2011, pp. 608–611.

[21] J. Rubin, M. Chechik, and S. M. Easterbrook, "Declarative approach for model composition," in *Proceedings of the 2008 international workshop on Models in software engineering*. ACM, 2008, pp. 7–14.

[22] M. Alwanain, B. Bordbar, and J. K. Bowles, "Automated composition of sequence diagrams via alloy," in *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*. IEEE, 2014, pp. 384–391.

[23] T. J. Eggebraaten, J. W. Tenner, and J. C. Dubbels, "A health-care data model based on the hl7 reference information model," *IBM Systems Journal*, vol. 46, no. 1, pp. 5–18, 2007.

[24] C. N. M. Gunther SCHADOW, "The hl7 reference information model under scrutiny," in *Ubiquity: technologies for better health in aging societies: proceedings of MIE2006*, vol. 124. IOS Press, 2006, p. 151.

[25] L. Schmitt, T. Falck, F. Wartena, and D. Simons, "Novel iso/ieee 11073 standards for personal telehealth systems interoperability," in *High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007. HCMDSS-MDPnP. Joint Workshop on*, June 2007, pp. 146–148.

[26] Continua. (2015, Dec.) Continua - Official Website. [Online]. Available: http://www.continuaalliance.org/