Interfaces and Metainterfaces for Models and Metamodels

Anders Hessellund and Andrzej Wąsowski

IT University of Copenhagen, Denmark
{hessellund,wasowski}@itu.dk

Abstract. Evolution and customization of component-based systems require an explicit understanding of component inter-dependencies. Implicit assumptions, poor documentation and hidden dependencies turn even simple changes into challenges. The problem is exacerbated in XML-intensive projects due to the use of soft references and the lack of information hiding. We address this with dependency tracking interface types for models and metamodels. We provide automatic compatibility checks and a heuristic inference procedure for our interfaces, which allows easy and incremental adoption of our technique even in mature projects. We have implemented a prototype and applied it to two large cases: an enterprise resource planning system and a healthcare information system.

1 Introduction

The challenge of evolving and customizing component-based systems is well-known in both academia and industry. The solution strategies devised by computer scientists, however, often fail to deliver due to an underlying misconception of what a *component* is. As Bosch [1, p.14] points out, the classic academic view is that a component is a small black-box asset with a narrow interface and a single point of access. The industrial view, on the other hand, is that components are large assets with no encapsulation and no distinction between interface and non-interface entities. A large class of model-driven systems exacerbates this problem by relying on XML-based domain-specific languages (DSLs) which use *soft references*—untyped, string-based references between XML documents.

XML-based DSLs are often used in a significant class of model-driven systems following an approach which we term *interpretative.* These systems use models expressed in XML as first-class artifacts. The models are not used by code generators in the development phase but rather by interpreters at load- or runtime. The DSLs are defined by an XML Schema and cover a range of concerns from simple configuration to full-blown programming. Typically, multiple such languages are used in concert, as described in [2], to address different concerns of an individual application. Components in such systems are often large, have blurred boundaries, and consist of a mixture of models conforming to various schemas, and of code snippets in a general-purpose language for custom functionality.

The main advantage of using an XML-based language is the availability of generic tools for parsing and checking for schema conformance. However, there are also some serious problems. One of the most important of these problems is the use of soft references to tie together different XML models (possibly conforming to different metamodels). In order to refer to elements in other XML models, simple strings are used. The string references can not be checked by a standard schema conformance checker but rather by special tools such as SmartEMF [2] or Xlinkit [3] that require manual specification of every single reference in advance. Furthermore, since XML lacks an information hiding construct, there is no explicit definition of an interface between different XML models. To reveal dependencies between components implemented in such languages, one has to track every single reference instance, i.e., check every single XML model.

This paper presents an automatic, compositional approach to dependency tracking based on use of interfaces. The approach is applicable to a large class of model-driven systems, specifically *interpretative* systems using XML-based DSLs. Using a simple set of heuristics, we can infer interfaces for models and metainterfaces for metamodels. We discuss composition rules for model interfaces such that interfaces can be composed to create descriptions of larger units, like entire components. The approach is intended to be lightweight and easy to adopt. To validate this hypothesis, we have tested our approach on two existing industrial cases, an enterprise resource planning system [4] and a healthcare information system [5], and the results are promising. Our contributions are:

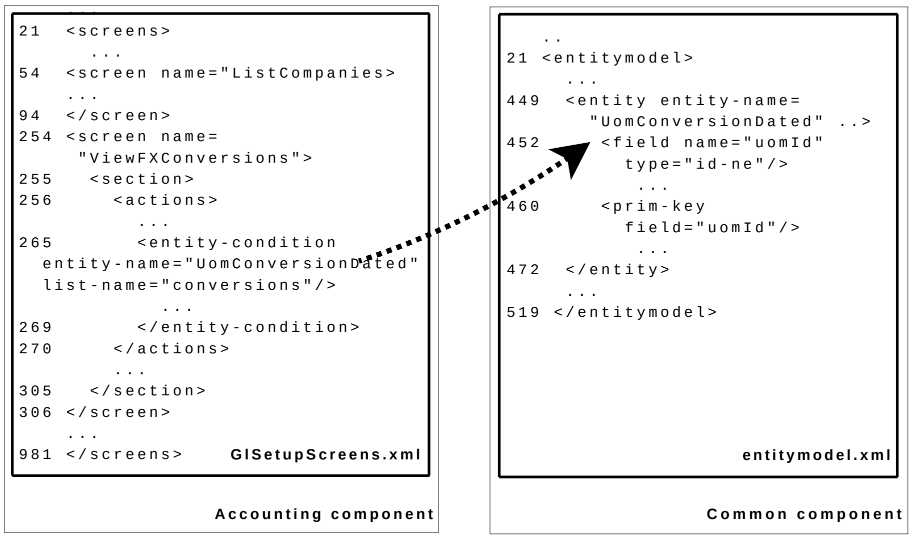
– A compositional approach to tracking soft dependencies based on interfaces
– An interface language for specification of interfaces and metainterfaces
– A notion of component interfaces via composition of model interfaces
– A lightweight bootstrapping procedure via inference for easy adoption
– An empirical evaluation in the form of two industrial case studies

Outline. A motivating problem selected from a case study is examined in Sect. 2. Sections 3–4 introduce interfaces for models and metamodels. Section 5 describes the interface inference implemented in our prototype, and Sect. 6 reports results of empirical validation. We address limitations and open problems in Sect. 7, and the related work in Sect. 8.

2 Dependencies and Soft References in OFBiz

We begin by demonstrating how hidden dependencies and soft references among models appear in actual systems. The Apache Open For Business (OFBiz) project [4], described in greater detail in Sect. 6, uses 17 different DSLs to define and implement various parts of its functionality. Two examples of these are: the *Screen Language* used to define user interface screens, and the *Entity Language* used to define business objects, their attributes and associations. A screen model usually refers to an entity model to render a business object in the user interface.

Figure 1 shows actual screen and entity models from the OFBiz project. The `ViewFXConversions` screen is located in the `Accounting` component and is used


```
<screens>
  ...
<screen name="ListCompanies>
...
</screen>
<screen name=
   "ViewFXConversions">
  <section>
    <actions>
      ...
      <entity-condition
  entity-name="UomConversionDated"
  list-name="conversions"/>
        ...
      </entity-condition>
    </actions>
    ...
  </section>
</screen>
...
</screens>
```

```
 ..
<entitymodel>
  ...
 <entity entity-name=
    "UomConversionDated" ..>
    <field name="uomId"
      type="id-ne"/>
       ...
    <prim-key
      field="uomId"/>
       ...
 </entity>
  ...
</entitymodel>
```

Fig. 1. The *GlSetupScreens.xml* contains general ledger screens, such as `ListCompanies` and `ViewFXConversions`, from the *accounting* component in OFBiz. The `ViewFXConversions` screen has a soft reference to the *common* component since it needs to render the `UomConversionDated` business object from the *entitymodel.xml*.

to display rates for foreign exchange conversions. The concept of a conversion rate is encapsulated in the `UomConversionDated` business object defined in the `Common` component. The `entity-name` attribute in the screen model is a reference to the entity model. When the interpreter encounters this attribute, it performs a lookup in the entity models and, in this case, renders the `UomConversionDated` object on the `ViewFXConversions` screen. This is a *soft reference.* Borrowing vocabulary from programming language theory, we call the attribute in the entity model the *name declaration* and the attribute in the screen model the *name use.* The reference must point to either an `entity` or a `view-entity` in the *Entity Language* but this is not checked until calltime when the screen is rendered. Developers must take care to manually update all references when they change the name declaration of a business object.

OFBiz is partitioned into a set of different components, such as Accounting and Inventory. In the example, the screen is defined in the Accounting component and the business object is defined in the Common component (that contains a library of basic, reusable objects and services). Since the screen refers to the entity model, the Accounting component has an implicit dependency on the Common component. XML does not have any language constructs for information hiding, so it is impossible to specify which elements of an XML model can be referred to. In other words, we can not state that the Common component provides `entity` and `view-entity` objects.

This means that replacing or updating an existing model or component in OFBiz requires a lot of work since every dependency must be revealed and checked manually. A previous study of OFBiz has shown that this is a significant cause of errors and a concern of the developers [2]. We address these difficulties by introducing an interface concept stating the provisions and requirements on three levels: First, on the metamodel level, we state which names are *provided*, i.e. declared for public access, and which external names are used in a metamodel, i.e. are *required* to exists. Second, on the model level, we state which exact (attribute instance,value)-pairs are provided and required. Third, a component is a collection of models, so on the component level, model interfaces should be composed to establish what a component requires and provides.

3 Interface Theory for Soft References

An *interface* is an abstract description of the way in which a *component* communicates with its *context*; It serves as a requirements specification for the developer of the component, a use specification for the user of the component, and a correctness specification for automatic verification. Our components are models stored in XML files at two levels of the modeling hierarchy: models and metamodels to which these models conform. A context for a model may be other models in the project. In this paper we restrict the communication between the model and a context to an ability to refer to, or being referred from, the context objects by means of soft references.

Since our interface theory for models only tracks soft references, our view of XML is grossly simplified. We define an *attribute instance* to be a fully qualified path to an attribute occurrence in an XML file, identifiable by an XPath expression. An attribute instance in a schema file is its conceptual counterpart. It corresponds to the `xs:attribute` notion in the XML Schema Specification. In our perspective, attribute instances, whether in XML files, or in schema files, are just abstract, distinct atomic units.

XML and schema files are just collections of attribute instances. For an XML file att_1 and a corresponding schema att_2 a many-to-one implementation relation $\mathcal{R}(\mathsf{att}_1 : \mathsf{att}_2) \subseteq \mathsf{att}_1 \times \mathsf{att}_2$ relates every attribute instance in att_1 to a single attribute instance in att_2. $\mathcal{R}$ is established by the standard XSD semantics.

Each model can potentially appear in multiple contexts. Since contexts are just sets of attribute instances from sets of model files, it seems natural to model them as unions of models. But a union of sets containing attribute instances is itself a set of attribute instances. So we define a context to be a set of attribute instances. Again in practice we distinguish contexts for models (other models) and contexts for meta models (other schema), but in our abstract view they are all just sets of distinguishable items. The interpretation is that if an attribute instance a is in some context ctx, so $a \in \mathsf{ctx}$, then ctx declares a, and it can be referred from other files using a soft reference. Such a generic treatment of contexts allows us to use interfaces both in small scale, where context can be fragments of models or entire models, and in the large, for collections of models, i.e., components.

Definition 1 (Interface). *An interface is a quadruple* $\mathcal{I} = (\mathsf{use}, \mathsf{decl}, \mathsf{req}, \mathsf{pro})$ *where* use *and* decl *are disjoint sets of attribute instances,* req *is a consistent propositional logic formula over* use*, and* pro *is a formula over* $\mathsf{use} \cup \mathsf{decl}$*. Moreover for every context* $\mathsf{ctx} \subseteq \mathsf{use}$ *we have that* $(\mathsf{req} \rightarrow \mathsf{pro})_{\mathsf{ctx}}$ *is consistent.*

Expression $(\varphi)_{\mathsf{ctx}}$ denotes a specialization of φ in the context ctx. Each occurrence in φ of a name $v \in \mathsf{use}$ is substituted with true if $v \in \mathsf{ctx}$ and with false otherwise.

Intuitively, the req formula expresses the necessary condition for a component to deliver its functionality; req must be formulated in terms of attribute instances that are found in the use set. In simple interfaces req just requires one attribute instance to exist. When a component refers to two or more attribute instances, then req is a conjunction of them. A reference to an abstract concept can be modeled as a disjunction of several references.

Consider the example in Table 1. On top, the signature (use and decl) of the interface is stated: four names might be used, and two might be guaranteed to be declared in this interface. The signature merely limits the symbols used in the remaining part of the interface. The *requires* block, which corresponds to req in Definition 1, states that this interface refers to an instance of attribute `entity-name` with value `'UomConversionDated'`, which can be either provided by an `entity` or a `viewentity`. A sum of the two XML scopes (`entity` and `view-entity`) effectively constitutes an abstract scope here, which is modeled by a disjunction in the interface. Similarly, this interface also requires an `entity` or a `view-entity` named `'PartyRole'` that is referenced from the `ListCompanies` screen. This entity has not been shown in Figure 1 for space reasons.

The formula pro, of Definition 1, describes ways in which the model can satisfy the interface. In the most common form of an interface pro is formulated in terms of declared names found in the decl set, and is just a simple conjunction, as in Table 1. In more sophisticated cases, general expressions can be used to express abstract interfaces—for example to summarize several similar components that differ slightly in what they provide. We allow use of names from use in the requires part in order to be able to express these more nuanced interfaces,

Table 1. An interface for `GlSetupScreens.xml` (Ofbiz's general ledger UI)

```
names declared: /screens/screen[@name='ViewFXConversions']
                /screens/screen[@name='ListCompanies']
names used: /entitymodel/entity[@entity-name='UomConversionDated']
            /entitymodel/view-entity[@entity-name='UomConversionDated']
            /entitymodel/entity[@entity-name='PartyRole']
            /entitymodel/view-entity[@entity-name='PartyRole']
requires: (/entitymodel/entity[@entity-name='UomConversionDated']
           xor /entitymodel/view-entity[@entity-name='UomConversionDated'])
      and (/entitymodel/entity[@entity-name='PartyRole']
           xor /entitymodel/view-entity[@entity-name='PartyRole'])
provides: /screens/screen[@name='ViewFXConversions']
      and /screens/screen[@name='ListCompanies']
```

```
names declared: /entitymodel/entity[@entity-name='UomConversionDated']
requires: true
provides: /entitymodel/entity[@entity-name='UomConversionDated']
```

Table 2. An interface for `EntitityModel.xml`

for example when a part of the functionality is only available given a certain precondition. This generalization creates a potential problem: we might write a *provides* block that is inconsistent with the associated *requires* block (for example require a and guarantee $\neg a$). Since such interfaces are obviously not useful, we restrict ourselves to *well formed* interfaces in the last sentence of Def. 1.

There are several kinds of verification to be considered in the presence of interfaces. First, a *conformance* procedure checks if a model obeys its own interface. Second, *refinement* or *subtyping* means that one interface properly strengthens another one, without breaking any other models, which might be using it. Third, the most central of these, *compatibility* of interfaces, means that for any two interfaces their composition models the composition of any two components implementing them in such a way that if the interfaces are compatible then there exist contexts that can use these two components combined.

The subtyping question is contravariant in nature: a subtype of an interface may relax the assumptions, and may strengthen the guarantees. Formally:

Definition 2 (Subtyping). *For two interfaces* $\mathcal{I}_1 = (\mathsf{use}_1, \mathsf{decl}_1, \mathsf{req}_1, \mathsf{pro}_1)$ *and* $\mathcal{I}_2 = (\mathsf{use}_2, \mathsf{decl}_2, \mathsf{req}_2, \mathsf{pro}_2)$ *say that* $\mathcal{I}_1$ *is a subtype of* $\mathcal{I}_2$*, written* $\mathcal{I}_1 \preceq \mathcal{I}_2$*, iff*

- $\mathsf{use}_1 \subseteq \mathsf{use}_2$ *and* $\mathsf{decl}_1 \supseteq \mathsf{decl}_2$*,*
- *and formulas* $(\mathsf{req}_2 \rightarrow \mathsf{req}_1)$ *and* $(\mathsf{pro}_1 \rightarrow \mathsf{pro}_2)$ *are valid.*
- *and* $(\mathsf{req}_1 \rightarrow \mathsf{pro}_1) \rightarrow (\bigvee_{\mathsf{decl}_2 - \mathsf{decl}_1} \mathsf{req}_2 \rightarrow \mathsf{pro}_2)$ *is valid.*[1]

Recall that req_i are interpreted as constraints on the contexts in which $\mathcal{I}_i$ may be used. The implication $\mathsf{req}_2 \rightarrow \mathsf{req}_1$ means that $\mathcal{I}_1$ constrains the environment no more than $\mathcal{I}_2$ does. Similarly pro_i are interpreted as resources guaranteed by $\mathcal{I}_i$, so $\mathsf{pro}_1 \rightarrow \mathsf{pro}_2$ means that any guarantee given by $\mathcal{I}_1$ must be stronger than the one given by $\mathcal{I}_2$. The final formula states that the interface refinement is conservative: it should not change the guarantees already specified in the part being refined. The refining interface can freely introduce constraints on new variables though.

Interface theories take an *existential* approach to compatibility: two interfaces are *compatible* if there exists a context in which they can be legally used.

Definition 3 (Compatibility). *Two interfaces* $\mathcal{I}_1 = (\mathsf{use}_1, \mathsf{decl}_1, \mathsf{req}_1, \mathsf{pro}_1)$ *and* $\mathcal{I}_2 = (\mathsf{use}_2, \mathsf{decl}_2, \mathsf{req}_2, \mathsf{pro}_2)$ *are compatible iff* $\mathsf{decl}_2 \cap \mathsf{decl}_1 = \emptyset$ *and there exists a context* $\mathsf{ctx} \subseteq \mathsf{use} = (\mathsf{use}_1 \cup \mathsf{use}_2) - (\mathsf{decl}_1 \cup \mathsf{decl}_2)$ *such that the formula* $[(\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \rightarrow \mathsf{pro}_1) \wedge (\mathsf{req}_2 \rightarrow \mathsf{pro}_2)]_{\mathsf{ctx}}$ *is consistent.*

[1] Notation $\bigvee_V \varphi$ means elimination of variables in V from φ in propositional logic (a form of finite existential quantification): for one variable $\bigvee_v \varphi \equiv \varphi[0/v] \vee \varphi[1/v]$.

Let us elaborate on the above definition. First of all two compatible interfaces need to have disjoint sets of declared attribute instances, as otherwise it would be impossible to resolve soft references to any shared name. Second, there must exists a context in which the composition can be legally used. Third, in this context one must be able to satisfy an assumption of one or the other of the interfaces (or both) while still being able to fulfill their obligations. Mind that a context ctx turns all the variables in use into constants, so satisfiability of the last two terms implies that it is possible to assign declared variables in $\mathsf{decl} = \mathsf{decl}_1 \cup \mathsf{decl}_2$ so that the original constraint of both interfaces is fulfilled.

Any two compatible interfaces can be composed, and we give rules to synthesize the interface for the composition. Assume names as in Definition 3. Then the result of composing $\mathcal{I}_1$ and $\mathcal{I}_2$ is an interface $\mathcal{I} = (\mathsf{use}, \mathsf{decl}, \mathsf{req}, \mathsf{pro})$, where $\mathsf{decl} = \mathsf{decl}_1 \cup \mathsf{decl}_2$, $\mathsf{req} := \bigvee_{\mathsf{decl}} .(\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \rightarrow \mathsf{pro}_1) \wedge (\mathsf{req}_2 \rightarrow \mathsf{pro}_2)$, and $\mathsf{pro} = (\mathsf{req}_1 \rightarrow \mathsf{pro}_1) \wedge (\mathsf{req}_2 \rightarrow \mathsf{pro}_2)$.

Example 1. Consider a composition of the interface from Table 1 with the entity interface in Table 2. The entity interface fulfills the first assumption in the screen interface, but the second one remains to be pending. The two interfaces are compatible in the sense of Definition 3. The resulting composed interface is:

```
names declared: /screens/screen[@name='ViewFXConversions']
                /screens/screen[@name='ListCompanies']
                /entitymodel/entity[@entity-name='UomConversionDated']
names used: /entitymodel/view-entity[@entity-name='UomConversionDated']
            /entitymodel/entity[@entity-name='PartyRole']
            /entitymodel/view-entity[@entity-name='PartyRole']
requires: (not /entitymodel/view-entity[@entity-name='UomConversionDated'])
          or (/entitymodel/entity[@entity-name='PartyRole']
          xor /entitymodel/view-entity[@entity-name='PartyRole'])
provides: /entitymodel/entity[@entity-name='UomConversionDated']
      and ((not /entitymodel/view-entity[@entity-name='UomConversionDated']
          and (/entitymodel/entity[@entity-name='PartyRole']
               xor /entitymodel/view-entity[@entity-name='PartyRole']))
          implies (/screens/screen[@name='ViewFXConversions']
               and /screens/screen[@name='ListCompanies']))
```

We encourage the reader to perform the calculation herself. In practice, this synthesis can be performed using binary decision diagrams (BDDs) or a theorem prover, and further compatibility checking can also be automated. Observe that the synthesized interface promises to deliver the entity named `'UomConversion-Dated'` unconditionally. However delivery of `'ViewFxCompanies'` and `'ListCom-panies'` depends on absence of a view named `'UomConversionDated'` (as this conflicts with the entity present in the interface), and availability of either an entity or a view named `'PartyRole'`. In order to use this interface (the requires section) you need to either refrain from providing the conflicting view, or make `'PartyRole'` available. Otherwise there is nothing this component can offer.

Other interesting uses of composition are possible. For example, an interface for a complicated file, can be created by treating parts of the file as separate models, specifying interfaces for them individually, and automatically synthesizing the interface for the entire file by applying the above composition rule. If you want to know what needs to be guaranteed to safely access `'ListCompanies'`, you can use the composition algorithm to get the answer, namely that `'UomConversionDated'` should be provided. Simply compose the interface that provides `'ListCompanies'`, with another one that requires it.

We admit that the synthesised interfaces are complex, even unsuitable to be read by humans. We should reiterate that these are not to be written manually, but synthesized and checked for compatibility automatically. In the long run the framework should incorporate an interactive exploration tool, which allows the user to investigate combinatorial spaces in the spirit of interactive product configuration [6]—without reading the specifications .

Last but not least, as any typical interface theory, ours also offers *substitutability* by design: for any two compatible interfaces $\mathcal{I}$ and $\mathcal{J}$, if $\mathcal{I}' \preceq \mathcal{I}$ and $\mathcal{J}' \preceq \mathcal{J}$ then $\mathcal{I}'$ and $\mathcal{J}'$ are compatible, allowing to safely substitute $\mathcal{I}'$ in place of $\mathcal{I}$ and, $\mathcal{J}$ in place of $\mathcal{J}'$ without introducing any obligation to recheck compatibility, as long as $\mathcal{I}'$ and $\mathcal{J}'$ do not *declare* any new names, which are already *used* in $\mathcal{I}$ or $\mathcal{J}$. This crucial property means that a developer can substitute a model without rechecking compatibility as long as the original and the new model implement the same interface.

4 Metainterfaces

Known interface theories do not explicitly distinguish between the meta level and the model level. They typically treat implementations uniformly, as completely refined specifications. Such interface frameworks are simpler to implement, and simpler to reason about. However since in development with DSLs, both models and metamodels are first-class artifacts that are being developed and evolved, it makes sense to exploit their co-existence and the special relation between the two kinds of models to obtain a more precise interface inference with less burden on users.

In Sect. 3, we have defined the schemas and XML files to be just sets of attribute instances at the two levels. Consequently the framework presented therein applies just as well to schemas as to the models, and all the definitions and claims are still valid (interfaces, subtyping, compatibility, composition, and substitutability). Even though at the metalevel we model associations between elements, as opposed to references between objects, at the plain logical level there is no difference between composing models and composing metamodels.

We show how metainterfaces together with metamodels can be useful in establishing conformance of models to interfaces. The correctness of a model is established in two steps: first the conformance of the XML file to its schema is checked using regular XML technology. Then the conformance of the XML file to its interface is checked, involving the schema interface. Why involve the schema? Ideally we would like to claim that if a model conforms to an interface, then it declares all the required names and does not use any names that are not mentioned in the `requires` block. While the former can be robustly checked, the latter cannot due to lack of information. Any attribute instance in the XML file can be a soft reference, but we do not know whether it is one and, if so, which attribute it uses. To provide a check for this conformance we enrich interfaces used for metamodels with additional information—links between the used names and the attribute instances (`xs:attribute`) that use them. This information, specified once per metamodel, can be efficiently reused for all conforming models.

Consider the example of a metainterface in Table 3. Note that the syntax is essentially the same, except that the names no longer include particular values, and we add `by` clauses to indicate the internal attributes responsible for the use. A formal definition of the metainterface follows.

Definition 4 (Metainterface). *A metainterface is a 6-tuple* $\mathcal{I} = (\mathsf{use}, \mathsf{decl}, \mathsf{req}, \mathsf{pro}, \mathsf{int}, \mathsf{by})$ *where* $(\mathsf{use}, \mathsf{decl}, \mathsf{req}, \mathsf{pro})$ *is an interface and* int *is a set of internal attribute instances disjoint with both* use *and* decl*, while* $\mathsf{by} \in \mathsf{int} \mapsto \mathsf{use}$ *maps internal attribute instances to external ones.*

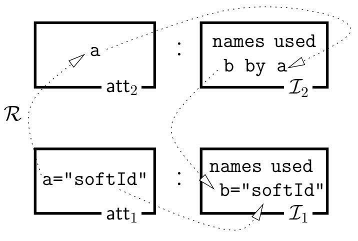


Fig. 2. Checking conformance of name uses

An XML file att_1 conforms to its interface $\mathcal{I}_1$, iff (i) att_1 actually contains the attribute instances that are provided in $\mathcal{I}_1$ and (ii) the interface $\mathcal{I}_1$ declares the use of all the attribute instances used in att_1. Since (ii) cannot be checked automatically we approximate it using a metainterface. See the overview in Fig. 2. For every attribute instance $a \in \mathsf{att}_1$ (here a is $\mathsf{a} =$ "`softId`") detect whether a makes an external reference by checking whether the corresponding attribute (here `a`) is mentioned in the metainterface $\mathcal{I}_2$, in a `by` clause. If so, we require that the corresponding used name, seen in the metainterface (here `b`), is declared as used in the interface with the same value as a (here `softId`).

Definition 5 (Conformance). *Let* att_1 *be an XML file,* att_2 *be a schema file to which* att_1 *conforms, and* $\mathcal{R}$ *be an implementation relation between the two. Let* $\mathcal{I}_1 = (\mathsf{use}_1, \mathsf{decl}_1, \mathsf{req}_1, \mathsf{pro}_1)$*, and* $\mathcal{I}_2 = (\mathsf{use}_2, \mathsf{decl}_2, \mathsf{req}_2, \mathsf{pro}_2, \mathsf{int}_2, \mathsf{by})$ *be the corresponding interface and metainterface. We say that* att_1 *conforms to* $\mathcal{I}_1$*, written* $\mathsf{att}_1 : \mathcal{I}_1$*, iff* $(\mathsf{req}_1 \rightarrow \mathsf{pro}_1)_{\mathsf{att}_1}$ *is a valid formula and for every attribute instance* $a \in \mathsf{att}_1$ *such that* $\mathcal{R}(a) \in \mathsf{att}_2$ *and* by *is defined for* $\mathcal{R}(a)$ *in* $\mathcal{I}_2$*, we have that* $\mathsf{by}(R(a)) = \mathit{value}(a)$ *and* $(\mathsf{req}_1)_{\mathsf{att}_1} \rightarrow \mathsf{by}(R(a))$

Let us summarize the ingredients of our framework. We provide interfaces and meta-interfaces for models and metamodels, with algorithms for checking compatibility and subtyping and for computing compositions. For models we also allow conformance checking. Complete conformance checking for metamodels is impossible due to lack of knowledge on which attributes are soft references (unless a metametamodel is available). We do not think this is a large problem. Metainterfaces tend to be much simpler and smaller than interfaces and are specified by DSL designers. At the same time, interfaces, are used and written by regular users of DSLs, for whom the automatic support is much more valuable.

5 Interface Inference

We have seen how interfaces are specified, verified and composed. If this technology is to be useful in realistic development projects, we need to address another pressing question: how to easily specify a large number of interfaces? Below we propose a heuristic technique to automatically *infer* interfaces reflecting the current dependencies in a project. In Sect. 6, we evaluate the technique by applying its implementation to two large development projects.

The central idea is to discover associations between metamodels by heuristically mining all references between attribute instances in the available collection of models. Once metainterfaces are in place, a refinement procedure is applied to generate model interfaces. Our hypothesis is that this *bootstrapping* process, given a sufficiently large number of models, results in fairly precise approximations of the interfaces.

The inferred interfaces explicitly represent combinatorial dependencies, addressing the problems described in Sect. 1–2, allowing for safer evolution and customization of model-based components. Since we do not require any special conventions from the systems under analysis, the technique can be applied incrementally and for a large class of systems. Optional configuration files allow us to cater for domain-specific heuristics and coding conventions.

Identifying Potential Soft References. We begin with identifying all soft reference candidates in the models. A soft reference consists of a name declaration and a name use that are attributes with the same value. An attribute is a *potential name declaration* if it always takes unique values (so it can be used as a primary key). An attribute is a *potential name use* if all values it takes are a subset of the values taken by some potential name declaration attribute. Formally if ref and key are two different attributes, and ref_1 and key_1 their corresponding nonempty instance sets there is a potential simple soft reference with ref being the *name use* and key its name declaration if $ref_1 \subseteq key_1$.

In the example of Sect. 2, attribute `entity-name` on the `entity-condition element` refers to either an `entity` or a `view-entity`. This is because both `entity` and `view-entity` are subtypes of some implicit abstraction. The above simple inclusion check fails in such cases because the name declaration of a potential soft reference is really a union of all names participating in the abstraction;

Table 3. Metainterface for `widget-screens.xsd`, metamodel of `GlSetupScreens.xml`

```
names declared: /screens/screen[@name]
names used: /entitymodel/entity[@entity-name]
                by (/screens/screen/actions/entity-condition[@entity-name])
            /entitymodel/view-entity[@entity-name]
                by (/screens/screen/actions/entity-condition[@entity-name])
requires: /entitymodel/entity[@entity-name]
          or /entitymodel/view-entity[@entity-name]
provides: /screens/screen[@name]
```

here `entity` and `view-entity`. The values of a *name use* referring to such an abstraction will be a subset of this union and not of any of the individual sets. Mining uses of such abstractions is similar to mining for class hierarchies.

Let *ref* be an attribute, *keys* be a set of attributes from the same metamodel, and ref_1 and $key_{1..n}$ be their corresponding, non-empty pairwise disjoint instance sets. We identify a potential soft reference to an abstraction with *ref* as its name use and an abstraction over *keys* as its name declaration, if for each i we have that $ref \not\subseteq key_i$ while $ref \subseteq \bigcup_{i=1}^{n} key_i$.

Our prototype loads and parses all XML files in a project, identifying potential simple references and references to abstractions as defined above. After that a graph representing potential references is created. Every vertex in the graph represents either an attribute or an abstraction over attributes. There is an edge between every potential declaration and use, and between every abstraction and its members. Figure 3 shows a small fragment of the graph created for the example of Sect. 2 where an abstraction is created over `entity` and `view-entity` in the entity model. The `entity-condition[@entity-name]` attribute is shown to be a possible name use of a soft reference between itself and this newly formed abstraction. Effectively edges in this graph visualise cross-model references.

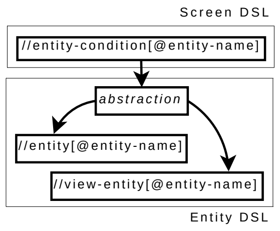


Fig. 3. A excerpt of the graph from the example.

As the last step, all incoming edges of non-abstract vertices with both a positive in- and out-degree are removed, assuming that it is unlikely that an attribute can serve as both name declaration and name use at the same time.

Synthesize interfaces for metamodels. The next step in the process is the synthesis of an interface for each metamodel. The graph created above contains all the necessary information. For each metamodel, we simply have to locate vertices in the graph that corresponds to attributes or abstractions from that metamodel. For each of these vertices, we create an entry in the interface description of the metamodel. A vertex with an incoming edge results in a `provides` clause and a vertex with any outgoing edges results in a `requires` clause in the interface description. Synthesizing an interface for entity language based solely on the graph in figure 3 will give the following result:

```
names declared: n/a
requires: true
provides: /entitymodel/entity[@entity-name]
          /entitymodel/view-entity[@entity-name]
```

In other words, a metamodel provides a set of attributes that are name declarations of soft references. Similarly, a metamodel requires a set attributes that are name uses for each soft reference that originates in this metamodel. Interestingly, for abstractions we synthesize disjunctions in the metamodel where the name use is located. This is, for instance, shown in the *requires* clause of Table 3.

Refine interfaces from metamodels to models. We shall now generate interfaces for the models that were used to synthesize the graph and the metainterfaces. The synthesis forgoes by refining the metainterfaces.

Each entry in the interface description of a metamodel has a corresponding XPath expression. Evaluating such an XPath expressions results in a set of attribute instances. These instances are either name declarations or name uses of soft references as specified in the metamodel interface. The model interface entry is synthesized by taking an entry from the metamodel interface description and adding the concrete value of the attribute instance. Models do not just provide and require attributes but also attribute values. If two models are composed then not only must the correct attributes be present but these must also have specific values in order for the model interfaces to be compatible.

6 Empirical Report

In order to validate our claims we have tested the prototype on two industrial case studies—two *interpretative*, model-driven systems heavily relying on XML-based DSLs. We describe the nature of the two case studies now, before delving into a discussion of the results of experiments and limitations of our analysis.

Our main requirement when choosing the cases was to allow easy good access to code, so that our claims can be independently validated. Each case should furthermore be implemented in an *interpretative*, model-driven manner with XML based primary modeling languages. We selected one case which actually uses models more than regular code and another case which has the more common approach of using a 1:5 ratio between models and code.

The first case study is the Apache Open For Business (OFBiz) project [4]. OFBiz is an open source enterprise resource planning (ERP) system. It consist of a set of modules with basic ERP functionality such as accounting, manufacturing, e-commerce, inventory etc. It has been deployed in a large number of medium-sized companies. Larger users include United Airlines and British Telecom [7]. The implementation is in J2EE and XML and as a rough estimate of the size: the out-of-the-box solution consists of approximately 180 000 lines of Java code and 195 000 lines of XML.

The second case study is the District Health Information System (DHIS) [5] which is developed under the Health Information Systems Programme (HISP). DHIS is an open source healthcare information system which is developed as a joint project between participants in Cape Town and Oslo. It has been deployed in several African and Asian countries such as Mozambique, Malawi, Ethiopia, Nigeria, Vietnam and India. The implementation is in Java and XML and has a size of roughly 51 000 lines of Java code and 11 500 lines of XML.

We have implemented our approach in a Java-based program and tested it on the two cases. For each case, we ran a test with and without heuristic rules in order to see how performance and results were affected. We were interested in finding the number of inclusions and abstractions to see how many soft reference candidates were identified by our initial algorithm. Finally, the prototype

Table 4. Results of testing the prototype on OFBiz and DHIS with and without heuristics. Columns, *#inclusions* and *#abstractions*, are the number of potentiel simple references and potential references to abstractions. The *edges* column corresponds to the final number of potential soft reference candidates after running all checks. The rightmost column, *result*, indicates the ratio between true and false positives for a sample language

Setup	time	#models	#m.models	#inclusions	#abstractions	#edges	result
OFBiz, 8 rules	80.5s	609	14	61	12	46	29:1
OFBiz, 0 rules	81.3s	609	14	1112	39	391	n/a
DHIS, 3 rules	32.3s	176	6	58	7	73	8:1
DHIS, 0 rules	24.3s	176	6	245	8	106	n/a

cleans up the graph as described in section 5 and produces a set of edges which form the basis for the synthesized interfaces. Table 4 shows the results of these experiments[2].

The heuristics that we have used are typically rather simple and fairly application-specific. For the OFBiz case, we were, as table 4 shows, able to reduce the number of edges, i.e., soft reference candidates, from 391 to 46 by 8 simple heuristics. An example of such heuristic is: *ignore all attributes from any metamodel if the value of an instance of this attribute contains the character sequence '${'*. The reasoning behind this heuristic is that the character sequence '${' indicates a reference to a java properties file as used in most localization schemes. The heuristics were devised in an incremental manner by looking at the produced graph and then writing heuristics to capture detected false positives. Table 4 clearly shows how especially the number of inclusions and edges decrease after the introduction of heuristics.

To evaluate our findings, we would ideally have to manually inspect all edges and check them against the documentation of the 20 DSLs involved in the experiment. This exercise would give us an exact ratio between false and true positives in the two experiments. Instead we have chosen a more limited evaluation where we approximate these figures by manually inspecting 1 sample language from each experiment. We used the Spring Beans Configuration language for DHIS and the Entity language for OFBiz, the results can be seen in the rightmost column in table 4. Further inspection of the remaining 18 DSLs will of course be necessary in order to completely ascertain the value of the approach. We do, however, claim that ratios 8:1 and 29:1 between true and false positives in the two case studies are very convincing results.

7 Discussion and Future Work

Identifying abstractions. Mining abstractions from attribute instances by means of searching for a partition built of all overlapping attribute instance sets was

[2] We excluded Minilang, entityconfig and serviceconfig DSLs from the setup for OFBiz

well-suited for our case studies (see Sect. 5). In general, the partition of the reference value might be realized by a *selection* of overlapping sets, as opposed to all of them. A more general mining algorithm for abstractions could be devised by solving vertex cover problems [8, ch.6.3]. We have not attempted this so far.

Identifying false negatives. One of limitations of our approach is that results depend on the kinds of models that serve as input. If none of the models use a certain language feature, e.g., such as a soft reference, then this reference will not be detected. In terms of the evaluation that we described in section 6, these undetected soft references can be categorized as false negatives. An evaluation of the number of false negatives in the two cases remains future work.

Applicability to other systems. We assert that this approach is widely applicable to *interpretative* model-driven systems. Two XML-based DSLs, particularly interesting as test cases, are the XML User Interface Language (XUL) from the Mozilla Foundation and the Extensible Application Markup Language (XAML) from Microsoft. Both use a variant of soft references. On top of that XUL embeds JavaScript that poses an additional (static analysis) challenge, which we would have to address. Then using our prototype one could synthesize interfaces for libraries of user interface components in both languages in order to support evolution and upgrade of applications relying on such libraries.

8 Related Work

Our interfaces are mildly inspired by assume/guarantee interfaces of Alfaro and Henzinger [9], adapted to tracking soft references between models. Interface theories [9–13] have been so far developed using small scale case studies in embedded systems [14, 15] and reliability engineering [16], but not in development of large scale software intensive systems. We are not aware of any attempts of interface inference in the interfaces community, or of exploitation of the relation between interfaces and metainterfaces.

The study of usage of multiple domain-specific languages in a single application has gained more interest in recent years. Work on static checking of models that span multiple DSLs can be found in tools such as Xlinkit [3] and SmartEMF [2]. These tools do, however, require manual specification rather than automatic extraction of the soft references before they can be checked.

Our approach can be seen as an attempt to uncover types in a set of untyped models similar to *soft types* in Scheme [17], JavaScript [18], and embedded DSLs in Java [19]. Our approach is different since our soft types are extracted through the automatic bootstrapping process described in Sect. 5.

9 Conclusion

We have offered a compositional approach to tracking soft dependencies based on interfaces. Using a new language for interfaces and metainterfaces, we can express complex dependencies and describe compositions of models into components. Our approach is lightweight and can be used to infer interfaces for existing systems. We have demonstrated this by implementing a prototype and applying it to two cases: an enterprise resource planning system and a healthcare information system. The approach and prototype are highly configurable and could be applied to a larger class of similar systems.

References

1. Bosch, J.: Design & Use of Software Architectures - Adopting and evolving a product-line approach. Addison Wesley (2000)
2. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided development with multiple domain-specific languages. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS. Volume 4735 of LNCS., Springer (2007) 46–60
3. Nentwich, C., Emmerich, W., Finkelstein, A.: Static consistency checking for distributed specifications. In: ASE, IEEE Computer Society (2001) 115–
4. The Apache Software Foundation: The Open For Business Project (OFBiz). `http://ofbiz.apache.org/` (2008) Seen May 7th, 2008.
5. Health Information Systems Programme (HISP): District Health Information Software (DHIS). `http://www.hisp.info/` (2008) Seen May 7th, 2008.
6. Junker, U.: Configuration. In Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier Science Inc., New York, NY, USA (2006)
7. Chen, S.: Opening Up Enterprise Software: Why Enterprises are Adopting Open Source Applications (2006) `http://www.opensourcestrategies.com/slides/`.
8. Valiente, G.: Algorithms on Trees and Graphs. Springer (2002)
9. Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In Henzinger, T.A., Kirsch, C.M., eds.: EMSOFT. Volume 2211 of LNCS. (2001)
10. Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In Sangiovanni-Vincentelli, A.L., Sifakis, J., eds.: EMSOFT. Volume 2491 of LNCS. (2002)
11. Alfaro, L., Henzinger, T.A.: Interface automata. In: 9th Annual Symposium on Foundations of Software Engineering (FSE), ACM Press (2001) 109–120
12. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In Nicola, R.D., ed.: ESOP. Volume 4421 of LNCS. (2007)
13. Larsen, K.G., Nyman, U., Wąsowski, A.: Interface input/output automata. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of LNCS. (2006)
14. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In Alur, R., Lee, I., eds.: EMSOFT. Volume 2855 of LNCS. (2003)
15. Easwaran, A., Lee, I., Sokolsky, O.: Interface algebra for analysis of hierarchical real-time systems. In: Foundations of Interface Technologies (FIT). (2008)
16. Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Rich interfaces for dependability: Compositional methods for dynamic fault trees and Arcade models. In: Foundations of Interface Technologies (FIT). (2008)
17. Wright, A.K., Cartwright, R.: A practical soft type system for scheme. In: LISP and Functional Programming. (1994) 250–262
18. Thiemann, P.: Towards a type system for analyzing javascript programs. In Sagiv, S., ed.: ESOP. Volume 3444 of LNCS., Springer (2005) 408–422
19. Hessellund, A., Sestoft, P.: Flow analysis of code customizations. In Vitek, J., ed.: ECOOP. Volume 5142 of Lecture Notes in Computer Science., Springer (2008) 285–308

20. Michail, A.: Data mining library reuse patterns using generalized association rules. icse **00** (2000) 167
21. Michail, A.: Data mining library reuse patterns in user-selected applications. ase **0** (1999) 24

A The Formal Interface Theory

In general in the following the view of context is as characteristic functions of sets (as used in the main paper). This is a small technical difference, but useful to have in the feature. This way the context also carries its scope with it.

The following is an equivalent (and more formal) definition of well-formedness of interfaces than the one in the text. Well in fact we add one thing, which is only implicit in the main text, namely that an interface should be usable (so we rules out vacuous well-formedness).

Definition 6 (Well-formedness). *An interface $\mathcal{I} = (\mathsf{use}, \mathsf{decl}, \mathsf{req}, \mathsf{pro})$ is well formed if (i) for every context $\mathsf{ctx} \subseteq \mathsf{use}$ we have that $(\mathsf{req} \to \mathsf{pro})_{\mathsf{ctx}}$ is consistent, and (ii) $(\mathsf{req})_{\mathsf{ctx}}$ is consistent itself.*

The first claim to prove is that a result of a composition of two compatible well-formed interfaces is itself well-formed.

Lemma 1. *Let $\mathcal{I}_1 = (\mathsf{use}_1, \mathsf{decl}_1, \mathsf{req}_1, \mathsf{pro}_1)$ and $\mathcal{I}_2 = (\mathsf{use}_2, \mathsf{decl}_2, \mathsf{req}_2, \mathsf{pro}_2)$ be two compatible interfaces. Then the result of composing $\mathcal{I}_1$ and $\mathcal{I}_2$, the interface $\mathcal{I} = \mathcal{I}_1 | \mathcal{I}_2$ is well-formed.*

Proof. Consider an arbitrary context $\mathsf{ctx} \subseteq \mathsf{use} = \mathsf{use}_1 \cup \mathsf{use}_2 - (\mathsf{decl}_1 \cup \mathsf{decl}_2)$. Let $\mathsf{decl} = \mathsf{decl}_1 \cup \mathsf{decl}_2 = \{d_1, \ldots, d_k\}$, $\mathsf{req} := \bigvee_{d_1} \cdots \bigvee_{d_k} (\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \to \mathsf{pro}_1) \wedge (\mathsf{req}_2 \to \mathsf{pro}_2)$, and $\mathsf{pro} = (\mathsf{req}_1 \to \mathsf{pro}_1) \wedge (\mathsf{req}_2 \to \mathsf{pro}_2)$. We need to show that $(\mathsf{req} \to \mathsf{pro})_{\mathsf{ctx}}$ is satisfiable, so that

$$\left[\bigvee_{d_1} \cdots \bigvee_{d_k} (\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \to \mathsf{pro}_1) \wedge (\mathsf{req}_2 \to \mathsf{pro}_2)\right] \\ \to ((\mathsf{req}_1 \to \mathsf{pro}_1) \wedge (\mathsf{req}_2 \to \mathsf{pro}_2)) \quad (1)$$

is consistent (which is obvious). Then we also need to argue that req itself is consistent, so that

$$\bigvee_{d_1} \cdots \bigvee_{d_k} (\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \to \mathsf{pro}_1) \wedge (\mathsf{req}_2 \to \mathsf{pro}_2) \quad (2)$$

is satisfiable. This however follows directly from compatibility of $\mathcal{I}_1$ and $\mathcal{I}_2$. □

In the paper we have claimed substitutability by design. Let us restate it here as a theorem. Observe a small but an important detail, that we require the new, stronger interface do not have any clashes of newly introduced names. This is a usual additional requirement, as subtyping usually does not track name clashes with yet unknown components.

Theorem 1. *For any two compatible interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, if $\mathcal{I}'_1 \preceq \mathcal{I}_1$ and $\mathcal{I}'_2 \preceq \mathcal{I}_2$ and $\mathsf{decl}'_1 \cap \mathsf{decl}'_2 = \emptyset$, $(\mathsf{decl}'_1 - \mathsf{decl}_1) \cap \mathsf{use}_2 = \emptyset$, $(\mathsf{decl}'_2 - \mathsf{decl}_2) \cap \mathsf{use}_1 = \emptyset$ then $\mathcal{I}'_1$ and $\mathcal{I}'_2$ are compatible.*

It is quite clear that the following simpler lemma suffices to prove the above:

Lemma 2. *For any two compatible interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, if $\mathcal{I}'_1 \preceq \mathcal{I}_1$ and $\mathsf{decl}'_1 \cap \mathsf{decl}_2 = \emptyset$, $(\mathsf{decl}'_1 - \mathsf{decl}_1) \cap \mathsf{use}_2 = \emptyset$ then $\mathcal{I}'_1$ and $\mathcal{I}_2$ are compatible.*

The condition that $\mathsf{use}_2 \cap (\mathsf{decl}'_1 - \mathsf{decl}_1) = \emptyset$ above, corresponds to usual static binding assumption. If $\mathcal{I}_2$ is linked against something implementing $\mathcal{I}_1$ then any refinement of $\mathcal{I}_1$ should not provide other names that $\mathcal{I}_2$ needs – normally the linking mechanism would search for these names in the environment for the composed component. This is a side effect of introducing stating typing into a dynamic mechanism. Perhaps we could do better, and lift it for dynamic cases like soft refs in XML, but this requires more research.

Proof. Consider a complete context ctx be such that:

$$[(\mathsf{req}_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}_1 \rightarrow \mathsf{pro}_1) \wedge (\mathsf{req}_2 \rightarrow \mathsf{pro}_2)]_{\mathsf{ctx}} = 1 \tag{3}$$

We know that such a context exists due to compatibility of $\mathcal{I}_1$ and $\mathcal{I}_2$. We shall find a context ctx' that satisfies the corresponding compatibility requirement between $\mathcal{I}'_1$ and $\mathcal{I}_2$, so that:

$$[(\mathsf{req}'_1 \vee \mathsf{req}_2) \wedge (\mathsf{req}'_1 \rightarrow \mathsf{pro}'_1) \wedge (\mathsf{req}_2 \rightarrow \mathsf{pro}_2)]_{\mathsf{ctx}'} = 1 \tag{4}$$

is consistent.

Observe that ctx is exhaustive for (3), so its scope includes almost all variables in Figure 4 (more precisely it does not include variables declared afresh in $\mathcal{I}'_1$ — the gray area in the figure).

1° Assume that $[\mathsf{req}'_1]_{\mathsf{ctx}|_{\mathsf{use}_1}} = 0$. Then also $[\mathsf{req}_1]_{\mathsf{ctx}|_{\mathsf{use}_1}} = 0$ (by the first condition of subtyping), and $[\mathsf{req}_1]_{\mathsf{ctx}} = 0$. Consequently, since ctx satisfies (3), we get that it also satisfies (4). The satisfaction of the two outer clauses carries over from (3), while the middle clause is satisfied vacuously as per the previous sentence.

2° Assume that $[\mathsf{req}'_1]_{\mathsf{ctx}|_{\mathsf{use}_1}} = 1$ and that $[\mathsf{req}_1]_{\mathsf{ctx}} = 1$ (otherwise we proceed like in step 1°). Then by well-formedness of $\mathcal{I}'_1$ we have that $[\mathsf{pro}'_1]_{\mathsf{ctx}|_{\mathsf{use}_1}}$ is consistent. Moreover it is consistent even for $[\mathsf{pro}'_1]_{\mathsf{ctx}|_{\mathsf{use}_1 \cup \mathsf{decl}_1}}$ because of the third condition of subtyping (that refinement introduces no changes in the combinatorics of the refined interface).

The only variables missing in $\mathsf{ctx}|_{\mathsf{use}_1 \cup \mathsf{decl}_1}$ are those in the gray area in Figure 4, but these are exactly the variables that do not occur in ctx at all, so they are free. We thus extend ctx to ctx' by assigning variables from $\mathsf{decl}'_1 - \mathsf{decl}_1$ in a way which makes $[\mathsf{pro}'_1]_{\mathsf{ctx}'} = 1$ (we do exploit the fact that the intersection of use_2 and decl'_1 is empty) . Since ctx' agrees with ctx on all other variables we get that the last clause of (4) is satisfied in ctx' (by transfer from (3)). The middle clause is satisfied since we have argued that $[\mathsf{req}'_1 \wedge \mathsf{pro}'_1]_{\mathsf{ctx}'} = 1$. The last one follows form $[\mathsf{req}'_1]_{\mathsf{ctx}'} = 1$. $\square$

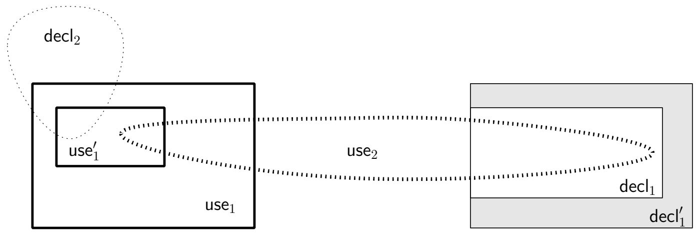


Fig. 4. Possible intersections between sets of variables (sorts) involving $\mathcal{I}_1$, $\mathcal{I}'_1$, and $\mathcal{I}_2$ as in the assumptions of Theorem 1.

B Varia

Richness of the Interface Language. Such structure of interfaces allows us to express combinatorial constraints in interfaces, which is particularly useful in components that allow several different use cases. One could argue that our interfaces are too rich in that they detail not only the required and provided part, but the also expose which of the internal model parts refer to the required objects. This seemingly excessive information, is instrumental in automatic synthesis of interfaces for the model level. Consider the following example of a model level interface synthesized from the model and the metamodel-interface used in the example above.

Automatically generated keys. Our implementation works well for manually written named based references, it is very easy to incorporate into mature software development projects. Initial interfaces are generated efficiently, and they can be easily adapted by developers to contain more fine grained information. The weakness is that the initial interfaces are all singleton: they have a uniform set of assumptions, and make one global guarantee. The algorithm attempts no guesses at the internal semantic meaning of the XML file, which would be needed in order to offer more fine grained interfaces, where parts of the guarantees only require part of the assumptions.

Our interface language, and the conformance and compatibility checking algorithms are suitable to work with references of various kinds, including machine generated identifiers (such like consecutive integers). However the inference algorithm assumes that identifiers are unique within the project (not just within a type). This assumption normally holds for human created models, and it was no problem in the projects we have used for evaluation.

- ref to class diagram mining [20, 21]
- impose in configuration file
- association mining - why we haven't used it [after a second thought AW thinks that itemset mining with data mining techniques is so much different that it should not even be mentioned to avoid confusion]
- XAML component libraries
- XUL