Solving the Instance Model-View Update Problem in AADL

Rakshit Mittal
Dominique Blouin
rakshitm@acm.org
dominique.blouin@telecom-paris.fr
LTCI Lab, Telecom Paris
Palaiseau, France

Anish Bhobe
anish.bhobe@ip-paris.fr
Institut Polytechnique de Paris
Palaiseau, France

Soumyadip Bandyopadhyay
soumyadipb@goa.bits-pilani.ac.in
BITS Pilani Goa Campus
Sancoale, Goa, India

ABSTRACT

The Architecture Analysis and Design Language (AADL) is a rich language for modeling embedded systems through several constructs such as component extension and refinement to promote modularity of component declarations. To ease processing AADL models, OSATE, the reference tool for AADL, defines another model (namely 'instance' model) computed from a base 'declarative' model/s. An instance model is a simple object tree where all information from the declarative model is flattened so that tools can easily use this information to analyze the system. However for modifications, they have to make changes in the complex declarative model since there is no automated backward transformation (deinstantiation) from instance to declarative models. Since the instance model is a 'view' of the declarative model, this is a view-update problem. In this paper, we propose the OSATE Declarative-Instance Mapping Tool (OSATE-DIM[1]), an Eclipse plugin for deinstantiation of AADL models implementing a solution of this view-update problem. We evaluate OSATE-DIM with a benchmark of existing AADL model processing tools and verify the correctness of our deinstantiation transformations. We also discuss how our approach could be useful for decompilation of Object-Oriented languages' intermediate representations.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; • **Computer systems organization** → *Embedded and cyber-physical systems*; • **Software and its engineering** → Software creation and management.

KEYWORDS

Model-Driven Engineering, Cyber-Physical Systems, Embedded Systems, View-Update Problem, AADL

[1]https://mem4csd.telecom-paristech.fr/blog/index.php/osate-dim/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552396

ACM Reference Format:
Rakshit Mittal, Dominique Blouin, Anish Bhobe, and Soumyadip Bandyopadhyay. 2022. Solving the Instance Model-View Update Problem in AADL. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552396

1 INTRODUCTION

The Architecture Analysis and Design Language (AADL) [9] was developed to model real-time embedded systems composed of software and physical execution platform components tightly coupled with actuators and sensors (also physical components) to interact with their environments. It is standardized by the Society of Automotive Engineers (SAE-AS5506[2]) for scheduling/flow-control analyses and code generation for various embedded platforms. It is supported by the Open-Source AADL Tool Environment (OSATE[3]), the reference tool for AADL released under the Eclipse Integrated Development Environment (IDE).

Factorization of component declarations in AADL is made possible through constructs like extensions, refinements, inheritance, and different levels of component abstractions allowing for a rich specification of the structural and behavioral characteristics of embedded systems. This richness of the language complicates the analysis of an AADL model. To solve this, OSATE provides another simpler *Instance* metamodel. An *Instance* model represents the runtime configuration of a system. It is generated from the original *Declarative* model through a transformation called *Instantiation*. During *Instantiation*, all properties of the system components and their elements like *Features*, *Connections* and *Modes* are collected from all parent classifiers/specifications, and collapsed into one entity. The architecture of the system is represented in the *Instance* model through a containment tree of components and other elements. Traces in the form of references relate the generated *Instance* elements to their corresponding *Declarative* elements.

Many AADL analysis tools such as RAMSES [19], MC-DAG [17], Cheddar [20] and OSATE itself use the generated *Instance* model for analysis, but for refinement/modification they have to use the traces to identify locations in the *Declarative* model where changes are to be made. Updates performed by tools include (but are not limited to) addition of computed properties to system components or change in the structure of the system by application of some design patterns.

Artifact available: https://doi.org/10.5281/zenodo.6971720
[2]https://www.sae.org/standards/content/as5506d/
[3]https://osate.org/

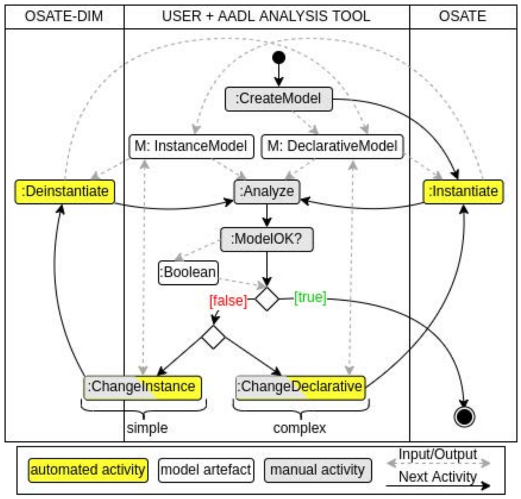

Figure 1: Activity diagram of AADL model analysis in OSATE

It becomes increasingly difficult to track each change in the *Declarative* model as it could (undesirably) propagate to other elements through extensions, refinements, and property inheritances. Also, it is imperative to decide the level of abstraction at which the modification should be made. Conversely, directly modifying the *Instance* model is much simpler. To maintain consistency of information in such a case, the changes should be reflected back in the *Declarative* model. Since the *Instance* model is essentially a mutable *View* of the *Declarative* model, this is the *Instance Model-View Update Problem in AADL*, derived from the well known *View-Update Problem* in database theory.

It will be very beneficial for the AADL-users/developers community to have an automated solution for the *Instance Model-View Update Problem in AADL*, so that users/tools need only focus on modification of the *Instance* model directly, which will greatly simplify tools development.

In this paper, we present an approach and tool for *Deinstantiation* (backward transformation from *Instance* to *Declarative* model) of AADL models in the form of an Eclipse-based plugin as an extension of OSATE. The proposed novel tool is called the OSATE-based Declarative Instance Mapping (OSATE-DIM[3]), which greatly simplifies the processing of AADL models throughout the design-space exploration process illustrated in Fig. 1.

The contributions of this paper are:

(1) an approach and Eclipse plugin tool, OSATE-DIM, for automated incremental *Deinstantiation* of AADL models,
(2) a scope for the application of the proposed approach to Object-Oriented Programming (OOP) languages like C++, Rust, and LLVM-IR,
(3) and improvements suggestions for the current *Instantiation* of AADL models.

This paper is structured as follows: Section 2 describes the theoretical background of the work including AADL and its *Declarative* and *Instance* metamodels released in OSATE, and the *View-Update Problem* and its possible solutions. Section 3 discusses the methodology for *Deinstantiation* through OSATE-DIM, as a case-by-case analysis of the possible modifications using a running example. Section 4 explains the implementation of OSATE-DIM, including the supported *Deinstantiation* scenarios. Section 5 describes the benchmark and case-studies used to validate OSATE-DIM. Section 6 discusses the lessons learnt and our recommendations. Section 7 concludes the paper.

2 BACKGROUND

In this section, we first introduce a subset of the AADL language required to understand this work and its reference tool OSATE. Then we present background on the View-Update problem and show how its notions map to the problem addressed in this work.

2.1 AADL and OSATE

The AADL language, formally specified as a grammar in the standard, is implemented as a metamodel in OSATE. The core concepts of the AADL language are shown in the right part of Fig. 2 as depicted by the OSATE *Declarative* metamodel[4].

2.1.1 AADL Core Language. AADL is a component-based architecture description language. Its *Components* represent hardware or software entities as parts of a modeled system. They can be categorised as data, subprogram, subprogram group, thread, thread group, process, memory, bus, virtual bus, processor, virtual processor, device, system, and abstract. Their structure is defined through *Component Classifiers* which are of two kinds: *Component Type* and *Component Implementation*. Similar to interfaces in OOP languages a *Component Type* defines the external structure of a component and its connection points for interaction with other components. Similar to classes in OOP languages, a *Component Implementation* defines the internal structure of a component including *Subcomponents* and their *Connections*. A *Component Implementation* "implements" a *Component Type* and there can be several implementations defined for a given *Type*.

The aforementioned connection points defined by *Component Types* for transmission of information to and/or from other components consist of *Features*. *Features* can be of different kinds like *Ports (event, data, event-data)*, *Parameters*, *Access (data, bus, subprogram)* with different semantics.

The aforementioned *Subcomponents* contained in a *Component Implementation* are *Components* within a *Component*. They are specified with a *Component Classifier* to define them. *Connections* are linkages between *Features* to define sharing of data or control between *Components*. They have the same categories as *Features* (port, access, parameter), since a *Connection* can only connect *Features* of the same category.

2.1.2 Richness (Classifier Extensions, Subcomponent/Feature Refinements, Property Visibility, Component Libraries). Similar to OO languages, the AADL *Declarative* language defines various kinds of relations between model elements in order to favor reuse of component specifications. A *Type* can extend another *Type* subject to conditions like category, etc. Similarly, an *Implementation* can extend another *Implementation*. Within a *Type* that extends another,

[4]Although not shown in Fig. 2, there are subclasses of *Component Implementation*, *Component Type* and *Subcomponent* for every component category in the *Declarative* metamodel. Those are not shown due to space limitations.

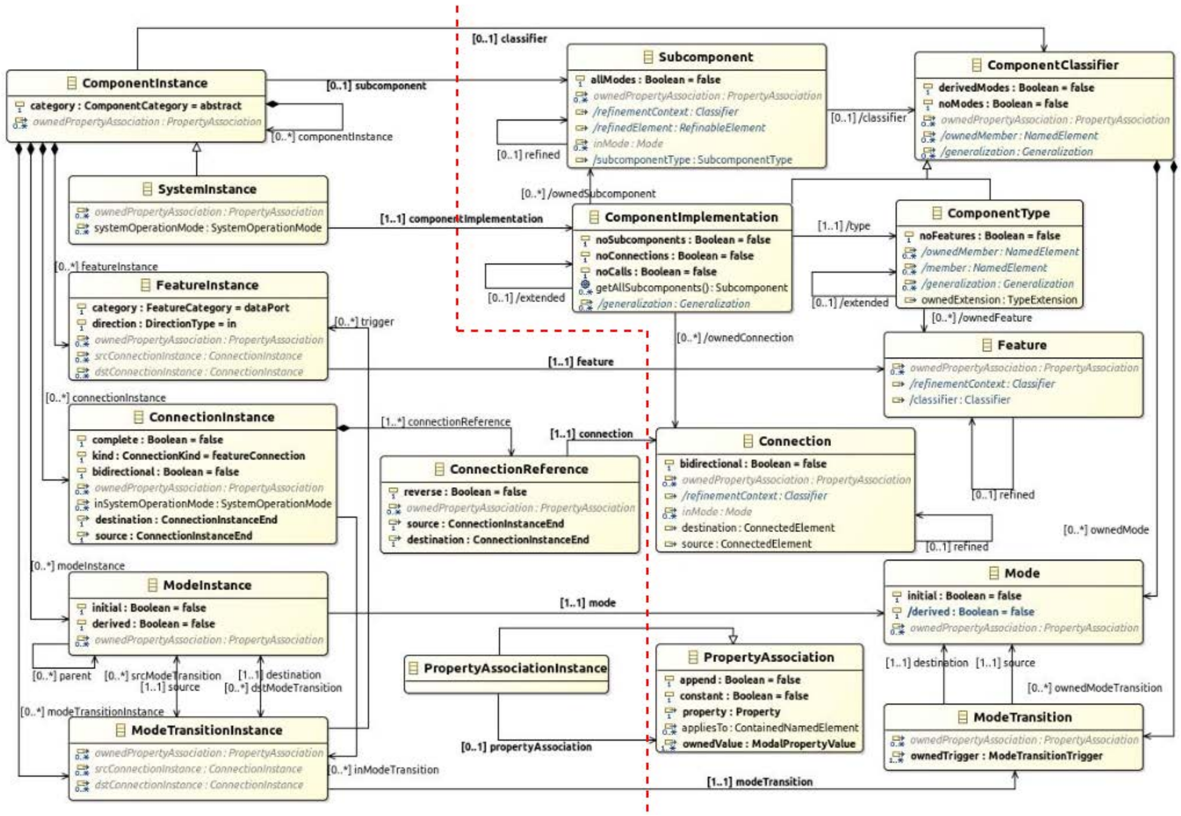

Figure 2: Subset of AADL. The red dotted line separates the Instance (left) and Declarative (right) metamodels.

Features from the super-*Type* can be refined. Similarly, *Subcomponents* and *Connections* in an *Implementation* can be refined subject to conditions. The structure of the system is emulated through a containment hierarchy of *Components* and their *Subcomponents* specified in the *Implementation*. *Properties* can be set to any element of an AADL model, and their visibility follow a complex search algorithm starting from the element itself to its classifier classifier extension, etc. In addition, *Properties* can be of *inherit* kind, meaning that a *Property* set in a parent *Component Implementation* will be inherited by all its *Subcomponents*.

These constructs allow the definition of complex cyber-physical systems in a highly modular fashion. There are many standard component libraries in the literature [13], which allow reuse of previously defined components. Users can import these component declarations located in *Packages* to be used into their own *Packages*.

2.1.3 Instance Metamodel. In OSATE, an *Instance* model is generated from a *Declarative* model, albeit with significant loss of information. A subset of the *Instance* metamodel is shown in the left part of Fig. 2. During *Instantiation*, a *System Implementation Classifier* is selected as the root *Declarative* element from which a *System Instance* is created and set as the root element of the newly created instance model. This element is computed by collecting all *Properties*, *Subcomponents*, *Features*, *Connections*, *Modes*, etc. from its *Implementation*, its super-*Implementations*, its *Type*, and its super-*Types*. Similarly, its *Subcomponents* are recursively instantiated to *Component Instances* to arrive at a simple tree-graph model of containment.

Due to the *Instantiation* procedure, information such as *Classifier* extensions, *Subcomponent*/*Feature*/*Connection* refinements, *Property* inheritance, *Subprogram* calls, *Requires-modes* clause, etc. are unavailable in an *Instance* model. One special characteristic of *Instantiation* is the formation of *Semantic Connections*, where a *Connection Instance* is formed from the ultimate source to the ultimate destination by following a sequence of *Connection* declarations on the *Declarative* model.

It should be noted that the name *Instance* metamodel may be confusing since it does not refer to the standard type-instance relationship in MDE. Indeed, there is only one *Instance* model for a given *System Implementation* of the *Declarative* model, but in general there will be several *Instance* models for several *System Implementations* so that different designs can be explored. The OSATE *Instance* metamodel can be seen as the semantic domain of constructs of the *Declarative* language used to modularize specifications such as *Component* extension, *Subcomponent* refinement and *Property* visibility. Nevertheless, we keep using these terms in this paper since they have been used for many years by the AADL community.

2.2 View-Update Problem

The *View-Update* is a classic problem in database theory appearing first in [2]. A *View* is a subset of the core database obtained as a

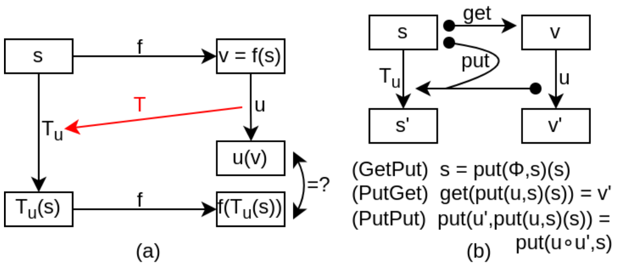

Figure 3: (a) Generalized *View-Update Problem* (b) Delta-based *Lens* with 3 'very-well-behavedness' laws adapted from [8]

result of a user query, and acts as an interface to the core database. It is often a security measure by allowing the *Viewer* access to only authorized information [5] generated for that *View*. In application of this problem to the model-driven domain, the core database is called the *Model State*, say s. In our case, the *Model State* is the *Declarative* AADL model. On the other hand, the *View* is the *Instance* model.

The *View-Update* formalisms illustrated in this section are standard within the community. They originated in [14], and are further reported in [6, 11]. We refer the reader to these articles for more information.

Mathematically, for *Model State* s, in the *Model Space*, say S (i.e. $s \in S$), the *View-Generating Function* $f : S \twoheadrightarrow V$, is a non-injective surjective function ($\twoheadrightarrow$ means surjective relation) over V, the *View Space*. The *View Space* is the set of all *Views*, v (i.e. $v \in V$). Therefore, $f(s) = v$. In our case, the *View-Generating Function* f, is the *Instantiation* model transformation that generates *Instance* models.

The *View-Update* $u \in U : V \longrightarrow V$ defines the modification of the *View*, where U is a complete set of possible *View-Updates*. The *View-Update* produces a refined/modified *View-State*, say v' ($\because v' = u(v)$). The ordering relation in U comes from priority-based ranking of the *View-Updates*. In our case, the *View-Update* is the set of modifications (automated or manual) carried out on the original *Instance* model, to produce the refined/modified *Instance* model.

An update u in the *View* has to be translated to the source *Model State*, s, to maintain consistency of information. Let the *Translation* be T, and the *Model-Update* corresponding to u be $T(u) = T_u : S \longrightarrow S$. The *Translation* T is the solution of the *View-Update* problem. The *Translation* T is said to "have no side effects", iff $\forall u \in U, f(T_u(s)) = u(f(s))$. In our case, this *Translation* is the task for OSATE-DIM.

There are many theories and solutions in the literature for the *View-Update Problem*. In the following, we discuss the theory of *Lenses*, which is the most relevant for the *View-Update Problem*.

2.2.1 State/Delta-based Backward Transformation. *Lenses* are an asymmetric Bidirectional Transformation (BX) framework [8]. Asymmetric meaning: in the two models being synchronized, one (*Instance*) is derived from the other (*Declarative*), and the *View-Generating Function* (*Instantiation*) is non-injective. Two functions, *get* and *put*, define the *Lens*. *get* is the *View-Generating Function* f, whereas *put* is the *Translation* T.

Most BX frameworks are state-based (rather than delta-based), where *put* takes states of models as input and ignores how updates were actually done. This is due to the fact that it was only recently realized that deltas could be valuable, but also because deltas are not always available since modifying tools may not have recorded them.

However the state-based approach introduces many synchronisation problems. It has been proven in the literature, $\exists u_1, u_2 \in U | u_1 \neq u_2, u_1(v) = u_2(v), T_{u_1} \neq T_{u_2}, f(T_{u_1}(s)) = f(T_{u_2}(s))$ where $f(s) = v$. This means there exist different *View-Updates* that have the same net result on the *View*, but a state-based framework fails to capture this difference since it ignores the update procedure. For example, an update-modification of an element can also be achieved through deletion and addition of the modified element.

The updated *Model State* s' can be significantly different for these two updates. This is an ambiguity that cannot be captured in state-based *Lenses*. Hence, we opt for the Delta-based framework that uses the precise update information u as input and outputs a corresponding T_u. In the Delta-based *Lens* the *put* uses information from the original *Model State*, s, and the *View-Update*, u, as input, to generate the *Model-Update*, T_u as shown in Fig.3(b). A "very-well behaved" *Lens* satisfies the three laws given in Fig. 3 (b). The laws are derived for a delta-based *Lens* from [8].

In this article, the *get* function of the BX *Lens* is defined by the current Java-based *Instantiation* forward transformation defined in OSATE, and the *put* function is the *Deinstantiation* performed by OSATE-DIM.

3 METHODOLOGY

3.1 Principles, Requirements and Assumptions

As shown earlier, the *Deinstantiation* of AADL models is a *View-Update* problem, which allows us to borrow concepts from this domain. From [7], for a given *Translation* T, we get the conditions for 'translatability' of a *View-Update*, which is a desirable characteristic. A *View-Update* u, is translatable to T_u if:

(1) there is a unique translated update T_u, corresponding to each u,
(2) there are no extraneous (unnecessary) updates in T_u, i.e. unnecessary side-effects on *Model State* s,
(3) there are no side-effects on the *View State* $f(T_u(s))$, $f(T_u(s)) = u(f(s))$,
(4) and u preserves model validity.

The uniqueness criterion is not trivial to meet since the *View-Generating Function* f is non-injective. Therefore, for *Deinstantiation* we need to find a unique $T_u \forall u \in U$ through incremental *Translations* given semantic/side-effect constraints that allow disambiguation of the multiple *Deinstantiation* choices. [7] stresses on the desire for T_u to be minimal to help accomplish requirement (2) for 'translatability'. This also aligns with our principle of making the least *Model Update* as possible, to avoid (undesirable) propagation of these modifications.

There are many *Lens Laws* in literature, which define the relationship between the *get* and *put* functions of the *Lens*. The literature suggests that *Lenses* be at least 'very-well behaved' to simplify analysis and composition (if required) of the *Lens*. Hence, it is desirable for the *Lens* to satisfy the three laws defined in Fig.3(b).

We also desire, given the complexity of AADL and various choices to be made for *Deinstantiation*, to give users flexibility in deciding the course of the *Deinstantiation* depending on their knowledge about the model being *Deinstantiated*. *Flexibility* is an important principle given the complexity of the language, and the methodology for providing flexibility in the OSATE-DIM approach will be described in Section 4.2.

We assume that:

(1) The root model element is a *System Implementation*,
(2) *Features* are only contained within *Components*, i.e. there are no *Features* within *Feature Groups* (*Feature Groups* are collections of *Features* which can be connected as a single unit outside the *Component*),
(3) The user ensures *View-Updates* preserve semantic consistency. If the updated *View* does not satisy semantic constraints, OSATE-DIM and OSATE return errors while serializing the *Declarative* model during the model save operation.

3.2 Running Example

Our running example is a simplified version of the *sampled-communications* example project from the RAMSES tool. Its AADL *Instance* specification (left part of Fig. 4) consists of a system containing a *Memory Subcomponent* 'the_mem', a *Processor Subcomponent* 'the_cpu', and a *Process Subcomponent* 'the_proc' having two *Thread Subcomponents*, 'the_sender' and 'the_receiver' that communicate with each other through a *Port Connection*. It is a model of a simple producer-consumer pattern for which implementation code can be automatically generated for different operating systems platforms.

The *Declarative* model of *sampled-communications* is shown on the right part of Fig. 4. Traces from the *Component Instances* to their *Classifiers* on the *Declarative* side are represented as red edges, whereas traces from *Component Instances* to their *Subcomponent* definitions on the *Declarative* side are shown as blue edges.

3.3 *View-Update* Translation Rules

In this Section, we describe the *Model Update* T_u corresponding to each *View-Update* u, in the set of updates U, currently supported by OSATE-DIM. This is done through the running example. Due to lack of space, we cannot describe the rules for *Deinstantiation* of all kinds of *Instance* objects. However, for an in-depth analysis of the *Translation* rules associated with OSATE-DIM, we refer the reader to the rules webpage[5].

3.3.1 Updating a *Component Instance's* Name. -

Example: Change the name of 'the_cpu' *Component Instance* to 'new_cpu'.

The simplest *Translation* of such a *View-Update* is to change the name of the corresponding *Subcomponent* 'the_cpu' contained within the *Component Implementation* 'main.linux'.

3.3.2 Adding a new *Component Instance*. -

Example: Addition of a third *Thread Component Instance* named 'the_viewer' to 'the_proc'.

To add the new *Component Instance*, the simplest *Translation* is to add a *Thread Subcomponent* in the *Implementation* of 'the_proc', i.e. 'proc.impl'. On the *Declarative* side, a *Subcomponent* is typed by a *Component Classifier*. Consequently, a new *Component Type* for 'the_viewer' should be created within the *Package* 'dim_test_experiment' to define 'the_viewer'. Since 'the_viewer' has no *Subcomponent* or *Connection*, only a *Component Type* and not a *Component Implementation* is created. If more such elements are added to 'the_viewer' in later *View-Updates*, a *Component Implementation* for 'the_viewer' may be created then.

If the new *Thread Component* is defined incrementally, i.e. it is first added as a child to 'the_proc', and then its characteristics like name, classifier, and category are set, then the definition of these characteristics triggers rules of component updates, and not component creation. Instead, if the whole *Component Instance* is first constructed separately, and then attached to 'the_proc', then only the component creation transformation rules are triggered.

3.3.3 Changing (not refining) a *Component Instance's* Category. -

Example: Change the category of 'the_cpu' *Component Instance* to virtual processor.

In this case, the simplest *Translation* is to replace the *Processor Subcomponent* 'the_cpu' with a *Virtual Processor Subcomponent* with the same name, and other characteristics.

Consequently, to define the structure of the newly created *Virtual Processor Subcomponent*, a new *Virtual Processor Type*, say 'the_virtual_cpu', has to be created in the *Package* 'dim_test_experiment'. 'the_virtual_cpu' should contain all properties/features that are contained in previous the classifier i.e. 'cpu.impl', to avoid any unnecessary side-effects.

3.3.4 Refining a *Component Instance's* Category. -

Example: Assume there is another *Memory Subcomponent* 'new_mem' in 'main_linux_Instance' with an abstract *Classifier*. Change the category of 'new_mem' to memory.

AADL supports the partial specification of *Component Classifiers*, as a template, which can be completed according to the use-case. For *Components*, their category is said to be refined when changed from abstract to concrete category like thread, process, etc.

In the case of refinement (i.e. when the old category of the *Component* being refined was abstract), the *Translation* should be different to adhere to our principle of maximum information preservation as described next. In this case, the *Translation* is to create a new *Classifier* which extends the *Classifier* of the parent of 'the_cpu' i.e. to create a new *Component Implementation* 'main_linux_Instance_new', such that it extends 'main_linux_Instance'. Then if possible, we can refine the category of the 'the_mem' *Subcomponent*.

3.3.5 Deleting a *Component Instance*. -

Example: Deletion of the *Component Instance* 'the_mem'.

The simplest *Translation* in this case is to delete the 'the_mem' *Subcomponent* from 'main.impl'. We do not need to delete the corresponding *Classifiers* ('mem.impl' and 'mem'), since an important value for us is maximum information preservation as described in Section 3.1.

3.4 Avoiding Undesirable Change Propagation

Consider the following three statements:

(1) If a *Classifier* contributes to the definition of more than one different *Component Instances*, a *Model-Update* in the *Classifier* due to a *View-Update* in one of the *Component Instances*

[5] https://mem4csd.telecom-paristech.fr/blog/index.php/osate-dim/rules/

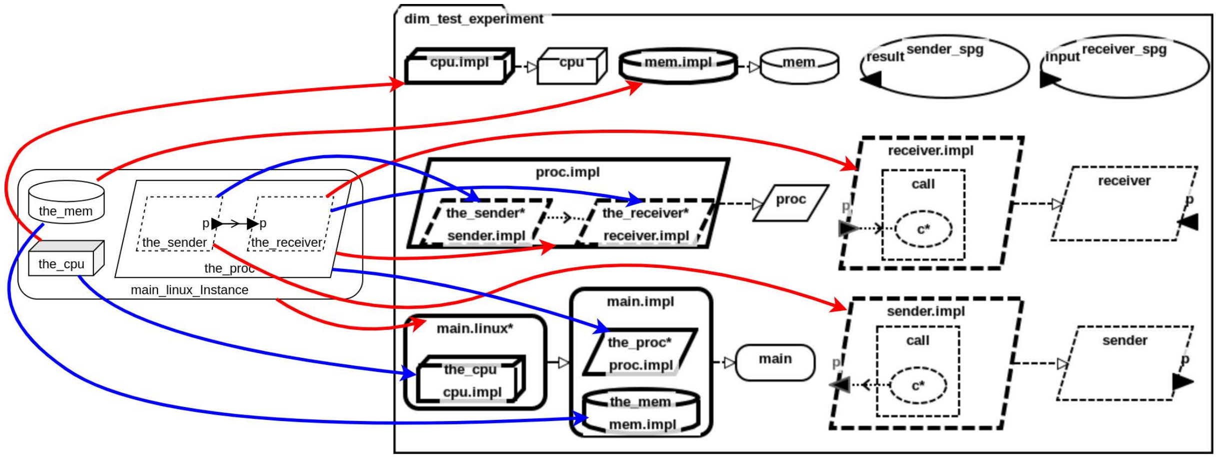

Figure 4: Running example. Blue edges represent *Subcomponent* traces and red edges represent *Classifier* traces from the *Instance* (left) to *Declarative* (right) model. Refer the AADL Graphical Syntax for more information.

will propagate (often undesirably) to the other *Component Instances*. An example of this for the running example is shown in Fig. 5 where a second process component 'the_proc2' with the same classifier as 'the_proc' has been added to improve fault tolerance.

(2) A change in the *Classifier* of a *Subcomponent*, is also a change in the structure and characteristics of the *Classifier* which contains the *Subcomponent*. For example, a deletion of the *Feature* 'p' within 'the_sender' in 'the_proc' (a change in 'sender' *Component Type*) is also a change in 'the_proc'. Consequently, the corresponding feature 'p' within 'the_proc2' will also be deleted.
(3) Statement (2) can not only be applied to a parent-child relations, but also to ancestor-child relations, i.e. a change in the *Classifier* of a child *Subcomponent* is not only a change in the characteristics of the containing *Classifier*, but also all its ancestors (through the *Implementation-Subcomponent* containment hierarchy), i.e. the change is not just propagated to the immediate parent *Component*, but all the containing ancestor *Components* as well. In the example, this means, the deletion of *Feature* 'p' is also a change in the *Classifier* of 'main_linux_Instance' i.e. 'main.linux'.

The three statements imply that the *Translations* described in Section 3.3 are not enough to ensure that the *Model-Updates* do not propagate undesirably to other *Components*. To handle this, and avoid undesirable propagation, a special method is built into OSATE-DIM, which is called within all *Translation* rules.

It is undesirable to change the definition of *Classifiers* in component libraries. Hence, the method crawls the *Declarative* side to find the *Classifier* at the highest level that has been reused multiple times, or is from a *Component* library (tracked using OSATE-DIM *Property Set*, see Section3.4.1). Once identified, it goes on to copy each *Classifier* at a lower level, and recreates the extension and inheritance relations, to ensure there is no undesirable propagation.

3.4.1 'DIM_Properties' Property Set. OSATE-DIM provides a *Property Set* containing 3 properties:

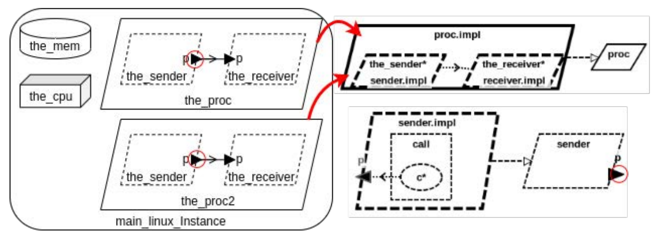

Figure 5: Example showing undesirable update propagation

(1) Is_Library_Classifer : Boolean *Property* to tag *Classifiers* which are part of a Component Library.
(2) Is_Classifier_Library : Boolean *Property* to tag if a *Package* is part of a *Component Library*.
(3) DIM_Classifier : see Section 4.2.3

4 TOOL IMPLEMENTATION

The proposed tool, OSATE-DIM, has been implemented as a set of Eclipse IDE-based plugins. The source code for the tool is available in a GitLab repository[6]. Users can install this tool into their Eclipse installations through the update-site[3].

4.0.1 VIATRA. OSATE-DIM uses VIATRA[7] for executing graph-transformations. The *Instance* and *Declarative* models are graphs where objects are nodes, and their relationships are edges. VIATRA is a scalable reactive framework, which allows for incremental execution of transformations. Incrementality is offered by separating the pattern matching and transformation steps. A pattern matcher identifies patterns, and whether they are newly created, updated, or deleted. The transformation uses information of each pattern (and its state of creation, updation, or deletion) as input. When the state of a pattern changes, the corresponding transformation rules are 'fired'. A new match for a pattern is a creation, the disappearance of a match is a deletion, and a change in the properties of

[6]https://gitlab.telecom-paris.fr/mbe-tools/osate-dim/
[7]https://www.eclipse.org/viatra/

objects in the match is an update. Patterns are specified through a Domain-Specific Language (DSL) called Viatra Query Language. The transformations are written in an Xtend-based DSL providing a model manipulation Application Programming Interface (API).

Our choice of VIATRA was guided by our recent benchmark of incremental model transformation tools with an industrial AADL case study [18]. Out of the four benchmarked tools, our benchmark indicated that VIATRA was the best choice given its maturity and expressiveness, especially compared to TGG approaches for which shortcomings were found regarding expressivity. This is particularly important given the richness and complexity of AADL. Besides, implementation constraints were given favoring a Java-like model transformation language (the current instantiation transformation is also implemented in Java) implemented with the Eclipse Modeling Framework on which OSATE is based.

4.1 *Deinstantiation* Scenarios

OSATE-DIM supports *Deinstantiation* in many different scenarios. A *View-Update*, u may be an in-place or an out-of-place transformation. Consequently, the *Model-Update*, T_u should be an in-place or out-of-place transformation respectively. The transformation rules for all the scenarios are the same. The difference is only in the interface to the transformation rules.

4.1.1 State-based. Firstly, OSATE-DIM supports a state-based deinstantiation scenario as shown in Fig.6.(i). This state-based case is derived from the delta-based case as a pure backward transformation when it is assumed there is no original *Declarative* model s. It is basically a $put(v_\phi,\phi)$, where $v_\phi \in V_\phi \subset U|v_\phi(\phi) = v$. V_ϕ is the set of view-updates v_ϕ, which result in the updated *View* v, when applied on empty *View* ϕ.

In this scenario, OSATE-DIM takes all information from v and creates the simplest *Declarative* model from that information. By comparing the output of this scenario with the original *Declarative* model (if it exists) used to create the *View*, the user can also understand what kind of information is lost in the *Instantiation* transformation.

4.1.2 Delta-Inplace. In an in-place transformation, the changes are made directly in the model. Hence, for an in-place *View-Update* u, the corresponding translated *View-Update* T_u has to be in-place as well. That is, it should make changes directly on the *Declarative* model.

OSATE-DIM detects the changes in this scenario, using VIATRA's built-in transformation engine that listens for changes to query patterns in the *View* model as shown in Fig.6.(ii).

4.1.3 Delta-outplace. The scenario where an entirely new *Instance* model (with modifications) is computed from the base *Instance* model, is the out-place scenario as shown in Fig.6.(iii). In this case, a new corresponding *Declarative* model should be constructed reflecting the modifications, instead of directly modifying the *Declarative* model.

In this scenario, the *View-Updates* can be computed from differences between the new and original *Instance* models. OSATE-DIM provides the delta-trace model to define trace relations between an instance model and its out-of-place update. The delta-trace model borrows concepts from EMF Change [21] to store information regarding the specific change operations that were performed on the *Instance* model to lead to the new *Instance* model.

4.1.4 Delta-trace Model. The Delta-trace meta-model (provided in OSATE-DIM) consists of an 'Aaxl2AaxlTraceSpec' class that relates the roots of the two instance models, and contains the 'Aaxl2AaxlTrace' trace. Each trace relates elements from the original to refined instance models. An 'Aaxl2AaxlTrace' has three properties: 'ObjectsToAttach', 'ObjectsToDetach', and 'ObjectChanges' which respectively represent addition, deletion, and modification of objects. 'ObjectsToAttach' is a non-containment property that references objects in the new *Instance* model that have been added. Similarly, 'ObjectsToDetach' refers to objects in the old *Instance* model that have been removed in the *View-Update*. 'ObjectChanges' contains FeatureChanges (an object from the EMF Change meta-model) to represent updates in values of features of objects.

4.2 Preferences and Utilities

As discussed in Section 3.1, flexibility is an important principle for us to give the user more control over the *Translation*. We accomplish this through user-defined preferences which decide the course of the *Deinstantiation*. Fig.7 displays the preferences dialog for OSATE-DIM. Some preferences enable model manipulation utilities that ease the modification of *Instance* models. Other preferences enable users to direct the course of *Deinstantiation* according to their knowledge of the models.

4.2.1 Instance Model Property Inheritance Utility. To simplify the modification of *Instance* models, OSATE-DIM provides a utility which automatically adds inherited properties to newly-created *Instance* objects. If the parent object of a newly-created *Instance* object contains some *Property Association*, if the *Property* is inheritable, and if the *Property* applies to the child as well, then it inherits this property. In the *Instance* model, the *Property Association* is copied into the child (while in the *Declarative* model, it is simply inherited from the parent's definition without copying). This is a requirement to ensure semantic validity of the *Instance* model. Instead of the users having to manually copy the *Property Association* from the parent to the child, OSATE-DIM does it automatically, if the corresponding preference is set to TRUE. If TRUE, when a new child *Instance* element is created, OSATE-DIM checks for inheritable properties in the parent, and automatically copies them to the child element.

4.2.2 Instance Model Mode Inheritance Utility. Similar to *Property Associations*, *Modes* are also inherited. If a parent object is only active in certain *Modes*, then its children can only be active in a subset of these *Modes*. Hence, if the corresponding preference is set to TRUE, OSATE-DIM automatically copies these *Modes* to the child object. The user can choose to delete some/all of these (automatically added) modes later, according to the intended behavior of the object.

4.2.3 Modification of multiply-used Classifier*.* A classifier can contribute to the definition of more than one *Component Instances*. In such a case, an *Update* on a classifier may unwantedly propagate to other *Component Instances*. To avoid this, OSATE-DIM provides an

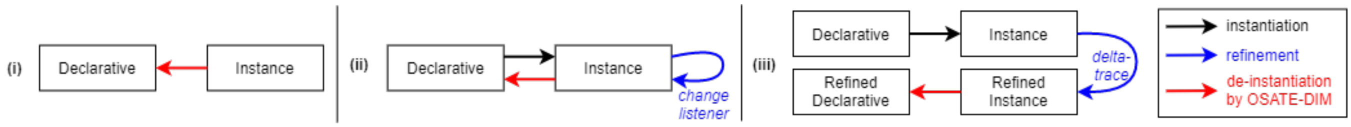

Figure 6: Possible AADL deinstantiation scenarios supported by OSATE-DIM

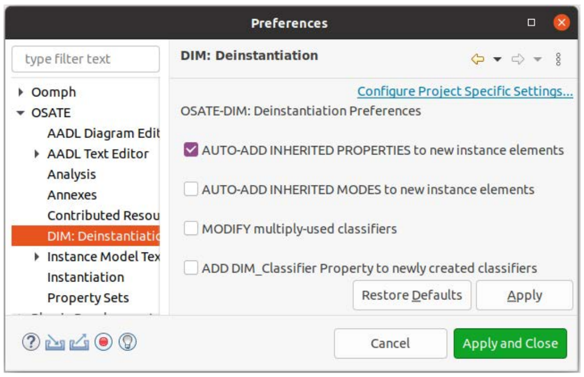

Figure 7: OSATE-DIM preferences menu

option to the user, such that *Updates* on multiply-used *Classifiers* are not performed directly, but instead through extension of the *Classifier* (in the case of addition update), or copying the *Classifier* and performing *Updates* on the copy (in the case of modification and deletion updates), similar to the flow for library support described in Section 3.4.

Depending on the user's knowledge: if the *Declarative* model is simple and no classifier contributes to the definition of multiple *Component Instances* then the user can set it to TRUE/FALSE (it won't make any difference); if the *Declarative* model is complex with multiply-used *Classifiers*, and if the user does not want *Translated Updates* to propagate, the user should select TRUE; on the other hand, if the user wants *Translated Updates* to propagate, then the user should select FALSE for the corresponding preference.

4.2.4 Addition of DIM_Classifier Property to new Classifiers. For additional 'book-keeping' of OSATE-DIM's actions, we provide a Boolean Property within the 'DIM_Properties' *Property Set* called 'DIM_Classifier'. If the corresponding preference is set to TRUE, every time a new *Classifier* is created by OSATE-DIM, an additional 'DIM_Classifier' *Property* is added to it, and set to TRUE.

5 VALIDATION

We empirically verify the 'very-well behavedness' (defined in Section 2.2.1) of the BX *Lens* composed of OSATE-based *Instantiation* and OSATE-DIM-based *Deinstantiation*. All the tests performed in this section are available as JUnit tests in the test package in the OSATE-DIM code repository[6].

5.0.1 GetPut Law. The GetPut law states 'if no *View-Update* has been performed on the *View* v, then the *put* function should return the identity function for *Model-State* i.e. there should be no difference between the original and updated *Model State*'.

This is verified by the incremental *Deinstantiation* in OSATE-DIM. If no *View-Update* u is performed, no *Translated Update* T_u is performed either.

5.0.2 PutGet Law. The PutGet Law states 'a well-behaved *Lens* has no side-effects', i.e. the *Instantiation* of the updated *Model-State*, $f(s')$, should be equal to the updated *View-State* u'. This is also described in Fig.3.(a).

We verify this through test cases generated for each of the *View-Updates* mentioned in Section 3.3. The updated *Model-State* is *Instantiated* and compared with the updated *View-State* using EMF Compare. In addition, tests are also generated for the case studies described in Sections 5.1 and 5.2.

5.0.3 PutPut Law. The PutPut Law states that '*View-Updates* should completely overwrite the effect of the previous *View-Update*'. So, the effect of two *puts* in a row, should be the same as just the second. Mathematically, if there is a *View-Update* u, followed by u', then, $T_{u \cdot u'}(s) = T_{u'}(T_u(s))$.

OSATE-DIM is incremental and reactive, where each *View-Update* is considered atomic. Hence, even if *Deinstantiation* is performed after composing the *View-Updates*, the *Deinstantiation* would involve the same steps, in the same order, had the *Deinstantiation* been done reactively. Hence, the two *Declarative* models will be the same.

5.1 Case Study: MC-DAG

The Mixed-Criticality Directed Acyclic Graphs (MC-DAG) framework developed by our group computes scheduling tables for AADL systems with mixed-critical tasks. This case study from the MC-DAG benchmark [16] is a system with a multi-core processor and a process. The process *Component* has eight threads, each with a different task. There are two *System Operation Modes* for low-criticality and high-criticality in the case of a failure. The transitions from one mode to another are relayed through *Connections* from the processor to the process.

The *View-Update* in this case study is the addition of static scheduling tables for each *Thread*. Such tables are modeled in AADL using the RAMSES-specific 'Execution_Slots' Property, so that RAMSES can take those into account when generating OS configuration files during code generation. These tables specify for each *Thread*, its binding to one of the two cores of the processor and its execution start and stop times. Overall, there is an addition of nine rich *Property Associations*, which are not just value-based, but also contain references to other *Instance* objects.

5.2 Case Study: RAMSES

RAMSES is an AADL-based tool available as an additional OSATE component and developed by our group for the automatic generation of C code for POSIX, ARINC653, and OSEK-compliant

Operating Systems (OS). Starting from an *Instance* model, RAMSES refines the model by adding details for a specific OS platform as selected by the user. From this refined *Instance* model, analyses can be performed, which are typically more precise given the lower level of abstraction of the model.

In its current version, RAMSES must first create a new *Declarative* model from of the selected *Instance* model, to which the OS-specific details are added. A new *Instance* model is then computed by OSATE from the newly created *Declarative* model. The RAMSES refinement model transformation, implemented with the ATL model transformation tool and its EMFTVM (EMF Transformation Virtual Machine) [22] virtual machine is very complex and hard to maintain and evolve. In this context, OSATE-DIM will be extremely useful as it will allow simplifying the transformation by avoiding the need for this transformation to create the *Declarative* model. Hence, RAMSES will only need to update the *Instance* model and OSATE-DIM will take care of creating the corresponding *Declarative* model. Versions of these simplified RAMSES transformations, processing only *Instance* models, have already been developed for our benchmark of Incremental Model Transformation Tools [18].

Following this, we have used a simple AADL producer-consumer communications module instance model, refined for a Linux-based platform by transforming *Port Connections* to *Shared Data Accesses* and by adding several *Data Components* and RAMSES-specific *Properties* as second case study for OSATE-DIM. The *View-Updates* in this case consist of the addition of 41 *Data Components* to a process component, which are shared by two threads. The *Port Features* interfacing the two threads with each other are changed to a *Data Access* kinds. New *Data Access Connections* are also added between the shared *Data Component* and the threads. The added *Data Components* have varying numbers of *Properties*, and the total number of newly added properties is 122. This RAMSES case study is included as a much more complex example than the simple *View-Updates* performed in the MC-DAG case study.

6 DISCUSSION

While developing OSATE-DIM we experienced some shortcomings and unnecessary complications of AADL that need to be discussed within the community:

(1) The current *Declarative* metamodel is highly complicated, since it includes specific classes for each category of *Component*, *Feature*, and *Connection*. This was done to introduce constraints on component composition. The Object Constraint Language (OCL) could be used to expressed these constraints instead of component category subclasses.
(2) In the current version of OSATE, there is no class in the *Instance* model to represent *Subprograms* and *Subprogram Calls*. Information for these elements is only available from the *Declarative* model.
(3) Another issue relates to *Annexes*. The core AADL language can be extended by embedding sub-languages such as the Behavioral Annex (BA) and the Error Model Annex (EMV2). Currently those are not represented in the *Instance* model although it is currently under development for EMV2.

6.1 Envisioned Utility of OSATE-DIM

OSATE-DIM has been developed keeping in mind the needs of both AADL-based tool developers and AADL-users. Support for various scenarios allows for integration of OSATE-DIM into a wide array of workflows for AADL-based research and development. For users, OSATE-DIM is envisioned to provide incremental model-synchronization capabilities, which ensures no loss and consistency of information. It also simplifies the modification of AADL models for users, who previously had to make changes in the *Declarative* models directly.

For the AADL-based tool developer, OSATE-DIM is useful for them by simplifying the development of their tool. Instead of having to design complex algorithms to modify the *Declarative* model at the correct location, the developer can simply implement algorithms to modify the *Instance* model. They can integrate OSATE-DIM within this modification (whether in-place our out-of-place) to automatically perform the synchronization with the quality of being the simplest changes having least loss of knowledge.

This especially will simplify the migration of tools to the AADL Version 3 when it will be released. AADL targets significant changes of the *Declarative* metamodel on the lines of suggestion (1) in Section 6. If developers have integrated OSATE-DIM into their pipeline being therefore isolated from the *Declarative* model, they need not worry about such updates of the *Declarative* language, since OSATE-DIM after having been updated for AADL V3 will take care of interfacing tools with *Declarative* models.

6.2 Generalization of Concepts to OOP

While the methodology we propose for the *Deinstantiation* of AADL models in this paper was clearly developed to solve a real problem, we envision that it could also be useful for High-Level Languages (HLLs), since constructs of these languages can be related to those of the AADL *Declarative* model. The compiled source code of HLLs where several flattening operations occur can be related to the generation of *Instance* models. For instance, languages such as Rust, Swift, and C/C++ are compiled to Low Level Virtual Machine Intermediate Representation (LLVM-IR)[8]. LLVM-IR is a typed-bitcode that unifies multiple programming languages in a common representation which can be used with the LLVM Optimizer tool[9].

Compilation of most HLLs to LLVM-IR is non-injective surjective[10] as shown in Fig.8. Since high-level constructs such as inheritance and virtual functions are not supported by LLVM-IR, the inheritance hierarchy needs to be 'flattened' to a simpler structure: virtual functions are simplified to function pointers with virtual function dispatch tables resulting in different inheritance hierarchies being flattened to similar LLVM-IR structures. An OO-class, its member variables and its functions are translated to a single IR-struct containing variables and IR-functions operating on the IR-struct. The variables of an OO-class are copied into the IR-structs corresponding to both, the base OO-class and its derived classes. The IR-functions corresponding to the base OO-class can use the IR-struct corresponding to the derived OO-class to access its members.

[8] https://llvm.org/
[9] https://llvm.org/docs/CommandGuide/opt.html
[10] https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/

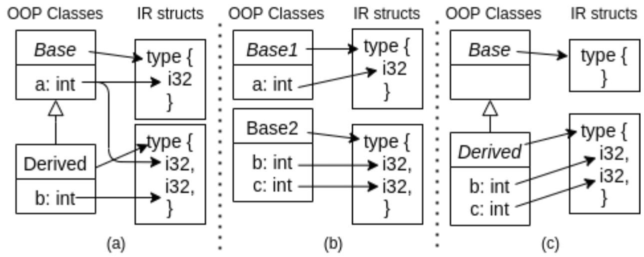

Figure 8: OO Classes mapping to LLVM-IR structs

After optimizations have been applied, the IR is no longer consistent with the source code and is inconvenient to debug. Decompiling the IR to source language can improve accessibility for debugging [10]. since source HLLs are more widely known and more accessible than LLVM-IR. Besides, de-compilation from IR to source language could allow translation between languages (ex. C++ to Rust).

Modification of the inherited members of a derived IR-struct can be *Translated* as changes local to the derived OO-class or a change inherited from the parent class. Deciding the correct *Translation* in such a case is a challenge. We can illustrate the similarity between this problem and our methodology by relating the OO concepts to AADL classifiers:

- Interface → *Component Type*: Interfaces define the exposed structure of the implementing classes and its member accessors and public functions. This matches the Component Types which expose the ports and properties.
- Class/Struct → *Component Implementation*: Classes and Structs in OO Languages contain the actual data members and function implementations for any Interfaces being implemented.
- Member Variables → *Subcomponents*: Member variables are instances of other existing interfaces or classes. They relate to subcomponents in AADL and any calls to their methods map to AADL connections.
- Functions → *Subcomponents*: Functions are internally stored as pointers with or without virtual function dispatch tables. They are data members and thus mapped to Subcomponents.
- Overriding → Refinement: Derived classes can override function implementations in the base class. This can be matched to the refinement/*View-Update* in AADL.

By modifying the *Deinstantiation* methods discussed in the paper to apply to the OO constructs as illustrated above, we can extend the concepts to apply to the decompilation methods. Additionally, the State-Based method similar to 4.1.1 can be applied to LLVM-IR that is generated from one HLL to decompile into another HLL. LLVM languages such as Objective-C, which are phased out can be decompiled to newer languages like Swift in order to update their codebase and enhance language interoperability. This could also be useful for the specific case of Rust ↔ C++ given that Rust may become a replacement to C++ for embedded systems.

6.3 Related Work

Many tools apply the concept of views to MDE. EMF Views [3] uses an SQL-like DSL to define a virtualization engine. It looks at views as non-concrete entities, and implements them as virtualization of real base models so that there is no data duplication. Thus, changing data in the view implies change of the data in the base model.

OpenFlexo [4] is a tool for homogeneously handling and relating data from various sources. As soon as a view is computed, it is connected with different base models for synchronization. The synchronization is conceptually similar to EMF Views.

ModelJoin [12] is a tool for the creation of heterogeneous models. Its DSL is used to define not just the elements of the view, but also the meta-model of the view. Support for editability inside the views is provided using OCL constraints.

Orthographic Software Modelling (OSM) [1] is a hub-and-spoke architecture-based approach and tool that allows for the definition of multiple views from a Single Underlying Model (SUM). The definition is through a unique bidirectional transformation between the SUM and each view. Vitruvius [15] is based on the OSM approach but instead of a SUM, it uses a Virtual-SUM that is a non-invasive combination of many legacy metamodels. Flexible view definition allows restriction of possible view updates. Updating a view results in execution of corresponding synchronizing transformations.

These methods provide light-weight backward transformations, through virtualization using invertible transformations to define virtual views (EMF Views, OpenFLexo), or through constraints and restrictions on possible model edits (ModelJoin, OSM, Vitruvius). They are not as flexible and as specifically made for AADL as OSATE-DIM and often require to severely limit the class of possible updates to the view to guarantee well-behavedness.

7 CONCLUSION AND FUTURE WORK

We presented our approach for the automatic *Deinstantiation* of OSATE *Instance* models. It will be very useful for the AADL community, since it allows model processing tools to only manipulate the *Instance* model and have their modifications automatically synchronized with the original *Declarative* models. This is achieved thanks to an incremental model transformation implemented with VIATRA solving this *View-Update* problem for AADL. The presented approach can be generalized for other similar problems such as decompilation of programming languages' intermediate representations for debugging.

Future work for OSATE-DIM will consist of completing the out of place scenario to make use of model change information (deltas) stored in the OSATE-DIM-provided trace models, on which processing tools can write their changes as they are performed. Besides, the current OSATE *Instantiation* transformation written in plain Java could be redeveloped in VIATRA to make it a truly BX *Lens*. Reducing the assumptions listed in Section 3.1, to increase the scope of usability of OSATE-DIM, as well as providing further flexibility to the user to alter the *Deinstantiation* by providing preferences such as choice of level of *Deinstantiation* (whether the user wants *Model-Updates* to occur at the *Subcomponent*, *Implementation*, or *Type* abstraction level) are also part of future work.

ACKNOWLEDGMENTS

This work has been supported by US Army CCDC-ATLANTIC DEVCOM.

REFERENCES

[1] Colin Atkinson. 2010. Orthographic Software Modelling: A Novel Approach to View-Based Software Engineering. In *Modelling Foundations and Applications*, Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–1.

[2] F. Bancilhon and N. Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (dec 1981), 557–575. https://doi.org/10.1145/319628.319634

[3] Hugo Bruneliere, Jokin Garcia, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models, Vol. 9381. https://doi.org/10.1007/978-3-319-25264-3_23

[4] Erik Burger, Jörg Henß, Martin Küster, Steffen Kruse, and Lucia Happe. 2014. View-Based Model-Driven Software Development with ModelJoin. *Software and Systems Modeling* 14 (2014), 1–24. https://doi.org/10.1007/s10270-014-0413-5

[5] Erik Johannes Burger. 2013. Flexible Views for View-Based Model-Driven Development. In *Proceedings of the 18th International Doctoral Symposium on Components and Architecture* (Vancouver, British Columbia, Canada) *(WCOP '13)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/2465498.2465501

[6] Haitan Chen and Husheng Liao. 2011. A Survey to View Update Problem. *International Journal of Computer Theory and Engineering* 3 (2011), 23–31. https://doi.org/10.7763/IJCTE.2011.V3.278

[7] Umeshwar Dayal and Philip Bernstein. 1978. On the Updatability of Relational Views. *Proc. 4th VLDB Conf.*, 368–377.

[8] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10 (2011). https://doi.org/10.5381/jot.2011.10.1.a6

[9] Peter Feiler, David Gluch, and John Hudak. 2006. *The Architecture Analysis & Design Language (AADL): An Introduction.* Technical Report CMU/SEI-2006-TN-011. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[10] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *2011 18th Working Conference on Reverse Engineering*. 347–356. https://doi.org/10.1109/WCRE.2011.49

[11] Enrico Franconi and Paolo Guagliardo. 2012. The View Update Problem Revisited. *CoRR* abs/1211.3016 (2012). arXiv:1211.3016 http://arxiv.org/abs/1211.3016

[12] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. 2013. Conceptual interoperability through Models Federation. (10 2013).

[13] Jerome Hugues. 2013. AADLib, a library of reusable AADL models. In *2013 SAE AeroTech Congress & Exhibition*. 1–8. https://doi.org/10.4271/2013-01-2179

[14] Arthur Keller. 1986. The Role of Semantics in Translating View Updates. *Computer* 19, 1 (1986), 63–73. https://doi.org/10.1109/MC.1986.1663034

[15] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software* 171 (2021), 110815. https://doi.org/10.1016/j.jss.2020.110815

[16] Roberto Medina, Etienne Borde, and Laurent Pautet. 2018. Availability enhancement and analysis for mixed-criticality systems on multi-core. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1271–1276. https://doi.org/10.23919/DATE.2018.8342210

[17] Roberto Medina, Étienne Borde, and Laurent Pautet. 2018. Scheduling Multi-periodic Mixed-Criticality DAGs on Multi-core Architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 254–264. https://doi.org/10.1109/RTSS.2018.00042

[18] Hana Mkaouar, Dominique Blouin, and Etienne Borde. 2022. A benchmark of incremental model transformation tools based on an industrial case study with AADL. *Software and Systems Modeling* (2022), 1–27.

[19] Smail Rahmoun, Asma Mehiaoui-Hamitou, Etienne Borde, Laurent Pautet, and Elie Soubiran. 2019. Multi-objective exploration of architectural designs by composition of model transformations. *Software & Systems Modeling* 18 (02 2019). https://doi.org/10.1007/s10270-017-0580-2

[20] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. 2004. Cheddar: A Flexible Real Time Scheduling Framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies* (Atlanta, Georgia, USA) *(SIGAda '04)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1032297.1032298

[21] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[22] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. 2011. Towards a general composition semantics for rule-based model transformation. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 623–637.