Augmenting Field Data for Testing Systems Subject to Incremental Requirements Changes

DANIEL DI NARDO, University of Luxembourg
FABRIZIO PASTORE, University of Luxembourg
LIONEL BRIAND, University of Luxembourg

When testing data processing systems, software engineers often use real world data to perform system level testing. However, in the presence of new data requirements software engineers may no longer benefit from having real world data with which to perform testing. Typically, new test inputs complying with the new requirements have to be manually written.

We propose an automated model-based approach that combines data modelling and constraint solving to modify existing field data to generate test inputs for testing new data requirements. The approach scales in the presence of complex and structured data, thanks to both the reuse of existing field data and the adoption of an innovative input generation algorithm based on slicing the model into parts.

We validated the scalability and effectiveness of the proposed approach using an industrial case study. The empirical study shows that the approach scales in the presence of large amounts of structured and complex data. The approach can produce, within a reasonable time, test input data that is over ten times larger in size than the data generated with constraint solving only. We also demonstrate that the generated test inputs achieve more code coverage than the test cases implemented by experienced software engineers.

CCS Concepts: •**Software and its engineering** → **Software verification and validation;**

Additional Key Words and Phrases: System Testing, Data Processing Systems

ACM Reference Format:
Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand, 20XX. Augmenting Field Data for Testing Systems Subject to Incremental Requirements Changes. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 00 (2016), 39 pages.
DOI: 0000001.0000001

1. INTRODUCTION

When testing data processing systems, software engineers often take advantage of the availability of a huge quantity of real world data to perform system-level testing. For example, when developing a web crawler, software engineers can rely upon existing web pages to verify the robustness of the system. However, in the presence of new requirements, where there is a need to deal with new data formats, software engineers may no longer have the benefit of having existing real world data with which to perform testing. Typically, new test inputs that comply with the new format would have to be written.

This situation is very common in industry, where requirements are continuously changing. For example, this paper was motivated by the needs of SES, a satellite operator. SES develops data acquisition systems for satellite transmissions. One such

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR 4082113), an SES grant, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).
Author's addresses: D. Di Nardo and F. Pastore and L. Briand, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, 29 Avenue J.F Kennedy, L-1855 Luxembourg, Luxembourg.

DOI: 0000001.0000001

system has been developed for the European Space Agency Sentinel series of satellites [ESA 2015]. The first of the Sentinels is already in orbit and more Sentinel satellites will be launched in the coming years. Real transmission data for the first of the Sentinel mission types is available for testing the data acquisition system. For other Sentinel mission types, real transmission data is not yet available. Additionally, during the development process it is not uncommon for the transmission data specifications to continue to change. Hence, an approach that supports the automatic generation of valid synthetic transmission data files is necessary to ensure that the data acquisition system can be thoroughly tested throughout development.

In practice, testing data processing systems involves the handcrafting of inputs, which requires the creation and editing of large and complex data structures saved in binary files. Furthermore, given available time and resources, test files should be as realistic as possible in terms of size and content, as large sizes will stress the system more and are more likely to reveal faults. The size and complexity of the test inputs makes this process error-prone and expensive, especially in the presence of changing requirements that force software engineers to modify or rewrite already defined complex test inputs.

Most existing approaches for the automatic generation of test inputs cannot be used because they require extensive specifications in the form of context free grammars, which cannot capture all the complex relationships between data fields. A few approaches that use extended grammars to capture such relationships exist (e.g. [Xiao et al. 2003]); however, these approaches are limited in the types of relationships they can express.

Constraint solvers that process constraints expressed using the OCL language [Ali et al. 2013], Alloy [Anastasakis et al. 2007], or constraint programming [Cabot et al. 2008] can be used to generate test inputs from scratch. However, existing approaches do not scale in the presence of numerous constraints and complex and highly structured data, as visible for Alloy in our empirical study. An additional limitation of these approaches is that they require software engineers to model all the constraints on input data, which may require a lot of time.

To limit the modelling effort, other approaches generate test inputs by mutating existing field data [Shan and Zhu 2009; Di Nardo et al. 2015a]. The technique described in [Di Nardo et al. 2015a], for example, generates test inputs by mutating available field data represented by a data model. This technique does not require exhaustive modelling of all the characteristics of the test inputs; however, it is not applicable when existing field data do not comply with new data requirements.

In this paper, we propose an approach that modifies existing field data to generate test inputs for testing new requirements. The approach combines data modelling and constraint solving. Models of both the original data format as well as the updated data format must be provided as inputs to the approach, along with field data complying with the original data model. The approach scales in the presence of complex and structured data, thanks to both the reuse of existing field data and the adoption of an innovative input generation algorithm based on slicing the model into parts; it reuses field data for parts unaffected by requirements changes, and then iteratively updates parts that are affected, by using data generated by means of constraint solving.

The approach proposed in this paper makes use of previous work in which we proposed a modelling methodology dedicated to modelling the structure and content of complex input/output data stores (e.g. files), and their relationships for systems where the complexity lies in these elements, such as data acquisition (DAQ) systems [Di Nardo et al. 2013].

Given that the solution proposed in this paper creates valid synthetic field data that complies with an updated data model, test engineers can proceed to use this

synthetic field data to test the updated target system. Although in our experiments we evaluated the effectiveness of the approach on a case study where test inputs for the system under test are bytestreams, the underlying data modelling methodology is generic, and enables the adoption of the proposed approach to test systems that process different data formats (e.g. structured text files or XML files). Additionally, the newly generated test inputs can be further processed by other automated testing techniques. For example, the mutation technique that we introduced in [Di Nardo et al. 2015a] can be used to automatically modify the synthetic field data generated to perform robustness testing. Also, the outputs that result from the execution of the software under test against the synthesised field data can be automatically evaluated by relying upon a technique that makes use of data constraints in the data model [Di Nardo et al. 2013].

The contributions of this paper are:

— An automated, model-based approach to modify field data to fit new data requirements for the purpose of testing data processing systems.
— A scalable test generation algorithm based on data slicing that allows for the incremental invoking of a constraint solver to generate new or modified parts of the updated field data.
— An industrial empirical study demonstrating (1) scalability in generating new field data and (2) coverage of new data requirements by generated field data in addition to a comparison with expert, manual testing.

The paper is structured as follows. Section 2 describes the data modelling approach adopted to capture the characteristics of the input data. Section 3 summarises the challenges of the research problem addressed in this paper. Section 4 overviews the approach. Sections 5 and 6 present the details of the core contributions of the paper: reuse of existing data and generation of missing or invalid data with constraint solving. Section 7 shows, by means of an example, how the algorithm proposed by this article correctly generates a complete solution. Section 8 discusses the empirical results obtained. Section 9 summarises related work. Section 10 concludes the paper.

2. BACKGROUND ON DATA MODELLING

To define data requirements, we rely upon a data modelling methodology described in [Di Nardo et al. 2013; Di Nardo et al. 2015a] that uses UML class diagrams to capture the structure of inputs and outputs, relies upon Object Constraint Language (OCL) [OMG 2015] expressions to define relationships between the inputs and outputs, and uses UML stereotypes and OCL expressions to capture a fault model.

To briefly present the methodology, we show how it can be applied to model Sentinel-1 mission transmission data processed by SES-DAQ, a DAQ system developed by SES. SES-DAQ processes bytestreams of transmitted satellite data. The Sentinel-1 mission is the first of several planned Sentinel missions. One Sentinel-1 satellite is already in orbit.

Fig. 1 shows a simplified example of a satellite transmission processed by SES-DAQ. Each transmission consists of a sequence of *Virtual Channel Data Units (VCDUs)* [CCSDS 2006]. Each *VCDU* contains a *Header* and a packet zone that contains a sequence of *Packets* [CCSDS 2003]. The *VCDUs* in a transmission may belong to different virtual channels; a unique virtual channel identifier (VCID) number identifies each virtual channel. *VCDUs* can be active (i.e. they transmit data) or idle (i.e. they do not transmit anything). A special VCID is used to transmit idle data. Fig. 1 shows a transmission with six *VCDUs*: four belonging to virtual channel 1; one belonging to virtual channel 2; and one belonging to virtual channel 0, which indicates idle data.

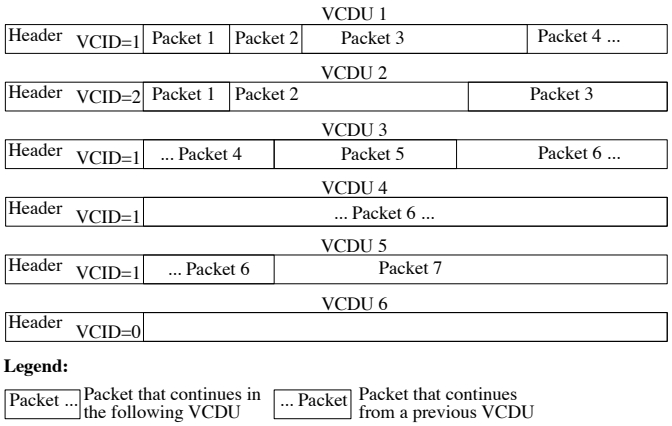

Fig. 1. A simplified example of the transmission data processed by the SES data acquisition system. The keyword *VCID* indicates the virtual channel each *Virtual Channel Data Unit (VCDU)* belongs to.

To create the data model for the Sentinel-1 mission, we first studied the SES-DAQ system using design and test documents and held several modelling sessions with the system testers and developers. The data model and constraints for the system were implemented by the first author of this paper, in an iterative manner, following the modelling methodology. Although the proposed approach has been used only in one pilot project, it is currently being disseminated in the organisation.

Fig. 2 shows how we model the Sentinel-1 input data. The model captures the structure of a transmission: we use UML classes to represent elements that contain multiple fields, while we use UML attributes to model elements that cannot be further decomposed. For example, it shows that each transmission consists of a sequence of *VCDUs*. Each *VCDU* begins with a *VcduHeader*, followed by a *PacketZone* that contains a sequence of *Packets* (if the packet zone is active). The *VCDUs* in a transmission may belong to different virtual channels.

Class attributes are used to represent the transmitted binary information (e.g. checksums, frame counters, or data). For example, attribute *sequenceCount* of class *Packet* is used to store information about the packet order.

Associations are used to represent containment relationships. In Fig. 2, the classes that model the *VCDU* and its *VcduHeader* are connected by an association. We use generalisations to indicate when a data field can have multiple different definitions. For example, each *Packet* has a *PacketHeader* whose content may vary according to the type of packet. Note that for Sentinel-1, the *PacketHeader* has two possible values: (1) the *SarPacketHeader* is used by packets containing data generated by the Synthetic Aperture Radar (SAR) instrument and (2) the *GpsrPacketHeader* is associated with packets containing Global Positioning System Receiver (GPSR) data.

Data models also capture the structure of configuration files. In the case of SES-DAQ, the structure of configuration files is captured by classes *Configuration* and *VcduConfig*. The configuration files specify how the SES-DAQ software should process data and also define what data values are valid for received transmissions. For example, the attribute *checkCrc* of class *Configuration* indicates whether or not the software should check for packet correctness by using cyclic redundancy check information. A *Configuration* also contains a collection of *VcduConfig* instances, one for each valid virtual channel. Class *VcduConfig* provides run-time information characterising the expected contents of a valid virtual channel, for example: a valid VCID value, *vcid*

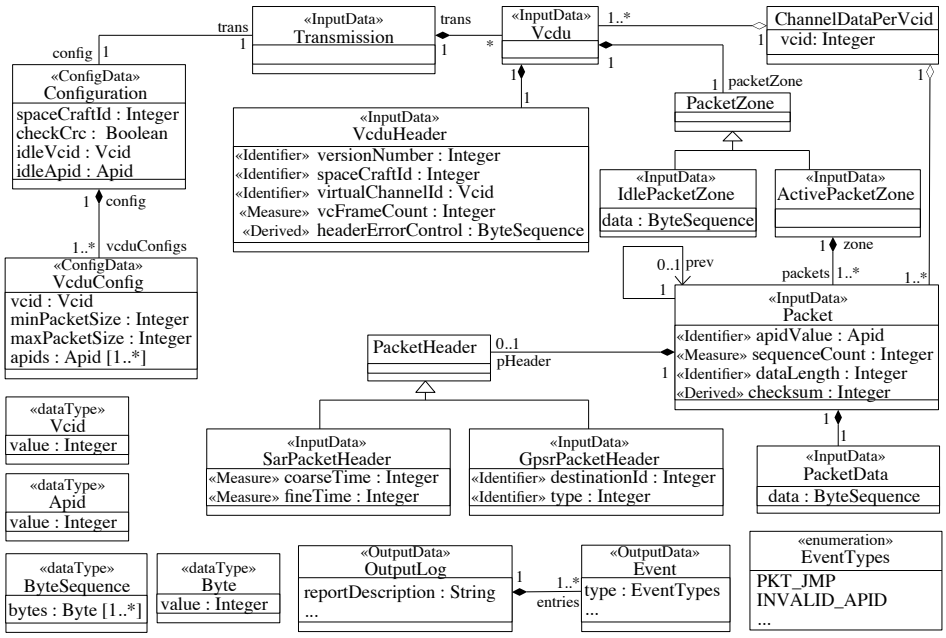

Fig. 2. Simplified data model for the SES-DAQ working with Sentinel-1 satellites.

```
1  context Packet inv:
2      ( self.apidValue.value=0 ) or
3      ( self.apidValue.value=1 and self.pHeader.oclIsTypeOf(SarPacketHeader) ) or
4      ( self.apidValue.value=2 and self.pHeader.oclIsTypeOf(GpsrPacketHeader) )
```

Fig. 3. Mapping of packet type numbers to specific *PacketHeader* subclasses.

in Fig. 2; and a collection of valid packet identifiers (specifically, application identifier (APID) values), *apids* in Fig. 2.

The outputs of SES-DAQ are modelled by classes *OutputLog* and *Event* that capture the event messages reported by the application in the output log file.

A data model also captures the characteristics of valid data by means of constraints written using the OCL language. Fig. 3 shows a simple input constraint specific to the Sentinel-1 data model that indicates that a *PacketHeader* is of type *SarPacketHeader* only if the APID value of the packet is equal to 1. Fig. 4 shows a more complex input constraint—one having collection operators; the constraint indicates that the *apidValue* for a given *Packet* must either be equal to (1) an active apid value, as specified by *VcduConfig* entries that indicate the valid *apids* for a given *vcid* value (Lines 3 and 4) or (2) the idle apid value, as specified by the *idleApid* entry of the *Configuration* (Line 6). For the SES-DAQ, there are 4 complex input constraints (having collection operators) and 17 simpler ones. Output constraints might be defined as well. An output constraint may, for example, indicate that an error event *PKT_JMP* should exist in the system output log file if the frame count of a VCDU is not greater by one than the frame count of the previous VCDU on the same virtual channel.

```
1  context ActivePacketZone inv:
2      self.packets→forAll(x : Packet |
3          self.vcdu.trans.config.vcduConfigs→select(y : VcduConfig |
4              y.vcid = self.vcdu.vcduHeader.virtualChannelId).apids→exists(z : Apid | z = x.apidValue)
5          or
6          (x.apidValue = self.vcdu.trans.config.idleApid)
7      )
```

Fig. 4. OCL constraint that indicates the allowed *apidValue* entry for a given *Packet* according to the configuration.

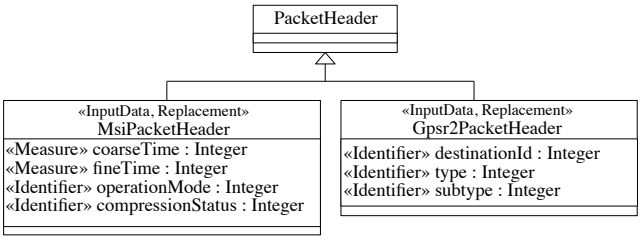

Fig. 5. Portion of the data model for SES-DAQ that handles new data requirements specific for Sentinel-2 satellites.

```
1  context Packet inv:
2      ( self.apidValue.value=0 ) or
3      ( self.apidValue.value=1 and self.pHeader.oclIsTypeOf(SarPacketHeader) ) or
4      ( self.apidValue.value=2 and self.pHeader.oclIsTypeOf(GpsrPacketHeader) ) or
5      ( (self.apidValue.value=3 or self.apidValue.value=4) and
6          self.pHeader.oclIsTypeOf(MsiPacketHeader) ) or
7      ( (self.apidValue.value=5 or self.apidValue.value=6) and
8          self.pHeader.oclIsTypeOf(Gpsr2PacketHeader) )
```

Fig. 6. New OCL constraint that replaces the one in Fig. 3. The constraint is updated to include the new packet header types.

3. RUNNING EXAMPLE

New data requirements potentially result in changes to both the data model and the contents of the configuration files of the system. A change to the data model corresponds to a modification of the class diagram or the OCL constraints, while changes in configuration files consist of changes in the values assigned to configuration parameters.

Fig. 5 shows a portion of the data model of SES-DAQ that has been updated to process data transmitted by Sentinel-2 mission satellites. In the case of Sentinel-2, a *PacketHeader* can be either of type *MsiPacketHeader* or *Gpsr2PacketHeader*. These two kinds of packet headers contain information that is different from the packet headers transmitted by Sentinel-1 satellites. If we compare, for example, the *GpsrPacketHeader* transmitted by Sentinel-1 and the *Gpsr2PacketHeader* transmitted by Sentinel-2 we notice that both provide information about the *destinationId*, and the *type* of the content being sent, while only the latter provides a *subtype* field that provides additional information characterising the content.

Data constraints might change as well. Fig. 6 shows an example for SES-DAQ where the OCL constraint in Fig. 3 has been modified by specifying the new mappings between the packet type and the two new *PacketHeaders*.

```
1  context VcduHeader inv:
2      self.virtualChannelId = self.vcdu.trans.config.idleVcid
3      or
4      self.vcdu.trans.config.vcduConfigs.vcid→exists(x | x = self.virtualChannelId)
```

Fig. 7. OCL constraint involving configuration parameters.

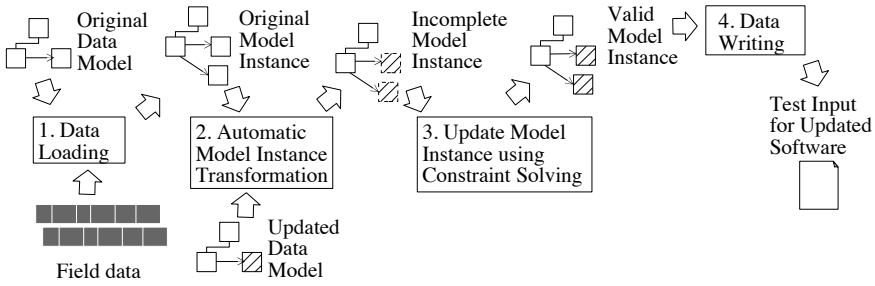

Fig. 8. Automatic generation of test inputs for new data requirements.

In addition to changes in the data model, new data requirements often imply changes in the configuration files used to run the software. Different software versions may require different configuration parameters, although changes in the content of the configuration files may not imply changes in the structure (or the related constraints) of the configuration classes captured by the data model (e.g. the configuration file for SES-DAQ has the same structure whether it is used to process Sentinel-1 or Sentinel-2 data). The configuration values to be used with a given version of the software are typically set in the field, before executing the software. When generating test inputs for the new requirements, it is thus necessary to properly set the values appearing in configuration files, because they are referenced by OCL constraints involving configuration parameters. An example is given by the constraint in Fig. 7 that states that the virtual channel identifier specified in a *Vcdu* header (attribute *virtualChannelId* of class *VcduHeader*) must either be equal to the idle virtual channel identifier (attribute *idleVcid* of class *Configuration*) (line 2) or to one of the active virtual channel identifiers (attribute *idleVcid* of class *Configuration*) (line 4) present in the configuration file.

4. AUTOMATIC GENERATION OF TEST INPUTS FOR NEW DATA REQUIREMENTS

We automatically generate test inputs for new requirements by adapting existing field data. To this end, we combine model transformations with constraint solving. Model transformations enable the partial reuse of existing field data, while constraint solving allows for the generation of missing data that fulfils the updated constraints. Fig. 8 shows the four steps of the approach.

In step 1 we load a chunk of field data in memory as an instance of the original data model (Original Model Instance). In the case of SES-DAQ, the process is automated by using a parser that follows the approach described in [Di Nardo et al. 2015a]. The parser uses stereotypes in the UML class diagram to determine the mapping between class attributes and the bytes stored on disk.

In step 2, we generate an instance of the Updated Data Model by means of a model transformation applied to the Original Model Instance. The result of the model transformation is an instance of the Updated Data Model that is incomplete (Incomplete Model Instance). The Incomplete Model Instance contains only the information that can be directly derived from the Original Model Instance: instances of classes and at-

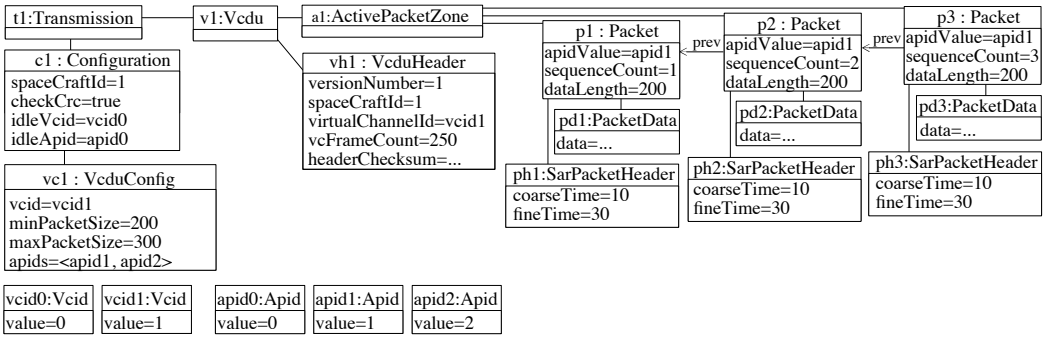

Fig. 9. Example of an instance of the Original Data Model of SES-DAQ visualised using the object diagram notation.

tributes that have been introduced in the Updated Data Model are missing from the Incomplete Model Instance (these instances are the ones generated in the next steps of the algorithm).

In step 3 we generate a valid instance of the updated data model by means of constraint solving. As the underlying solver we use the Alloy Analyzer [Jackson 2015]. Alloy is a modelling language for expressing complex structural constraints [Jackson 2002], which has been successfully used to generate test inputs for testing object-oriented programs [Khurshid and Marinov 2004]. We rely upon UML2Alloy [Anastasakis et al. 2007] to generate an Alloy model that corresponds to the class diagram and the OCL constraints of the data model.

Finally, in step 4, to generate the concrete test inputs to be processed by the software under test (e.g. a binary file in the case of SES-DAQ), the content of the Valid Model Instance is written in the format processed by the software under test. For example, to produce test inputs for SES-DAQ, we rely upon a toolset that we already used in previous work [Di Nardo et al. 2015a]; this toolset writes the content of the Valid Model Instance back to a file as a stream of bytes.

The following sections describe in detail Steps 2 and 3, which are the core contributions of this paper.

5. AUTOMATIC MODEL TRANSFORMATIONS TO GENERATE INCOMPLETE MODEL INSTANCES

The proposed technique is able to automatically generate an incomplete instance of the updated data model in the presence of changes that alter the information provided by the data model. These changes correspond to removals, additions and replacements of classes and attributes.

The technique does not deal with model refactoring (i.e. changes that alter the structure of the data model but preserve the information provided by the data model). Model refactoring can be effectively implemented by means of model transformations [Mens and Tourwé 2004].

The technique initially requires an instance of the original data model. For example, using the original data model of the SES-DAQ given by Fig. 2, and some sampled field data, an original model instance is generated as shown in Fig. 9. The example of Fig. 9 shows an instance of class *Transmission* (*t1*) that contains an instance of class *Vcdu* (*v1*). The instance *v1* contains an instance of class *ActivePacketZone* (*a1*), which in turn contains three *Packet* instances (*p1, p2, p3*), and so on.

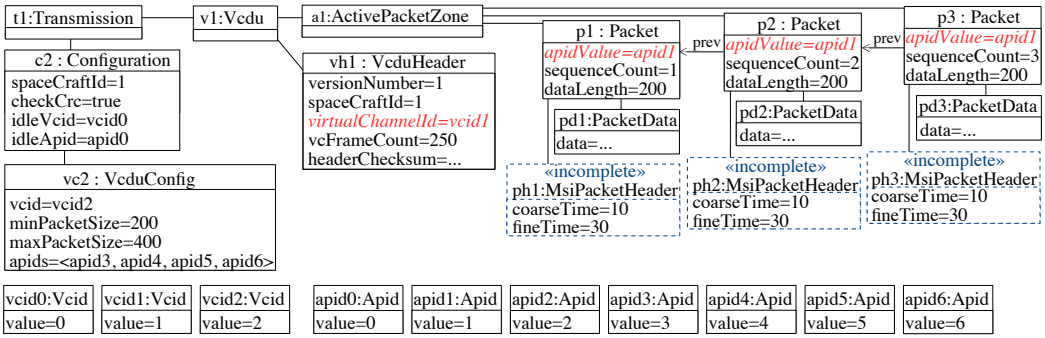

Fig. 10. Incomplete Model Instance derived from the Original Model Instance in Fig. 9. Incomplete instances are blue and dotted, attribute values invalidated by modified constraints are italic and red.

To create an instance of the updated data model, the technique copies and adapts the instance of the original data model. When attributes or classes have been removed, the technique simply ignores the deleted attributes or classes when creating the copy. To deal with classes added to the data model, the technique creates an instance of each new class along with a new association instance linking the new class instance to its containing class instance. The new class instances are tagged as being *incomplete*. Similarly, the technique tags as *incomplete* the instances of classes with attributes that have been introduced in the updated data model. In the case of the replacement of classes, we rely upon a stereotype, named *«Replacement»*, that is used by software engineers in the data model to indicate that a class replaces another one. Fig. 5 shows that the stereotype *«Replacement»* is used for classes *MsiPacketHeader* and *Gpsr2PacketHeader*. The stereotype *«Replacement»* also enables software engineers to specify, for each replacement class, the name of the class whose instances should be replaced. For example, to generate a Sentinel-2 input, class *MsiPacketHeader* replaces class *SarPacketHeader* in the field data of Sentinel-1 satellites.

Fig. 10 shows the Incomplete Model Instance derived from the Original Model Instance of Fig. 9. Since class *MsiPacketHeader* replaces class *SarPacketHeader*, each instance of class *SarPacketHeader* in the Incomplete Model Instance has been replaced by an instance of class *MsiPacketHeader*. Each instance of class *MsiPacketHeader* has been tagged as *incomplete*.

An instance of the updated data model also often differs from an instance of the original data model by the parameter values used in the configuration file. To deal with this case, the technique automatically updates the Incomplete Model Instance to include the content of the new configuration file. To this end, the technique automatically loads the content of the configuration file into memory and replaces the instances of the configuration classes in the Incomplete Model Instance with instances that capture the new given configuration. The instance *c2* of class Configuration appearing in the Incomplete Model Instance of Fig. 10 replaces the instance *c1* appearing in the Original Model Instance of Fig. 9.

Fig. 10 also shows that the updates related to the *Configuration* and the *Packet* classes lead to invalid attributes. According to the OCL constraints of Figs. 4 and 6, the attribute *apidValue* of class *Packet* is expected to be either equal to *apid3* or *apid4*, while the field *virtualChannelId* of class *VcduHeader* is now expected to be equal to *vcid2*. Invalid attributes are automatically detected by our proposed solution while generating an instance of the Updated Data Model.

6. GENERATION OF VALID MODEL INSTANCES

To generate a Valid Model Instance, the technique updates the Incomplete Model Instance with values generated by means of constraint solving. The technique uses constraint solving both to generate data that is completely missing from the Incomplete Model Instance (i.e. classes or attributes tagged as *incomplete*) and to replace data that no longer satisfies the constraints of the Updated Data Model.

In principle, constraint solvers can be used to automatically generate in a single run a solution that matches the shape of the Incomplete Model Instance and satisfies all of the constraints. Unfortunately, constraint solvers often present scalability issues if the data model includes multiple collections of items with constraints among their elements, which is often the case when dealing with the data models of data processing systems. For this reason, we built an algorithm, *IterativelySolve*, that, instead of generating a complete valid model instance in a single run, iteratively generates valid instances of a portion (i.e. a slice) of the updated data model, and assigns the generated values to the attributes in the updated model instance. This iteratively leads to a valid instance of the updated data model.

A slice contains a subset of the class instances that belong to a data model instance. Slices are defined by traversing a graph that corresponds to the data model instance. Let G_{DM} be a graph that corresponds to an instance of a given data model I_{DM} if it contains a set of nodes N, such that for each class instance in I_{DM} there exists a unique corresponding node n in G_{DM}, and for each pair of class instances i_{c1} and i_{c2} connected by an association, there exists an edge connecting the corresponding nodes n_{c1} and n_{c2}. We assume that the data model has a single root node r. Slices are built by means of a depth-first visit of the graph G_{DM}. Each branch of the of the graph corresponds to a slice; a slice is a sequence of nodes that belong to the path between the root r and a leaf node n_l. Leaf nodes are identified during the depth-first graph visit and correspond to class instances without any association edges that point to class instances not yet visited. Slices contain nodes sorted according to the order in which they are traversed in the depth-first visit. We define a parent-child relationship between two nodes in a slice, $n1$ and $n2$, such that $n1 = parent(n2)$ if $n1$ and $n2$ are connected by an association link, and $n1$ was visited before $n2$. The depth-first graph visit does not traverse again nodes already visited so far. For this reason, by construction, a slice cannot contain two nodes with the same parent.

In the case of Fig. 10, we can identify eight slices, whose nodes are grouped as follows having node *t1* selected as the root (we use the identifiers of the instances in Fig. 10 to denote the corresponding nodes): {t1, v1, a1, p1, ph1}, {t1, v1, a1, p1, pd1}, {t1, v1, a1, p2, ph2}, {t1, v1, a1, p2, pd2}, {t1, v1, a1, p3, ph3}, {t1, v1, a1, p3, pd3}, {t1, v1, vh1}, {t1, c2, vc2}. Please note that although the node *p1* can be reached both from nodes *a1* and *p2*, node *p1* does not belong to any slice containing node *p2* (a slice cannot contain two nodes with the same parent).

IterativelySolve relies upon a constraint solver to modify the assignments of the attributes in the slices such that the constraints of the data model are satisfied. We use the expression *slice solving* to indicate the process of executing a constraint solver to identify the values to assign to the attributes of a slice in order to satisfy the constraints of the data model.

The consistency of the solution is guaranteed by the incremental nature of the algorithm: items of collections are generated assuming that previously generated items are valid.

Figs. 11, 12, and 13 show, respectively, the algorithm *IterativelySolve*, function *SolveSlice*, and function *EnableFactsAndSolve*, which implement the logic for the incremental solving of slices.

```
Require: IMI, the incomplete model instance
Require: DM, the data model with the OCL constraints
Require: rootImiClass, the name of the class that specifies the root node of the IMI
Require: configClass, the name of the class that captures the content of the configuration file
Ensure: VMI, a valid model instance generated by means of constraint solving (i.e. the test input data)
 1: defined ← new List()
 2: toRegenerate ← null
 3: VMI ← null
 4: slices ← depthFirstVisit(IMI, rootImiClass, configClass)
 5: odg ← buildOCLDependencyGraph(IMI, DM)
 6: alloyModel ← uml2Alloy(DM)
 7: repeat
 8:     used ← new List()
 9:     for slice in slices do
10:         IMI, used, defined, toRegenerate ←
11:                 SolveSlice(alloyModel, odg, slice, IMI, used, defined, toRegenerate)
12:         if toRegenerate ≠ null then
13:             break
14:         end if
15:     end for
16: until toRegenerate = null
17: if IMI ≠ null then
18:     VMI ← IMI
19: end if
```

Fig. 11. The algorithm *IterativelySolve*.

IterativelySolve performs 5 main activities:

— *Slices detection:* identifies a set of slices of the Incomplete Model Instance that can be solved separately and then recomposed to obtain a Valid Model Instance.
— *Solving with Alloy:* for each slice, derives an Alloy model that is given to the Alloy Analyzer to produce assignments for the attributes that (a) satisfy the constraints of the data model and (b) reflect the actual values observed in the Incomplete Model Instance (this is implemented by function *SolveSlice*).
— *Removal of invalid values:* iteratively removes from the Alloy model the assignments that prevent the solving of slices—that is, attribute values that invalidate OCL constraints (this is implemented by function *EnableFactsAndSolve*).
— *Consistency check:* in order to generate consistent solutions, the algorithm may solve a same slice multiple times—this occurs when the solution of a slice changes a value used by previously solved slices.
— *Update of Incomplete Model Instance:* reads the values generated by the Alloy Analyzer from the Alloy solution, and copies them into the Incomplete Model Instance (implemented by function *SolveSlice*).

The last four activities are repeated for all the slices. By iteratively updating the Incomplete Model Instance, *IterativelySolve* attempts to obtain a Valid Model Instance. The following paragraphs provide a detailed explanation of the algorithm.

6.1. Slices detection

IterativelySolve first identifies a set of slices of the Incomplete Model Instance by performing a depth-first visit of the Incomplete Model Instance (Line 4, Fig. 11). Slices are identified while performing the visit; more specifically, the algorithm keeps track of each path (i.e. each slice) traversed from the root to the leaf nodes of the Incomplete Model Instance (cyclic paths are not traversed).

The software engineer is expected to specify the root node of the Incomplete Model Instance (e.g. class *Transmission* in Fig. 10). Engineers also specify the name of the class that captures the contents of the configuration file (e.g. class *Configuration* in Fig. 10). This is required to avoid the generation of slices containing configuration items because configuration values are specified by the software engineer and are not

```
Require: alloyModel, the initial Alloy model that corresponds to the data model
Require: odg, the OCL Dependency Graph
Require: slice, a slice generated from the IMI
Require: IMI, the incomplete model instance
Require: used, a list of variables appearing in the facts used to solve previous slices
Require: defined, a list of variables whose values have been previously defined using the results generated by the solver
Require: toRegenerate, a list of attributes (if any) that need to be regenerated
Ensure: IMI, the incomplete model instance with values updated to satisfy the constraints for each slice
Ensure: used, an updated list of variables used in the facts
Ensure: defined, an updated list of the variables defined by the Alloy solver
Ensure: toRegenerate, a list of attributes that have been modified in the current execution and need to be regenerated
    by restarting the incremental solving from the first slice
 1: function SolveSlice(alloyModel, odg, slice, IMI, used, defined, toRegenerate)
 2:     prunedODG ← pruneODG(odg, slice)
 3:     a ← generateAugmentedSlice(prunedODG, slice)
 4:     instanceM, facts ← augmentAlloyModel(alloyModel, a)

 5:     if slice.processed = true then
 6:         // this is a re-execution of SolveSlice
 7:         // this slice only needs to be solved if it contains one of the variables to regenerate
 8:         if toRegenerate.contains(DefinedVars(facts)) = false then
 9:             return IMI, used, defined, toRegenerate
10:         end if
11:     end if

12:     solution ← ExecuteAlloy(instanceM)
13:     if solution ≠ null then
14:         used ← SetAllFactVariablesAsUsed(facts, used)
15:     end if
16:     if solution = null then
17:         solution, used, defined, modified ←
18:                 EnableFactsAndSolve(instanceM, facts, used, defined)
19:     end if
20:     if solution = null then
21:         return null, used, defined, null
22:     end if
23:     IMI ← update(IMI, solution)
24:     slice.processed = true // simply trace that slice has been solved at least once
25:     if modified.size > 0 then
26:         return IMI, used, defined, modified
27:     end if
28:     return IMI, used, defined, null
29: end function


30: function SetAllFactVariablesAsUsed(facts, used)
31:     for fact : facts do
32:         if isConfiguration(fact) = false or isShape(fact) = false then
33:             used ← used ∪ DefinedVar(fact)
34:         end if
35:     end for
36:     return used
37: end function
```

function DefinedVar returns the variable defined in a fact (i.e. the left hand side of the assignment in the fact).

Fig. 12. Function *SolveSlice*.

to be generated by means of constraint solving. For example, in the case of Fig. 10, it is the software engineer who specifies the value of the attribute *spaceCraftId* in the configuration file; the constraint solver is not expected to change the given identifier. To prevent the generation of slices containing configuration items, function *depthFirstVisit* does not traverse the configuration class during the depth-first visit.

After generating the slices, *IterativelySolve* builds the OCL Dependency Graph (*ODG*, see Line 5, Fig. 11). An *ODG* is a directed graph whose nodes correspond to the class instances in the Incomplete Model Instance, while its edges connect all the instances that are traversed when evaluating the OCL constraints (please note that a given OCL constraint might traverse multiple paths in the *ODG*). Fig. 14 shows an ex-

```
Require: instanceM, the alloy model that captures the content of a single slice
Require: generatedFacts, a list of facts generated from the instanceM
Require: used, a list of variables appearing in the facts used to solve previous slices
Require: defined, a list of variables whose values had been previously defined using the results generated by the solver
Ensure: solution the result generated by Alloy, or null if the formula cannot be satisfied
Ensure: used, an updated list of variables used in the facts
Ensure: defined, an updated list of the variables defined by the Alloy solver
Ensure: modified, a list of variables used in previous iterations that had been redefined to solve the current slice
 1: function EnableFactsAndSolve(instanceM, generatedFacts, used, defined)
 2:     modified ← new List()
 3:     for fact : generatedFacts do
 4:         if isConfiguration(fact) = false or isShape(fact) = false or DefinedVar(fact) ⊆ defined then
 5:             instanceM ← disable(instanceM, fact)
 6:         end if
 7:     end for
 8:     solution ← ExecuteAlloy(instanceM)
 9:     if solution = null then
10:         return solution, used, defined, modified
11:     end if
12:     for fact : generatedFacts do
13:         if isDisabled(fact) then
14:             instanceM ← enable(instanceM, fact)
15:             tempSolution ← ExecuteAlloy(instanceM)
16:             if tempSolution = null then
17:                 instanceM ← disable(instanceM, fact)
18:                 if DefinedVar(fact) in used then
19:                     modified ← modified ∪ DefinedVar(fact)
20:                 end if
21:                 defined ← defined ∪ DefinedVar(fact)
22:             else
23:                 used ← used ∪ DefinedVar(fact)
24:                 solution ← tempSolution
25:             end if
26:         end if
27:     end for
28:     return solution, used, defined, modified
29: end function
```

Fig. 13. Function *EnableFactsAndSolve*.

ample *ODG* built from the Incomplete Model Instance of Fig. 10; the figure also shows the constraints used to produce the *ODG*. Observe, for example, that nodes $p1$, $a1$, $v1$, $t1$, $c2$, and $vc2$ in Fig. 14 are connected by edges because they are traversed to evaluate constraint *C3*. By looking at the associations in Fig. 2 we can observe that association end *zone* allows for navigation to the *ActivePacketZone* instance *a1* from the *Packet* instance *p1*. The association end *vcdu* allows for navigation to the *Vcdu* instance *v1* from *a1*. Association end *trans* allows for navigation to the *Transmission* instance *t1* from *v1*. Association ends *config* and *vcduConfigs* allow for navigation to instances *c2* (*Config*) and *vc2* (*VcduConfig*), respectively.

6.2. Solving with Alloy

Lines 7 to 16 of Fig. 11 implement the logic to solve the slices of the incomplete model instance; the function *SolveSlice* is invoked in Line 11 to incrementally generate a valid model instance.

SolveSlice calls the function *generateAugmentedSlice* (Line 3, Fig. 12) to generate an augmented slice for each slice *s* identified in the previous steps. The augmented slice includes all the class instances that are required to evaluate if the class instances in the slice *s* violate the constraints of the data model.

To build the augmented slice, the function *generateAugmentedSlice* first traverses the *ODG* to identify all the nodes that can be reached from each node in the slice *s*. The augmented slice contains all the class instances that correspond to the nodes traversed in the *ODG*. For example, the slice *AugmentedSlice1* in Fig. 14 includes the

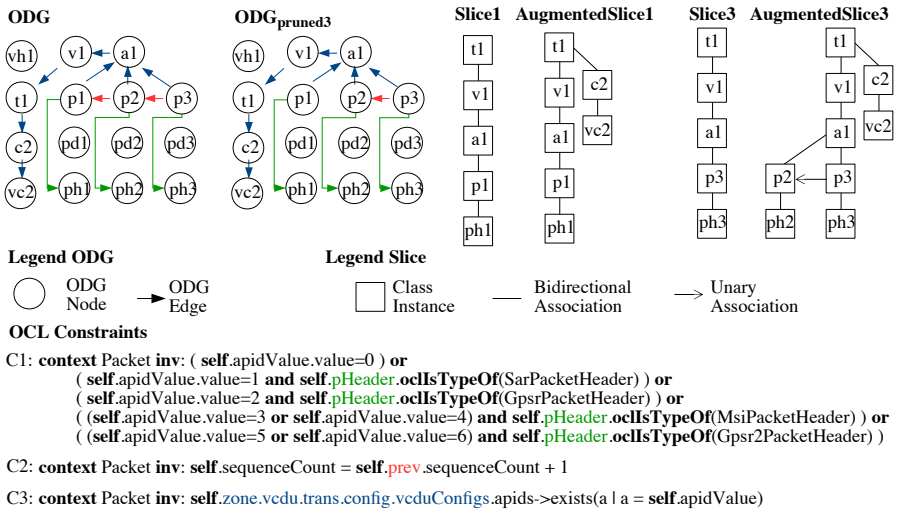

Fig. 14. Example of the artefacts generated to perform slicing: *ODG*, *pruned ODG*, *slices*, and *augmented slices*. These artefacts are built from the Incomplete Model Instance of Fig. 10. The figure also shows the constraints used to produce the *ODG*; colours are used to show which attributes are related to the different edges in the *ODG*.

class instances $c2$ and $vc2$, which are reached when traversing the *ODG* starting from $p1$. Note that $c2$ and $vc2$ are required to evaluate the constraint *C3*.

Constraints on the items of collections may lead to a huge set of nodes that can be reached from a slice *s*. To generate smaller sets of reachable nodes, *SolveSlice* prunes the *ODG* (Line 2, Fig. 12) by removing all the edges that connect collection items with their predecessors with the exception of the items belonging to the current slice, which remain linked to their predecessor. $ODG_{pruned3}$ in Fig. 14 shows the result of the pruning operation performed when processing Slice3. $ODG_{pruned3}$ contains only the edge that links $p3$ with its predecessor ($p2$), but not the edge that connects $p2$ with $p1$. Accordingly, the OCL constraints written for the model must still be valid following the pruning operation. Our approach assumes that, at most, only a single predecessor of a given instance type be required to reach a solution. For example, in constraint C2 of Fig. 14 a packet instance needs only an association with its immediate predecessor (via the association *prev*); should additional predecessor packets have been required for the constraint, then the given approach would fail.

The augmented slice contains both data belonging to the original field data and incomplete data (i.e. incomplete class instances or attributes). *SolveSlice* executes the solver to generate valid class instances or attributes in place of the missing data.

IterativelySolve uses UML2Alloy to generate an initial Alloy model that corresponds to the data model. Since the same Alloy model can be used to solve multiple slices, *IterativelySolve* generates the Alloy model before invoking *SolveSlice* (Line 6, Fig. 11). UML2Alloy implements a model transformation that maps UML class diagrams and OCL constraints to the Alloy format. UML2Alloy generates an Alloy signature for each class and its contained attributes, facts capturing the associations between classes, and a predicate for each OCL constraint.

Given a model specified using the Alloy language, the Alloy Analyzer can generate a valid instance of the data model; however, to reuse existing field data, we need to generate a solution that also has the same 'shape' as the Incomplete Model Instance (i.e. the Incomplete Model Instance and the Alloy solution must be isomorphic). This

```
abstract sig Vcdu {
  trans:one Transmission,
  packetZone:one PacketZone }

abstract sig Transmission {
  config:one Configuration,
  vcdu:some Vcdu }

abstract sig Configuration {
  trans:one Transmission
  spaceCraftId:one Int,
  checkCrc:one Bool,
  idleApid:one Apid,
  idleVcid:one Vcid,
  vcduConfigs:some VcduConfig }

abstract sig VcduConfig {
  config:one Configuration,
  vcid:one Vcid,
  apids:some Apid }

abstract sig Apid {
  value:one Int }

abstract sig Vcid {
  value:one Int }

abstract sig PacketZone {
  vcdu:one Vcdu}

abstract sig ActivePacketZone
  extends PacketZone {
  packets:some Packet }

abstract sig Packet {
  apidValue:one Apid,
  sequenceCount:one Int,
  pHeader:lone PacketHeader,
  prev:lone Packet }

abstract sig PacketHeader {
  Packet:one Packet }

abstract sig MsiPacketHeader
  extends PacketHeader {
  coarseTime:one Int,
  fineTime:one Int }

//declarations for class instances
one sig p2 extends Packet {}
one sig p3 extends Packet {}
one sig apid0 extends Apid {}
one sig apid1 extends Apid {}
one sig apid3 extends Apid {}
one sig apid4 extends Apid {}
one sig apid5 extends Apid {}
one sig apid6 extends Apid {}
one sig c2 extends Configuration {}
one sig vc2 extends VcduConfig {}
one sig t1 extends Transmission {}
one sig v1 extends Vcdu {}
one sig a1 extends ActivePacketZone {}
one sig ph2 extends MsiPacketHeader {}
one sig ph3 extends MsiPacketHeader {}

//facts for data types
fact { apid1.value = 0 }
fact { apid1.value = 1 }
fact { apid3.value = 3 }
fact { apid4.value = 4 }
fact { apid5.value = 5 }
fact { apid6.value = 6 }

//facts for associations
fact { vc2.config = c2 }
fact { c2.vcduConfig = vc2 }
fact { t1.config = c2 }
fact { c2.trans = t1 }
fact { t1.vcdu = v1 }
fact { v1.trans = t1 }
fact { v1.packetZone = a1 }
fact { a1.vcdu = v1 }
fact { a1.packets = p2 + p3 }
fact { p2.zone = a1 }
fact { p3.zone = a1 }
fact { p2.pHeader = ph2 }
fact { s2.packet = p2 }
fact { p3.pHeader = ph3 }
fact { s3.packet = p3 }

//facts for config variables
fact { vc2.apids = apid3 +
       apid4 + apid5 + apid6 }

//facts for non-config variables
fact { p2.apidValue = apid3 }
fact { p3.apidValue = apid1 }
```

Fig. 15. Portion of the Alloy model generated to capture the part of the Incomplete Model Instance containing *AugmentedSlice3* (shown in Fig. 14).

way we can easily copy values from the Alloy solution to the corresponding Incomplete Model Instance. Function *augmentAlloyModel* (Line 4, Fig. 12) modifies the Alloy model in order to enforce the generation of a solution that fits the shape of the slice. Fig. 15 shows a portion of the Alloy model generated by function *augmentAlloyModel* from the Incomplete Model Instance of Fig. 10. The keyword *sig* indicates a signature; that is, a set of atomic definitions (atoms) that we use to model classes with Alloy. Signatures share similar properties with classes of UML class diagrams; in fact, they can be *abstract*, if they cannot be instantiated, and can be used to extend other signatures (see the keyword *extends*). The keyword *one* is used to indicate singletons (i.e. signatures for which only a single instance can exist). The keyword *fact* is used to indicate a property that must hold in the Alloy solution.

To be isomorphic, the Alloy solution and the Incomplete Model Instance must share the same number and types of instances. To this end, the function *augmentAlloyModel* sets all the signatures in the Alloy model as abstract, and then creates a specialisation (i.e. a signature that extends another one), for each class instance in the augmented slice. Each specialised class is a singleton (this way we create the same exact instances observed in field data). The third column of Fig. 15 shows the declarations generated for the different class instances appearing in the Incomplete Model Instance; for example, the first two lines declare $p2$ and $p3$, the two instances of class $Packet$ present in the portion of the Incomplete Model Instance of Fig. 10 that are present in *AugmentedSlice3* (Fig. 14). Signature $Packet$ is abstract; consequently, the solver will not generate any instance of class Packet other than $p2$ and $p3$.

To preserve associations, function *augmentAlloyModel* generates a fact (i.e. a constraint) for each association between the class instances in the data model. For example, the fact '*a1.packets = p2 + p3*' in the top block of the fourth column of Fig. 15 indicates that *p2* and *p3* belong to the collection *packets*.

Finally, function *augmentAlloyModel* also generates Alloy facts that capture the actual values present in the Incomplete Model Instance (e.g. the facts in the bottom block

of the fourth column of Fig. 15).[1] This is done to obtain a solution that reuses the data values observed in the original field data.

6.3. Removal of invalid values

Facts reflect the values observed in the field data (e.g. fact '*p3.apidValue = apid1*' in Fig. 15 that states that the *apidValue* of *Packet 3* is *apid1*). In the presence of updated (or new) OCL constraints that do not match the data used in the original test inputs, the solver cannot generate a solution (fact '*p3.apidValue = apid1*' breaks constraint C1). For this reason, when the solver determines that the set of given constraints is unsatisfiable, *IterativelySolve* relaxes the Alloy model by disabling the facts that prevent the identification of a solution (see function *EnableFactsAndSolve* in Fig. 13). The disabling of a fact is performed by adding a comment at the beginning of the line; this way facts can be easily disabled and re-enabled. The disabling of facts is what enables *IterativelySolve* to replace existing values with new ones.

Function *EnableFactsAndSolve* proceeds by disabling all the facts (Lines 3 to 7, Fig. 13), and then iteratively enabling facts one by one to identify the ones that allow for the generation of a new solution (Lines 12 to 27, Fig. 13). Facts that prevent the generation of a solution are left out (Line 17, Fig. 13). *EnableFactsAndSolve* disables facts that capture actual values of the field data but not facts that preserve the shape of the solution. Additionally, the facts that capture configuration data (i.e. data that is not meant to be regenerated by the solver) are also not considered for removal.

For example, to create a solution from the Alloy model of Fig. 15, it is necessary to relax the model. In fact, this model cannot be used to generate an instance that satisfies the constraints C1, C2, and C3. In particular, constraint C1 cannot be satisfied because the *apidValue* of $p3$ is $apid1$ but its packet header is of type *MsiPacketHeader* (see the facts '*p3.apidValue = apid1*' and '*apid1.value = 1*'). *IterativelySolve* will solve this constraint only after disabling the fact *apid1 in p3.apidValue*, and will then generate a solution with *p3.apidValue* equal to either *apid3* or *apid4*. If a solution is not found even after removing all the facts, *IterativelySolve* terminates without generating a test input (see Lines 20 to 22, Fig. 12). In this case, the technique simply continues the test generation process by sampling a new chunk of field data (Step 1 in Fig. 8).

6.4. Update of incomplete model instance

Once a solution is found, *SolveSlice* updates the data in the Incomplete Model Instance (see Line 23, Fig. 12). In particular, *SolveSlice* uses the values generated by constraint solving to update both the incomplete attributes of the Incomplete Model Instance, and any values that break the updated OCL constraints. By updating the Incomplete Model Instance, *SolveSlice* can incrementally generate a consistent solution: each augmented slice contains an item of a collection and its immediate predecessor (if any); this guarantees that the item is populated with data consistent with the previously generated item.

Although multiple values are valid for some of the attribute instances being reassigned, when generating multiple instances of the same type (e.g. packet instances containing packet headers of type *MsiPacketHeader*), the same data (e.g. for *apidValue*) is always generated by the Alloy solver for a given attribute. If deemed important, a strategy to randomise attribute solving outcomes could be devised.

[1] To minimise execution time, function *augmentAlloyModel* creates facts only for those attributes that appear in the OCL constraints.

6.5. Consistency check

To enforce data consistency, when relaxing a model, function *EnableFactsAndSolve* checks if the disabled fact regards a variable that has been already observed when solving a previous slice. Lines 18 to 20 in Fig. 13 show that *EnableFactsAndSolve* adds to the list *modified* the names of the variables[2], used by previously solved slices, that have been modified during the current execution of *EnableFactsAndSolve*. If a solution for a slice is generated by changing a value used by previous slices, the algorithm restarts the solving from the beginning (see the loop in Lines 7 to 16 of Fig. 11). This is done to ensure that the slices already solved will still satisfy the constraints of the data model. To prevent infinite loops, *EnableFactsAndSolve* does not disable facts that define variables whose values have already been redefined by Alloy in previous iterations (see the clause '$DefinedVar(fact) \subseteq defined$' in Line 4 of Fig. 13).

Lines 5 to 11 of Fig. 12 are an optimisation. Since *SolveSlice* is executed multiple times, it should only call the Alloy Analyzer to solve slices that contain one of the variables that have been redefined or slices that have not yet been solved.

7. ANALYSIS OF TERMINATION, CORRECTNESS AND COMPLETENESS

This section shows, by means of an example, how the incremental solving of slices combined with the consistency check contributes to the generation of a correct solution. A discussion on termination and completeness follows.

7.1. Termination and Correctness

The algorithm proposed in this paper, *IterativelySolve*, incrementally updates the Incomplete Model Instance by modifying the values assigned to the attribute instances of each slice. The Alloy Analyzer guarantees that all the values assigned to a slice satisfy the constraints of the data model.

In the unlikely case the data model instance contains only a single slice, the result given by the algorithm is correct by definition: the solver is executed once and it provides a set of assignments that satisfy all the constraints. The generated solution is trivially correct also whenever the data model instance contains only two slices without any shared variables. In this case, the final solution is the union of the two separate sets of assignments generated by the solver.

In the case of two slices with a shared variable, an incorrect solution may be generated if the values assigned when solving the second slice invalidate constraints that were true for the first slice. By means of an example, we show that, with our incremental, slice-based approach, no incorrect solution can be generated. To simplify the discussion, we model each slice by considering only the attributes belonging to the class instances in the slice, thus ignoring the associations between classes. Associations are not modified by *IterativelySolve*.

Let us take an example of a data model instance containing two slices. The two slices can be represented by the sets of variables *{b,a}* and *{c,a}*, representing class attributes, where *a* is shared by the two slices.

In our example we consider two simple inequalities as constraints, $C_1 : a >= b$ and $C_2 : a <= c$. Let us assume that the field data contains the values x_1, x_2, and x_3 assigned to variables a, b, and c, respectively. In our demonstration, we distinguish two cases that we identified by considering the possible valuations of the constraint $a >= b$: in *case 1*, $x_1 < x_2$, while in *case 2*, $x_1 >= x_2$.

[2]We use the generic term *variable* to indicate an *attribute instance* of the data model instance. The name of a variable is the name used in the *facts* of the generated Alloy model to assign values to attribute instances. An example variable name is *apid1.value* in Fig. 15, which refers to the attribute *value* of the *Apid* instance *apid1*.

Case 1: $x_1 < x_2$

Solving the first slice. To solve the slice *{b,a}*, our algorithm generates an Alloy model that corresponds to the formula $a >= b\ AND\ b = x_2\ AND\ a = x_1$.

If $x_1 < x_2$, the formula cannot be satisfied. As a consequence, the algorithm will invoke function *EnableFactsAndSolve* (Line 18, Fig. 12). Function *EnableFactsAndSolve* starts by disabling facts that correspond to variable assignments (Lines 3 to 7, Fig. 13), and then solves the Alloy formula that contains the remaining facts (Line 8, Fig. 13). In this case, the formula contains just constraint C_1 (i.e. $a >= b$); the formula can be solved. Then the algorithm proceeds by enabling the facts one by one (Lines 12 to 27, Fig. 13).

After enabling the first fact, function *EnableFactsAndSolve* solves $a >= b\ AND\ b = x_2$, which results in a solution where $a = y_1$, $b = x_2$ and $y_1 >= x_2$. After enabling the second fact, the algorithm tries to solve $a >= b\ AND\ b = x_2\ AND\ a = x_1$, which is not feasible. Function *EnableFactsAndSolve* thus keeps the previous solution and updates the Incomplete Model Instance (Line 23, Fig. 12). The Incomplete Model Instance will thus contain the assignments $a = y_1$, $b = x_2$, $c = x_3$ (with $y_1 >= x_2$). The algorithm then starts solving the second slice.

Solving the second slice. The formula built by the algorithm to solve the second slice is $a <= c\ AND\ c = x_3\ AND\ a = y_1$. If $y_1 <= x_3$, the solution is immediately generated by the Alloy Analyzer in Line 12 (Fig. 12) and the algorithm returns with a correct model instance.

However, we are interested in understanding if the algorithm can generate an incorrect solution, which happens if the algorithm assigns to a a value lower than b. The values assigned to the model instance are updated by function *EnableFactsAndSolve*, which is executed if $y_1 > x_3$.

During the execution of function *EnableFactsAndSolve*, the fact $a = y_1$ cannot be disabled because variable a was assigned when generating data for the first slice. The formula to be solved in Line 8 (Fig. 13) is thus $a <= c\ AND\ a = y_1$, which leads to a solution with the assignments $a = y_1$, $c = y_2$ with $y_1 <= y_2$. This solution is returned and used to update the Incomplete Model Instance, which will then satisfy all the constraints (this is trivial since when solving *slice 2* the algorithm did not replace any value belonging to *slice 1*).

Case 2: $x_1 >= x_2$

Solving the first slice. If $x_1 >= x_2$, the field data already contains values that satisfy the formula $a >= b\ AND\ b = x_2\ AND\ a = x_1$, and the algorithm will proceed with the solving of the second slice without changing any variable values. Please note that in this case the algorithm variable $used$, which is a list data structure, is populated with all the variables appearing in the facts (see function *SetAllFactVariablesAsUsed*, Lines 30 to 37, in Fig. 12).

Generating data for the second slice. The formula built to solve the second slice is $a <= c\ AND\ c = x_3\ AND\ a = x_1$.

If $x_1 <= x_3$, the solution is immediately generated by the Alloy Analyzer and the algorithm returns with a correct model instance.

A more interesting case occurs when $x_1 > x_3$. In this case, function *EnableFactsAndSolve* is executed.

Function *EnableFactsAndSolve* disables all the facts and then re-enables them one by one. After enabling the first fact, *EnableFactsAndSolve* solves the formula $a <= c\ AND\ c = x_3$ (Line 15, Fig. 13), which leads to the assignments $a = y_3$, $c = x_3$ with $y_3 <= x_3$. After enabling the second fact, *EnableFactsAndSolve* tries to solve the

formula $a <= c\ AND\ c = x_3\ AND\ a = x_1$, which cannot be satisfied because $x_1 > x_3$. The algorithm thus keeps the previously generated solution and adds variable a to the list $modified$ (Line 19, Fig. 13) to indicate that the value of variable a, which was used to solve a previous slice (the variable belongs to the list $used$), had been modified. Variable a is also added to the list $defined$ (Line 21, Fig. 13).

Repetition of the main loop of IterativelySolve. After function *EnableFactsAndSolve* returns, *SolveSlice* updates the Incomplete Model Instance with the generated values (Line 23, Fig. 12) and then checks if the list $modified$ has a size greater than zero (Line 25, Fig. 12), which is true in this case. *SolveSlice* then returns the updated Incomplete Model Instance and the contents of the list $modified$.

Since $toRegenerate \neq null$ is true (in Line 12 of Fig. 11), *IterativelySolve* re-executes the loop in Lines 7 to 16 in Fig. 11, which means that it again invokes the function *SolveSlice*. This time the Incomplete Model Instance contains the following assignments $a = y_3$, $b = x_2$, $c = x_3$ (with $y_3 <= x_3$).

Solving the first slice, second iteration. To generate data for the first slice, the solver must satisfy the formula $a >= b\ AND\ b = x_2\ AND\ a = y_3$. If $y_3 >= x_2$, the formula trivially evaluates to true and the algorithm proceeds by solving the next slice.

If $y_3 < x_2$, *SolveSlice* invokes function *EnableFactsAndSolve*. Variable a has already been set in previous iterations, so *EnableFactsAndSolve* ends up by solving the formula $a >= b\ AND\ a = y_3$, which leads to the assignments $a = y_3$, $b = y_4$ with $y_3 >= y_4$.

Solving the second slice, second iteration. To solve the second slice, *EnableFactsAndSolve* builds the formula $a <= c\ AND\ c = x_3\ AND\ a = y_3$, which trivially evaluates to true ($y_3 <= x_3$, according to the previous iteration of *IterativelySolve*).

The solution generated for the first slice might have lead to $y_3 >= x_2$, or to $y_3 < x_2$. If $y_3 >= x_2$, the assignments in the Incomplete Model Instance are thus $a = y_3$, $b = x_2$, $c = x_3$, which satisfy the constraints C_1 (with $y_3 >= x_2$) and C_2 (with $y_3 <= x_3$). If $y_3 < x_2$, the assignments in the Incomplete Model Instance are thus $a = y_3$, $b = y_4$, $c = x_3$, which also satisfy the constraints C_1 (with $y_3 >= y_4$) and C_2 (with $y_3 <= x_3$).

General case

The example presented in this section shows that the algorithm is able to guarantee that all the constraints are satisfied even in the presence of variables contained in slices that are redefined while solving subsequent slices. For this to be true in the general case, we need to show that the algorithm always terminates and that, at any given step, solving yields correct results, when it can return a result (see completeness discussion in section 7.2).

Termination. The example has shown that *IterativelySolve* does backtracking (i.e. it restarts the solving process from the first slice) every time *EnableFactsAndSolve* redefines a variable included in a slice solved by a previous iteration. However, given that *EnableFactsAndSolve* is allowed to modify the value assigned to a variable only once, we can guarantee the termination of the algorithm. *EnableFactsAndSolve* behaves the same way in the presence of one or more variables. Thus, the termination of the algorithm is guaranteed also in presence of multiple constraints working on multiple shared variables.

Correctness. We can guarantee that the final result generated by *IterativelySolve* is correct if we can guarantee that the union of the results generated for the single slices satisfies all the constraints. The Alloy Analyzer gives guarantees about the correctness of the results generated for a single augmented slice, while the backtracking mecha-

nism of *IterativelySolve* guarantees that slice solving does not invalidate previously validated constraints.

We provide a formal proof of correctness for constraints that capture binary relations by relying upon set theory. A data model instance can be seen as a set M whose elements are the variables (attribute instances) belonging to the model instance. Given an OCL constraint, we may express it in terms of a relation defined over a subset S of M, $S \subseteq M$, whose elements are all the variables of the model instance referenced in the OCL expression. If a constraint holds for M and is formalised as a relation $R \subseteq S \times S$, it means that the corresponding relation R is total for S, so that, $\forall a, b \in S$, the constraint is true for either (a, b) or (b, a). An augmented slice is a subset T_i of M, with $T_i \subseteq M$. By construction *IterativelySolve* builds n augmented slices of M such that $M = \bigcup_{i=1}^{n} T_n$.

For every slice T_i, *IterativelySolve* identifies a set of assignments that satisfy the given constraint. This means that *IterativelySolve* identifies a set $R_i \subseteq R$ that is total for a subset S_i of S, with $S_i \subseteq T_i$. The backtracking part of *IterativelySolve* guarantees that if all the slices are solved, then all the R_i are total. The final solution produced by *IterativelySolve* is the union of all the R_i generated by Alloy and *IterativelySolve* is correct if this union yields R, e.g., $R \subseteq \bigcup_{i=1}^{n} R_i$. This is what we need to prove.

We prove by contradiction that the result produced by *IterativelySolve* is correct for *binary relations*. The result produced by *IterativelySolve* is not correct if there exists a pair $(a, b) \in R$ such that $(a, b) \notin \bigcup_{i=1}^{n} R_i$. However, *IterativelySolve* guarantees that if $a, b \in M$ then a, b belongs to at least one augmented slice T_i (recall that an augmented slice includes all the variables reachable by traversing the *ODG*). Since R_i is total for $S_i \in T_i$, then there must exist an R_i such that any pair a, b belonging to S_i also belongs to R_i. This implies that for all $a, b \in M$ there must exist at least an R_i such that $(a, b) \in R_i$ Therefore, we have shown that the union of R_is yields R.

Beyond binary relations, in the general case we observe that a constraint that holds on the individual slices T_1 ... T_n may not hold for the whole model M only if $S \not\subseteq \bigcup_{i=1}^{n} S_i$ (i.e. when the constraint applied on the whole model refers to variables not referenced in at least one augmented slice). However, this is not possible because *IterativelySolve* uses the *ODG* to build the individual augmented slices: it includes in a single augmented slice all the instance variables that are required to validate the OCL constraints on a single slice. The only exception occurs with collections; augmented slices only contain predecessors of collection items. This implies that we can give guarantees about the correctness of the algorithm if it is used to solve constraints that follow our restrictions (see section 6.2); for example, they do not refer to arbitrary collection elements. The algorithm may not provide correct results, for example, in the presence of constraints that restrict the number of collection items with a given property, e.g., in the case of SES-DAQ, having at most five *Packet* instances with *apidValue* equal to zero. According to our experience these types of constraints are seldomly used. The test inputs generated by *IterativelySolve* can be validated against the OCL constraints of the data model. In the rare cases where some of the constraints do not hold, software engineers can manually modify the generated test input to obtain a valid one and overcome the current restrictions of *IterativelySolve*.

7.2. Completeness

The discussion in the previous paragraphs has shown that *IterativelySolve*, our iterative, slice-based approach, generates data that can only result in a solution that is valid or it will generate no solution at all.

IterativelySolve is thus incomplete; that is, it does not guarantee the generation of an instance of the Updated Data Model even if it exists. This is mostly due to the fact that *EnableFactsAndSolve* is allowed, in our current implementation, to modify the value assigned to a variable only once. The modification of variable values is what enables *EnableFactsAndSolve* to modify an Incomplete Model Instance to satisfy the constraints for the Upgraded Data Model; that is, *EnableFactsAndSolve* modifies the value assigned to a variable when it does not satisfy an existing constraint. The limit on the number of times a variable can be reassigned is what guarantees the termination of the algorithm.

IterativelySolve may not be able to identify a solution if the OCL constraints lead to circular dependencies between the attributes in the data model instance; however, the conditions under which this happens are not frequent in practice. *IterativelySolve* does not identify a solution when, in the presence of two augmented slices, for example *Slice1* and *Slice2*, *IterativelySolve* reassigns a value to variable v when solving *Slice1*, and then the same variable needs to be reassigned when solving *Slice2*. To figure out the context under which this happens we must recall that the augmented slices generated by *IterativelySolve* include all the variables of the Incomplete Model Instance that are related by the OCL constraints in the Data Model, except for variables belonging to collections. Collection items never belong to the same slice, except for immediate predecessors. Thus, *IterativelySolve* may not identify a solution if there are two constraints $C_{va}(v, i_a)$, e.g. $v > i_a$, and $C_{vb}(v, i_b)$, e.g. $v > i_b$, relating two pairs of variables (v, i_a) and (v, i_b) belonging to *Slice1* and *Slice2*, respectively, and *v* is a variable shared between the two augmented slices. Please note that to belong to different slice variables, i_a and i_b should be collection items belonging to the same collection, and neither of these collection items should be an immediate predecessor of the other.

Let us investigate further the type of complex circular dependencies among OCL constraints that may render *IterativelySolve* unsuccessful. In the above example, note that variable v needs to be reassigned when solving *Slice2* if a valid value for i_b cannot be found given the current value of v. This case occurs only if there is a constraint $C_x(i_b, x)$, e.g. $i_b > x$, with x being a variable that cannot be reassigned. Variable x cannot be reassigned in two cases: either because it belongs to a configuration item (i.e. it is a constant value) or it has been reassigned in a previously solved slice (i.e. there is a circularity in the OCL dependencies). If variable x is a constant it is very likely that the constraint C_x is defined over all the items of the collection (i.e. we may have a generic constraint $i > x$ for all the items i of the collection), which implies that the two slices should satisfy exactly the same constraints and thus a result is always generated by *IterativelySolve*. A simple and common case of circular dependency is one in which C_x relates i_b with i_a, but this case cannot occur because we require that constraints involving collection items affect immediate predecessors only (and these are always included in the same slice). More complex circular dependencies (e.g. involving multiple collections) might occur but they are rare in practice.

In some cases, even in the presence of complex circular dependencies, *IterativelySolve* may be able to find solutions for shared variables. Both the characteristics of the data model instance and the criteria used to select the variables to be reassigned using Alloy results determine whether this is possible. The choice of the variables to be reassigned depends on the order in which *EnableFactsAndSolve* re-enables disabled facts (see Fig. 13). *EnableFactsAndSolve* does not implement any complex heuristic for

selecting the facts that should be re-enabled—it simply processes facts in their order of appearance in the Alloy model. However, since facts are added in the Alloy model according to the order of appearance in the slice, in practice, *EnableFactsAndSolve* tends to reassign variables that belong to the leaf nodes in the augmented slices, while it tends to keep values of parent and root items[3]. This heuristic is meaningful in our context because of the hierarchical structure of the data model considered in our case study. The development of heuristics that work better on case studies with different characteristics is part of our future work.

Two distinct characteristics of the data model instance may ease the identification of proper values for shared variables. The first is the presence of field data that remain valid even in the presence of new data requirements, thus limiting the number of shared variables that need to be reassigned with the solver. The second is that the variables that are shared across multiple slices are often constant configuration parameters that cannot be altered by *EnableFactsAndSolve*, and thus do not lead to any conflict when two different slices are solved.

Last, *IterativelySolve* could be easily extended to enable *EnableFactsAndSolve* to reassign variable values more than once, thus augmenting the probability of building a solution. This might be done by introducing a counter for each variable that keeps track of the number of times a variable is reassigned.

8. EMPIRICAL EVALUATION

We performed an empirical evaluation to answer four research questions: the first two questions address the scalability and performance of our approach, while the remaining two questions address whether the data generated by the approach can, in fact, be used to effectively test new requirements. The research questions are:

— RQ1: Does the proposed approach scale to a practical extent?
— RQ2: How does the proposed approach compare to a non-slicing approach?
— RQ3: Does the proposed approach allow for the effective testing of new data requirements?
— RQ4: How does the use of the proposed approach compare to a manual approach?

The following subsections overview the subject of the study and the experimental setup, and describe, for each research question, the measurements performed and the achieved results.

8.1. Subject of the study and experimental setup

We implemented our approach as a Java prototype that: (a) relies upon the Eclipse UML2 Library for the processing of data models (i.e. class diagrams), (b) implements wrapping code to integrate UML2Alloy and the Alloy Analyzer (using the integrated SAT4J solver [Le Berre and Parrain 2010]), and (c) implements *IterativelySolve* and all the supporting functionality.

As subject of our study, we considered SES-DAQ, the industrial data processing system introduced in Section 2 that processes satellite input data. Recall from Section 3 that the data model originally created to represent the Sentinel-1 satellite input data is updated to reflect the new data requirements of the Sentinel-2 satellite input data. SES-DAQ is a non-trivial system written in Java having 32,469 bytecode instructions, whose data model contains 82 classes, 322 attributes, and 56 associations. To capture the constraints of the SES-DAQ data model we defined 52 OCL constraints (28 input constraints, 24 input/output constraints).

[3]Please recall that a variable is reassigned if its current value prevents the generation of a solution that satisfies the data model constraints.

As an input for our approach, we considered a large transmission file containing Sentinel-1 mission field data provided by SES. The size of the transmission file is about 2 gigabytes, containing 1 million VCDUs belonging to four different virtual channels.

Because of the large number of runs performed for this experiment and considering that some runs took up to 110 hours (each of which had to be repeated ten times), we used a large cluster of computers to run these experiments [Varrette et al. 2014]. To allow for a fair comparison between the different techniques and the various file sizes considered, the experimental runs were each executed on computing nodes having the same characteristics. The experiments were run on a bullx B500 blade system [Atos 2016] with each node having two processors (2 $\times$ Intel Xeon L5640 @ 2.26 GHz). Altogether, the experiments took over 143 days of run time to execute.

8.2. RQ1: Does the proposed approach scale to a practical extent?

8.2.1. Measurements and setup. *RQ1* deals with the practical applicability of the proposed approach.

The generation of new data should be fast enough and scale effectively as file sizes increase. For this reason, to respond to *RQ1*, we applied the proposed approach to automatically generate test input files of various sizes. More specifically, we randomly sampled chunks of field data used by SES to test Sentinel-1 satellite requirements, and used those data chunks to generate inputs that cover the data requirements related to the processing of Sentinel-2 satellite data.

We automatically generated test inputs containing from 50 to 500 VCDUs, in steps of 50 VCDUs. We chose these values because of our experience with SES-DAQ. Our previous research results, in fact, show that test inputs with 50 VCDUs can be effectively used to perform conformance testing [Di Nardo et al. 2015a]. Test inputs with 500 VCDUs, instead, have been effectively adopted for robustness[4] testing, to stress the behaviour of the software in the presence of inputs containing multiple invalid data values [Di Nardo et al. 2015b]. In general, software engineers aim to generate test files that are as realistic as possible in terms of size and content, as large sizes will stress the system more and are more likely to reveal faults. For example, in the case of SES-DAQ, larger input data files are more likely to be able to accommodate more diversity of patterns in the data and reveal faults related to the handling of large amounts of data.

For each given VCDU value, we generated ten test inputs using our approach. We measured the execution times for creating test inputs with the proposed approach; specifically, we analysed the relationship between execution time and input size.

In this paper we do not deal with the problem of automatically generating test oracles. However, in the presence of input/output constraints defined in the data model by means of OCL, the approach that we presented in [Di Nardo et al. 2015a] can be adopted to automatically determine if the output generated by the system after processing an automatically generated test input is wrong; that is, if the output invalidates some of the OCL constraints in the data model (see [Di Nardo et al. 2015a] for more details).

8.2.2. Results. Fig. 16 shows a plot with the average execution time (in hours) required to generate a test input versus the number of VCDUs contained in each test input (see the curve named *Solving time*); box plots are also shown to demonstrate that the variance across runs is low in most cases. Fig. 16 also shows that the approach scales to a practical extent. In the case of test inputs containing 50 VCDUs, the

[4]Robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [ISO/IEEE 2010].

approach requires on average 35.6 minutes to generate a single test input. In the case of test inputs containing 500 VCDUs, the approach requires on average 108.2 hours to generate a test input. A test input containing 500 VCDUs is particularly complex to generate because of the presence of multiple collections, each containing items with multiple references to other data items contained in the test input (e.g. in the case of SES-DAQ, test inputs with 500 VCDUs contain on average 24,861 class instances and 28,827 association instances). Such big inputs cannot be handcrafted by software engineers, which highlights the usefulness of our approach.

We consider the time required to generate big inputs to be acceptable in practice. The approach provides the benefit of automated test generation (i.e. no human effort is required to generate the test cases) and, furthermore, thanks to its model-based nature, does not negatively impact on the deadlines of the software testing process even if test generation may require days to complete. Given that the proposed approach requires only an updated data model and existing field data, the test input generation process can be started immediately after new data requirements are defined. Test generation can be executed while the requirements are implemented; for this reason, the generated test inputs are likely to be available before the system is ready to be tested.

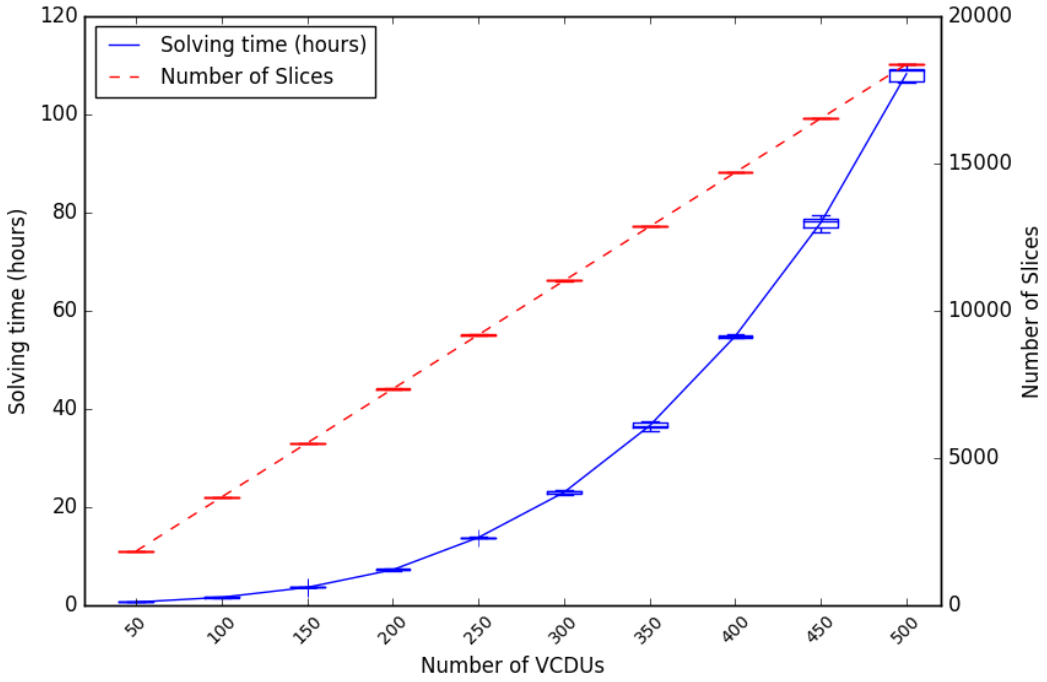

Fig. 16. Average execution time required to generate test inputs and average number of slices versus number of VCDUs in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

The curve for solving time in Fig. 16 shows an exponential growth. This is mainly due to the nature of the input data processed by SES-DAQ. To better understand this behaviour we also report in Fig. 16 the average number of slices per test input size, and in Fig. 17, we show the average number of calls to function *SolveSlice* and the average

number of times function *ExecuteAlloy* (i.e. the Alloy Analyzer) had been invoked during the generation of a test case. The plot in Fig. 16 shows that the number of slices grows linearly with the number of inputs. Recall that the iterative process restarts the slice solving loop if *EnableFactsAndSolve* alters the value of a variable used in a previous slice. Consequently, *SolveSlice* can be invoked multiple times—as was the case for our experiments—against the same slices when generating a test input. A direct consequence of this is that the average number of calls to function *SolveSlice* grows exponentially with the size of the inputs (Fig. 17). This trend depends on the presence of several data items shared by multiple slices, and constitutes an indirect indicator of complexity of the input data. Although the average number of calls to function *SolveSlice* grows exponentially, the average numbers of calls to the Alloy Analyzer does not, which means that function *SolveSlice* often does not invoke the Alloy Analyzer; this is an effect of the optimisation implemented in Lines 5 to 11 of *SolveSlice* (Fig. 12). Therefore, the most plausible explanation for the exponential growth in solving time (Fig. 16) is the exponential growth in calls to SolveSlice (Fig. 17), which consumes computation time, even though it does not always invoke the Alloy Analyzer.

To collect empirical evidence that the repeated calls to *SolveSlice* (Line 11, Fig. 11) are responsible for the exponential growth in execution time, we inspected some of the 500 VCDU executions. For example, in one of them 4,600,848 calls to *SolveSlice* were made: in 376,250 of these calls, the slice contained no constrained attributes and the function returned in a fraction of a millisecond; in 14,413 of these calls, the slice already satisfied the constraints of the model—the Alloy Analyzer was called once (the function returned in, on average, 572 ms); in 3,966 of these calls, the slice did not initially satisfy the constraints of the model—the initial call to the Alloy Analyzer failed, *EnableFactsAndSolve* was executed, and as a result multiple additional calls were made to the Alloy Analyzer (this process took on average 17 s); in 4,206,219 of these calls, the slice was being re-executed and contained no modified attributes—the Alloy Analyzer was not called (in this case, the function took on average 75 ms); this corresponds to '$toRegenerate.contains(DefinedVars(fact))$' being false in Line 8 of Fig. 12). Even in cases where the Alloy Analyzer was not called, the combined calls to *SolveSlice* took over 87 hours to execute.

8.3. RQ2: How does the proposed approach compare to a non-slicing approach?

8.3.1. Measurements and setup. To be justified, the proposed approach should provide an advantage over a more straightforward approach that does not use slicing. To respond to RQ2, we thus compared the performance of the approach proposed in this paper with an approach that generates test inputs from scratch without relying upon a slicing algorithm.

We built a solution that uses a modified version of *IterativelySolve* that does not apply slicing. We refer to this approach as *NonSlicingSolving*. *NonSlicingSolving* processes the entire Incomplete Instance Model to derive an Alloy model that captures the shape of the input data but not the actual values of attributes. All of the attribute values are thus generated from scratch through a single execution of the Alloy Analyzer. To compare the scalability of the two approaches, we apply them to generate test inputs containing different numbers of VCDUs and we measure the execution time required to generate new test inputs.

NonSlicingSolving does not scale; in fact, it cannot generate test inputs containing 50 VCDUs because of out of memory errors. In 10 separate executions performed to generate test inputs with 50 VCDUs, the Alloy Analyzer always crashed because of out of memory errors, even when 16 GB of RAM had been dedicated to the Alloy Analyzer process. *NonSlicingSolving* is thus useless for performing robustness testing; it is unable to generate sufficiently large augmented field data files.

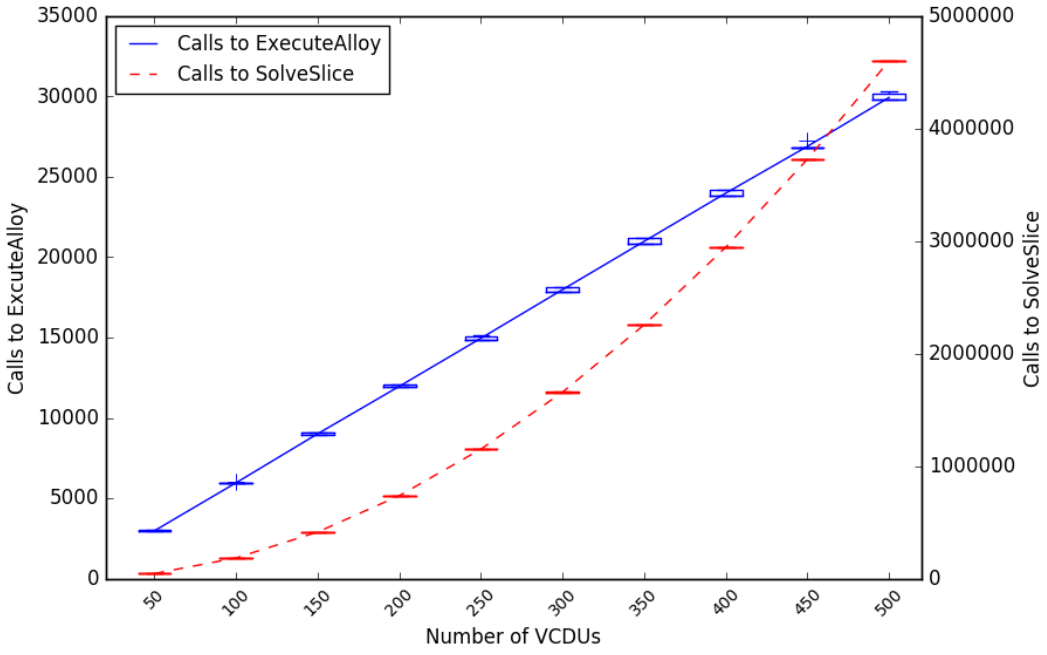

Fig. 17. Average number of calls to functions SolveSlice and ExecuteAlloy versus the number of VCDUs in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

To better compare the two approaches and study the effect of input size on execution time, we ran several experiments to generate test inputs containing 1 to 40 VCDUs. We used both *IterativelySolve* and *NonSlicingSolving* to generate 10 different test inputs for each possible input size containing from 1 to 40 VCDUs. To perform the experiment, we randomly sampled chunks of Sentinel-1 field data. Each sample contained the required number of VCDUs (i.e. 1 to 40 VCDUs), and we then applied the two approaches to generate a test input to validate Sentinel-2 requirements. As a metric of performance, we measured the solving time to generate a valid model instance for the two approaches. For the *NonSlicingSolving* approach, we studied the performance using maximum heap sizes of both 8 and 16 GB. For the approach proposed in this paper we used a maximum heap size of 8 GB[5].

8.3.2. Results. Fig. 18 shows the obtained results for both *NonSlicingSolving* and *IterativelySolve*; the x-axis reports the input size measured in number of VCDUs and the y-axis reports the execution time taken in minutes. Fig. 18 shows that the *NonSlicingSolving* is more efficient for very small test inputs, but it becomes increasingly inefficient with a growing number of VCDUs. *IterativelySolve* always performs better than *NonSlicingSolving* with test inputs containing more than 17 VCDUs. *NonSlicingSolving* shows an exponential growth, this result is in line with research indicating that the Alloy Analyzer shows an execution time that grows exponentially in the presence of relations that involve thousands of elements [Leuschel et al. 2011]. When

[5]Recall that the approach proposed in this paper always terminated, while *NonSlicingSolving* crashed because of out of memory errors.

NonSlicingSolving is executed using an 8 GB maximum heap size, out of memory failures begin to occur at 22 VCDUs, and no solution is possible with 23 VCDUs or more. When executing *NonSlicingSolving* using a 16 GB maximum heap size, out of memory failures begin to occur at 29 VCDUs; no solution is possible with 29 VCDUs or more.

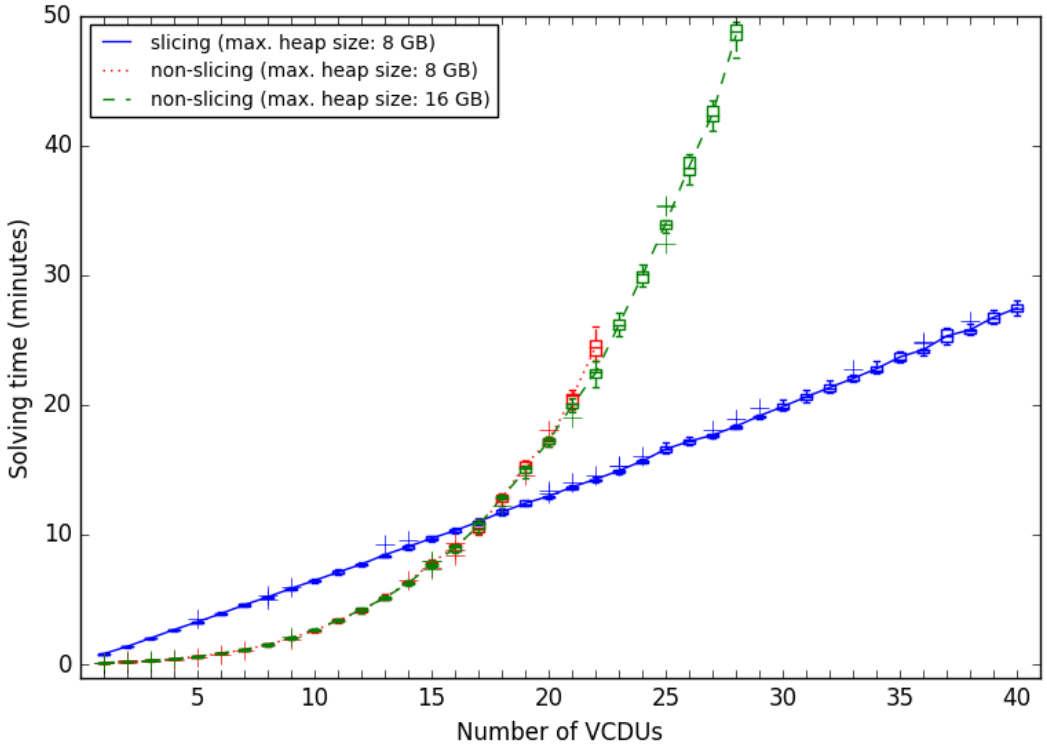

Fig. 18. Comparison of the performance of *IterativelySolve* (that uses slicing) with a non-slicing approach. Average execution time required to generate test inputs versus number of VCDUs in each generated test input. Boxplots are given for each data point. Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range.

8.4. RQ3: Does the proposed approach allow for the effective testing of new data requirements?

8.4.1. Measurements and setup. *RQ3* aims to evaluate the effectiveness of the approach—that is, the ability of the approach to generate test inputs that are effective to test the new requirements of the software system.

One of the key features of SES-DAQ, our case study system, is the ability to automatically identify and discard invalid inputs (e.g. the system should be able to automatically identify and discard data units containing out-of-order packets); more importantly, the system is expected to be robust enough in the presence of invalid inputs to continue functioning without failing. For this reason, robustness testing (i.e. testing the capability of the system to deal with invalid data) plays a fundamental role in evaluating whether the software meets its requirements.

To respond to RQ3, we thus applied the approach proposed in this paper to automatically generate new data intended for use in robustness test cases targeting new data requirements. Since robustness testing deals with the generation of invalid test data, we must adapt the proposed approach to automatically generate invalid test inputs for Sentinel-2 requirements. To this end, we integrated the proposed technique with an approach that we developed previously, which generates robustness test cases by automatically mutating chunks of valid field data [Di Nardo et al. 2015a]. Fig. 19 shows how the technique presented in this paper is integrated with our previously developed data mutation approach [Di Nardo et al. 2015a]. In practice, since field data for the new requirements are not available, we relied upon the approach proposed in this paper to generate valid chunks of augmented field data that meet the new data requirements.

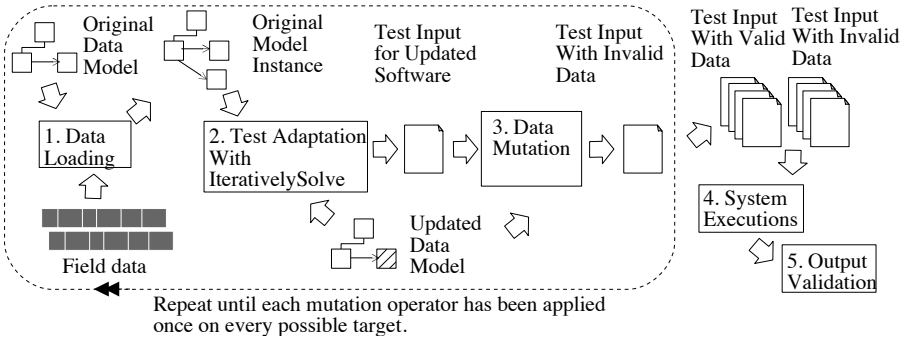

Fig. 19. Testing process followed to respond to research questions RQ3 and RQ4.

The technique described in [Di Nardo et al. 2015a] is based on data mutation. The technique samples a chunk of field data and loads it into memory as an instance of the data model. The loaded data is then modified by applying a set of generic mutation operators. The mutation operators enable the automatic generation of invalid test inputs. There are six mutation operators defined in [Di Nardo et al. 2015a]: three working at the class level (Class Instance Duplication, Class Instance Removal, Class Instances Swapping), and three working at the attribute level (Attribute Replacement with Random, Attribute Replacement using Boundary Condition, Attribute Bit Flipping).

To enable data mutation, software engineers use stereotypes in the data model to configure the mutation operators in such a way that the generated invalid data fits a given fault model. The stereotypes associated to the entities of the SES-DAQ data model are shown in Fig. 2. Only instances of classes tagged with the stereotype *InputData* are mutated. The stereotype *Identifier* is used for attributes that should be replaced with random values. The stereotype *Measure* is used to indicate attributes that should be mutated by using the operator *Attribute Replacement using Boundary Condition*. *Attribute Bit Flipping* is applied to other attributes. The mutation operators working at the class instance level are applied against instances of classes that belong to collections of elements. For example (refer to Fig. 2), given an *ActivePacketZone* instance that contains an ordered set of *Packet* instances, the mutation operator *Class Instances Swapping* can be used to swap the positions of two of the contained packets.

Fig. 19 shows that the test inputs generated by the technique presented in this paper are given as input to the data mutation technique presented in [Di Nardo et al. 2015a] that randomly selects a mutation operator and a possible target for the mutation (i.e.

Table I. Coverage of instructions/branches implementing Sentinel-2 specific data requirements.

Test suite	S2 Instructions Covered			S2 Branches Covered			#Tests
	Avg	Min	Max	Avg	Min	Max	
Auto S1+S2	74 (77.9%)	74 (77.9%)	74 (77.9%)	9 (81.8%)	9 (81.8%)	9 (81.8%)	103.1†
Manual S1+S2	68 (71.6%)	–	–	8 (72.7%)	–	–	32

†Average value. One of the 10 test suites generated contains an additional test case for an attribute that only occurs very rarely in the field data.
Note: Auto, automatically generated test cases according to our methodology; Manual, test cases written manually by SES; S1, Sentinel-1; S2, Sentinel-2

an attribute or a class instance) among the ones not already considered in previous iterations. The process is repeated until all the mutation operators have been applied on one instance of every attribute (or class) on which they can be applied. The generated test inputs are then executed against the software, and the OCL input/output constraints included in the data model are used to verify if the resulting outputs are correct.

To answer *RQ3*, we used test inputs generated with the process in Fig. 19. Since the latest version of SES-DAQ implements both Sentinel-1 and Sentinel-2 data requirements, we created test suites containing both test cases for the original Sentinel-1 data requirements (i.e. input data generated using data mutation only) and test cases for the new Sentinel-2 data requirements (i.e. input data generated with the process in Fig. 19). Though not targeting new code written to implement the new data requirements, the Sentinel-1 test cases serve as additional regression tests that might execute code modified to accommodate the new Sentinel-2 data requirements. We generated ten test suites to assess the impact of the randomness of the process. For example, different samples of field data are used; also, the same mutation operator might target different locations within the sampled transmission data. Each generated test input is 50 VCDUs in size (i.e. the same size adopted in [Di Nardo et al. 2015a]).

The ten test suites were then executed against the SES-DAQ system. No faults were detected during our test suite executions. This was no surprise as the version of the SES-DAQ we tested had already successfully undergone testing using an existing manually written test suite. Therefore, we relied on code coverage as a surrogate measure for fault detection effectiveness [Ammann and Offutt 2008]. Note that maximising test suite code coverage is a common requirement in industry and for our partner, and is therefore an important aspect to consider. We used a code coverage tool, EclEmma [Mountainminds 2006], to measure and evaluate code coverage in terms of bytecode instructions and branches. An analysis by Li et al. [2013] has determined that the implementation of branch coverage in EclEmma, which measures the branches covered at the bytecode level, provides the equivalent of clause coverage (i.e. it checks that each clause of a predicate evaluates to both true and false, and, for switch statements, each branch is considered to end with a break statement). We measured the number of instructions and branches implementing Sentinel-2 data requirements that were covered by the test inputs generated with the proposed approach; the instructions and branches implementing the new data requirements were determined via a manual code analysis. Branch coverage captures the effectiveness of the test suite to spot failures that depend on specific values for certain clauses.

8.4.2. Results. The top section of Table I shows the results of the different test suites considered for our evaluation. Column *Test suite* lists the name of the test suites, *Auto S1+S2* is the test suite automatically generated to cover both Sentinel-1 and Sentinel-2 data requirements. Column *S2 Instructions Covered* reports the average, minimum and maximum number of bytecode instructions implementing Sentinel-2

data requirements that have been covered by the generated test suites. Column *S2 Branches Covered* reports the average, minimum and maximum number of Sentinel-2 specific branches covered by the generated test suites.

The new data requirements related to the processing of Sentinel-2 data have been implemented in 11 branches, for a total of 95 bytecode instructions. The results show that, for each automatically generated test suite, the proposed approach covers 9 (81.8%) of the branches related to the new data requirements (see column *S2 Branches Covered*). The proposed approach does not cover two branches because these branches have been written to handle a particular error that cannot be introduced through the data mutation operators of the technique presented in [Di Nardo et al. 2015a]. However, the proposed approach proved to be able to cover most, 77.9%, of the instructions implementing the new data requirements in a fully automated manner. Uncovered instructions correspond to the two uncovered branches mentioned above.

8.5. RQ4: How does the use of the proposed approach compare to a manual approach?

8.5.1. Measurements and Setup. In general, the costs of software testing depend on the time required to design test cases and prepare the test suites, and the time required to execute the test cases. In our approach, the former corresponds to the cost of modelling while, for manual testing, this is the cost of defining and writing test cases. As for execution time, in many situations it is sufficiently small to have no practical impact on the test process and is then of negligible importance.

Another important aspect is that, in our context, manual testing was performed by highly experienced engineers, with domain expertise. In a context where change is frequent, versions are many, and test engineer turnover is high, this expertise may not always be available. In such situations, automation brings a significant advantage as test cases can be automatically regenerated. However, changes must be made to the model and the assumption is that such changes are less expensive than reviewing and possibly changing every test case in a test suite.

In the case of SES, for example, software engineers have to handcraft test inputs (i.e. binary files) containing proper values so that they resemble a valid satellite transmission. We report here some data that shows that manually written test inputs are expensive to produce. Although each manually written test input contains only 6 VCDUs, it has a complex structure that is labour intensive to produce manually. On average, the manually written test inputs for SES-DAQ contain the equivalent of 130 class instances of the data model, and 261 attribute values. The complexity of the inputs does not depend only on the structure of the file, but also on the constraints that a test input must satisfy to be a valid input for SES-DAQ. On average, each test input contain 36 attribute values that must satisfy at least one constraint, for a total of 1152 attribute values that are constrained within the whole test suite. This data clearly shows that manually writing and maintaining test inputs is not straightforward.

On the contrary, data modelling is a software engineering practice that can play a key role when designing the software, which means that the model required by the proposed approach may coincide with the models already produced by software engineers without any additional cost. Furthermore, in the case of engineering companies like SES, data modelling has additional benefits. SES engineers, for example, find data modelling particularly useful because it allows for the structure of data and data constraints to be characterised with a high level notation that facilitates discussions with the management of the company (usually engineers who can read class diagrams and constraints written in OCL).

Last, when comparing two testing approaches, it is particularly important to take into account the fault detection effectiveness of the test cases. As for RQ3, we use instruction and branch coverage as a surrogate measure for fault detection effectiveness.

Table II. Coverage of instructions/branches of SES-DAQ.

Test suite	Bytecode Coverage Avg / Min / Max	Branch Coverage Avg / Min / Max
Auto S1+S2	23,432.1 (72.2%) / 23,273 (71.7%) / 23,529 (72.5%)	978.7 (51.3%) / 957 (50.2%) / 987 (51.8%)
Manual S1+S2	23,046 (71.0%)	950 (49.8%)

Note: In total, SES-DAQ has 32,469 bytecode instructions and 1,907 branches.

To answer RQ4, for each of the automatically generated test suites (the same as those used for the evaluation of RQ3) and the test suite written manually by SES software engineers with a high degree of domain expertise: we measured test suite size and test execution time; furthermore, to measure test suite effectiveness, we measured the branch and instructions coverage. We specifically considered the coverage of branches and instructions implementing the functionality that deals with new data requirements. The manually written test suite for SES-DAQ tests both Sentinel-1 and Sentinel-2 data requirements; there are 32 test cases in the test suite, three of which were specifically written to test Sentinel-2 data requirements.

We compared the size of the manually written SES test suite with the number of test cases in the automatically generated test suites. We also compared the average time required to execute the test suites. These measurements are useful because if execution time does not introduce important delays in the testing process, which is common for this type of system, it would suggest that the test suite size is not practically relevant in our context. Finally, we compared the coverage of the manual test suite for SES-DAQ with the ten test suites automatically generated to respond to RQ3.

8.5.2. Results. The bottom section of Table I shows the coverage results for the test suite written manually by SES software engineers (see line *Manual S1+S2*). Column # *Tests* shows the number of test cases of the two test suites.

When using the approach proposed in this paper, the number of test cases (103.1, on average) is larger than in manual testing (32), and is determined by the testing strategy. In our context, the complete automatically generated test suite can be run in under 31 minutes, and, therefore, executing the test suite can easily be accommodated on a daily (or even more frequent) basis; the additional execution time required by the automatically generated test suite has little practical impact. Test execution is therefore a negligible cost factor and we will focus on test design. This is likely to be the case for many data processing systems because they are built to process megabytes of data in few seconds.

What matters most for our evaluation is thus whether the automated approach proposed in this paper covers as many or more Sentinel-2 specific instructions and branches than the manual test cases, written by experts. The results show that the *Manual S1+S2* test suite covers 8 (72.7%) of the branches related to the new data requirements and 68 (71.6%) instructions, while the automated approach covers 9 (81.8%) branches and 74 (77.9%) instructions. Automated testing therefore performs slightly better than manual testing in terms of coverage. Although small, the difference in the number of instructions/branches covered is of practical importance for increasing confidence in the reliability of the system—uncovered branches may trigger critical faults (e.g. runtime exceptions), while the software is running in the field.

As an additional note, we report in Table II the overall number of instructions (see column *Bytecode Coverage*) and branches (see column *Branch Coverage*) covered by the automated and manual test suites. Results show that the proposed approach covers more bytecode instructions overall; comparing *Auto S1+S2* with *Manual S1+S2* we observe that, on average, 386.1 additional bytecode instructions are executed in the case of the test suites generated with the proposed approach. The test suites generated

according to our technique take approximately 21 minutes more to execute compared with manual testing; in practice, 21 minutes are negligible.

To conclude, our model-based, automated approach fares slightly better than manual testing, as performed by experts, in terms of instruction and branch coverage corresponding to new requirements. It also achieves better coverage overall, when considering all requirements. In terms of cost, though this is very context dependent, in a situation like the one at SES, where data processing systems incur frequent changes and new versions must be produced, relying on the availability of experts is not always possible or easy. Our model-based approach is therefore a valuable alternative.

8.6. Threats to Validity

To limit the threats to the internal validity of the empirical evaluation—that is, a faulty implementation of our toolset that may lead to erroneous results—we carefully inspected a subset of the generated test inputs to look for the presence of errors: data values of the original field data not preserved in the updated model instance, a shape of the updated model instance that did not coincide with the shape of the original model instance, or values that did not satisfy data constraints. To further validate all the generated test inputs, we performed two additional validation activities: (1) we relied upon the Eclipse UML2 library [Eclipse Foundation 2016] to automatically verify that all the generated test inputs satisfied the OCL constraints of the SES-DAQ data model, and (2) we checked that SES-DAQ error handling code was not covered when SES-DAQ was executed to process the valid test inputs generated with the proposed approach.

Threats to the external validity regard the generalisability of results. The algorithm *IterativelySolve* may show different performance when executed to generate test inputs for other case studies. Factors that affect the performance of *IterativelySolve* are the size of the test inputs to generate and the characteristics of the data model (i.e. number of classes, attributes, associations, and OCL constraints). We have run experiments considering an industrial and complex case study system as benchmark for our evaluation. We have shown that the data model is complex: it contains 82 classes, 322 attributes, 56 associations, and 52 OCL constraints. Working with a nontrivial system that is already in use, gives some confidence that the scalability results can generalise to many of the industrial data processing systems on the market. Furthermore, we studied the effect of input size on the algorithm performance, dealing with test inputs that correspond to model instances containing up to 24,861 class instances and 28,827 association instances, which gave us confidence about the general scalability of the algorithm with respect to test input size.

Results on the effectiveness of the proposed approach may not generalise as well. We have shown that the technique allows for the generation of test inputs such that most of the code implementing new data requirements is executed in the presence of a nontrivial data model. In systems where repeated handcrafting of test inputs is significantly less expensive than modelling, the benefits provided by our technique might be less evident, even if the generated test cases are more effective in covering new data requirements. However, in the general case of complex data processing systems, we believe that the assumption of modelling costs being less expensive than manual testing holds.

In addition, the characteristics of the data model may also affect the results obtained with our algorithm—more specifically, both the operators and the data types used in the data model constraints may slow down the performance of the underlying constraint solver or prevent the solving of constraints. Known Alloy limitations, for example, regard the solving of constraints involving integer variables that might be assigned with big values or constraints with strings. The approach presented in this paper aims to tackle the scalability issues of Alloy, and potentially other constraint

solvers, that arise when the solver is used to generate a huge amount of instances of classes (or user-defined data types) with multiple constraints among them. The approach does not aim to tackle Alloy scalability issues related to the management of basic data types such as integers or strings since these issues have already been addressed by a complementary approach proposed by Ganov et al. [Ganov et al. 2012] (see Section 9 for a discussion), and the extended Alloy solver proposed by Ganov et al. could be integrated into *IterativelySolve* in case constraints involving big integers are used in the data model.

In addition, to address scalability issues related to big integers and strings, our experience has shown that in practice it is much easier to implement domain-specific generators than resorting to constraint solving. As further discussed in Section 9, generators are also employed by other test input generation approaches, such as UDITA [Gligoric et al. 2010]. In our context, domain-specific generators can be executed in pre- or post-processing steps where existing values of the input data are replaced with properly generated values. The implementation of domain-specific generators comes at a cost, but they tend to be simple compared to the problems related with constraint solving (e.g. the solver does not terminate in useful time or it crashes). This is the case of our case study where we did not include in the data model constraints relating variables that might be assigned with big integer values or strings, but we had to deal with the generation of correct big integer values for some of the data attributes (e.g. packet checksums). For these attributes we relied upon the same generators implemented for the data mutation approach integrated into our experiments [Di Nardo et al. 2015a]; that is, we regenerated these attribute values in a post-processing step by means of domain-specific functions that update the attribute values after input data generation and mutation.

To address issues related to the generalisability of results in the presence of different sets of operators used in the model constraints, we evaluated the approach using a case study where different OCL operators expressing properties over collections were used. In particular, our case study includes properties expressed with the operators $forAll$ (used to express properties of all the elements of a collection; please note that the same properties expressed with $forAll$ can be expressed also with the operator $iterate$), and $exists$ (used to indicate the existence of at least one instance of an element satisfying a given property). Since we rely upon UML2Alloy to generate an Alloy model from a given data model, evaluating the approach against all the possible Alloy operators is outside the scope of this paper. For example, the Alloy models generated in our experiments did not include the operators '*' (reflexive transitive closure) and '^' (transitive closure). However, membership operators ':' and 'in' are included.

9. RELATED WORK

Most of the existing approaches that deal with the problem of testing evolving software are based on static program analysis techniques that aim to achieve high source code coverage [Cadar and Palikareva 2014; Santelices et al. 2008; Xu et al. 2013]. These approaches do not deal with the generation of complex structured inputs; furthermore, although effective in achieving high code coverage, these approaches cannot guarantee that the generated tests cover all the system requirements.

Existing work on the generation of test cases in the presence of evolving models is related to the generation of test input sequences from evolving state machines [El-Fakih et al. 2004; Pap et al. 2007; Rapos and Dingel 2012] and does not deal with the generation of complex data structures.

Most of the existing approaches for the generation of test inputs with complex data structures deal with the problem of generating test inputs from scratch. Bounded exhaustive testing approaches generate all the possible test inputs that match the struc-

ture of a given data model up to a given bound, and work with models specified in different formats: Java classes [Boyapati et al. 2002], constraint logic [Senni and Fioravanti 2012], Alloy [Khurshid and Marinov 2004], or Z specifications [Hörcher 1995]. These techniques have been proven to be effective for testing software systems that process classical data structures like trees, but they may not scale once adopted to generate more complex structures like the ones required to test SES-DAQ. Other approaches generate class diagram instances that satisfy a set of given OCL constraints by executing appropriate constraint solvers after having transformed the OCL constraints into other formalisms such as Alloy models [Anastasakis et al. 2007], constraint satisfaction programs [González et al. 2012], SMT [Przigoda et al. 2016], or SAT problems [Soeken et al. 2011]. In principle all of these approaches can be used to automatically generate test input data in the presence of a data model. However, with the exception of Alloy, we found that the transformation tools that implement the techniques presented in most of these papers are not usable in practice, either because they are not available [Soeken et al. 2011; Przigoda et al. 2016] or they are outdated [González et al. 2012]. The results achieved in this paper show that (1) Alloy can be effectively used as the underlying formalism to represent the data model in a format that can be processed by a constraint solver, and (2) the solution proposed in this paper makes the Alloy solver scale when complex input data has to be generated. The study of the applicability of the approach proposed in this paper using other solvers such as SMT is part of our future work.

An efficient black-box test generation technique is UDITA [Gligoric et al. 2010]. What contributes to the efficiency of UDITA is the combination of both generator methods and predicates. Generator methods are used to build instances of the data structure, while predicates are used to validate the generated instances. UDITA relies upon the Java Path Finder model checker [Visser et al. 2004] to generate all the instances that satisfy the given predicates. The implementation of these generator methods that define the complete structure of a complex data model instance and lead to realistic test inputs can be quite expensive. In the case of SES-DAQ, the use of UDITA would require for the implementation of generators that lead to sequences of VCDUs that belong to different channels, and packets that span over multiple VCDUs with a given frequency. In our case study, we also resorted to (domain-specific) generators, but their scope and complexity was very limited and focused on the generation of values for a few specific attributes that were difficult to handle for Alloy. With UDITA, one would need to define a complete set of predicates, which is unrealistic in practice, and would lead to time-consuming constraint solving. In contrast, in our approach, we rely on field data to limit the number of constraints to be solved.

Existing approaches use context free grammars to generate both valid and invalid input data structures, but the existing approaches do not model the complex relationships among data fields [Hoffman et al. 2009; Zelenov and Zelenova 2006]. Context free grammars, for example, cannot be used to model the SES-DAQ relationship that indicates that the *sequenceCount* of a *Packet* must be greater by one than the *sequenceCount* of the previous *Packet*.

Other model-based approaches only focus on the generation of invalid test inputs. These techniques focus on security testing and use models to capture the characteristics of typical malicious (and invalid) inputs that should be properly handled by the software under test. Models like attack trees [Morais et al. 2011], UML state machines [Hussein and Zulkernine 2006], and transition nets [Xu et al. 2012], are used to generate sequences of illegal actions, which are not relevant for testing data processing systems where the complexity of the testing process lies in the definition of the input data and mappings to complex, structured outputs. Mutation-based approaches instead alter valid field data or existing test inputs to generate invalid test inputs [Shan

and Zhu 2009; Bertolino et al. 2014; De Jonge and Visser 2012]. The main limitation of these approaches is that they cannot generate test inputs from scratch to test new data requirements.

Fuzz testing approaches rely upon random inputs [Miller et al. 1990; Miller et al. 1995; Forrester and Miller 2000] or random permutations of valid inputs generated by means of grammar-based specifications [Xiao et al. 2003; Godefroid et al. 2008]. Similarly to grammar-based approaches, fuzz testing cannot deal with inputs with a complex data structure and constraints leading to many trivially invalid inputs—unlike the ones generated by the approach presented in this paper.

In this paper, we introduced an algorithm that relies upon model slicing to improve the performance of constraint solving. Slicing is a solution known to be effective for making constraint solving scale [Xie and Aiken 2007]. Other approaches that rely upon model slicing to improve the performance of constraint solvers exist, but they focus mostly on the slicing of static models for satisfiability purposes (i.e. to verify whether it is possible to create instances of the class diagram without violating any constraint [Shaikh et al. 2010; Balaban and Maraee 2013]). The generated slices typically contain a subset of the elements belonging to the static models. The approach proposed in this paper instead performs slicing on an instance of a class diagram; furthermore, the proposed approach does not simply verify satisfiability but also supports the incremental augmentation of incomplete model instances.

Kato [Uzuncaova and Khurshid 2008] is a tool that incrementally builds a solution for an Alloy model by grouping the predicates in two formulas (i.e. slices). A solution for the Alloy model is generated by solving the base slice first, and then by conjoining the solution with the predicates in the other slice. This approach has also been used to speed up the generation of test cases for testing product lines [Uzuncaova et al. 2010]. Kato deals with performance issues that depend on the presence of several constraints in a same Alloy model but it does not deal with the scalability problems related to the generation of large collections of elements. The approach proposed in this paper is thus complementary to Kato.

Ganov et al. extended Alloy to include a slicing procedure that identifies multiple slices each including constraints related to specific data types, in particular integer and string, to build satisfiability formulas that can be solved by domain-specific solvers that perform better than Alloy [Ganov et al. 2012]. This Alloy extension could be clearly integrated into the solution presented in this paper to efficiently deal with performance issues related to constraints involving integer and string data types.

Finally, data structure repair approaches have a goal similar to that of the approach proposed in this paper—the regeneration of a portion of an invalid data structure instance (by relying upon ad hoc search [Demsky and Rinard 2003], by using symbolic execution [Elkarablieh et al. 2007], or SAT solving [Nokhbeh Zaeem et al. 2012]). For example, Cobbler generates an Alloy model that captures the content of an invalid data structure along with its invariants and method post-conditions, and then uses Alloy to reassigning the data structure fields that appear in the Alloy subformulas and are not satisfied by the invalid data [Nokhbeh Zaeem et al. 2012]. Different from data structure repair approaches, the approach proposed in this paper does not simply modify an existing invalid model instance, but also integrates a slicing algorithm that makes the approach scale in the presence of data structures with a large quantity of items.

10. CONCLUSION

In this paper, we presented an approach to automatically generate test inputs for testing new data requirements of data processing systems. More specifically, we deal with changes that regard the structure of the input data accepted by a data processing system, or the constraints that regulate the content of the different data fields.

When test inputs coincide with complex data structures containing thousands of data items related with each other by multiple constraints, which is often the case when dealing with data processing systems, traditional approaches based on constraint solving cannot be applied because of scalability issues.

The proposed technique makes use of existing field data, a data model describing the original structure and content of the input data, and an updated data model reflecting the modifications to the structure and the content of the input data. The data model is a class diagram capturing the structure of the inputs, with constraints among classes and attributes. *The proposed approach overcomes the limitations of traditional approaches thanks to the integration of model slicing with constraint solving.*

The proposed approach generates test inputs by augmenting and adapting existing field data that matches the original data requirements. The reuse of existing field data reduces the amount of data values that need to be generated by a constraint solver. In particular, the approach uses constraint solving to generate only the data items introduced by the new data requirements, or to regenerate data items that break new or modified data constraints. As the underlying constraint solver, we rely upon the Alloy Analyzer. The proposed technique also integrates a new slicing algorithm that allows for the incremental invoking of the constraint solver to generate portions of the new test input. The algorithm guarantees the consistency of the generated test input, which results from the composition of the data belonging to the different slices.

We validated the scalability and effectiveness of the proposed approach using an industrial case study, a satellite data acquisition system working with the European Space Agency Sentinel series of satellites [ESA 2015]. In our study, we considered a version of the system that had been modified to accept new packet types associated with new missions. The empirical study shows that the proposed approach scales in the presence of complex data structures. In particular, the study shows that the input generation algorithm based on model slicing presented in this paper can produce, in a reasonable amount of time, test input data that is over ten times larger in size than the data that can be generated with constraint solving only. To evaluate the effectiveness of the proposed approach, we generated test inputs to stress software robustness and we measured the code covered when executing the generated test inputs against the software. Robustness testing is critical for testing the case study system considered. The results show that the generated test inputs cover most of the source code instructions of the updated software written to implement new data requirements. The generated test inputs also achieve more code coverage than the test cases implemented by experienced software engineers, thus highlighting the benefits of the proposed approach.

ACKNOWLEDGMENTS

The authors would like to thank Raul Gnaga, Vincent Masquelier, Tomislav Nakić-Alfirević, and David Valcárcel Romeu for their valuable feedback and assistance.

REFERENCES

Shady Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C Briand. 2013. Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1376–1402.

Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing*. Cambridge University Press, Cambridge, UK.

Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2007. UML2Alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*. Springer, Berlin, Heidelberg, 436–450.

Atos. 2016. Bull Welcome - bullx B500 blade system. http://www.bull.com/bullx-b500-range. (2016).

Mira Balaban and Azzam Maraee. 2013. Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. *ACM Transactions on Software Engineering and Methodology* 22, 3, Article 24 (July 2013), 42 pages. DOI:http://dx.doi.org/10.1145/2491509.2491518

Antonia Bertolino, Said Daoudagh, Francesca Lonetti, Eda Marchetti, Fabio Martinelli, and Paolo Mori. 2014. Testing of PolPA-based usage control systems. *Software Quality Journal* 22, 2 (2014), 241–271. DOI:http://dx.doi.org/10.1007/s11219-013-9216-0

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133.

Jordi Cabot, Robert Clarisó, and Daniel Riera. 2008. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW)*. IEEE, Los Alamitos, CA, 73–80. DOI:http://dx.doi.org/10.1109/ICSTW.2008.54

Cristian Cadar and Hristina Palikareva. 2014. Shadow Symbolic Execution for Better Testing of Evolving Software. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 432–435. DOI:http://dx.doi.org/10.1145/2591062.2591104

CCSDS. 2003. Space Packet Protocol, CCSDS 133.0-B-1. http://public.ccsds.org/publications/archive/133x0b1c2.pdf. (2003).

CCSDS. 2006. AOS Space data link protocol, CCSDS 732.0-B-2. http://public.ccsds.org/publications/archive/732x0b2c1s.pdf. (2006).

Maartje De Jonge and Eelco Visser. 2012. Automated Evaluation of Syntax Error Recovery. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 322–325. DOI:http://dx.doi.org/10.1145/2351676.2351736

Brian Demsky and Martin Rinard. 2003. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. ACM, New York, NY, USA, 78–95. DOI:http://dx.doi.org/10.1145/949305.949314

Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, Elizabeta Fourneret, Tomislav Nakić-Alfirević, and Vincent Masquelier. 2013. Model based test validation and oracles for data acquisition systems. In *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*. IEEE, Los Alamitos, CA, 540–550.

Daniel Di Nardo, Fabrizio Pastore, Andrea Arcuri, and Lionel Briand. 2015b. Evolutionary Robustness Testing of Data Processing Systems Using Models and Data Mutation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, Los Alamitos, CA, 126–137. DOI:http://dx.doi.org/10.1109/ASE.2015.13

Daniel Di Nardo, Fabrizio Pastore, and Lionel Briand. 2015a. Generating Complex and Faulty Test Data Through Model-Based Mutation Analysis. In *8th International Conference on Software Testing, Verification and Validation (ICST'15)*. IEEE Computer Society, Los Alamitos, CA, Article 11, 10 pages.

Eclipse Foundation. 2016. Eclipse Model Development Tools (MDT). (June 2016). http://www.eclipse.org/modeling/mdt/

Khaled El-Fakih, Nina Yevtushenko, and Gregor v. Bochmann. 2004. FSM-based incremental conformance testing methods. *Software Engineering, IEEE Transactions on* 30, 7 (July 2004), 425–436. DOI:http://dx.doi.org/10.1109/TSE.2004.31

Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. 2007. Assertion-based Repair of Complex Data Structures. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 64–73. DOI:http://dx.doi.org/10.1145/1321631.1321643

ESA 2015. Overview / Copernicus / Observing the Earth / Our Activities / ESA. (June 2015). http://www.esa.int/Our_Activities/Observing_the_Earth/Copernicus/Overview4

Justin E Forrester and Barton P Miller. 2000. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*. Seattle, USENIX Association, Berkeley, CA, USA, 59–68.

Svetoslav Ganov, Sarfraz Khurshid, and Dewayne E. Perry. 2012. Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers. In *Proceedings of the 14th International Conference on Formal Engineering Methods, ICFEM*, Toshiaki Aoki and Kenji Taguchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 414–429. DOI:http://dx.doi.org/10.1007/978-3-642-34281-3_29

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. IEEE, Los Alamitos, CA, 225–234.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 206–215. DOI:http://dx.doi.org/10.1145/1375581.1375607

Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA '12)*. IEEE Press, Piscataway, NJ, USA, 44–50. http://dl.acm.org/citation.cfm?id=2663689.2663697

Daniel Hoffman, Hong-Yi Wang, Mitch Chang, and David Ly-Gagnon. 2009. Grammar Based Testing of HTML Injection Vulnerabilities in RSS Feeds. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART '09)*. IEEE Computer Society, Washington, DC, USA, 105–110.

Hans-Martin Hörcher. 1995. Improving Software Tests Using Z Specifications. In *Proceedings of the 9th International Conference of Z Usres on The Z Formal Specification Notation (ZUM '95)*. Springer-Verlag, London, UK, UK, 152–166.

Mohammed Hussein and Mohammad Zulkernine. 2006. UMLintr: A UML Profile for Specifying Intrusions. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS '06)*. IEEE Computer Society, Washington, DC, USA, 279–288. DOI:http://dx.doi.org/10.1109/ECBS.2006.70

ISO/IEEE. 2010. Systems and software engineering – Vocabulary. (Dec 2010). DOI:http://dx.doi.org/10.1109/IEEESTD.2010.5733835

Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions of Software Engineering and Methodology* 11, 2 (April 2002), 256–290. DOI:http://dx.doi.org/10.1145/505145.505149

Daniel Jackson. 2015. Alloy Analyzer website. http://alloy.mit.edu/. (2015).

Sarfraz Khurshid and Darko Marinov. 2004. TestEra: Specification-Based Testing of Java Programs Using SAT. *Automated Software Engineering* 11, 4 (2004), 403–434. DOI:http://dx.doi.org/10.1023/B:AUSE.0000038938.10589.b9

Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. 2011. Automated property verification for large scale B models with ProB. *Formal Aspects of Computing* 23, 6 (2011), 683–709. DOI:http://dx.doi.org/10.1007/s00165-010-0172-1

Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*. IEEE, IEEE, Pasadena, CA, 380–389.

Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (Feb. 2004), 126–139. DOI:http://dx.doi.org/10.1109/TSE.2004.1265817

Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

Barton Paul Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Technical Report. University of Wisconsin-Madison, Computer Sciences Department.

Anderson Morais, Ana Cavalli, and Eliane Martins. 2011. A Model-based Attack Injection Approach for Security Validation. In *Proceedings of the 4th International Conference on Security of Information and Networks (SIN '11)*. ACM, New York, NY, USA, 103–110. DOI:http://dx.doi.org/10.1145/2070425.2070443

Mountainminds. 2006. EclEmma - Java Code Coverage for Eclipse. (2006). http://www.eclemma.org

Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. 2012. History-Aware Data Structure Repair Using SAT. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '12)*, Cormac Flanagan and Barbara König (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–17. DOI:http://dx.doi.org/10.1007/978-3-642-28756-5_2

OMG. 2015. The Object Management Group, The Object Constraint Language. http://www.omg.org/spec/OCL/. (2015).

Zoltán Pap, Mahadevan Subramaniam, Gábor Kovács, and Gábor Árpád Németh. 2007. A Bounded Incremental Test Generation Algorithm for Finite State Machines. In *Testing of Software and Com-*

municating Systems, Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp (Eds.). Lecture Notes in Computer Science, Vol. 4581. Springer, Berlin, Heidelberg, 244–259. DOI:http://dx.doi.org/10.1007/978-3-540-73066-8_17

Nils Przigoda, Robert Wille, and Rolf Drechsler. 2016. Ground Setting Properties for an Efficient Translation of OCL in SMT-based Model Finding. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16)*. ACM, New York, NY, USA, 261–271. DOI:http://dx.doi.org/10.1145/2976767.2976780

Eric James Rapos and Juergen Dingel. 2012. Incremental Test Case Generation for UML-RT Models Using Symbolic Execution. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 962–963. DOI:http://dx.doi.org/10.1109/ICST.2012.205

Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-Suite Augmentation for Evolving Software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE Computer Society, Washington, DC, USA, 218–227. DOI:http://dx.doi.org/10.1109/ASE.2008.32

Valerio Senni and Fabio Fioravanti. 2012. Generation of Test Data Structures Using Constraint Logic Programming. In *Proceedings of the 6th International Conference on Tests and Proofs (TAP'12)*. Springer-Verlag, Berlin, Heidelberg, 115–131. DOI:http://dx.doi.org/10.1007/978-3-642-30473-6_10

Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. 2010. Verification-driven Slicing of UML/OCL Models. In *Proceedings of the 2010 IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, New York, NY, USA, 185–194. DOI:http://dx.doi.org/10.1145/1858996.1859038

Lijun Shan and Hong Zhu. 2009. Generating Structurally Complex Test Cases By Data Mutation. *Comput. J.* 52, 5 (Aug. 2009), 571–588. http://dx.doi.org/10.1093/comjnl/bxm043

Mathias Soeken, Robert Wille, and Rolf Drechsler. 2011. Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In *Proceeding of Tests and Proofs: 5th International Conference, TAP 2011*, Martin Gogolla and Burkhart Wolff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 152–170. DOI:http://dx.doi.org/10.1007/978-3-642-21768-5_12

Engin Uzuncaova and Sarfraz Khurshid. 2008. Constraint Prioritization for Efficient Analysis of Declarative Models. In *FM 2008: Formal Methods*, Jorge Cuellar, Tom Maibaum, and Kaisa Sere (Eds.). Lecture Notes in Computer Science, Vol. 5014. Springer-Verlag, Berlin Heidelberg, 310–325. DOI:http://dx.doi.org/10.1007/978-3-540-68237-0_22

Engin Uzuncaova, Sarfraz Khurshid, and Don Batory. 2010. Incremental Test Generation for Software Product Lines. *IEEE Transactions on Software Engineering* 36, 3 (May 2010), 309–322. DOI:http://dx.doi.org/10.1109/TSE.2010.30

Sebastien Varrette, Pascal Bouvry, Hyacinthe Cartiaux, and Fotis Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, Bologna, Italy, 959–967.

Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. ACM, New York, NY, USA, 97–107. DOI:http://dx.doi.org/10.1145/1007512.1007526

Shu Xiao, Lijun Deng, Sheng Li, and Xiangrong Wang. 2003. Integrated TCP/IP protocol software testing for vulnerability detection. In *2003 International Conference on Computer Networks and Mobile Computing (ICCNMC'03)*. IEEE, IEEE Computer Society, Washington, DC, 311–319.

Yichen Xie and Alex Aiken. 2007. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 16 (May 2007), 43 pages. DOI:http://dx.doi.org/10.1145/1232420.1232423

Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. 2012. Automated Security Test Generation with Formal Threat Models. *IEEE Transactions of Dependable and Secure Computing* 9, 4 (July 2012), 526–539.

Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. 2013. Continuous Test Suite Augmentation in Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC '13)*. ACM, New York, NY, USA, 52–61. DOI:http://dx.doi.org/10.1145/2491627.2491650

Sergey Zelenov and Sophia Zelenova. 2006. Automated Generation of Positive and Negative Tests for Parsers. In *Formal Approaches to Software Testing*, Wolfgang Grieskamp and Carsten Weise (Eds.). Lecture Notes in Computer Science, Vol. 3997. Springer Berlin Heidelberg, Berlin, Heidelberg, 187–202.

Received XXXX; revised XXXX; accepted XXXX