Model-Driven Techniques to Enhance Architectural Languages Interoperability

Davide Di Ruscio, Ivano Malavolta, Henry Muccini,
Patrizio Pelliccione, and Alfonso Pierantonio

University of L'Aquila, Dipartimento di Informatica
{davide.diruscio,ivano.malavolta,henry.muccini,
patrizio.pelliccione,alfonso.pierantonio}@univaq.it

Abstract. The current practice of software architecture modeling and analysis would benefit of using different architectural languages, each specialized on a particular view and each enabling specific analysis. Thus, it is fundamental to pursue architectural language interoperability. An approach for enabling interoperability consists in defining a transformation from each single notation to a pivot language, and vice versa. When the pivot assumes the form of a small and abstract kernel, extension mechanisms are required to compensate the *loss of information*. The aim of this paper is to enhance architectural languages interoperability by means of hierarchies of pivot languages obtained by systematically extending a *root* pivot language. Model-driven techniques are employed to support the creation and the management of such hierarchies and to realize the interoperability by means of model transformations. Even though the approach is applied to the software architecture domain, it is completely general.

1 Introduction

Architecture descriptions shall be developed to address multiple and evolving stakeholders concerns [1]. Being impractical to capture all concerns within a single, narrowly focused Architectural Language (AL) [2], i.e., a form of expression used for architecture description [1], we must accept the co-existence of different domain specific ALs, each one devoted to specific purposes. The use of various ALs requires interoperability among them since bridging the different descriptions to be kept consistent and coherent is of paramount relevance [3]. The need of interoperability at the architecture level is clearly demonstrated by international projects like Q-ImPrESS [4], and ATESST [5] where correspondences among different languages have to be created and maintained.

An approach for enabling interoperability among various notations which is recently getting consensus in different application domains (e.g., [6,7]) consists in organizing them into a star topology with a pivot language in its center: in these cases *interoperability is enabled by defining a transformation from each single notation to the pivot language, and vice versa*. Thus, the pivot language acts as a bridge between all the considered notations and avoids point-to-point direct transformations among them. While how to build a pivot language is still a craftsman activity, two different trends can be noted: (i) building a (rich) pivot language that contains each element required by any AL, like in the Q-Impress project, and (ii) building a (small) kernel pivot language containing a set of core elements common to most of the involved ALs, like in KLAPER [8]. On one hand, the adoption of a rich pivot language tends to reduce the *loss of information* in the pivot-based transformation process from one AL to another. On the other hand, the use of a kernel pivot may give rise to loss of information, since concepts in some of the ALs might be missing in the pivot language (due to the kernel pivot language minimality).

The use of a rich pivot is ideal when ALs have to be related under a closed-world-assumption, i.e., when the set of ALs to be used is a-priori defined. However, a rich pivot difficultly *scales* when new ALs are introduced in the star topology: the rich pivot has to be updated to cover newly introduced concepts. This is an error-prone task that could easily introduce inconsistencies within the pivot. In such a scenario, while the kernel pivot solution is more scalable (since the kernel pivot language is defined once forever and is AL-independent), the addition of new ALs increases the loss of information when new ALs introduce new concepts not included in the kernel pivot. When the closed-world-assumption decays, a new solution is needed to support the interoperability among various ALs while reducing as much as possible the loss of information. This calls for kernel extensions, each extension defined for dealing with specific stakeholder concerns. Moreover, the construction of kernels must be properly controlled to support their coexistence and reuse. The information that can be lost consists of concepts that potentially could be transformed from a source model and properly represented in a target one, but for some reason are neglected by the transformation process.

In this paper we present a Model-Driven Engineering (MDE) approach to enhance the interoperability among ALs by using extensible kernel pivots. The approach (i) encompasses a systematically defined *extension process* that, starting from a small kernel pivot language permits the automated construction of a hierarchy of kernel pivots, and (ii) provides mechanisms to transform from an AL to another by minimizing the loss of information; this is realized by passing through the most informative pivot kernel in the hierarchy for the considered ALs. The overall approach is general and, while applied to the software architecture domain, may be adopted in different domains.

The remaining of the paper is organized as follows. Section 2 highlights limitations and challenges of current pivot-based solutions. Section 3 describes the proposed kernel pivot extension mechanisms. Section 4 applies the approach to a case study in the automotive domain. Section 5 compares our work with related works. Section 6 concludes the paper and highlights future research directions.

2 Interoperability via Pivot Languages

It is becoming common practice to use different ALs to model or to analyze different architectural aspects of the system under development. The Q-Impress project, for example, enables interoperability through a rich pivot language that unifies common aspects of the used ALs. The ATESST project provides means to integrate different model-based tools to develop automotive embedded systems. In the domain of reliability modeling and prediction, Klaper is a kernel language which can be used as the starting point to carry out performance or reliability analysis. **DUALL**Y [7] exploits model transformation techniques and any transformation among ALs is defined by passing through A_0, a kernel pivot metamodel defined as general as possible.

All the projects and research efforts described above adopt a pivot solution for supporting the interoperability among different description languages. Figure 1 shows the main difference between the use of a rich pivot language and a kernel one: filled circles represent modeling concepts, solid lines denote correspondences among AL and pivot language concepts, and finally dashed boxes and dashed lines represent added ALs and correspondences, respectively. A rich pivot language is built with the aim of including the highest number of concepts contemplated by all the interoperating ALs. As shown in Figure 1.a, each concept in any AL finds its correspondence with a rich pivot language element. Differently, a kernel language contains only a core set of concepts (as shown in Figure 1.b), and is kept as small as possible. Such a difference has positive and negative impacts on the way interoperability is realized. In the following we provide a summary of the main strengths and limitations of both solutions.

Interoperability Accuracy: the rich pivot is built with the intent to match any concept coming from the interoperating ALs. Thus, in principle, as soon as a correspondence exists among two ALs, it is caught by the pivot-based transformation. The kernel language solution, being minimal, may instead discard some correspondence, thus limiting the interoperability accuracy. For instance, see `a1` and `a2` in Figure 1.b: while a correspondence among them is found in the rich pivot, it is missing in the kernel-based solution. Information loss is thus introduced. The kernel-based approach is particularly limiting when domain-specific ALs are introduced in the star topology. *Overall*: the rich pivot solution is more accurate;

Pivot Scalability: as soon as a new AL has to be considered, the rich pivot needs to be revised in order to avoid information loss. As shown in Figure 1.a, the insertion of AL_4 implies the addition of the link between AL_4 and the already existing element `b1` in the rich pivot, and the addition of `b2`. This may require a strong revision of the entire rich pivot to solve possible conflicts and to avoid inconsistencies. When AL_4 is added to the kernel language in Figure 1.b, instead, only a new correspondence with `b1` is created. *Overall*: the kernel language approach scales better.

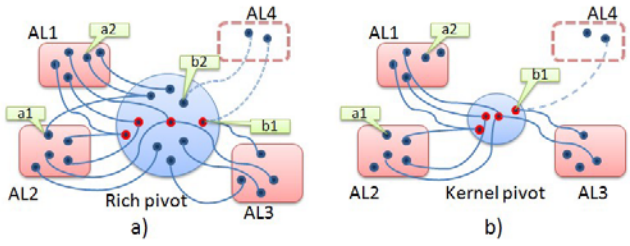

Fig. 1. Interoperability via a: a) rich pivot, b) kernel pivot

In summary, the rich pivot solution is more accurate in terms of interoperability correspondences, but it is less scalable and might require adjustments when a new notation is included. Contrariwise, the kernel solution shows complementary strengths and limitations. *A new solution is needed to support both interoperability accuracy and pivot scalability.*

An approach that is being used consists in making the kernel pivot *extensible*, thus adaptable to new ALs. Language extensibility in the software architecture domain has been adopted in the xADL [9] XML-based architecture description language (based on XML extension mechanisms), in AADL [10] (through its annexes), in UML (with its profiles), and in our approach for ALs interoperability named **DUALLY** [7]. However, **DUALLY**, which is at the best of our knowledge the most mature framework to support interoperability among various ALs, has shown a certain number of shortcomings. *Firstly*, it is not clear how to manage the extension process when two (or more) extensions are required. Let us suppose that both real-time and behavior extensions are needed. So far, three alternative solutions can be applied: i) extend the kernel with real-time concepts first, then with behavior, ii) extend the kernel with behavior concepts first, then with real-time ones, iii) extend the kernel with both concepts at the same time. The three scenarios may produce different kernel pivots, and so far there is no guideline on how to manage such a multiple extension. *Secondly*, current solutions tend to create ad-hoc extensions, not engineered to be reusable. Even when applying scenarios i) or ii) above, the intermediate kernels are typically lost and not stored for reuse. The extension itself is not considered as a first class element, but simply as an improvement to the original pivot.

The approach we propose in this paper satisfies the requirements of i) a systematic extension process, which provides clear guidelines on how and what to extend, ii) a compositional and reuse-oriented approach, where kernels are re-used and extended, iii) supporting both interoperability accuracy and pivot scalability.

3 The Extension Mechanisms

In this section we propose the mechanisms to extend an existing kernel A with a kernel extension e. In our approach the extension e is a metamodel, that can be re-used for extending different kernels. The proposed mechanisms rely on the adoption of weaving models [11] which relate a kernel A with an extension e. A weaving model wm contains links between elements of a kernel A and elements of an extension e.

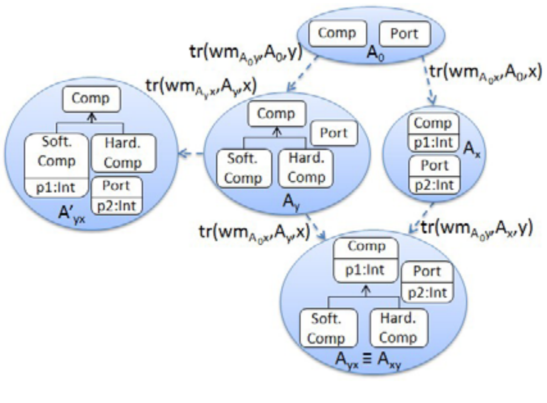

Fig. 2. Example of extensions of A_0

The generation of a kernel A_e, which is an extension of A with e, is performed by executing a transformation tr. tr is defined once forever and applies the extension e to A according to the extension operators used in wm (see Section 3.1). Figure 2 shows a small fragment of A_0 consisting of the metaclasses `Comp` and `Port` that represent a generic component and port, respectively (see [7] for a complete description of A_0).

Let us assume that y is a kernel extension containing the metaclasses `SoftComp` and `HardComp` to model software and hardware components, respectively. This extension can be applied to A_0 by means of the transformation tr which takes as input the weaving model wm_{A_0y}, the kernel A_0, and the extension y, and generates the new kernel A_y. The kernel A_y is shown in Figure 2 and contains the generic component concept specialized in software and hardware components. Let us assume also that x is another extension consisting of the performance annotations $p1$ and $p2$. This extension can be applied to A_0 by means of the transformation tr which takes as input another weaving model wm_{A_0x}, the kernel A_0, and the extension x. The obtained kernel called A_x is shown in Figure 2 and represents an extension of A_0 in which the $p1$ annotation is added to `Comp` and the $p2$ annotation is added to `Port`.

As previously said, weaving models are used to apply given extensions to existing kernels by specifying the metaclasses which are involved in the operation. Formally, a weaving model can be defined as in Def. 1.

Definition 1 (Weaving Model). *Let $\mathbb{A}$ be the set of all the possible kernels, let $\mathbb{E}$ be the set of all the possible extensions, and let $\mathbb{W}$ be the set of all the possible weaving models. We denote with $wm_{Ae} \in \mathbb{W}$ a weaving model defined between the kernel $A \in \mathbb{A}$ and the extension $e \in \mathbb{E}$. A weaving model $wm_{Ae} = \{wl^1_{Ae}, wl^2_{Ae}, \cdots, wl^n_{Ae}\}$ can be seen as a set of weaving links each establishing a correspondence between elements of A and elements of e. Each link is realized by means of extension operators.*

Referring to Figure 2, the weaving model wm_{A_0x} defined for A_0 can be used also to extend A_y, since A_y contains the metaclasses involved in wm_{A_0x}. In fact, A_y contains the metaclasses `Comp` and `Port` which are considered in the weaving model wm_{A_0x} to attach the annotation $p1$ to `Comp`, and $p2$ to `Port`. In the same way, wm_{A_0y} can be used to extend A_x by applying the extension y to the metaclass `Comp`, and specializes it with the metaclasses `SoftComp` and `HardComp`. These two independent extension journeys converge in a kernel called A_{yx} or A_{xy}. Focusing on the left-hand side of Figure 2, the weaving model wm_{A_yx} is another application of the extension x to the kernel A_y to add the annotation $p2$ to `Port` and the annotation $p1$ to `SoftComp`. In this case we obtain a kernel different from A_{xy}. Specifically, this kernel permits to add $p1$ exclusively to software components.

Extension hierarchies, like the one in Figure 2, contain three types of elements: kernels, extensions, and weaving models that apply extensions to kernels. In order to regulate how kernels and extensions can be involved in specific weaving models, we make use of a type system for kernels and extensions. In other words, a weaving model defined for a kernel can be re-used also for applying extensions to other kernels, under the assumption that these kernels have the metaclasses involved in the weaving model. Def. 2 defines our notion of model type substitutability, which is based on the following notion of model typing: the type of a model is defined "as a set of MOF classes (and, of course, the references that they contain)" [12]. We denote with $\mathbb{T}$ the set of all the possible model types. In our context $\mathbb{T}$ can be partitioned in $\mathbb{T}^{\mathbb{A}}$ and $\mathbb{T}^{\mathbb{E}}$ which denote the types of kernels and extensions, respectively.

Definition 2 (Model Type Substitutability). *Let $T_A \in \mathbb{T}^{\mathbb{A}}$ be the type of a given kernel A, and let $T_e \in \mathbb{T}^{\mathbb{E}}$ be the type of an extension e, then a weaving model wm_{Ae} can be used by the model transformation tr to extend a kernel typed with either T_A or any of its subtypes.*

In our context subtyping depends on a type's hierarchy obtained by means of the extension mechanism that produces a kernel typed T_B by exclusively adding new elements to an existing one, typed T_A (i.e., the deletion of elements from a kernel is not allowed). It is worth mentioning that our extension mechanism ensures that all the elements of an extension e are added to the kernel being extended. This type hierarchy introduces a *strict partial order* $<$ among kernel types: $T_A{<}T_B$ if T_B is obtained by extending T_A and then T_B can be substituted to T_A. Figure 3 is a generalization of Figure 2 and shows a simple hierarchy of extensions involving a generic kernel A_k and two extensions called x and y. The kernel extensions are regulated by four different weaving models (wm_{A_kx}, wm_{A_ky}, wm_{A_xy}, and wm_{A_yx}), thus producing five different new kernels. More specifically, A_x and A_y are obtained extending A_k with x and y and by means of the weaving models wm_{A_kx} and wm_{A_ky}, respectively. The weaving model wm_{A_kx} takes as input a kernel typed T_{A_k} and the extension x typed T_x. Similarly, the weaving model wm_{A_ky} takes as input a kernel typed T_{A_k} and the extension y typed T_y.

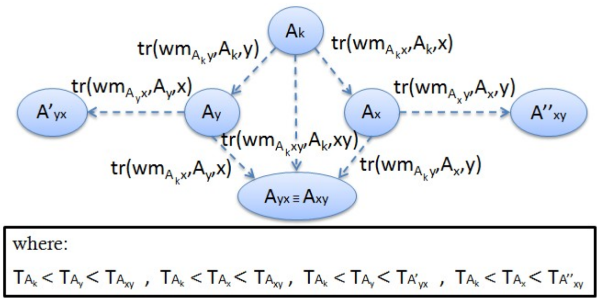

Fig. 3. A hierarchy of kernels

Let us focus now on A_x which is extended by applying the extension y in two different ways. The first way considers the weaving model wm_{A_xy} used by tr to apply the extension typed T_y to elements of a kernel typed T_{A_x}. This kernel contains the elements of A_k and those of x added by using wm_{A_kx}. The weaving model wm_{A_xy} can affect all of them since it considers a kernel typed T_{A_x}. This is not the case of wm_{A_kx}, which can only operate on elements of A_k. This justifies why the sequential compositions $tr(wm_{A_ky}, tr(wm_{A_kx}, A_k, x), y)$ and $tr(wm_{A_kx}, tr(wm_{A_ky}, A_k, y), x)$ lead to the same target metamodel A_{xy}, i.e., there is a confluence in the extension journeys. The generation of the target metamodel A_{xy} is performed by using a new weaving model wm_{A_kxy} which is the union of wm_{A_kx} and wm_{A_ky}. The execution of $tr(wm_{A_kxy}, A_k, xy)$, where xy is a metamodel consisting of the union of the elements of x and y, produces A_{xy}. Formally, the union of two weaving models is defined as in Def. 3.

Definition 3 (Union of Weaving Models). *Let $wm_{Ax}{\in}\mathbb{W}$ a weaving model defined between the kernel $A{\in}\mathbb{A}$ and the extension $x{\in}\mathbb{E}$, and $wm_{Ax}{=}\{wl^1_{Ax}, wl^2_{Ax}, \cdots, wl^n_{Ax}\}$. Let $wm_{Ay}{\in}\mathbb{W}$ a weaving model defined between the kernel $A{\in}\mathbb{A}$ and the extension $y{\in}\mathbb{E}$, and $wm_{Ay}{=}\{wl^1_{Ay}, wl^2_{Ay}, \cdots, wl^m_{Ay}\}$. The weaving models union $wm_{Ax}{\cup}wm_{Ay}{=}\{wl^1_{Ax}, wl^2_{Ax}, \cdots, wl^n_{Ax}, wl^1_{Ay}, wl^2_{Ay}, \cdots, wl^m_{Ay}\}$ is the set of all the weaving links in wm_{Ax} and wm_{Ay}.*

It is important to note that in general the confluence cannot be ensured since it depends on how the extensions have been applied, i.e., on the involved weaving models. In the following we explain why in our approach we have a confluence (see Section 3.1) and how to identify transformation paths from one AL to another by passing through the kernels hierarchy (see Section 3.2).

3.1 Extension Operators

The extension operators used to create weaving models are *Inherit*, *Reference*, *Expand*, and *Match*. These operators are defined by constraining the composition operators presented in [13] to exclusively enable extensions and avoid conflicts when structural features of the kernel and the extension being applied overlap. They always extend a kernel and then, in case of conflicts during the extension, the kernel element will be the one to be considered. Each operator is always applied on two metaclasses (one belonging to the kernel and one to the extension) that we refer to as source (s) and target (t) in the remainder of this section. The application of the operators consists of executing the transformation tr that, as explained before, takes as input a weaving model, a kernel, and an extension, and produces an extended kernel according to the applied operators. The extension operators are:
Inherit: This operator specifies that the concept s will be a subtype of t in the resulting extended kernel. If its application results in a cycle in the inheritance tree, then it is not executed and a warning is raised. The t metaclass must belong to the kernel metamodel.
Reference: In the extended kernel, s has a reference to t. The metaclasses s and t belong to the kernel or to the extension.
Expand: all the attributes of s are copied into t. Attributes with the same name are merged. The t metaclass must belong to the kernel metamodel.
Match: s and t represent the same concept; they are merged into a single metaclass which contains the union of all the structural features (i.e., both attributes and references) of s and t. Their supertype and subtype references are merged as well. The t metaclass must belong to the kernel metamodel.

The proposed extension operators have the following properties that underpin the construction of the type hierarchy previously presented.

Property 1 (Monotonicity - kernel preservation). Each operator can only add elements to the kernel being extended. The deletion of kernel elements is forbidden.

Property 2 (Extension integrity). All the elements of the extension metamodel are added to the kernel metamodel according to the operator semantics. In other words, it is not possible to use only a fragment of an extension. This is ensured by the default behavior of the extension mechanism which copies all the extension elements that are not considered by the used operators.

Property 3 (Parallel independence). An operator can be applied only if conflicts[1] among the structural features of the involved metaclasses do not occur. For instance, it is not possible to match a kernel metaclass A containing an attribute $p : Int$ with an extension metaclass B containing an attribute $p : String$ because of the conflicting types of the attribute p.

[1] According to the classification in [14], the conflicts that are considered in the *parallel independence* property are the so-called syntactic conflicts.

By referring to Figure 3, Properties 1, 2, and 3 ensure the confluence of the extension mechanism (see Theorem 1).

Theorem 1 (Confluence). *Given two weaving models wm_{A_kx} and wm_{A_ky} between the kernel A_k and the extensions x and y, respectively, and $wm_{A_kx} \cup wm_{A_ky}$ does not contain weaving links that refer to elements in x and y which are in conflict, then:*

$$tr(wm_{A_kxy}, A_k, xy) = tr(wm_{A_kx}, tr(wm_{A_ky}, A_k, y), x) = tr(wm_{A_ky}, tr(wm_{A_kx}, A_k, x), y)$$

where wm_{A_kxy} is the weaving between the kernel A_k and the extension xy is given as the union of wm_{A_kx} and wm_{A_ky}.

The proof of the theorem is given in Appendix.

3.2 Identification of transformation paths

ALs can be bound to different kernels of the built hierarchy. To better explain both the problematics of the transformation path identification and the provided solution, we use the example presented before.

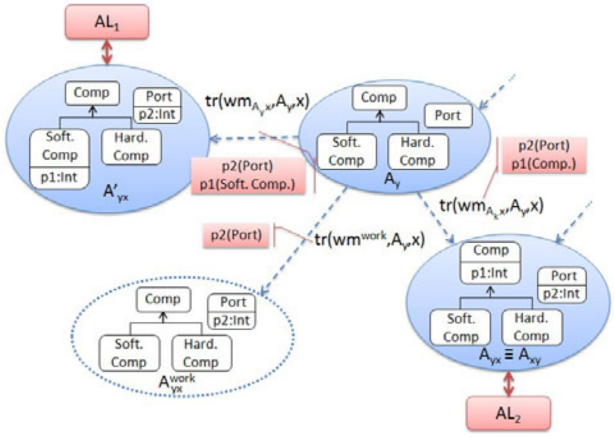

Fig. 4. AL-to-AL transformation management

Figure 4 describes two generic ALs, AL_1 and AL_2, bound to A'_{yx} and A_{yx}, respectively. As described before, A'_{yx} is an extension of A_y that contains the performance annotation $p1$ added to `SoftComp` and the performance annotation $p2$ added to `Port`. Whereas, A_{yx} contains the performance annotation $p1$ added to `Component` and the performance annotation $p2$ added to `Port`. In this simple example the performance annotation $p2$ is present both in A'_{yx} and in A_{yx}; therefore, when transforming from a model specified with AL_1 to a model conforming to AL_2, it is desirable to maintain also the $p2$ annotation. In a transformation realized by passing through A_y we lose such an information. For this reason our approach automatically builds a working kernel, A^{work}_{yx} in Figure 4, which contains also the $p2$ annotation. This working kernel contains the metaclass `Port` with the annotation $p2$, while $p1$ is ignored since in A'_{yx} $p1$ is attached to `SoftComp` and in A_{yx} it is attached to `Comp`. Thus, $p1$ represents information that cannot be automatically translated. Notice that once transforming from AL_1 to AL_2 and back, the values of the $p1$ annotations possibly attached to `SoftComp` instances of AL_1 are preserved by means of the *lost-in-translation* mechanism described in [7].

Formally, let A_l and A_m be the kernels which AL_1 and AL_2 are bound to, respectively. Moreover, let T_{A_l} and T_{A_m} the types of A_l and A_m, respectively. To identify the transformation path between A_l and A_m that minimizes the loss of information, we look for the most "specialized" common ancestor A_{anc} of A_l and A_m such that:

$$((T_{A_{anc}} < T_{A_l}) \wedge (T_{A_{anc}} < T_{A_m})) \wedge (\nexists A' \in \mathbb{A} | (T_{A'} < T_{A_l}) \wedge (T_{A'} < T_{A_m}) \wedge (T_{A_{anc}} < T_{A'}))$$

To understand if we can build a kernel useful to reduce the loss of information, we consider the extensions that have been applied from A_{anc} to A_l and from A_{anc} to A_m. The functions in Def. 4 and Def. 5 are introduced to construct such a kernel.

Definition 4 (extensionApplications). $extensionApplications$: $\mathbb{A} \times \mathbb{A} \rightarrow 2^{\mathbb{W}}$ *is a function that given as input the kernels* $A_i \in \mathbb{A}$ *and* $A_j \in \mathbb{A}$*, such that* $T_{A_j} < T_{A_i}$ *(i.e.,* A_j *is an ancestor of* A_i*) returns a set containing all the weaving models that have been applied to* A_j *for building the kernel* A_i*.*

Definition 5 (usedExtensions). $usedExtensions$: $\mathbb{A} \times \mathbb{A} \rightarrow 2^{\mathbb{E}}$ *is a function that given as input the kernels* $A_i \in \mathbb{A}$ *and* $A_j \in \mathbb{A}$*, such that* $T_{A_j} < T_{A_i}$ *(i.e.,* A_j *is an ancestor of* A_i*) returns a set containing all the extensions that have been used to extend* A_j *for building the kernel* A_i*.*

The transformation path that minimizes the loss of information between A_l and A_m is calculated by means of the $pathIdentification$ algorithm shown in the left-hand side of Figure 5. In particular, $pathIdentification$ gets as input A_l and A_m and calculates the common ancestor A_{anc} (see line 1). Then the next step is to find a kernel that while transforming can reduce the loss of information. To this purpose the algorithm checks if there is an intersection between (i) the extensions that have been applied (i.e., weaving models) to A_{anc} to build A_l, and (ii) those that have been applied to A_{anc} to build A_m. The extension applications are calculated in two steps. Firstly, the sets of weaving models applied to A_{anc} for building the kernels A_l and A_m are calculated (lines 2 and 3, respectively). Secondly, for each set, the union of all the weaving models is calculated. More precisely wm_L and wm_M are the weaving models that have been

```
Algorithm 1 pathIdentification($A_l, A_m$)
 1: $A_{anc} \leftarrow calculateCommonAncestor(A_l, A_m)$
 2: $L \leftarrow extensionApplications(A_l, A_{anc})$
 3: $M \leftarrow extensionApplications(A_m, A_{anc})$
 4: $wm_L \leftarrow \bigcup_{wm \in L} wm$
 5: $wm_M \leftarrow \bigcup_{wm \in M} wm$
 6: if ($wm_L \cap wm_M = \emptyset$) then
 7:     return $calculatePath(A_l, A_{anc}, A_m)$
 8: else
 9:     $wm_{lm}^{work} \leftarrow wm_L \cap wm_M$
10:     $E \leftarrow createWorkingExtension(wm_{lm}^{work},$
                    $usedExtensions(A_l, A_{anc}),$
                    $usedExtensions(A_m, A_{anc}))$
11:     $A_{lm}^{work} \leftarrow tr(wm_{lm}^{work}, A_{anc}, E)$
12:     return $calculatePath(A_l, A_{lm}^{work}, A_m)$
13: end if
```

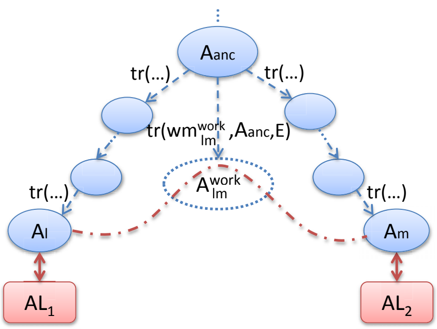

Fig. 5. Working kernel generation

obtained from the union of all the weaving models contained in L and M, respectively (lines 4 and 5). To understand if we can refine the hierarchy by building a new kernel that can reduce the loss of information, the intersection between wm_L and wm_M is calculated (line 6). If the intersection is empty, then all the information that is common to A_l and A_m is already contained into A_{anc}; consequently, the path that minimizes the loss of information between A_l and A_m starts from A_l, navigates the hierarchy up to A_{anc}, and then navigates the hierarchy down to A_m (see line 7).

If the intersection is not empty, then we have to refine the hierarchy as shown in Figure 5 in order to perform transformations (from AL_1 to AL_2 and vice versa) via a kernel more specific than A_{anc}. In other words, the idea is to extend A_{anc} with the information shared between A_l and A_m that is not contained in A_{anc}. The ad-hoc kernel is called A_{lm}^{work} and is automatically generated by using a working weaving model called wm_{lm}^{work}. This wm_{lm}^{work} is obtained from the intersection of wm_L and wm_M (line 9). As shown in the right-hand side of Figure 5, the weaving model wm_{lm}^{work} applies the working extension E to A_{anc} then generating A_{lm}^{work} (line 11). E is obtained by suitably merging the extensions that have been used to extend A_{anc} for building the kernel A_l and those that have been used to extend A_{anc} for building the kernel A_m (line 10). The merging of extensions is realized by means of the function $createWorkingExtension$ that considers only the portion of the extensions involved in at least one of the weaving links in wm_{lm}^{work}. $createWorkingExtension$ does not add new conflicts into A_{lm}^{work} since each weaving link added in A_{lm}^{work} belongs both to A_l and A_m; indeed having a conflict in A_{lm}^{work} would imply to have a conflict in both A_l and A_m. This is not possible since Property 3 of the extension operators ensures that A_l and A_m do not have conflicts (by construction).

It is worth noting that A_{lm}^{work} is a working kernel since it is exclusively used for transformation purposes and we do not allow ALs to be bound to A_{lm}^{work}. Finally, as shown in Figure 5, the path that minimizes the information loss between A_l and A_m starts from A_l, directly passes through A_{lm}^{work} and ends to A_m.

4 Case Study and Discussion

In Section 4.1 we present a case study to show how two real ALs can interoperate by means of the proposed approach. The scale of the considered case study does not allow us to show all technical aspects of the approach. Thus, we show the most automated parts, while more complex technicalities are better described by using small examples as done in Section 3. Then, Section 4.2 discusses issues related to the approach.

4.1 Putting the approach in practice

According to its business needs, an organization decided to draw and analyze the architecture of a system in the vehicular domain by using AADL [10] (with its behavioral annex), complemented with SaveCCM [15] (helpful to support the development of resource-efficient systems and to perform structural preventive analysis). The case study starts from an already existing kernel hierarchy (see the uppermost part of Figure 6) composed of three extensions of the root kernel A_0, namely *Behaviour*, *Embedded systems*, and *Real-time*. Due to space limitations, we do not describe the concepts contained into the extension metamodels. We assume that two ALs are already bound to the hierarchy: Acme [16] is bound to A_0 and Darwin/FSP [17] to the *Behaviour* kernel. In order to apply the proposed approach, we need to identify the suitable kernel on which each AL can be profitably bound. Focusing on SaveCCM, it contains both real-time and embedded systems concepts. A satisfying kernel does not exist but two existing kernels, namely the *Embedded systems* and the *Real-time*, can be suitably used to obtain a new kernel on which SaveCCM can be bound. In this example the kernel can be produced by reusing both the existing weaving models wm_{A_0E} and wm_{A_0RT}. The obtained kernel, named *RT+E*, is shown in Figure 6. This kernel metamodel is automatically obtained, as explained in Sections 3.1. It is important to note that during this extension a new weaving model, wm_{A_0RT+E}, is automatically generated by composing wm_{A_0E} and wm_{A_0RT}. As explained in Section 3.1 this weaving model is extremely important to support further extensions of the kernel *RT+E*.

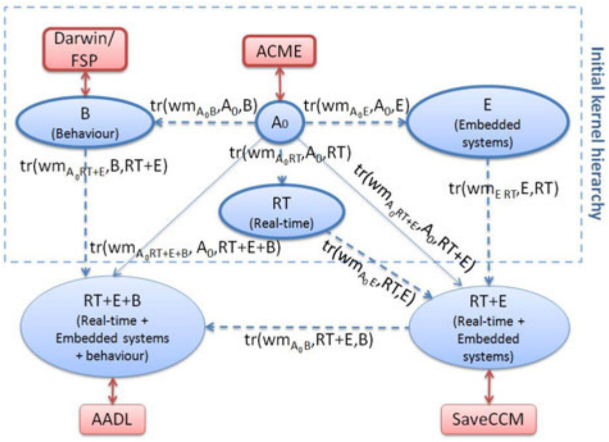

Fig. 6. SaveCCM and AADL into the hierarchy

Similarly to SaveCCM, AADL contains both real-time and embedded systems concepts; however, AADL contains also behavioral concepts since we are considering also its behavior annex. In this specific situation we look for a candidate kernel with real-time, embedded systems, and behavioral concepts. Building on the kernel *RT+E* and by considering also the extension B, we can build a new *RT+E+B* kernel by reusing both the wm_{A_0B} weaving model already used to extend A_0 with B and the

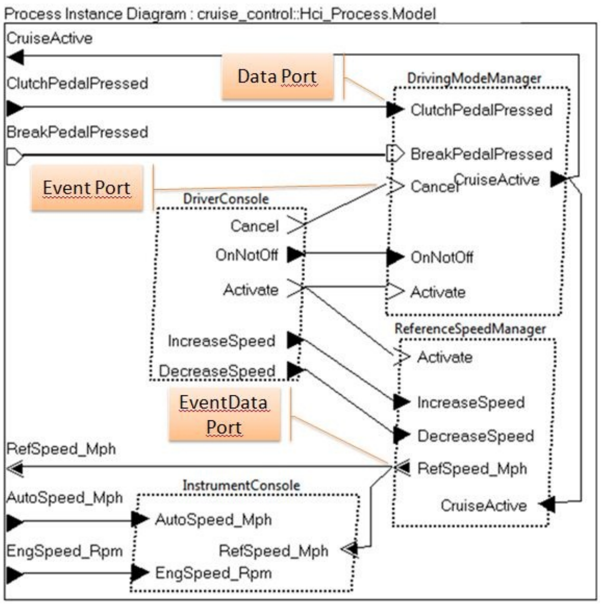

Fig. 7. AADL model of the HCI process

generated weaving model wm_{A_0RT+E}. Once the extension metamodel *RT+E+B* has been generated, AADL can be bound to the hierarchy. *RT+E+B* contains real-time, embedded systems, and behavioral concepts. Finally, suitable model transformations are generated from each weaving model as described in Sections 3.2. Now that the kernel hierarchy is ready to be used, we can proceed by modeling the system of interest. It is a cruise control system, i.e., a system that automatically controls the speed of a vehicle according to the driver settings [18]. In this paper we focus on the Human Control Interface (HCI) subsystem, which is the front-end to the driver. Figure 7 shows the HCI process modeled in AADL. This process is composed of four threads managing the driving mode (`DrivingModeManager`), the reference speed (`ReferenceSpeedManager`), the buttons panel (`DriverConsole`), and a console (`InstrumentConsole`) for special settings of the system.

In order to transform the AADL model to the corresponding SaveCCM model, the transformation chain is calculated as described in Sections 3.2. In this case the calculated path passes through the kernel *RT+E* that is the most specific common ancestor of *RT+E* and *RT+E+B*. By means of this transformation chain we ensure that both real-time and embedded system concepts are accurately translated. Therefore, the information that is lost while transforming is limited to behavioral concepts or to concepts specific to AADL; they cannot be translated to SaveCCM even by using an ad-hoc transformation. However, without a systematically defined extension process SaveCCM and AADL could have been bound to two extensions of A_0 with potential but unexpressed similarities. This may lead to the loss of real-time and embedded system concepts.

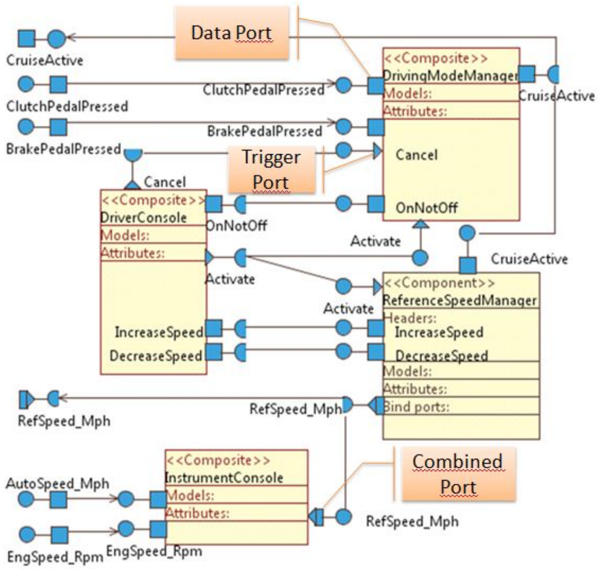

Fig. 8. SaveCCM model of the HCI component

Figure 8 shows the model of the HCI process automatically generated for SaveCCM. SaveCCM does not provide specific modeling constructs for processes and threads and then, as can be seen in the figure, both processes and threads become components; in particular the HCI process becomes a `Composite` component. This is because the generic component metaclass of AADL (which is a superclass of thread, process, memory, etc.) is linked to the component metaclass of the kernel *RT+E+B*, and the SaveCCM component metaclass is linked to the component metaclass of the kernel *RT+E*. We clearly have a loss of information when transforming from AADL to SaveCCM. However, the generated transformations are instrumented to maintain the information which is lost so to recover it when transforming back from SaveCCM to AADL. `Data`, `Event`, and `EventData` ports are linked to the corresponding concepts in *RT+E+B*, which are linked in turn with `Data`, `Trigger`, and `Combined` ports of SaveCCM, respectively. Therefore, the semantics of the modeled ports is maintained when transforming from AADL to SaveCCM. This is obtained thanks to the kernel hierarchy. Without such a hierarchy, i.e., by passing directly through A_0, we loose the specific information related to ports since A_0 has only the concept of generic port.

4.2 Discussion

In this section we discuss the following aspects: (i) generalization of the approach, (ii) its scalability, and (iii) overhead added by the kernel hierarchy to the transformation.

Generalization: the overall approach is applied to the software architecture domain and specifically to ALs. However, the kernel hierarchy and transformation management can be easily applied to different domains by simply substituting A_0 with a different root kernel metamodel. The definition of the root kernel metamodel is strategic and requires particular attention. Please refer to the discussion section in [19] for more details about the process we followed for defining A_0. Finally, we believe that the proposed approach could be used as a new "profiling" mechanism able to support the extensibility mechanisms envisioned by Jacobson and Cook in the UML of the future [20].

Approach Scalability: according to Section 3, a kernel can be extended in several different ways depending on the specified weaving model. As described in Section 3.2, some "working" metamodels need to be added to the hierarchy in order to properly manage the transformations. Thus, from the scalability point of view it is important to understand the order of magnitude of the hierarchy. As reported in [7] the number of available architecture description languages is around 50 or 60. An estimation of the possible extensions is more difficult to be performed but based on the number of available ALs we are confident that this will not compromise the approach applicability.

Overhead: the kernel hierarchy adds some overhead to the transformations. In order to quantify this overhead it is important to understand the operations that need to be performed during the transformations and to identify the operations that are performed once forever. In Section 3.2 we explained the need of having a working metamodel and the procedure to build it. This metamodel and related weaving models are created once forever. Therefore, this cannot be considered as overhead of the transformation from one AL to another. The overhead that is added to each transformation from one AL to another is related to the fact that the transformation is actually a chain of transformations instead of a direct transformation from one AL to another. Assuming a constant time t for each transformation, the overhead can be quantified as $(t \times x) - t$, where x is the number of transformations composing the considered chain. In the case study presented in this work, we used an Intel Pentium D-3.2Ghz, with 4GB DDR-II of RAM, running

Windows 7 Professional. The generation of the transformation chain and its execution took less than four seconds with a source AADL model consisting of 603 modeling elements. The experience we had with the case study was encouraging from the point of view of the efficiency of the overall approach.

5 Related work

State-of-the-art approaches on ALs interoperability have been discussed in Section 2 outlining what is missing and then motivating the proposed approach. In this section we compare our work with existing work in the area of model-driven engineering.

Over the last years a number of work has been proposed to cope with the problem of tool integration and interoperability in MDE. Such works can be classified into *Transformation-based approaches* and *Metamodel integration approaches* [21]. The former approaches, like [22,6], propose the adoption of model transformations which aim to serve as a bridge between the various tools that have to interoperate. In particular, model transformations are used to transform data required by heterogeneous tools. Differently to our work, such approaches rely on manually written transformations defined with respect to the notations adopted by the considered tools. Metamodel integration approaches, like [23], rely on the definition of a common metamodel to establish tool interoperability. Even though such approaches are similar to our work, they do not provide mechanisms supporting the extension of the common metamodel.

The problem of interoperability has been tackled also in the context of model-to-model transformation languages. In [24] the authors propose an approach based on a *Common Intermediate Language* to support interoperability between different model transformation languages. Differently from our approach the authors analyze a set of well-known transformation languages and identify common characteristics which are captured in a common metamodel which is not extensible.

In [25] the authors propose an approach based on consistency rules, and bidirectional model transformations to automate the synchronizations of AUTOSAR (Automotive Open System ARchitecture)and SysML (System Modeling Language) modelsEven though the approach is general and can be applied on any couple of modeling languages, it differs from our work since the used model transformations which underpin the synchronization mechanism are manually written and are not organized in an extension hierarchy as proposed in this paper.

Going back to the nineties, a family of works have been proposed to exploit a single formal kernel language to integrate specifications written in different languages. One of the most prominent work in this family is the one by Jackson and Zave [26] in which Z is used as a common semantic domain for the composition of partial specifications defined in different languages. The resulting composed specification is then used to check the consistency of the initial partial specifications. Our goal is quite different since we consider the kernels hierarchy as an intermediate means for transforming models across different languages, rather then a way to check their global consistency.

6 Conclusion and Future Work

Approaches to support architectural interoperability typically choose to organize the different notations in a star topology with an intermediate central pivot. In a context in which the set of involved notations cannot be *a-priori* established, the pivot assumes the form of a small kernel. Since the transformations are always performed through the small kernel that can be very abstract, important information can be lost during the transformation. This calls for kernel extensions. This paper proposes a model-driven approach to (i) build the extensions and organize them in a hierarchy, (ii) realize the interoperability (through the hierarchy) by means of model transformations, and (iii) manage the overall hierarchy. The extension is performed through operators that have properties that ensure the extension confluence.

We realized a prototype automatizing the overall approach: it is a plugin for Eclipse that allowed us to perform experiments on some systems. As future work we plan to release the tool as an open source project and to experiment it on industrial case studies.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons (2009)
2. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE TSE 26(1) (2000)
3. Giese, H., Neumann, S., Niggemann, O., Schätz, B.: 2 Model-Based Integration. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2010. LNCS, vol. 6100, pp. 17–54. Springer, Heidelberg (2010)
4. Q-ImPrESS consortium, `http://www.q-impress.eu` (last access, September 2011)
5. ATESST2 consortium, `http://www.atesst.org/` (last access, September 2011)
6. Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A Model Engineering Approach to Tool Interoperability. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 178–187. Springer, Heidelberg (2009)
7. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. IEEE TSE 36(1) (2010)
8. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. J. Syst. Softw. 80(4), 528–558 (2007)
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. TOSEM 14(2) (2005)
10. Feiler, H.P., Lewis, B., Vestal, S.: The SAE Architecture Analysis and Design Language (AADL) Standard. In: RTAS Workshop on Model-driven Embedded Systems, pp. 1–10 (2003)
11. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling 4(2), 171–188 (2005)
12. Steel, J., Jézéquel, J.M.: On model typing. Software and System Modeling 6(4), 401–413 (2007)
13. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing next generation ADLs through MDE techniques. ACM/IEEE ICSE 2010, 85–94 (2010)

14. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28(5), 449–462 (2002)
15. Kerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Jour. Syst. Softw. 80(5), 655–667 (2007)
16. Garlan, D., Monroe, R., Wile, D.: Acme: An Architecture Description Interchange Language. In: CASCON 1997, pp. 169–183 (1997)
17. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6) (1996)
18. Varona-Gomez, R., Villar, E.: Aads+: Aadl simulation including the behavioral annex. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, pp. 379–384. IEEE Computer Society, Washington, DC (2010)
19. Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: A model-driven approach to automate the propagation of changes among Architecture Description Languages. In: Software and Systems Modeling, SoSyM (2010)
20. Jacobson, I., Cook, S.: The Road Ahead for UML (2010), `http://www.drdobbs.com/architecture-and-design/224701702`
21. Seifert, M., Wende, C., Assmann, U.: Anticipating unanticipated tool interoperability using role models. In: Proc. of MDI 2010, pp. 52–60. ACM (2010)
22. Ehrig, K., Taentzer, G., Varró, D.: Tool Integration by Model Transformations based on the Eclipse Modeling Framework. EASST Newsletter 12 (2006)
23. Baumgart, A.: A common meta-model for the interoperation of tools with heterogeneous data models. In: Proc. of MDTPI 2010 (2010)
24. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. Sci. Comput. Program. 68(3), 114–137 (2007)
25. Giese, H., Hildebrandt, S., Neumann, S.: Towards integrating sysml and autosar modeling via bidirectional model synchronization. In: MBEES, pp. 155–164 (2009)
26. Zave, P., Jackson, M.: Conjunction as composition. ACM Trans. Softw. Eng. Methodol. 2, 379–411 (1993)

Appendix: Proof of Theorem 1

Let us assume (ad absurdum) that:

- $tr(wm_{A_kxy}, A_k, xy) = A'$,
- $tr(wm_{A_kx}, tr(wm_{A_ky}, A_k, y), x) = A''$, and
- $A' \neq A''$

(the symmetric, i.e., $tr(wm_{A_kxy}, A_k, xy) = A'$, $tr(wm_{A_ky}, tr(wm_{A_kx}, A_k, x), y) = A''$, and $A' \neq A''$ will directly follow). This can happen in four cases:
1. a metaclass C exists in A' and does not in A''. This means that C exists in A_k, in x, or in y. In case C exists in A_k, this implies that the application of wm_{A_kx} or wm_{A_ky} deletes it. This is absurd for Property 1. In case C exists in x or in y, this implies that wm_{A_kx} or wm_{A_ky} do not add it during the extension. This is absurd for Property 2.

2. a metaclass C exists in A'' and does not in A'. In case C exists in A_k, this implies that wm' deletes it. This is absurd since the operators that we use in wm' have to respect Property 1. In case C exists in xy, this implies that wm' does not add it during the extension. This is absurd since wm' is basically the union of wm_{A_kx} and wm_{A_ky} and then it respects Property 2.

3. a metaclass C exists both in A' and A'' and these two versions differ on some structural features, i.e., attributes and references. This can be caused exclusively due to deletion or conflicting additions performed by either wm_{A_kx} and wm_{A_ky} or wm'. This is absurd since Property 1 forbids the deletion and Property 3 prevents conflicts.

4. a metaclass C exists both in A' and A'' and these two versions differ on some parent. This can be caused by different applications of the *inherit* operator. This leads to an absurd since: i) a weaving model cannot delete a class parent for Property 1, ii) the sequential application of wm_{A_kx} and wm_{A_ky} cannot add class parents in a different way from wm' (wm' is the union of wm_{A_kx} and wm_{A_ky} and its existence ensures that Property 3 is satisfied).