S.IM.PL SERIALIZATION:
TYPE SYSTEM SCOPES ENCAPSULATE CROSS-LANGUAGE, MULTI-FORMAT INFORMATION BINDING

A Thesis

by

NABEEL SHAHZAD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Science

S.IM.PL Serialization: Type System Scopes Encapsulate

Cross-Language, Multi-Format Information Binding

Copyright 2011 Nabeel Shahzad

S.IM.PL SERIALIZATION:

TYPE SYSTEM SCOPES ENCAPSULATE CROSS-LANGUAGE, MULTI-FORMAT INFORMATION BINDING

A Thesis

by

NABEEL SHAHZAD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Andruid Kerne
Committee Members,	Bjarne Stroustrup
	Jaakko Järvi
	Philip Galanter
Head of Department,	Duncan Walker

December 2011

Major Subject: Computer Science

ABSTRACT

S.IM.PL Serialization:

Type System Scopes Encapsulate Cross-Language, Multi-Format Information Binding. (December 2011)

Nabeel Shahzad, B.S., National University of Computer and Emerging Sciences

Chair of Advisory Committee: Dr. Andruid Kerne

Representing data outside of and between programs is important in software that stores, shares, and manipulates information. Formats for representing information, varying from human-readable verbose (XML) to light-weight, concise (JSON), and non-human-readable formats (TLV) have been developed and used by applications based on their data and communication requirements. Writing correct programs that produce information represented in these formats is a difficult and time-consuming task, as developers must write repetitive, tedious code to map loosely-typed serialized data to strongly-typed program objects.

We developed S.IM.PL Serialization, a cross-language multi-format information binding framework to relieve developers from the burdens associated with the serialization of strongly-typed data structures. We developed *type system scopes*, a means of encapsulating data types and binding semantics as a cross-language abstract semantics graph. In comparison to representing data binding semantics and information structure through external forms such as schemas, configuration files, and interface description languages, type system scopes can be automatically generated from declarations in a data binding annotation language, facilitating software engineering. Validation is based on use in research applications, a study of how computer science graduate students use the software to develop applications, and performance

benchmarks. As a case study, we also examine the cross-language development of a Team Coordination (TeC) game.

To Ammi, Papa, Mona, Fariha, Yumna, and my lovely fianceé Marium

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Andruid Kerne, for his continuous support, commitment, and encouragement. I have acquired valuable skills and intellectual knowledge under his guidance, which will benefit me in years to come.

I would also like to thank my committee members, Jaakko Järvi, Bjarne Stroustrup, and Philip Galanter for their suggestions and feedback during the course of this research.

Thanks to my mixed-reality lab mates Zachary O. Toups and William A. Hamilton for their help, suggestions and support in every aspect of my research. Special thanks to my friend, Sashikanth Damaraju, for invaluable discussions, ideas, and use cases that refined my research. I wish great success to every member at the Interface Ecology Lab.

Finally, thanks to my parents, who have always shown confidence in me and supported me whenever I required. Everything that I have achieved in life was not possible without your unconditional love and support.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

In the current age of dense interconnectivity among diverse computer systems, there is an intense interest in developing software systems that communicate, share, and interoperate with each other. As more and more data are produced every day, software engineers are burdened with the task of writing software programs that process this data as information (data that informs). We categorize such software applications as *information-centric*. A crucial part of writing information-centric applications is to serialize and deserialize complex data structures. By serializing/deserializing information, we refer to the ability of a software program to convert data into an external representation so that it can be stored, transmitted over the network, and stored among software systems without losing the meaning or structure of data.

As part of object-oriented programming, writing correct programs that export and import structured representations of information is a difficult and time-consuming task. Developers must write repetitive, tedious code to map serialized data to typed program objects. The burden on software engineers increases, as data can be represented in multiple formats such as XML (Extensible Markup Language) [1], JSON (JavaScript Object Notation) [2], or YAML (YAML Ain't Markup Language) [3]. The problems are further multiplied when information is shared across platforms such as Java [4], .NET [5], and Objective-C [6]. Developers are burdened with writing separate serialization code in each platform for applications operating upon the same information. As a code base increases to cater to multiple formats and/or platforms, maintenance and documentation also adds to the burden on software engineers.

The journal model is *IEEE Transactions on Automatic Control.*

Programming languages such as C# and Java provide built-in capabilities for object serialization and deserialization. However, their serialization processes remain highly *opaque* to programmers, as they cannot completely control which fields are serialized and how. The serialized data itself is platform-dependent and obfuscated. Prior *data binding frameworks* offer some *transparency* by providing mechanisms for connecting program objects with serialized representations such as XML. However, the overhead of such frameworks, in terms of configuration during development and maintenance, and CPU runtime, may be prohibitive.

An example of a relatively high-level framework is JiBX [7], which offers XML-Java translation through requirement of writing XML binding definitions. These definitions are verbose, external to the source code and managed separately by the programmer. JiBX also uses a binding compilation step to augment Java byte-code for handling mappings, which must be managed by the programmer. These extra steps and external parallel definitions increase the burden on programmers, while reducing software maintainability.

We need efficient data binding frameworks that facilitate software maintenance and enable software engineers to specify a variety of data structures for de/serialization in different formats and across multiple platform. We present S.IM.PL serialization, a flexible, maintainable, cross-language, multi-format information binding framework hewn with object-oriented software design principles. It serves as a building block of a broader goal to integrate Support for Information Mapping in Programming Languages (S.IM.PL). The S.IM.PL initiative develops languages, architectures, and frameworks that integrate closely with popular programming languages to facilitate the development of object-oriented applications. S.IM.PL Serialization's goal is to reduce the burden on software engineers developing information-centric applications.

At the heart of S.IM.PL Serialization is the *type system scope* data structure, which we invented to encapsulate data bindings as *abstract semantics graphs*. Type system scopes are runtime data structures that cache reflection accessor objects to optimize serialization and deserialization processes. They are automatically generated from declarations in Data Binding Annotation Language (DBAL); no off-line compilation step is required. DBAL declarations reside within the source code with class and field declarations; therefore, its structural co-existence facilitates code maintenance and documentation.

Type system scopes are the key to S.IM.PL Serialization's support for multiple formats of data representation. They abstract translation semantics, enabling bindings to multiple serialization formats, such as XML, JSON, TLV (type-length-value) [8], and BibTeX [9]. Type system scopes also facilitate cross-language de/serialization. Currently data binding is supported across Java, C#, and Objective-C programming languages.

In the next chapter, we introduce S.IM.PL Serialization by examining a simple data binding example and explaining DBAL constructs. Then, we introduce type system scopes, describing their structure and processes that create instances of type system scopes. We also describe serialization and deserialization algorithms that leverage type system scopes. Later, Chapters IV and VI describe how type system scopes facilitate multi-format and cross-language data binding. Chapters VII, VIII, and IX present bottom-up validation of S.IM.PL Serialization. We validate design and features through research frameworks that builds on S.IM.PL Serialization. Robustness, flexibility, and error-handling is validated through S.IM.PL Serialization's application in real-world software systems. Ease-of-use and overall development experience is measured through feedback from student developers. Finally, we validated runtime performance benefits of S.IM.PL Serialization through benchmarking.

CHAPTER II

S.IM.PL SERIALIZATION

When building distributed applications, information is repeatedly serialized and deserialized across processes. Changing requirements can result in changes to structure of information, platform dependency, and the addition of new messages. Software engineers can benefit from an object-oriented flexible framework that requires minimal changes in code to cope with changing requirements.

S.IM.PL Serialization extends the prior open-source, Java-based, XML data binding framework Ecologylab•xml [10] [11]. In comparison to Ecologylab•xml, S.IM.PL Serialization provides a more precise annotation language and cross-language and multi-format translation. Currently, cross-language support is provided for Java, C#, and Objective-C programming languages. XML, JSON, and TLV formats comprise multi-format support. S.IM.PL Serialization is distributed as an open-source framework with complete source code, documentation, and guide publicly available on its website [12].

We begin with an example, using S.IM.PL Serialization for data binding of a class definition with an XML document in a multi-player game. Then, we categorize programming languages based on their reflection and annotation features, which are important for implementing de/serialization functionalities. We also define categories of data types based on the restrictions they impose when represented in a serialized form. Throughout this document, we will refer to these categories instead of a specific programming language or data type. Finally, we formally present an annotation language for expressing relationships between data objects and serialized representations.

A. Example Walkthrough

S.IM.PL Serialization is designed to simplify the process of serializing and deserializing data. We developed Data Binding Annotation Language (DBAL) (II, D) to enable software engineers to specify bindings between a class declaration and a serialized representation within programming language source code. Using language constructs, software engineers can specify which fields are de/serialized and how they are represented. We present an example of the syntax and semantics of translation of a simplified XML document from a multi-player application, the Team Coordination (TeC) game. TeC is a distributed application, in which a server serializes the runtime state of the game world, which in turn is subsequently deserialized by the game client. The client uses deserialized objects to render entities that comprise the game. In reality, a TeC game state consists of a large set of types of in-game data objects; however,

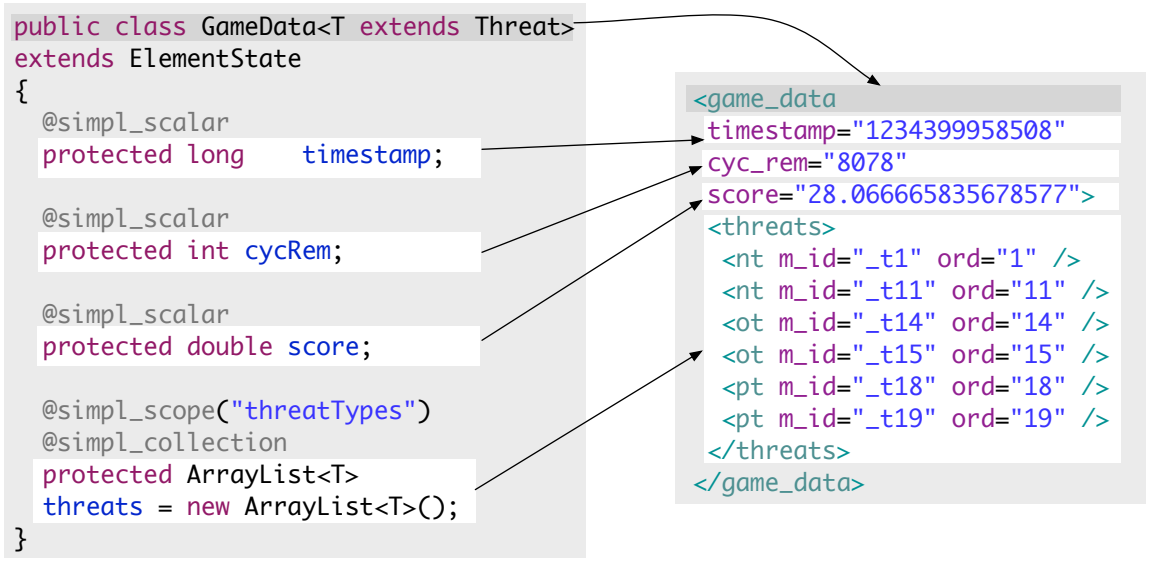

Fig. 1.: An XML document, which is a serialized representation of `GameData` instance. Class and field declarations are bound to specific tags and attributes in XML. Data is represented as values of these tags and attributes.

in this example, we only examine a subset of the actual data. An XML document that represents an instance of the game state, binds with class definitions in Java, as shown in Figure 1. By augmenting Java classes with declarations in DBAL, we can seamlessly de/serialize data without any handwritten parsing code.

```
@simpl_inherit
@simpl_tag("t")
public class Threat extends Entity
{
 @simpl_scalar
 protected String m_id;

 @simpl_scalar
 protected int ord;
}
```

```
@simpl_inherit
@simpl_tag("nt")
public class RepellableThreat extends Threat
{}
```

```
@simpl_inherit
@simpl_tag("pt")
public class PatrollingThreat extends Threat
{}
```

```
@simpl_inherit
@simpl_tag("ot")
public class OrbitingThreat extends   Threat
{}
```

Fig. 2.: Class declarations of a polymorphic base type `Threat` and derived sub-types. Subclasses override tag names of the base class through `@simpl_tag` annotation. Additional fields can also be added and annotated in sub-types.

We declared a top-level class `GameData` that encapsulates the information of an instance of a game state. Note that `GameData` extends `ElementState` and each field that binds to the XML document is augmented with annotations starting with "`@`" symbol. `ElementState` is the building block that provides methods for serialization;

subclasses function as program objects containing augmented fields that map to serialized representation. The `GameData` instance is serialized with a root tag `<game_data>` in XML. The tag name is derived automatically through camel-case conversion of the class name. We can also override automatic camel-case conversion by specifying a tag name using annotations explained in II, D, 4. Timestamps, cycles remaining, and score are scalar fields (II, C, 1) inside the `GameData` object. By default, these fields are serialized as attributes of the parent `<game_data>` tag. This default behaviour can be overridden by annotations explained in IV, B, such that these fields are serialized as leaf, text, or CDATA nodes in XML.

Later, we declared a polymorphic collection of threats. Threats in TeC can be orbiting, repellable, or patrolling types depending on their behaviour in the game. They are declared as sub-types of the common base class `Threat` (Figure 2). A polymorphic collection as declared in `GameData` can contain objects of any of the sub-types of `Threat`. The `@simpl_collection` annotation specifies a one-to-many relationship of a field with objects of the same base type. The `@simpl_scope` annotation specifies the types of objects the collection may contain at runtime. We defined a `SimplTypesScope` (see Chapter III) object in Figure 3 that specifies which Java classes functions as targets for translation within the collection. We also specified

```
threatTypesScope = SimplTypesScope.get("threatTypes",
                       Threat.class,
                       OrbitingThreat.class,
                       RepellableThreat.class,
                       PatrollingThreat.class);
```

Fig. 3.: Static accessor/factory `get()` method is invoked to create a type system scope instance. A unique identifier `threatTypes` is used for registering/finding the instance.

```
gameData.serialize(outputStream, FORMAT.XML);
```

Fig. 4.: Invoking the `serialize()` method on an instance of `GameData` serializes it to the output stream. The format of the serialized representation is specified as an argument to the method call.

an identifier for the declared type system scope object. This identifier can be used at runtime to reference the previously created `SimplTypesScope` object, instead of constructing a new instance. When serialized, the parent tag for the objects in a collection is serialized as `<threats>`; derived from field name of the collection. The tags for the child nodes in a collection are derived from tag names specified on each class by class-level `@simpl_tag` annotation.

```
gameTypesScope = SimplTypesScope.get("gamedata",
                                     threatTypesScope,
                                     GameData.class);
gameData = (GameData) gameTypesScope.deserialize(inputStream, FORMAT.XML);
```

Fig. 5.: Static accessor/factory `get()` method is invoked to create an instance of type system scope. Invoking `deserialize()` method on `gameTypesScope` deserializes data from input stream into an instance of `GameData`. Format of the input data is specified as an argument to the method call

To serialize, we call the `serialize` method on the `GameData` object (Figure 4). The `serialize` method is inherited from `ElementState` superclass. To deserialize, we declared a `SimplTypesScope` object, which inherits from `threatTypesScope` and adds the `GameData` class (Figure 5). The resultant `gameTypesScope` object contains specifications of all the required classes that binds to the serialized representation. Next, we call the `deserialize` method that populates the `GameData` object from data

in an XML document.

We presented a data binding example using an XML document and Java class definitions. S.IM.PL Serialization also supports similar data bindings in C# and Objective-C with XML, as well as JSON and TLV. In the next section, we define categories of programming languages and data types, based on how they restrict and/or support data binding.

B. Source and Target Languages

Different programming languages provide different features and constraints that define how binding semantics are expressed. S.IM.PL Serialization categorizes languages with support for reflection and annotations, such as Java and C#, as *source* programming languages. Programming languages without reflection and annotation language support, such as Objective-C [6], JavaScript [13] and C++ [14], are categorized as *target* programming languages. Table I shows the categorization of source and target programming languages.

C. S.IM.PL Type System

The goal of S.IM.PL Serialization is to facilitate the development of information-centric object-oriented software applications. Therefore, we only consider data types and structures associated with object-oriented programming languages. Different data structures define how they can be represented in a serialized form. We categorize data structures in the following sub-sections and present how their structure maps with a serialized representation.

Table I.: Shows the categorization of source and target languages based on their reflection and annotation features. S.IM.PL Serialization supports data bindings in Java, C#, and Objective-C

Language	Category	Support
Java	source	supports both annotations and reflection for inspecting the runtime object model
C#	source	supports both annotations and reflections; annotations are referred to as attributes
Objective-C	target	no support for annotations; reflection is supported by objective-c runtime
Python	target	no support for annotations; reflection is supported through introspection
JavaScript	target	no support for annotations; reflection is supported through introspection
C++	target	no support for annotations and reflection

1. Scalar Types

A scalar is a single unit of data. In serialized form, the value is represented as a *String* in cases of XML or JSON, and as a sequence of bytes in TLV. S.IM.PL Serialization categorizes data types that can be mapped onto a single unit of data, as scalar types. Enumerated types are also considered as scalar types. For enumerated types, their string constants specify their scalar values. Certain complex data types, such as Color, can also be considered as scalar types based on the general convention of their formatted representations as string. A color is commonly represented as a formatted

string of its hexadecimal value of RGB color space. However, alternate formats, such as an HSV representation, are also possible.

S.IM.PL Serialization provides an extensible scalar type system that supports de/serialization for all primitive data types as well as commonly used complex types such as String, StringBuilder, File, Color, Date and URL (or, alternatively, our convenient wrapper class, ParsedURL, which provides features such as lower case, domain identification, and suffix extraction by lazy evaluation). Software programmers can easily extend and override methods in `ScalarType` base class to provide alternate behaviours for representing complex data types or add new complex data types as scalars.

2. Composite Types

A complex unit of data consists of scalars and/or more complex units of data. Data in this form is best represented as a hierarchy or tree structure. Tree-structured data representation formats, such as XML, provide constructs for defining such a hierarchy. In object-oriented programming languages, object models are graphs. By making an exception with regard to cyclic and back references, we can consider object models as tree structures, forming a one-to-one mapping with a tree-structured data representation format. Cyclic and back references can be supported through addition of new constructs in the data representation format.

In the S.IM.PL type system, we refer to such complex data types as composite, meaning they are composed of one or more composite or scalar types. In practice, composite types are user-defined classes

3. Collection Types

Collections are an important kind of data structure in programming languages, as they enable co-allocation of multiple objects of the same base type. These objects can either be of scalar or composite types. Commonly used types of collections are *List* that hold multiple objects in order and *Map* that hold multiple objects relative to a unique key or hash code.

S.IM.PL Serialization supports data binding of collection objects, such as ArrayList(Java), List(C#), NSMutableArray(Objective-C), HashMap(Java), Dictionary(C#), and NSMutableDictionary(Objective-C) data structures.

4. Inheritance and Polymorphic Types

A key concept in object-oriented programming is *Inheritance*. A subclass inherits properties from its parent classes based on inheritance modifiers. A data binding framework must look up the type hierarchy to include properties from superclasses for data binding. However, in some cases, superclasses do not contain fields that need to be serialized. In S.IM.PL Serialization, software programmers limit the scope of lookup using annotations (II, D, 5) to avoid redundant processing.

Polymorphism is another key concept of object-oriented programming supported through inheritance. A polymorphic object can be of any data type with the same base class. The serialized representation of subclasses of the base type can be different. They can only be determined at runtime. The framework should also implement a mechanism to instantiate the correct subtype from a serialized representation. In S.IM.PL Serialization, data bindings of polymorphic objects and collections are handled through special annotations (II, D, 3).

D. Data Binding Annotation Language

In this section, we formally present the Data Binding Annotation Language (DBAL) for defining the semantics of translation between data objects and serialized representations. Using the DBAL constructs, software engineers can augment class definitions by specifying which fields are serialized and how they are represented. Later, instances of augmented classes can seamlessly be serialized and deserialized. Since declarations in DBAL reside within the source code, their structural co-existence with class and field declarations facilitate documentation and readability of the source code. As opposed to defining data bindings through external files, annotations are easier to maintain through refactoring and code completion features natively supported by integrated development environments such as Eclipse [15] or Visual Studio [16] .

Figure 6 and 7 presents a formal grammar of the Java definition of DBAL. This grammar is an augmented subset of the BNF-style grammar for Java presented in *The Java Language Specification* [4]. Sections 1-5 explain different types of annotations, describing their specific roles in defining the semantics of translation. A complete reference with examples is presented in Appendix B. Section 1 describes *SIMPLIndividualAugmentation*, which specifies a one-to-one mapping of scalar/composite fields and serialized representation. Section 2 describes *SIMPLCollectionAugmentation*, which specifies a one-to-many mapping between a collection field and serialized elements. Finally, additional constructs are discussed that enable polymorphism and flexibility in data bindings.

1. Individual Field Augmentation

SIMPLIndividualAugmentation functions as a constituent of a field declaration defined by *FieldDecl* production rule in Figure 6. The *Type* of the declared field can be

[x] denotes zero or one occurrences of x.
{x} denotes zero or more occurrences of x.
x | y means one of either x or y.

ClassBody:
{ {*ClassBodyDeclaration*} }

ClassBodyDeclaration:
;| [`static`] *Block*
| {***OtherModifier***} *MemberDecl*

OtherModifier*:*
OtherAnnotation | `public` | `protected` | `private`
| `static` | `abstract` | `final` | `native` | `synchronized`
| `transient` | `volatile` | `strictfp`

MemberDecl:
| *GenericMethodOrConstructorDecl*
| ***MethodDecl***
| ***FieldDecl***
| `void` *Identifier MethodDeclaratorRest*
| *Identifier ConstructorDeclaratorRest*
| *ClassOrInterfaceDeclaration*

ClassOrInterfaceDeclaration:
ModifiersOpt (*ClassDeclaration* | *InterfaceDeclaration*)

ClassDeclaration:
`class`
[`@simpl_inherit`**]**
[`@simpl_tag("`***TagName***`")`**]**
[*SIMPLOtherTags*]
Identifier [`extends` *Type*] [`implements` *TypeList*] *ClassBody*

Type:
Identifier [*TypeArguments*]{ . *Identifier* [*TypeArguments*]} {[ ]}
| *BasicType*

TypeArguments:
< *TypeArgument* {, *TypeArgument*} >

TypeArgument:
Type
| ? [(`extends` | `super`) *Type*]

Fig. 6.: Formal grammar of Data Binding Annotation Language. An augmented subset of BNF-style grammar for Java. (1 of 2)

FieldDecl:
[@simpl_tag("***TagName***")**]**
[*SIMPLOtherTags*]
[*SIMPLClasses*]
SIMPLIndividualAnnotation *Type Identifier MethodOrFieldRest* |
SIMPLCollectionAnnotation *Identifier MethodOrFieldRest*

SIMPLIndividualAnnotation:
@simpl_composite **[*SIMPLClasses*]** | **[*SIMPLScope*]**
|@simpl_scalar
|@simpl_hints({**HintName {, HintName }**})
|@simpl_filter(**regex = *"Expression"***)

SIMPLCollectionAnnotation:
@simpl_wrap | @simpl_nowrap
@simpl_collection(**["*TagName*"]**)
[*SIMPLClasses*] | **[*SIMPLScope*]**

Identifier TypeArguments
|@simpl_map(**["*TagName*"]**)

[*SIMPLClasses*] | **[*SIMPLScope*]**
Identifier TypeArguments

SIMPLClasses:
@simpl_classes({***ClassName*.class{, *ClassName*.class}**})

SIMPLScope:
@simpl_scope(***["SimplTypesScope"]***)

SIMPLOtherTags:
@simpl_other_tags({"***TagName***"{, "***TagName***"}})

TagName:
any allowed tag name

ClassName:
name of any existing class

SimplTypesScope:
the identifier of the type system scope

HintName:
XML_ATTRIBUTE, XML_LEAF,
XML_LEAF_CDATA, XML_TEXT,
XML_TEXT_CDATA, or UNDEFINED

Fig. 7.: Formal grammar of Data Binding Annotation Language. An augmented subset of BNF-style grammar for Java. (2 of 2)

a composite or a scalar field.

The `@simpl_scalar` annotation declares that a scalar-typed field should be de/serialized. In programming languages, a class can be composed of scalar type fields and objects of user-defined classes. Similarly, composite elements in a serialized form can be recursively composed of composite elements and multiple scalar elements. The `@simpl_composite` annotation is used to declare a one-to-one mapping between a field of typed user defined class, and its serialized representation. A field declared as `@simpl_composite` must be declared with a *Type* that is a subclass of `ElementState`. The `ElementState` base class is provided by S.IM.PL Serialization. It contains methods for serialization; instances of `ElementState` subclasses function as program objects that binds to serialized representation.

The `@simpl_hints` annotation is used for fine-grained control over serialized representation, specific for each format (IV, B). It is often necessary to write code for removing unwanted characters when deserializing data from the wild, such as XML from an RSS feed. The `@simpl_filter` annotation accepts a regular expression string that can be used to filter incoming data, as per the programmer's requirement.

2. Collection Field Augmentation

It is common that structured serialized representations contain nodes with a one-to-many relationship to its constituent child nodes. In programming languages, a similar relationship is represented by ArrayList (Java), List (C#), HashMap (Java), or Dictionary (C#) data structures. The *SIMPLCollectionAugmentation* production specifies a one-to-many relationship between fields and data. This annotation is used in conjunction with *Collection* classes, provided by a programming language Software Development Kit (SDK).

Within the *SIMPLCollectionAugmentation* production, the `@simpl_collection`

annotation declaration specifies one-to-many mapping of child objects to a parent, with sequential access to collection members. The generic *Type* declaration for such a field can either be of a composite or scalar type. The optional *TagName* parameter to `@simpl_collection` enables further flexibility in defining these mappings. The developer-specified tag name is required for non-polymorphic collections, as this tag name specifies the mapping between child nodes and parent nodes. For polymorphic collections, the developer should omit this parameter.

When deserializing data, it is often necessary for collections to be quickly and randomly accessed, based on the data they contain, rather than an ordinal index. The framework provides support for automatically generating keyed or hashed data structures directly from serialized representations. The `@simpl_map` annotation construct, in conjunction with a key-value data structure, can be used to specify deserialization a collection typed field. The elements of this collection should be of composite type, implementing the framework provided `Mappable` interface. The `Mappable` interface requires a simple implementation of method `key()` to identify the field that should be used for random access. When the implementation of method `key()` is specified, the framework can automatically establish efficient data structures such as Dictionary or HashMap.

We categorize collections as wrapped or unwrapped, based on how they are serialized. Elements of a wrapped collection are children of a parent collection element, while elements of unwrapped collections lacks this intermediate parent. When a collection is declared with `@simpl_wrap` annotation, elements of the collection are serialized as child of a parent collection element. The tag name of parent element is derived from the camel case conversion of the collection field name or overridden by `@simpl_tag` annotation (II, D, 4). When declared with `@simpl_nowrap`, the parent element of the collection is not serialized, thus collection objects are serialized directly

as children of the composite element.

3. Polymorphic Fields Augmentation

S.IM.PL Serialization supports polymorphic composite and collection types. For polymorphic collections the `@simpl_classes` annotation construct statically specifies a set of classes that an object might be of *Type* at runtime. An associated tag-class mapping is generated. The elements of the serialized representation are directly bound to an unqualified class name through camel case conversion. Tag-class mapping enables deserialization of polymorphic subclasses of a common type.

For additional flexibility, the `@simpl_scope` annotation allows developers to dynamically specify a polymorphic set of classes through a *type system scope*. The parameter to the `@simpl_scope` annotation specifies a unique identifier of a registered type system scope data structure (see Chapter III). `@simpl_scope` annotation is resolved through lazy-evaluation at runtime (III, B).

4. Custom Tags

The `@simpl_tag` annotation construct (*FieldDecl*, and *ClassDeclaration* productions) enables the programmer to explicitly specify a tag name for a given field or class. The `@simpl_tag` annotation overrides automatic camel case conversion of tag names. It handles names outside the possible scope of translation, enabling the characters to be used that are not allowed in class fields in a programming language, such as those including a dash ("-") or colon (":"), or names that collide with reserved words (such as `abstract` in Java). When `@simpl_tag` is used for a field declaration, as in the *FieldDecl* production, it remaps the serialized node name for the field. When used as part of a class declaration, as in the *ClassDeclaration* production, `@simpl_tag` overrides the name conversion for objects of the given class, when they are trans-

lated based on class name (as with the root element of a document, or polymorphic objects). `@simpl_tag` also allows the programmer to compress custom serialized documents as desired. Field names can be of arbitrary length (and readability), while the resulting serialized document (which may not be meant for human eyes) can be as small as possible. Compressing reduces bandwidth, which can be a shortcoming of verbose formats such as XML. These flexible mechanisms of specifying tag names provides maximum control over the output serialized representation, and enables reading messages from the wild.

Support for compatibility with old versions of serialized representations is provided by the `@simpl_other_tags` directive. This directive takes one or more strings as its argument. It can be applied to either a class or field declaration. The result creates extra one-way mappings from serialized representation to a class or field. These mappings will only be used when deserializing data. For serialization the primary tag names (camel case conversion or tag name from `@simpl_tag`) will be used.

5. Limiting the Scope of Translation

Translating to and from serialized representations is not computationally free, so it is useful for the programmers to be able to specify limits on its scope. The `@simpl_inherit` construct (*ClassDeclaration* production), by its presence in a class's declaration, indicates that the fields of the class's superclass should be translated. Otherwise, fields from the superclass and subsequent superclasses are not resolved at runtime. This construct requires programmers to consider streamlining execution of their code. Super classes that contain needed functionality but not needed for serialization are omitted from translation.

CHAPTER III

TYPE SYSTEM SCOPES

Type system scopes are at the heart of data binding in the S.IM.PL Serialization framework. They encapsulate translation semantics between run-time objects and their serialized representations as an abstract semantics graph (ASG). Type system scopes are immutable objects, which are automatically generated only once from runtime interpretation of declarations in Data Binding Annotation Language (DBAL) (II, D); therefore, no offline compile-step is required. Type system scopes are cached using efficient data structures, such as HashMap (Java) or Dictionary (C#) data types to improve runtime performance.

Type system scopes facilitate fine-grained control and flexibility over serialized representation. They create an abstraction with specific information about the semantics of translation, thus providing *connection-points* between object models and different forms of serialized representations. Through the structural abstraction provided by type system scopes, back and forth translation between a class's runtime object and serialized representation becomes a seamless process.

In the next section, we explain the data structures that constitute a type system scope. Then, we examine the runtime scenarios in which type system scopes are derived from interpretation of DBAL. Later, we present the data binding algorithms of serialization and deserialization that leverage type system scopes. Finally, we present features utilizing type system scopes to facilitate data binding.

An ASG is like an abstract syntax tree (AST), with embedded semantic information and the ability to make non-hierarchical references. Specifically, in an AST, a reference to an entity is represented by an edge pointing to a leaf node. In an ASG, reference is represented by an edge pointing to the root of the sub-graph in the ASG that represents the declaration of the entity.

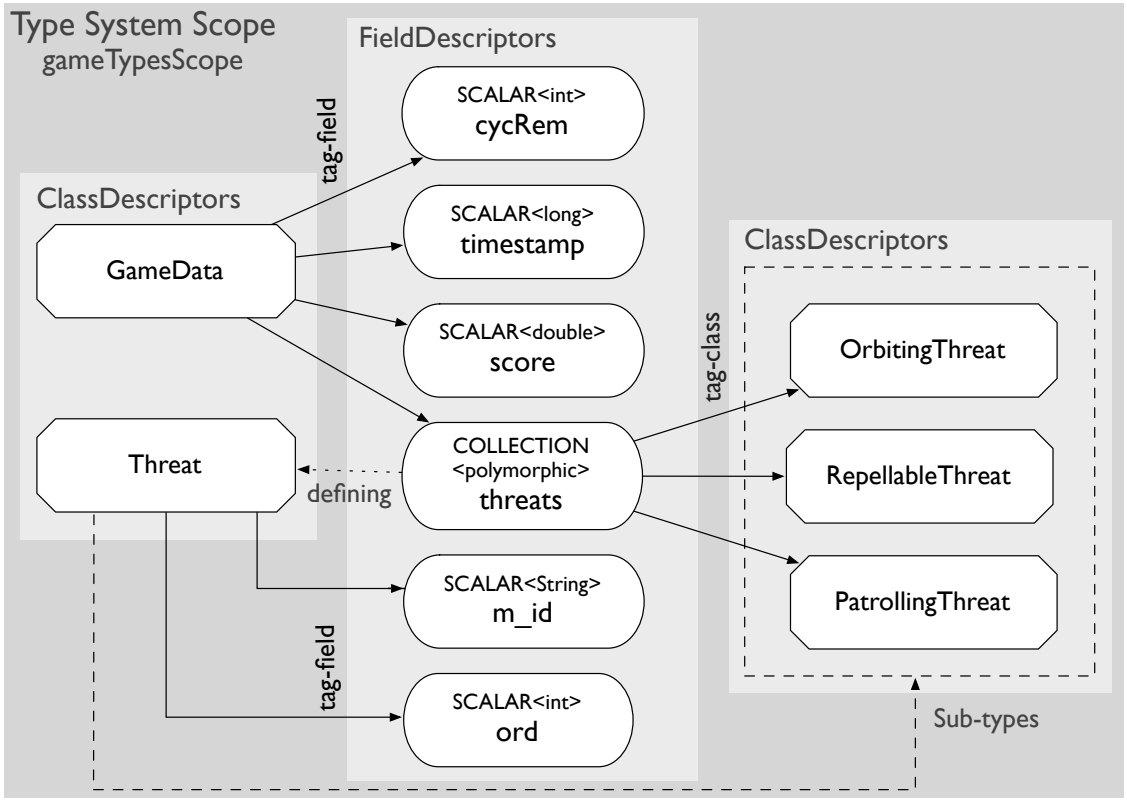

Fig. 8.: The type system scope ASG, generated from augmented class definition of the `GameData` (see Figure 1) type. `ClassDescriptor`s create tag-field mappings to `FieldDescriptor`s. For the polymorphic field `threats`, the `FieldDescriptor` creates tag-class mappings to other `ClassDescriptor`s.

A. Type System Scope Sub-structures

Processing for de/serialization uses introspection of the object model through reflection operations, which can be computationally expensive; resulting in inefficiencies during de/serialization. We improve efficiency by having type system scopes cache reflection accessor objects and translation semantics expressed by declarations in DBAL. Thus, subsequent requests for reflection accessor objects or translation semantics avoid repeated introspection of the object model.

Reflection accessor objects and translation semantics are encapsulated in the type system scope sub-structures `ClassDescriptor` and `FieldDescriptor`. A `ClassDescriptor` encapsulates the translation semantics of a class definition, while a `FieldDescriptor` encapsulates translation semantics of a field declared inside a class. Figure 8 shows the referential structure of the example `gameTypesScope`. Note that the `FieldDescriptor` of the collection *threats* contains a reference to the `ClassDescriptor` of its defining type, forming a backward edge. We refer to the type system scope as abstract semantics graph because it is constituted from `ClassDescriptor`s and `FieldDescriptor`s that abstract translation semantics and also contain backward and cyclic references to capture the structure of an object model.

1. Class Descriptors

A `ClassDescriptor` object explicitly describes the translation semantics of a class definition augmented with DBAL. An immutable instance of a `ClassDescriptor` is only created once per class, when required. Since, runtime introspection is computationally expensive, `ClassDescriptor`s cache reflection objects such as `Class` (Java), `Type` (C#), and `Class` (Objective-C) objects.

The framework maintains a set of tag-class mapping of `ClassDescriptor`s in each type system scope. Tag-class mappings are essential to efficiently find the correct `ClassDescriptor` from tag names when parsing a serialized representation. Figure 8 shows the tag name for `GameData` class. The `game_data` tag, which was computed from camel case conversion of the class name, is used to map the `GameData` `ClassDescriptor`. If the `@simpl_other_tags` annotation is used, additional tag-class mappings are created in the type system scope. When working with binary representations, such as TLV, `ClassDescriptor`s are also mapped by a unique integer id,

which is computed from a hash function that operates on the tag name.

For each constituent field, augmented with DBAL, a `ClassDescriptor` holds a mapping of its constituent `FieldDescriptor`s by their tag names and unique integer ids. Figure 8 shows the tag-field mappings between `ClassDescriptor`s and `FieldDescriptor`s. These mappings are essential to finding the correct `FieldDescriptor` when parsing a serialized representation. Tag names for `FieldDescriptor`s are optionally computed from field-level `@simpl_tag` and `@simpl_other_tags` annotations.

2. Field Descriptors

For each class field augmented with DBAL, an associated immutable `FieldDescriptor` is generated. Like `ClassDescriptor`s, each `FieldDescriptor` also contains tag names and a unique integer id for creating tag-field mappings in a `ClassDescriptor`. Tag-field mappings enables binding to serialized representations. Using the associated tags, the deserialization process can find the correct `FieldDescriptor`, and hence instantiate the correct data type.

`FieldDescriptor`s derived from scalar type fields contain generic methods for de/serialization utilizing the ScalarType (II, C, 1) system. The ScalarType system ensures the instantiation of the correct data type, while generating an appropriate error/warning if the data is invalid. A `ScalarType` base class provides generic methods for de/serialization, which developers can override to provide a type-specific implementation, enabling easy addition of new data types as scalars. For composite or collections of composite types, a `FieldDescriptor` maintains a reference to its defining `ClassDescriptor`, as shown in Figure 8. A reference to the defining `ClassDescriptor`, enables the deserialization algorithm (see III, C, 2) to refer to the correct `ClassDescriptor` in the type system scope, during parsing of deeply nested

data nodes.

For polymorphic fields, `FieldDescriptor`s maintain tag-class mappings of all valid types. The tag-class mapping refers to associated `ClassDescriptor`s for the valid types. As shown in Figure 8 the `FieldDescriptor` of the `threats` collection contains references to sub-types of `Threat`. Again, tag-class mappings are used to find the correct `ClassDescriptor` during deserialization, as polymorphic fields are serialized by their class's tag name, instead of the field's tag name. The valid runtime types are specified through `@simpl_classes` or `@simpl_scope` annotations. In case of `@simpl_classes`, tag-class mappings are immediately resolved when an instance of a `FieldDescriptor` is created. In case of `@simpl_scope`, tag-class mappings are lazily-resolved during deserialization (see III, B, 3).

Table II.: Shows the metadata in `FieldDescriptor`s derived from field types. This metadata is used by de/serialization processes to determine the abstract type of a field and execute specific functionality

Field Types	Semantics
scalar	describes scalar serializable field
composite_element	describes composite field with nested scalar/non-scalar fields
collection_element	describes collection field of composite objects
collection_scalar	describes collection field of scalar elements
map_element	describes map/dictionary data structure composite elements
wrapper	describes a wrapper tag for a collection or map
pseudo_field_descriptor	describes a place holder tag in a serialized document
ignored_attribute	error handling fields not bound with a serialized document

In addition to tag information, `FieldDescriptor`s also contain metadata about the *type* of the field, which guides the serialization and deserialization processes. Table II shows the metadata a `FieldDescriptor` can contain and it's meaning in the context of serialization. During de/serialization the framework inspects metadata contained in `FieldDescriptor`s instead of using computationally expensive reflection operations to determine the contextual type of an object.

B. Runtime DBAL Interpretation

Earlier, we explained that type system scopes are created from interpretation of DBAL declarations. In this section, we describe three distinct runtime invocation scenarios in which DBAL declarations are interpreted: (1) Specifying type system scope (2) Serialize (3) Deserialize. In the following sub-sections, we present details of each scenario explaining construction of type system scopes and constituent data structures.

1. Interpretation during Type System Scope Declaration

The user creates an instance of a type system scope by declaring a name and set of classes, each augmented with DBAL. The instance is created through a static accessor/factory method, instead of the constructor, as introduced earlier (II, A). The accessor/factory method creates an internal *GlobalSimplTypesScopeMap* (Figure 9) that maps type system scopes, with their names as identifiers. Repeated calls with the same name will not construct a new type system scope, but instead finds the previously constructed instance. This allows programmers to avoid concerns about inefficiencies associated with potential re-instantiations, without having to coordinate initialization sequencing control flows across modules.

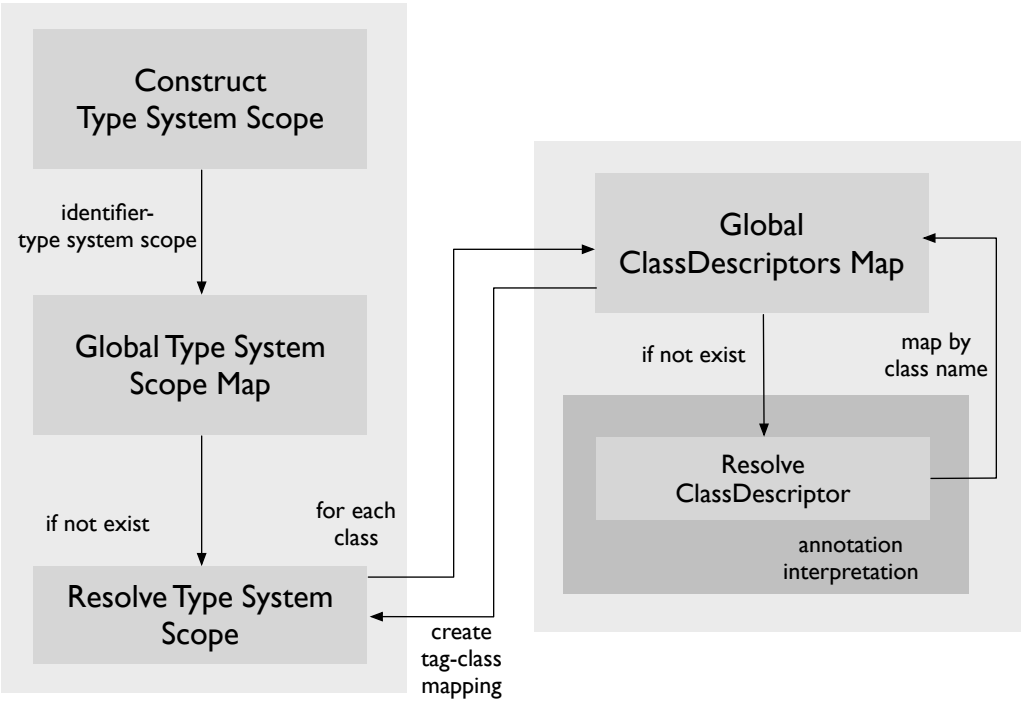

Fig. 9.: The sequence of operations during interpretation of DBAL declarations when a user specifies a type system scope. Resolved instances of `ClassDescriptor`s are mapped by class tag name in type system scopes.

If a type system scope object does not exists for a given name, the runtime resolves the `ClassDescriptor`s for each specified class. An associated `ClassDescriptor` for a given class is looked-up in the *GlobalClassDescriptorMap*, which maps all instances of generated `ClassDescriptor`s by qualified class name. Annotations are only interpreted when an associated `ClassDescriptor` does not exists in the *GlobalClassDescriptorMap*. Instances of resolved `ClassDescriptor`s are mapped with class tag name in type system scope. In this scenario, all class and field-level annotations are resolved, except `@simpl_scope`, which is lazily evaluated during a deserialize invocation scenario (see III, 3).

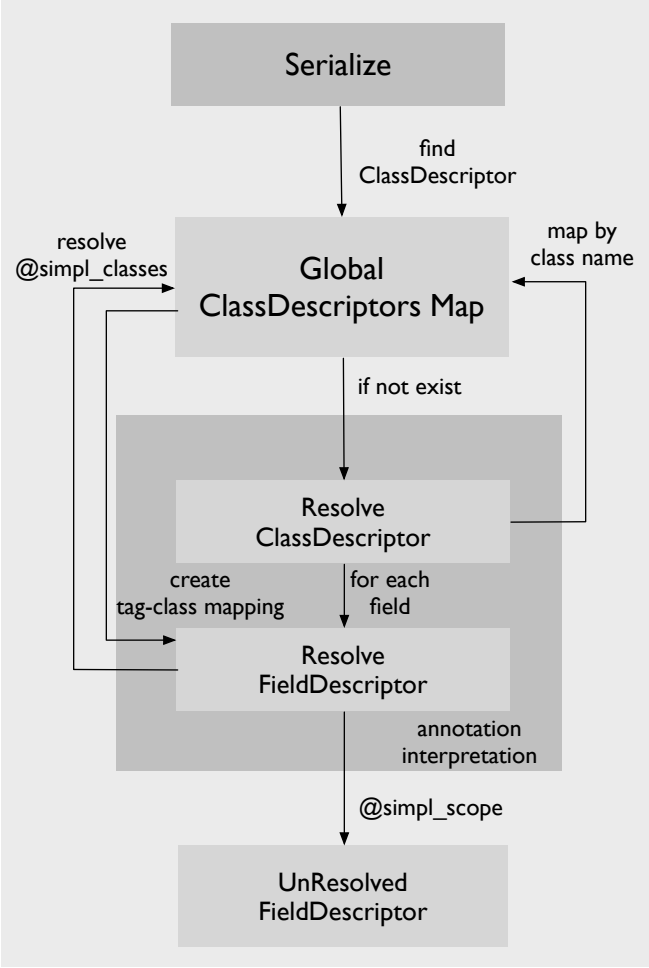

Fig. 10.: The sequence of operations during interpretation of DBAL declarations when the framework serializes data. Instances of `ClassDescriptor`s and `FieldDescriptor`s are created.

2. Interpretation during Serialization

In S.IM.PL Serialization, a user can serialize an object without specifying a type system scope, as it is required only in deserialization for mapping tags in data to `ClassDescriptor`s and `FieldDescriptor`s. However, `ClassDescriptor`s and `FieldDescriptor`s contain essential information, such as tag names, which are useful for serializing an object. Therefore, during serialization, instances of

`ClassDescriptor`s and `FieldDescriptor`s are created.

When serializing an object, the framework tries to find its associated `ClassDescriptor` in the *GlobalClassDescriptorMap* (Figure 10). If an associated `ClassDescriptor` does not exist, the framework resolves the `ClassDescriptor` for that object. The runtime interpreter utilizes reflection operations to inspect the class definition of the object, resolving class and field-level annotations. If a class definition contains the `@simpl_inherit` annotation, it implies that the parent class also contains fields required for binding to the serialized representation. The framework then recursively resolves the `ClassDescriptor` of the parent class.

When resolving field-level annotations, `FieldDescriptor`s are generated. Associated mappings in `ClassDescriptor`s are also created. If a field is declared with `@simpl_classes` annotation, it implies that the field is polymorphic; therefore, it can be of any type specified by the `@simpl_classes` declaration. Thus, the runtime interpreter resolves `ClassDescriptor`s of all classes specified in `@simpl_classes` declaration, mapping `ClassDescriptor`s in *GlobalClassDescriptorMap* by class name and creating tag-class mapping in `FieldDescriptor`s. If a polymorphic field is declared with the `@simpl_scope` annotation, it is marked as *unresolved* for lazy evaluation during deserialization.

3. Interpretation during Deserialization

When deserializing data, a user has already specified a type system scope and consequently all class and field-level annotations are resolved, except for any `@simpl_scope` annotations that are encountered. The `FieldDescriptor` of each field declared with `@simpl_scope` is marked as *unresolved*. During deserialization, the `@simpl_scope` annotation is resolved as shown in Figure 11. This lazy-interpretation is necessary, as the `@simpl_scope` annotation is specified with an identifier to a type system scope,

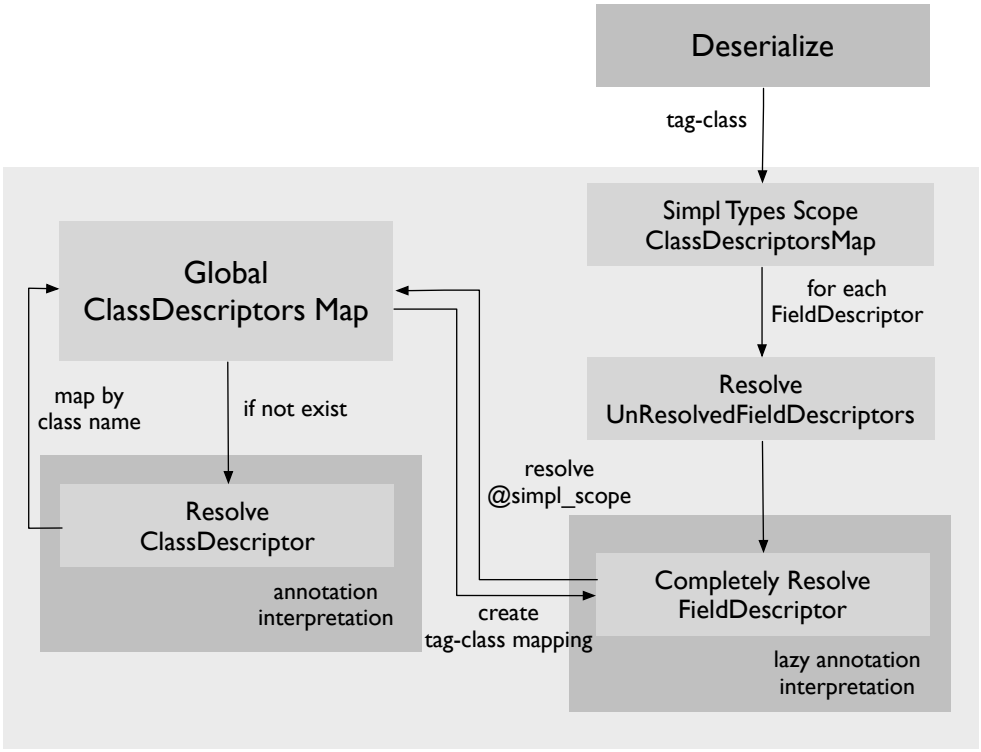

Fig. 11.: The sequence of operation in interpretation of DBAL declarations when the framework deserializes data. Un-resolved `FieldDescriptor`s are resolved, evaluating `@simpl_scope` declaration and creating tag-class mappings in `FieldDescriptor`s.

which may not exist when the user specifies a type system scope (B, 1) or serializes data (B, 2).

The framework identifies the `FieldDescriptor` that is marked as *unresovled*. Using the identifier to a type system scope from the `@simpl_scope` annotation, the framework finds all `ClassDescriptor`s in the *GlobalClassDescriptorMap* associated with the unresolved `FieldDescriptor`. The framework then creates tag-class mappings in `FieldDescriptor`s. Lazy-interpretation of `@simpl_scope` annotation supports self-referential graphs, enabling software engineers to specify a type system scope identifier in the `@simpl_scope` declaration and later specify a type system scope with the same identifier, without worrying about potential access to an unspecified

type system scope.

C. Data Binding Mechanism

We earlier explained the runtime interpretation of declarations in DBAL, which results in automatic creation of the type system scope ASG and its constituent `ClassDescriptor` and `FieldDescriptor` objects. Instead of utilizing reflection functions for the derivation of de/serialization semantics from annotations, S.IM.PL Serialization inspects encapsulated data in `ClassDescriptor`s and `FieldDescriptor`s for data binding.

In this section, we explain how type system scopes and their constituent `ClassDescriptor` and `FieldDescriptor` objects drive serialization and deserialization algorithms.

1. Serialization

The serialization process is straightforward, as type information and object hierarchy are available through introspection in programming languages that support reflection operations. The serialization process performs a depth-first traversal of the object model using type system scopes (Figure 12). Cyclic references are handled differently, which we will explain later, in Section E. A `FieldDescriptor` of the root element provides the tag information. For the starting tag in a serialized document, the `FieldDescriptor` contains an associated `ClassDescriptor`, which provides access to all containing `FieldDescriptor`s. In XML, a scalar field can be serialized as an attribute or child leaf node of the defining tag element. In `ClassDescriptor`s, attributes are indexed among *AttributeFieldDescriptors* and leaf nodes are indexed among *ElementFieldDescriptors*. This categorization is ignored in formats where

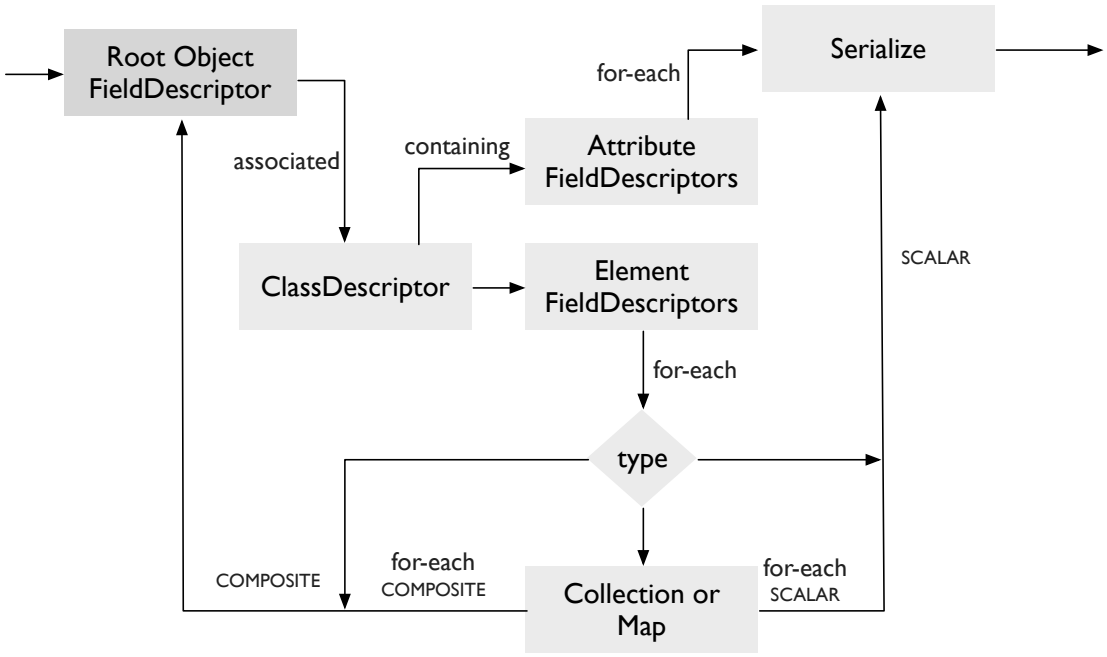

Fig. 12.: Serialization algorithm overview: Serializing instance of an object. `ClassDescriptor`s and `FieldDescriptor`s objects are used to serialize data into a structured representation.

scalar fields can only be serialized as attributes (e.g. JSON and TLV). The serialization process inspects the metadata information in the `FieldDescriptor`s as it iterates through scalar and collection elements and recursively resolves composite elements, avoiding use of reflection functions to determine the type of each field.

Scalar fields are serialized immediately by traversing through the list of `FieldDescriptor`s. For non-scalar fields such as composite fields, the serialization process recursively moves deeper into the object model, serializing objects until scalar elements are encountered. In the case of collection of scalars the framework can iterate through each element in collection, serializing it. Collections of composite elements are serialized by recursively resolving each composite element.

2. Deserialization

As opposed to serialization, deserialization is not straightforward, as type system scopes must map data nodes to the typed, language-specific object model. Further, deserialized objects must form an equivalent object model to that represented by the serialized data. Tag-class mappings in type system scopes help in instantiating the correct data type for each associated data node, while the type system scope ASG facilitates construction of a corresponding equivalent object model.

Figure 13 shows the control flow of the deserialization algorithm. For the root element tag, the framework looks up the tag-class mappings in the type system scope to find the associated `ClassDescriptor`. The framework then instantiates the corresponding root composite object. Later, for every child element tag, the framework finds the defining `ClassDescriptor` and instantiates its composite type. For polymorphic fields, the framework looks up tag-class mappings within a `FieldDescriptor` to find the correct `ClassDescriptor` corresponding to the child element tag. Using the `ClassDescriptor`, the framework instantiates the data type corresponding to the tag from the serialized representation. An instantiated composite type can either be added to a containing composite object or to a collection. For scalar fields, the framework resolves the associated `ScalarType` object. Sub-types of `ScalarType` defines translation semantics of a particular type of scalar field. Using the functionality provided by the sub-type, the framework deserializes the data and assigns it to its containing composite object or collection.

The programmer calls `TranslationScope`'s `deserialize()` method, specifying the format of the input data. The framework internally uses a format specific parser to parse the data. Currently the framework uses sequential access, and event-driven parsers. Such parsers are generally fast, as compared to other technologies such as

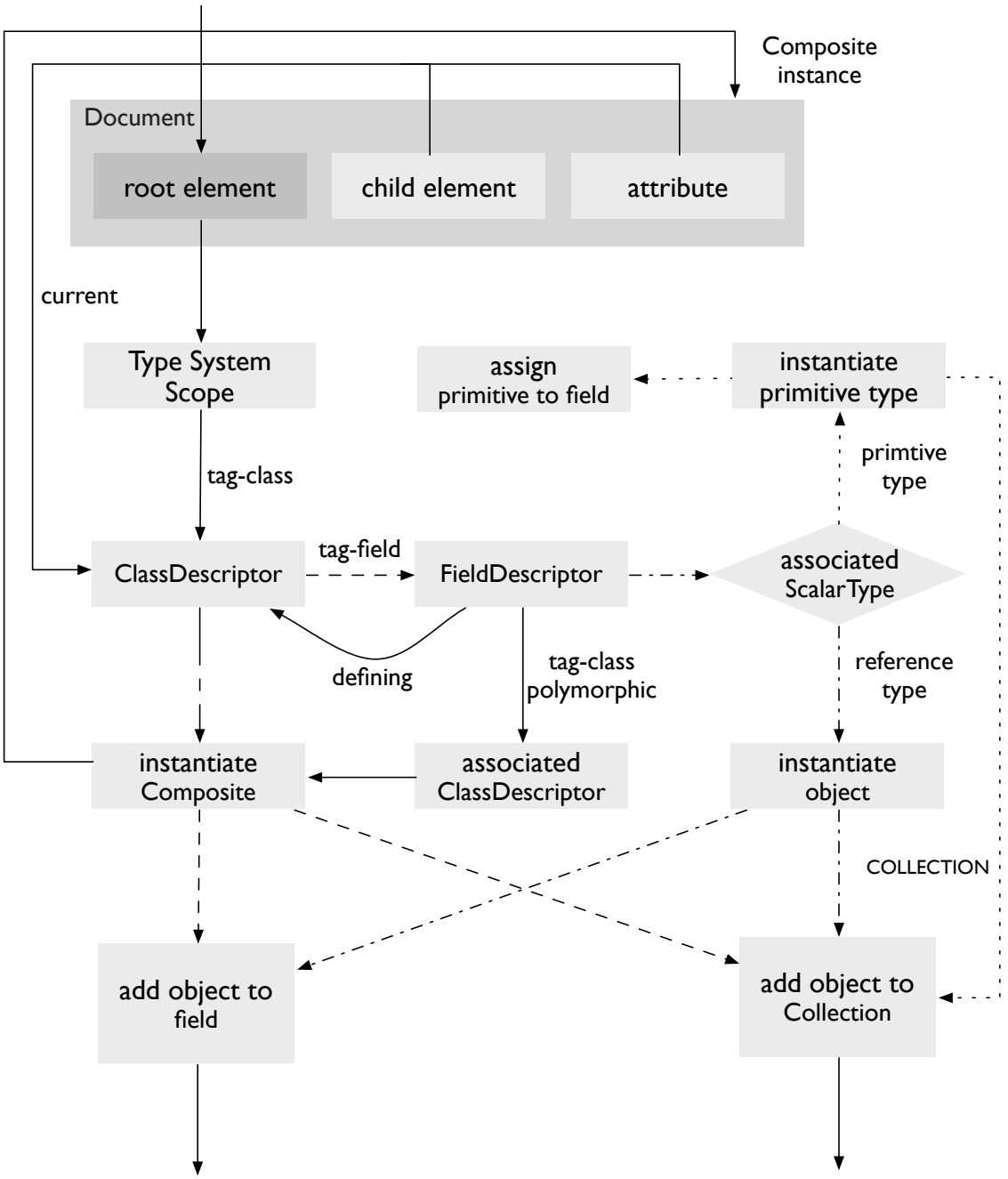

Fig. 13.: Deserialization algorithm overview using SAX parser: Parsing data from a structured representation, utilizing type system scope to create a corresponding typed object model.

DOM (Document Object Model) parsers for XML. However, programmers are not restricted to using a specific parser type. Programmers can implement a deserialization system on top of any data parser, as they are executed independently of the type system scope declaration.

D. Type System Scope Augmentation

In an iterative application development [17] model, where requirements are frequently changing, maintaining type system scopes can be cumbersome, as they are statically defined in the code. As new classes are defined and deprecated classes are removed from the code base, type system scope definitions are also required to be updated to handle de/serialization of new messages. The framework implements a runtime abstract semantics graph *augmentation* functionality that eases management of type system scopes in such cases.

The augmentation algorithm (Figure 14) performs a depth-first search on each instance of a `ClassDescriptor` mapped in the input type system scope. From the root `ClassDescriptor`, edges extend to its constituent `FieldDescriptor` objects. For every `FieldDescriptor`, the algorithm utilizes meta-information to determine if the described field is of a composite, collection, or scalar type. Scalar types are ignored. In case of composite or collection of composite types, the algorithm find the associated `ClassDescriptor` and recursively augments the new `ClassDescriptor` instance. In case of polymorphic types, the `FieldDescriptor` contains a list of classes that a field can bind to, specified by `@simpl_classes` or `@simpl_scope` annotations. In case of the `@simpl_classes` annotation, the algorithm finds the `ClassDescriptor`s for the specified classes and recursively augments each `ClassDescriptor`. Since, `@simpl_scope` annotation is used to dynamically specify the set of classes, the algorithm first re-

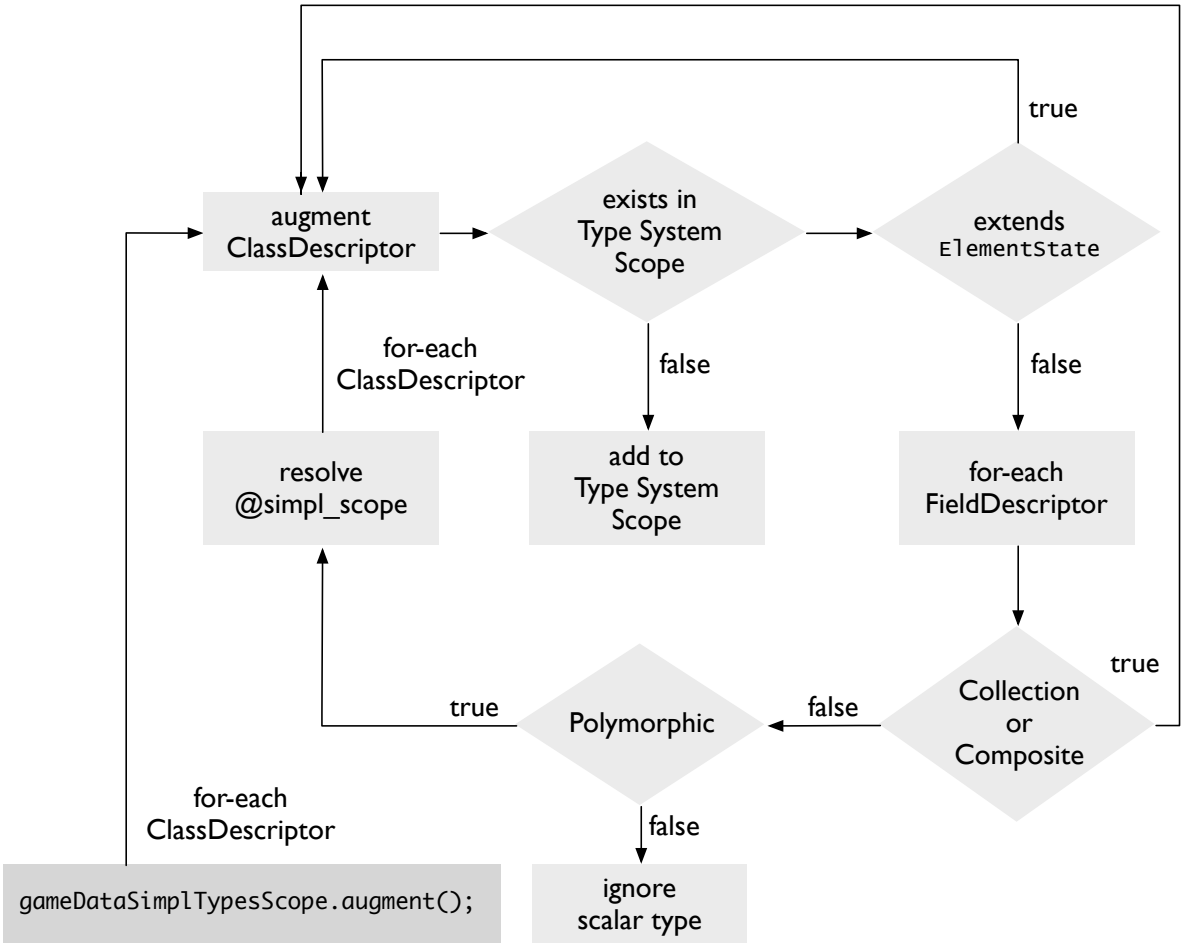

Fig. 14.: Augmentation algorithm overview: Recursive algorithm that computes the transitive closure of an input type system scope. For each `ClassDescriptor` mapped in type system scope, the algorithm recursively performs a depth-first search of ASG, adding `ClassDescriptor`s that are not present in the input type system scope.

solves the `@simpl_scope` declaration to find the set of classes and their associated `ClassDescriptor`s.

As new classes are encountered, their `ClassDescriptor`s are mapped augmenting the input type system scope. The algorithm ensures that every `ClassDescriptor` node in the type system scope is reachable from the root `ClassDescriptor` node, thus it effectively calculates the *transitive closure* [18] of the input type system scope.

Software engineers can augment a type system scope with a simple method call. This feature allows for easy maintenance of code, as the framework automatically manages bindings.

E. Handling Graph Data Structures

Object models are graph structures. They can contain back references to parent nodes. This may result in cyclic references. In comparison to in-memory object models, serialized data representations are tree-structured. No recursive back references or cycles can directly be represented. A data binding framework maps object graphs to tree structured representations. During serialization back references are resolved by recursively evaluating the serialized representation of the parent object. A cycle in the object graph would result in an infinite recursion during serialization, as resolving the parent object results in resolving a child, which again will resolve the parent object and so on. We needed an effective mechanism for handling graphs, as cyclic data structures are very common in object models.

To transform graphs into tree-structured representations, we introduced the `simpl:id` and `simpl:ref` special tags into serialized representations. Using these special tags, S.IM.PL Serialization ensures that an object is serialized only once. Further requests for serializing the same object will serialize a reference of the object, as a value of the `simpl:ref` tag, pointing to the unique id of the object, previously specified by the `simpl:id` tag. Using these special tags, the framework binds data graphs to textual representations. Figure 15 shows the definition of two classes, `ClassA` and `ClassB`. `ClassA` contains a reference to an object of `ClassB`, and also a reference to an object its own type. `ClassB` contains a reference to an object of `ClassA`. Creating an instance of `ClassA` and assigning its reference to an instance variable in `ClassB`,

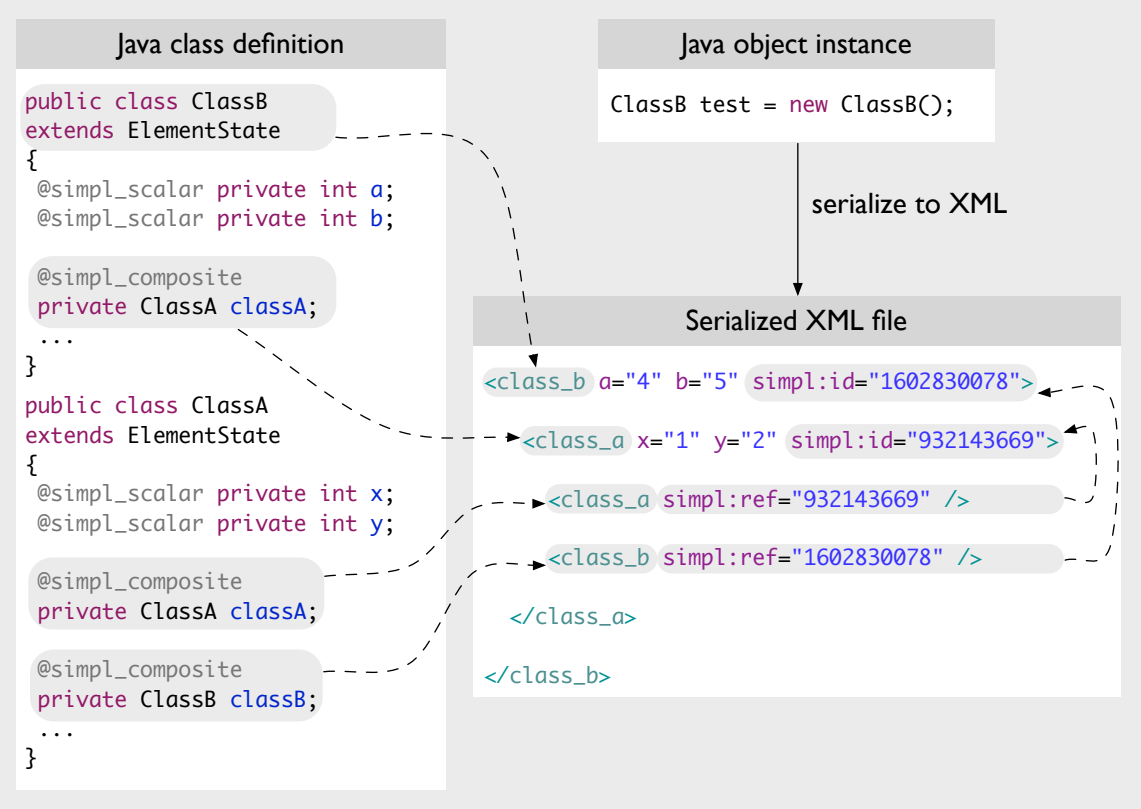

Fig. 15.: Class definitions of `ClassA` and `ClassB`, which produces cyclic references. Serialized represented of an instances of `ClassB` and `ClassA` make use of `simpl:id` and `simpl:ref` attributes that facilitates representation of cyclic references.

while also assigning the reference of `ClassB`'s instance to the instance variable in `ClassA` produces a cyclic reference.

S.IM.PL Serialization uses a two-pass algorithm to serialize graphs that contain cyclic references. The first pass performs a depth first search from the root node and recursively moves deeper into the object graph. Every visited node is added to a `visitedElements` hash table while generating a unique id for the object. For every node object, the algorithm determines if it has been visited by checking its entry in `visitedElements` . If an object has already been visited, the algorithm marks these objects as *requiring* `simpl:id`, and halts further recursive calls. The second

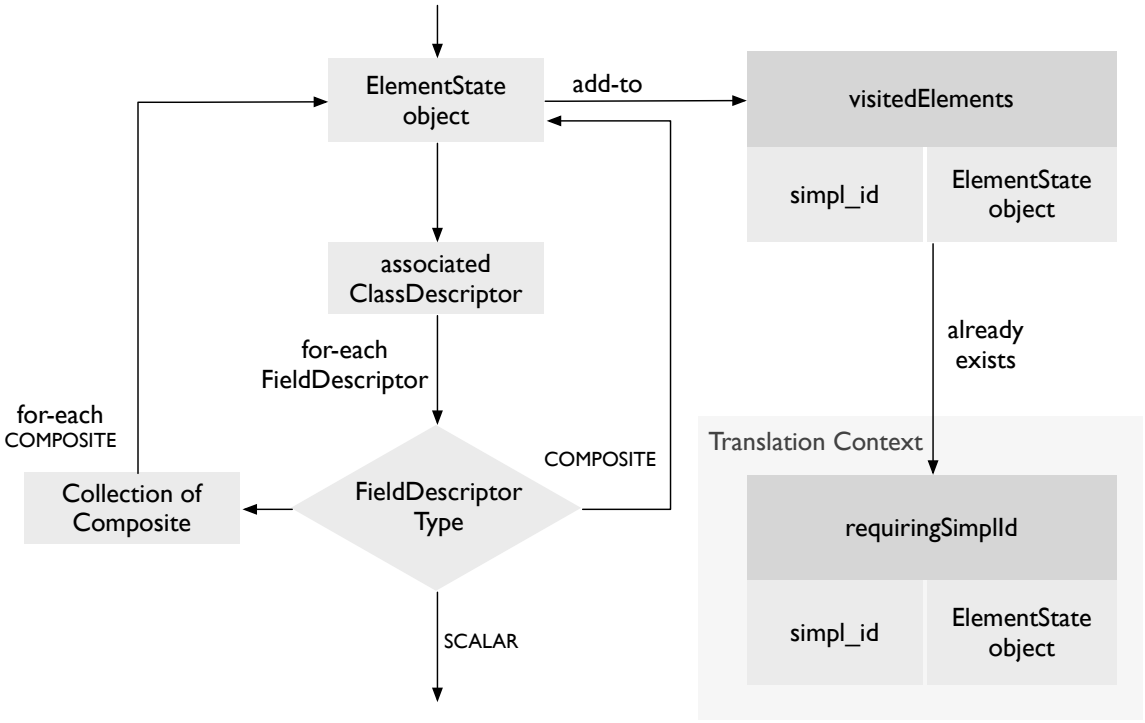

Fig. 16.: Resolve Graph Algorithm: The first-pass that populates `requiringSimplId` map in `TranslationContext`. The map contains references to `ElementState` objects that were referenced more than once with their corresponding `simpl_id` as key. Later in second-pass, `TranslationContext` facilitates graph handling.

pass serializes the root object, and recursively, the objects it is composed of. Each object marked as *requiring* `simpl:id` is serialized with an extra attribute: the unique id as the value of `simpl:id` attribute. Once serialized, these objects are marked as *serialized*. Subsequent references to these objects are only serialized by their tag name, with the unique id as the value of the `simpl:ref` attribute. Thus, recursive calls for back references are avoided.

The serialized document contains referential information so it can subsequently be deserialized. When deserializing, the framework maintains a map (`deserializedObjects`) of objects that contain a `simpl:id` attribute using the

value specified by the attribute. When a `simpl:ref` tag is encountered, it means that the specified object has already been deserialized. It can thus be accessed by its unique id from `deserializedObjects`. The framework finds an already created instance from the mapping using the unique id specified by `simpl:ref` attributed, instead of creating a new instance of the object. The deserialization process is thus able to create a correct object model, which contains cyclic or back references.

Handling graphs is computationally expensive as the framework needs to perform an extra pass on the object model before serialization. This could decrease performance of serialization. In cases, where developers can guarantee the absence of cyclic references, they can turn-off graph handling for better performance. In such cases, the framework will recognize and report an error when serialization detects a cyclic reference.

CHAPTER IV

MULTI-FORMAT SUPPORT

Data can be represented in multiple formats. Some formats are more readable, while others are more concise. Concise formats require less communication bandwidth. Due their conciseness, human-readability is reduced, which affects debugging. As software engineers, we choose data representation formats that best address software requirements, such as available bandwidth, ease of development and debugging, and integration with other software systems. However, choosing the right format is difficult, as software requirements may be unclear in the early stages of software development. In addition, readable formats are easy to debug, but verbose and inefficient in the production phase of a software application. Therefore, we are often faced with requirement of incorporating, switching, or migrating to another data representation format, which increases project cost in terms of development and testing time.

S.IM.PL Serialization addresses the problem of investing additional development and testing time in software applications that require transitioning to a different data representation format, by supporting multiple formats and allowing seamless switching between them. Type system scopes play a crucial role in supporting seamless switching, as they provide a format-agnostic abstraction, which is used by format-specific de/serialization methods.

In the next section, we describe the formats supported by S.IM.PL Serialization: XML, JSON, and TLV; explaining their advantages and disadvantages. Later, we describe the `@simpl_hints` annotation that enables fine-grained control over a specific data representation format. Finally, we examine support for de/serializing objects in BibTeX format.

A. Supported Formats

XML is a portable, extensible, human-readable, and document-oriented format for communication. XML is widely used by most distributed and web applications. However, XML is generally considered as verbose. In applications, such as on the Internet, where network bandwidth is a limited resource, software engineers may be more inclined to use less verbose data representation formats. JSON, a close alternative of XML, is popularized by dynamic programming languages, mainly JavaScript, due to its simplified syntax and key-value notations. JSON is a concise, non-extensible, data-oriented format. JSON is generally considered as lightweight and fast, but it is less readable.

Table III.: Shows advantages and disadvantages of data representation formats supported by S.IM.PL Serialization

Format	Advantages	Disadvantages
XML	extensible, human-readable format	verbose
JSON	concise, light-weight, readable	non-extensible, less readable than XML
TLV	very concise binary format	non-readable
BibTeX	concise, used as databases of bibliographies, third party tools can easily utilize	domain-specific, cannot support composite objects

The focus of the XML and JSON is readability, which facilitates debugging and integration with different software systems. Often, readability becomes less important in a *closed* distributed software system, when it is functionally stable and does not

require integration with any third party software system. A concise binary format, such as TLV is non-readable, but addresses better performance and reduced bandwidth requirements. Table III summarizes the advantages of data representation formats supported by S.IM.PL Serialization. Support for further data representation formats, such as YAML is planned.

B. Fine-grained Control

Flexibility in representing information is crucial for building scalable applications. As software engineers, we want to write application components that can be re-used in other software systems. S.IM.PL Serialization provides flexibility in data binding through annotations. Annotations discussed earlier (II, D) are generic. They are not specific to any particular data representation format. However, software engineers require control over specific elements in a particular data representation format. This fine-grained control over a particular serialized representation is provided through a special `@simpl_hints` annotation.

Table IV.: The `@simpl_hints` annotation accepts an array of the following enumerated parameters, allowing specification of hints for multiple formats.

Hints	Semantics
`HINT.XML_ATTRIBUTE`	translate field as an XML attribute
`HINT.XML_LEAF`	translate field as an XML leaf node
`HINT.XML_LEAF_CDATA`	translate as CDATA in XML leaf node
`HINT.XML_TEXT`	translate field as text node in XML
`HINT.XML_TEXT_CDATA`	translate field as CDATA in XML text node

The `@simpl_hints` annotation accepts parameter of enumerated type, which further specifies how an augmented field is represented in a particular format. For example, to serialize a scalar field, the `@simpl_scalar` annotation defines the field as scalar serializable. However, there are multiple ways to serialize a scalar field in XML: as an attribute, or as a single text field value in an element. Further, in the later case, the value may be optionally quoted as CDATA. Thus, one can add an additional annotation to the declaration: Table IV. The default is `Hint.XML_ATTRIBUTE`. The `@simpl_hints` can take an array of arguments, in case different hints for different formats need to be specified in a single declaration.

C. BibTeX Support

In addition to supporting serialization and deserialization in XML, JSON and TLV, the framework also implements support for serialization in the BibTeX [9] format.

```
@bibtex_type("inproceedings")
public class Entry extends ElementState
{
 @bibtex_key
 @simpl_scalar
 private String citationKey;

 @bibtex_tag("title")
 @simpl_scalar
 private String title;

 @bibtex_tag("authors")
 @simpl_nowrap
 @simpl_collection("author")
 private ArrayList<String> authors;
 ...
}
```

```
@inproceedings{Kerne:2008:CXB:1410152,
  title = {A concise XML binding
           framework facilitates
           practical object-oriented
           document engineering},

 author = {Kerne, Andruid and
           Toups, Zachary O. and
           Dworaczyk, Blake and
           Khandelwal, Madhur},
  ...
 }
```

Fig. 17.: An example of data binding BibTeX data. A publication's BibTeX entry in mapped to a Java class, utilizing BibTeX specific annotations for specifying BibTeX key, type, and alternative tags.

Table V.: Annotations specific to BibTeX data representation format.

Annotation	Semantics
`@bibtex_type`	defines the type of BibTeX entries represented by the class definition
`@bibtex_key`	defines a scalar field as BibTeX key
`@bibtex_tag`	defines an additional tag mapping for a field when serialized in BibTeX format
`@simpl_composite_as_scalar`	defines a scalar field as a scalar value for a composite element. This annotation can also be used for other formats that do not support the representation of composite elements

BibTeX is a widely used format for storing bibliography of scholarly articles and publications. BibTeX is a very limited format of representation and lacks hierarchy as represented by composite elements. Thus, BibTeX support is limited to scalars and collections of scalars. For BibTeX, composite elements can make use of an additional `@simpl_composite_as_scalar` annotation, which specifies a scalar field inside a composite element as its scalar value. Figure 17 shows an example of a Java class serialized in BibTeX.

Some BibTeX specific annotations (Table V) have been added to DBAL for use in conjunction with other S.IM.PL-annotations. `@bibtex_type` annotation defines the type of a BibTeX entry. Even though the bibtex database management utilities work with only a specific set of BibTeX types, a set of valid values is not enforced by the BibTeX language specifications. Thus, any string value can be passed as parameter for this construct. It is up to the developer to pass a BibTeX type that

will be handled correctly by external management utilities. `@bibtex_key` annotation specifies a Java field as key for a BibTeX entry. A BibTeX key uniquely identifies each entry in a BibTeX database. `@bibtex_tag` is an optional annotation, which specifies an alternate tag name in-case of BibTeX serialization. If this annotation is not present, the framework automatically utilizes the tag-name specified through `@simpl_tag` annotation or camel-case conversion of the field name. Again, the possible values for the tag names are not enforced by the language specification and developers should ensure that these tag names are correct for external tools. Table V summarizes these annotations.

Support for BibTeX serialization has enabled software engineers to develop software applications that allow users to export sets of store references of articles and publications to third party citation management tools, such as BibDesk [19] for Mac OSX.

D. Conclusion

Type system scopes enable multi-format support in S.IM.PL Serialization, as they abstract data binding semantics, which are independent of any particular format. Utilizing the abstract data binding semantics encapsulated in type system scopes, S.IM.PL Serialization implements de/serialization functionalities in different formats. Figure 18 presents an overview of translation of data into multiple formats.

Fine-grained control over a particular format is provided through DBAL constructs that further specifies how a particular field is represented in a particular format.

Switching between formats is seamless. Software engineers can specify the data format through parametrized de/serialization functions. This enables applications to

easily translate data from one format to another, integrate with third party software, and facilitates debugging, as software engineers can switch from non-readable format to readable format and identify errors in data.

S.IM.PL Serialization comes with built-in support for XML, JSON, TLV, and BibTeX formats of data representation. We plan to integrate support for other formats, such as YAML, an alternative to XML that is gaining popularity in web applications.

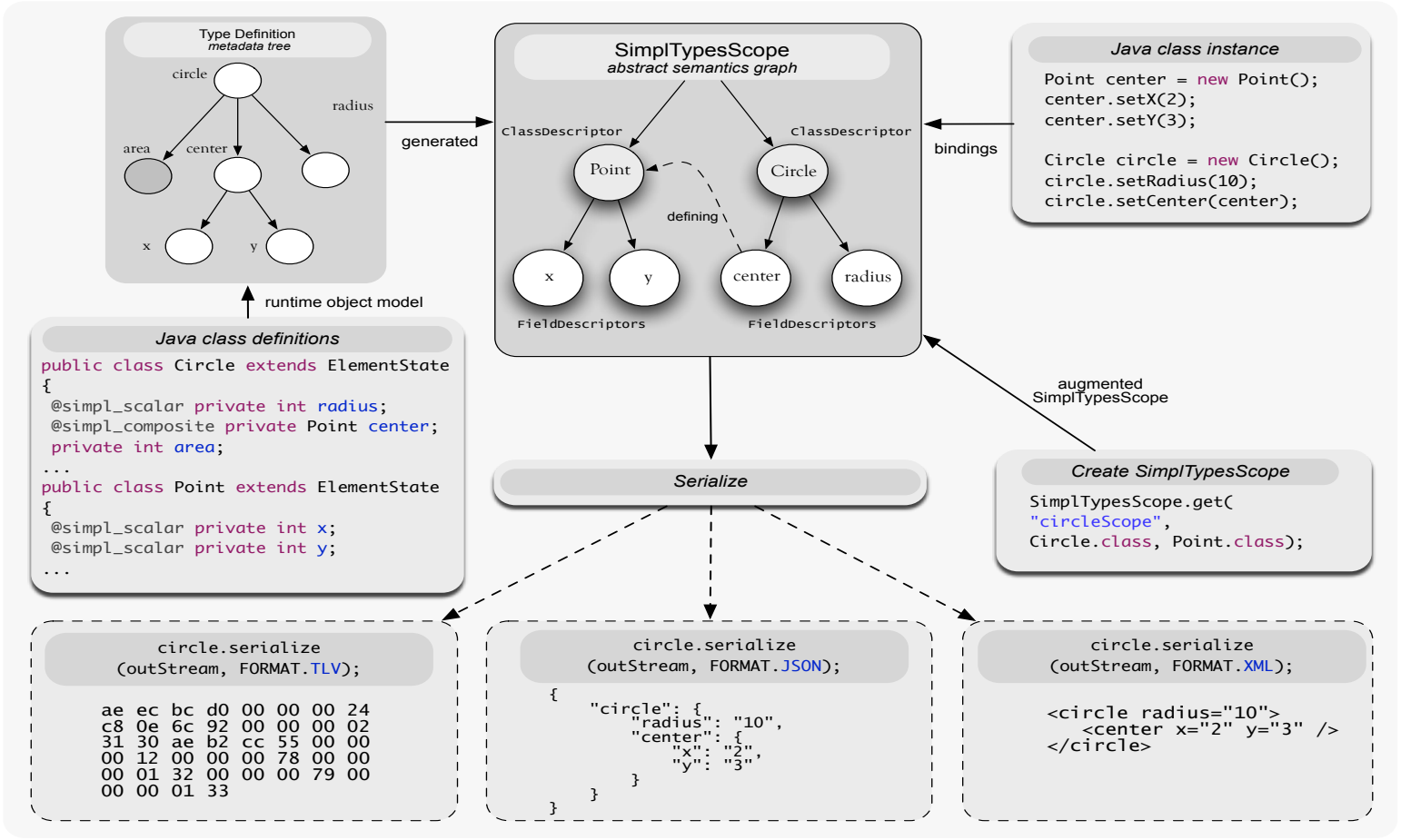

Fig. 18.: Multi-Format Translation Overview: A class definition of `Circle` containing a composite object of type `Point`. We generate the `SimplTypesScope` for DBAL-augmented fields. The abstract semantics graph encapsulate data bindings that facilitate translation of an instance of type `Circle` to supported data formats.

CHAPTER V

CROSS-LANGUAGE SUPPORT

As software engineers, we are often faced with requirements of building distributed software applications that rely on message passing across different platforms. Working with same information in different platforms is cumbersome as developers write platform-specific for parsing and manipulating data.

Various tools and techniques have been introduced to ease the burden on software engineers writing cross-language distributed software applications. However, these technologies have certain shortcomings. For example, the Component Object Resource Broker Architecture (CORBA) [20] standard is a widely used technology that enables cross-language information exchange. It requires software engineers to learn an Interface Description Language (IDL) that is used to specify the structure of messages. In addition to learning a new language, IDL files are maintained external to the source code, where they are difficult to manage, as changes in the source code requires changes in external files. Also, developers have no control over how information is serialized and represented. Therefore, integration with third party software systems and debugging are difficult.

S.IM.PL Serialization addresses shortcomings of prior technologies and eases the burden on software engineers. In S.IM.PL Serialization, software engineers are only required to write classes in one source programming languages (as of now, Java is supported as source programming language). Type system scopes are used to generate code in target programming languages (C# or Objective-C). This relieves software engineers from the burden of writing platform-specific object declaration code in multiple programming languages. The consistent structure of serialized representations

specified through Data Binding Annotation Language (DBAL) (II, D), which has simple specifications in comparison to IDL. DBAL declarations are within the source-code, therefore, they are easier to maintain. Software engineers also gain complete control over how information is serialized and represented through DBAL constructs.

In the next section, we present the supported data types and mappings in S.IM.PL Serialization. Then, we describe the code generation facilities that utilize type system scopes and data type mappings to generate code including documentation in target programming languages. Finally, we differentiate how cross-language data binding is supported in programming languages that support and do not support annotation features.

A. Supported Types and Mappings

An important aspect in providing cross-language support is the mapping of data types from one programming language to another. All primitives, as well as some complex scalar types, such as Date, URL, StringBuilder have one-to-one mappings between supported programming languages. S.IM.PL Serialization maintains mapping between language-specific data types, which are used used by code generation facilities (VI, B). Table VI shows the mappings between primitive data types, Table VII shows the mappings between non-primitive/complex data types. These data types are all a developer would use in many cases. However, new scalar types and their corresponding mappings can be easily added.

B. Cross-Language Class Translation

S.IM.PL Serialization currently supports cross-compilation from Java to C# and Objective-C. The process of cross-compiling Java code is closely integrated with type

Table VI.: The primitive data types in Java and their equivalent data types in C# and Objective-C

Java / size (bits)	C# / size (bits)	Obj-C / size (bits)	Java Default
int (32)	Int32 (32)	int (32)	0
float (32)	Single (32)	float (32)	0.0f
double (64)	Double (64)	double (64)	0.0d
byte (8)	Byte (8)	char (8)	0
char (16)	Char (16)	char (8)	\u0000
boolean (1)	Boolean (1)	bool (1)	false
long (64)	Int64 (64)	long long (64)	0L
short (16)	Int16 (16)	short (16)	0

system scopes. Since, type system scopes completely represent the abstract data types with `ClassDescriptor`s and `FieldDescriptor`s containing the appropriate metadata about each field and class, code translation is a straightforward process. The framework traverses through the type system scope ASG once, calculating dependencies of each class, and a second time to generate code for each class in the target programming language. Developers can also augment the type system scope (III, D) before generating code to ensure all dependencies are included.

While generating code, the framework utilizes the above mentioned data type mappings to generate an equivalent class. Certain limitation do apply during cross-language code translation. For example, variable names in the source programming language might be keywords in target programming language, such as `in` and `id`, which can be used as variable names in Java, but are keywords in Objective-C. The

Table VII.: Shows the non-primitive/complex data types in Java and their equivalent data types in C# and Objective-C

Java	C#	Objective-C
Date	DateTime	NSDate
StringBuilder	StringBuilder	NSMutableString
Url	Uri	NSURL
ParsedURL	ParsedURL	ParsedURL
Class	Type	Class
Field	FieldInfo	Ivar
enum	enum	*not-supported*
File	FileInfo	NSFileHandle
Color	Color	UIColor
ArrayList	List	NSMutableArray
HashMap	Dictionary	NSDictionary
HashMapArrayList	DictionaryList	DictionaryList

cross-compilation utility issues appropriate warnings in such cases, which a software developer must correct. In addition to these limitations, a developer must also account for the difference in type mappings from one programming language to another. For example, the size of a Java `char` is 16 bits as compared to the 8 bit `char` in Objective-C. Such differences can lead to error or loss of information in translating across platforms. Cross-compilation also supports framework provided classes such as HashMapArrayList and ParsedURL, as alternatives to optimization the basic

HashMap and URL classes. Also note that enumerated types are not cross-compiled to Objective-C. Their de/serialization is not supported by the Objective-C version of the framework, due to lack of Objective-C runtime introspection capabilities for enumerated types.

Generic data types are also recursively resolved to appropriate data types in target programming languages. In the case of Objective-C, which supports polymorphism, but does not support generic class definitions, generic types are ignored. This does not affect functionality in Objective-C, as correct data types will be instantiated at runtime through type system scopes. In the case of C#, the framework will generate generic type definitions, as well as generic classes with correct parameters. Figures 19 and 20 present cross-compiled `GameData` class (example walkthrough II, A) in C# and Objective-C.

1. Documentation Translation

An important aspect of building reusable software components is code documentation. When translating Java class files to a target programming language, Javadoc comments are preserved. The cross-compilation utility parses through the source code, extracting field and class level comments. The comments are mapped to comment styles in translated programming language. In addition to mapping comments, DBAL declarations are also inserted in comment descriptions for reference in target programming language. Javadoc comments are mapped to XML comments for C# and header doc comments for Objective-C. .NET documentation utilities such as NDoc [21], can effectively parse XML comments from C# class definitions, while HeaderDocs [22] can be used to parse comments in Objective-C header files.

```
/// <summary>
/// Encapsulates the run time state of the game
/// </summary>
public class GameData : ElementState
{
    /// <summary>
    /// time stamp value.
    /// </summary>
    [simpl_scalar]
    private Int64 timestamp;

    /// <summary>
    /// number of game cycles remaining
    /// </summary>
    [simpl_scalar]
    private Int32 cycRem;

    /// <summary>
    /// current score in the game
    /// </summary>
    [simpl_scalar]
    private Double score;

    /// <summary>
    /// objects containing information on threat entities
    /// </summary>
    [simpl_scope("threatTypes")]
    [simpl_collection]
    private List<Threat> threats;

    ..
}
```

Fig. 19.: The `GameData` class in C#. S.IM.PL code-generation facility generated code with equivalent DBAL declarations and documentation from the Java source code.

```
@interface GameData : ElementState
{
  /*!
       @var         timepstamp
       @abstract    annotated as : @simpl_scalar
       @discussion  time stamp value
  */
  long timestamp;

  /*!
       @var         cycRem
       @abstract    annotated as : @simpl_scalar
       @discussion  number of game cycles remaining
  */
  int cycRem;

  /*!
       @var         score
       @abstract    annotated as : @simpl_scalar
       @discussion  current scope in the game
  */
  double score;

  /*!
       @var         threats
       @abstract    annotated as : @simpl_scope("threatScope")
                    annotated as : @simpl_collection
       @discussion  objects containing information on threat entities
  */
  NSMutableArray *threats;
}

@property (nonatomic,readwrite) long    timestamp;
@property (nonatomic,readwrite) int     cycRem;
@property (nonatomic,readwrite) double  score;
@property (nonatomic,readwrite, retain) NSMutableArray *threats;
```

Fig. 20.: The `GameData` class in Objective-C produced by S.IM.PL code-generation facilities. Field declaration, annotations, and document comments are all appropriately translated from Java source code.

C. Portable Type System Scopes

For languages that support annotations, such as C#, S.IM.PL produces equivalent DBAL. The cross-compiler generates the correct declarations of DBAL for each field from the source programming language. This enables type system scopes to be automatically generated in target programming languages. Figure 19 shows the equivalent C# class definition, which make use of similar language constructs.

Type system scopes are data structures defined within S.IM.PL Serialization framework, augmented with DBAL. Thus they themselves can also be serialized. Target languages such as Objective-C, which do not support annotations, rely on data binding from serialized type system scopes. In S.IM.PL Serialization, a type system scope is serialized as XML or JSON, which can be parsed in a target programming language to create an equivalent type system scope that drives the de/serialization processes. Figure 21 shows a serialized type system scope XML for the example `gameDataSimpTypesScope`.

D. Conclusion

Cross-language support in S.IM.PL Serialization is facilitated through type system scopes, as they describe data structures in a language-independent type system and specify how objects bind with serialized representations.

Equivalent specification of type system scopes in particular programming languages, either through DBAL augmented class definitions or serialized type system scopes, enables cross-language data binding.

We provide code generation utilities that facilitate cross-language data binding. The code generation utility uses type system scopes to generate code in any supported target programming language. Code comments used for documentation of the source

```
<simpl_types_scope name="gamedata">
 <class_descriptor described_class="GameData" tag_name="game_data">
  <field_descriptor field="timestamp" tag_name="timestamp" type="18"
       scalar_type="LongType" xml_hint="XML_ATTRIBUTE"/>
  <field_descriptor field="cycRem" tag_name="cyc_rem" type="18"
       scalar_type="IntType" xml_hint="XML_ATTRIBUTE"/>
  <field_descriptor field="score" tag_name="score" type="18"
       scalar_type="DoubleType" xml_hint="XML_ATTRIBUTE"/>
  <field_descriptor field="score" tag_name="score" type="18"
       scalar_type="DoubleType" xml_hint="XML_ATTRIBUTE">
   <polymorph_class_descriptors>
    <class_descriptor described_class="Threat" tag_name="t"
      simpl:ref="1480462011">
     <field_descriptor field="m_id" tag_name="m_id" type="18"
       scalar_type="StringType" xml_hint="XML_ATTRIBUTE" />
     <field_descriptor field="online" tag_name="online" type="18"
       scalar_type="BooleanType" xml_hint="XML_ATTRIBUTE" />
     <field_descriptor field="m_in" tag_name="m_in" type="18"
       scalar_type="BooleanType" xml_hint="XML_ATTRIBUTE" />
     <field_descriptor field="safe" tag_name="safe" type="18"
       scalar_type="BooleanType" xml_hint="XML_ATTRIBUTE" />
     <field_descriptor field="ord" tag_name="ord" type="18"
       scalar_type="IntType" xml_hint="XML_ATTRIBUTE" />
    </class_descriptor>
    <polymorph_class_descriptor described_class="PatrollingThreat"
      simpl:ref="841752171">...
    </polymorph_class_descriptor>
    <polymorph_class_descriptor described_class="OrbitingThreat"
      simpl:ref="429405933">...
    </polymorph_class_descriptor>
    <polymorph_class_descriptor described_class="RepellableThreat"
      simpl:ref="107251772"> ...
    </polymorph_class_descriptor>
    <polymorph_class_descriptor described_class="PatrollingThreat">
      simpl:ref="1584946533"> ...
    </polymorph_class_descriptor>
   </polymorph_class_descriptors>
  </field_descriptor>
 </class_descriptor>
 <class_descriptor simpl:ref="1480462011" />
 <class_descriptor simpl:ref="841752171" />
 <class_descriptor simpl:ref="429405933" />
 <class_descriptor simpl:ref="107251772" />
 <class_descriptor simpl:ref="1584946533" />
</simpl_types_scope>
```

Fig. 21.: `GameData` type system scope serialized through S.IM.PL Serialization. This XML representation is utilized by S.IM.PL code generation facilities to generate code in target programming languages and data binding in Objective-C.

code are part of the type system scope enabling parsing by third party documentation utilities. Figure 22 shows an overview of cross-language translation of data objects between a source and target programming language.

Presently, cross-language support is provided between Java, C#, and Objective-C programming languages. Support for JavaScript as target programming language is currently under development. Support for further programming languages, such as C++ and Python, is planned.

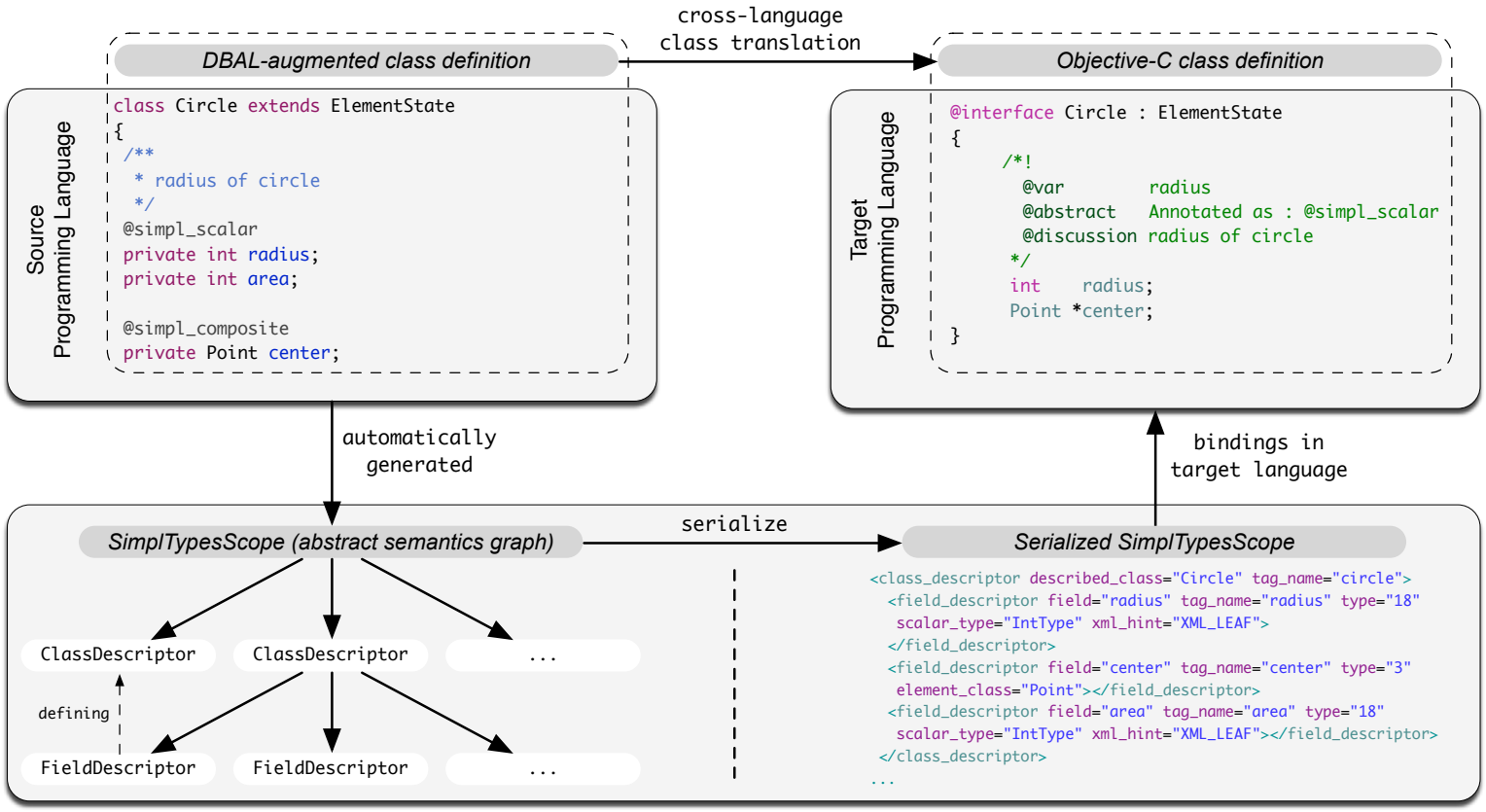

Fig. 22.: Cross-Language Translation Overview: A class definition of type `Circle` is translated to Objective-C. Code comments are also translated. Objective-C does not support annotations. Data binding is facilitated through `SimplTypesScope` declaration in XML.

CHAPTER VI

MULTI-FORMAT AND CROSS-LANGUAGE LIMITATIONS

In supporting multiple formats and programming languages, S.IM.PL Serialization must address limitations imposed by a particular data format or programming language. Similarly, developers writing cross-language or multi-format software, must account for such limitations.

In this section, we describe limitations in using multiple formats and programming languages and how they impact software development.

A. Format Specific Limitations

As mentioned earlier, S.IM.PL Serialization supports XML, JSON, TLV, and BibTeX data formats. Data formats are tree structured. How this tree hierarchy is represented, differs between formats and their syntax enables certain structures, which cannot be supported in other formats. In this section, we describe limitations specific to a particular format.

1. BibTeX with Composite Objects

The BibTeX data format allows one level of nesting, where the initial node must always be the BibTeX type. This limitation restricts BibTeX format to be able to represent an object containing scalars or collection of scalars, but greatly simplifies the data format. During serialization, composite objects are ignored by S.IM.PL Serialization unless the developer has specified `@simpl_composite_as_scalar` annotation, which specifies a scalar value of a collection.

2. JSON with Polymorphic Collections

The JSON data format allows multiple level of nesting, similar to XML, enabling representation of deeply nested composite objects. It provides additional syntax structures for representing collections, which enables concise representation of collections. Figure 23 shows wrapped and unwrapped collection of `threats`. Note that using the array syntax ('[' delimiter), the format assigns the tag to the collection, instead to each element in the collection. We utilize the array to determine the correct type of objects in the collection. If the collection is wrapped, the collection becomes the child of the `threat` object, which is associated to the collection field in the `GameData` class. This structure is useful to concisely represent collections that are not polymorphic, as the array tag is sufficient to determine the type of all objects in the collection.

In case of polymorphic collections, we must declare inner tags to specify how to

(1) JSON non-polymorphic (unwrapped)

```
{
   "game_data": {
      "timestamp": "1234399958508",
      "cyc_rem": "8078",
      "loaded": "true",
      "running": "true",
      "threat": [
         {
            "ord": "12"
         },
         {
            "ord": "13"
         }
      ],
      "score": "-28.066665835678577"
   }
}
```

(2) JSON non-polymorphic (wrapped)

```
{
   "game_data": {
      "timestamp": "1234399958508",
      "cyc_rem": "8078",
      "loaded": "true",
      "running": "true",
      "threats": {
         "threat": [
            {
               "ord": "12"
            },
            {
               "ord": "13"
            }
         ]
      },
      "score": "-28.066665835678577"
   }
}
```

Fig. 23.: The representation of the `GameData` object as JSON, containing non-polymorphic `threats` collection as wrapped and unwrapped.

(1) JSON polymorphic

```
{
   "game_data": {
      "timestamp": "1234399958508",
      "cyc_rem": "8078",
      "loaded": "true",
      "running": "true",
      "threats": [
         {
            "nt": {
               "ord": "12"
            }
         },
         {
            "ot": {
               "ord": "13"
            }
         }
      ],
      "score": "-28.066665835678577"
   }
}
```

(2) XML polymorphic (wrapped)

```
<game_data timestamp="1234399958508"
 cyc_rem="8078" loaded="true"
 running="true">
 <threats>
  <nt ord="12" />
  <ot ord="13" />
 </threats>
 <score>-28.066665835678577</score>
</game_data>
```

(3) XML polymorphic (unwrapped)

```
<game_data timestamp="1234399958508"
 cyc_rem="8078" loaded="true"
 running="true">
 <nt ord="12" />
 <ot ord="13" />
 <score>-28.066665835678577</score>
</game_data>
```

Fig. 24.: The representation of the `GameData` object in XML and JSON, containing polymorphic collection `threats`.

correctly map collection element object representations to their correct data types. In Figure 24 the `GameData` object contains a polymorphic collection of `threats`. The inner-tag determines the type of the object. Since, the array tag (`threats`) cannot be empty, we are limited to serializing polymorphic collection as wrapped.

B. Cross-Language Limitations

Programming language features, keywords, native data types, and language-specific development frameworks are not consistent among programming languages. When writing cross-language software we must account for such differences. In S.IM.PL Serialization, such differences impose limitations on how its cross-language support is utilized.

1. Keywords in Programming Languages

The set of keywords differs among programming languages. As S.IM.PL code-generation facilities generates code in multiple programming languages, a valid identifier in one programming language might be a keyword in another programming language, which will produce compile errors in generated code. Therefore, when writing cross-language software with S.IM.PL, developers are limited to not using identifies that are keywords in another programming language. We provide mechanisms for software developers to identify and correct such issues, if they occur.

2. Data Types and Mappings

Data type mappings, their runtime memory size, and how they function in one programming language may differ from another programming language. The `char` data type in Objective-C is 8 bits as compared to 16 bit `char` in Java and Objective-C, which may result in loss of information during cross-language data exchange.

The mappings of data types from one programming language to another is developed based on our experience as software developers. Some developers may disagree with the provided mappings. For example, the Java `HashMap` data type represents a hash table data structure. An exact equivalent of `HashMap` is not provided by C# or Objective-C programming language. The closest data type in functionality is `Dictionary` (C#) or `NSMutableDictionary` (Objective-C). The S.IM.PL code generation facilities will map `HashMap` to `Dictionary`, or `NSMutableDictionary`. It is up to the developers to manage differences in use of the mapped data type.

3. Use of Generics

The use of generics differs between Java and C#. In Objective-C, generic parameters are ignored, as they are not supported. Java supports wildcards ('?') for generic parameter declaration of types unknown at compile-time. A user can declare a generic collection in Java, as `ArrayList<?>`, which implies that the collection can be of any generic type or specify a constraint, as `ArrayList<? extends Threat>`, which implies that the collection is of any sub-type of the `Threat` class. In comparison, C# does not allow wildcards.

The S.IM.PL code generation ignores ambiguous generic parameters when translating code to C#. For example, the `ArrayList<?>` declaration will be translated as `List` and `ArrayList<? extends Threat>` is translated as `List<Threat>`.

CHAPTER VII

VALIDATION - RESEARCH FRAMEWORKS

We developed software frameworks that build on S.IM.PL Serialization by extending type system scopes and utilizing data binding features. By examining how these frameworks function, we validate S.IM.PL Serialization's extensible design and motivate the value of its data binding architecture. In this chapter, we examine software frameworks: Object-Oriented Distributed Semantics Services (OODSS), meta-metadata language and architecture, and Preferences Management System. These frameworks utilize multi-tiered capabilities of S.IM.PL Serialization.

OODSS utilizes flexible data binding and support for binding polymorphic data types to implement an object oriented and flexible framework for network communication and remote method invocation. The meta-metadata language and architecture utilizes the abstraction provided by type system scopes to develop a type-system for describing information found in digital repositories and on the Internet. The Preferences Management System utilizes support for binding polymorphic fields to develop an XML-based type-system that enables software engineers to easily specify and gain quick access to configuration settings in software applications.

The source code of these frameworks is available as open-source. They are used in research software applications, which we will examine in Chapter VII. Students in undergraduate and graduate courses in Department of Computer Science and Engineering at Texas A&M University have benefited from these frameworks to develop software applications for course assignments and research softwares.

A. Object Oriented Distribute Semantics Services

OODSS [23] is a research framework that focuses on facilitating software engineering principles in writing information-centric distributed software applications. It utilizes key S.IM.PL Serialization features such as flexible data binding, data binding polymorphic objects, and type system scopes to implement a framework for network communication and remote method invocation. The OODSS framework is utilized in research software applications: Team Coordination (TeC) game (VIII, B) and combinFormation (VIII, A). It is also used by students developing software applications for course assignments.

In the next section, we examine the role of S.IM.PL Serialization in OODSS's implementation of service call and return functionality. Later, we show how flexible data binding results in concise messages. Finally, we present how OODSS services can function across platforms in multiple formats through S.IM.PL Serialization's cross-language multi-format support.

1. Polymorphic Message Architecture

OODSS implements novel semantics for service call and returns in distributed applications utilizing the *command pattern* [24]. In the command pattern, an object encapsulates the complete information required to call a method at a later time.

In OODSS, a message is represented by a class definition annotated with declarations in Data Binding Annotation Language (DBAL). The instance of the message class encapsulates information required to invoke the remote method. Through subclassing messages take the form of *request* or *response* types. Request messages are sent by client applications to invoke the service method on the server; the reply from server is sent as a response message that invokes the response method on the client.

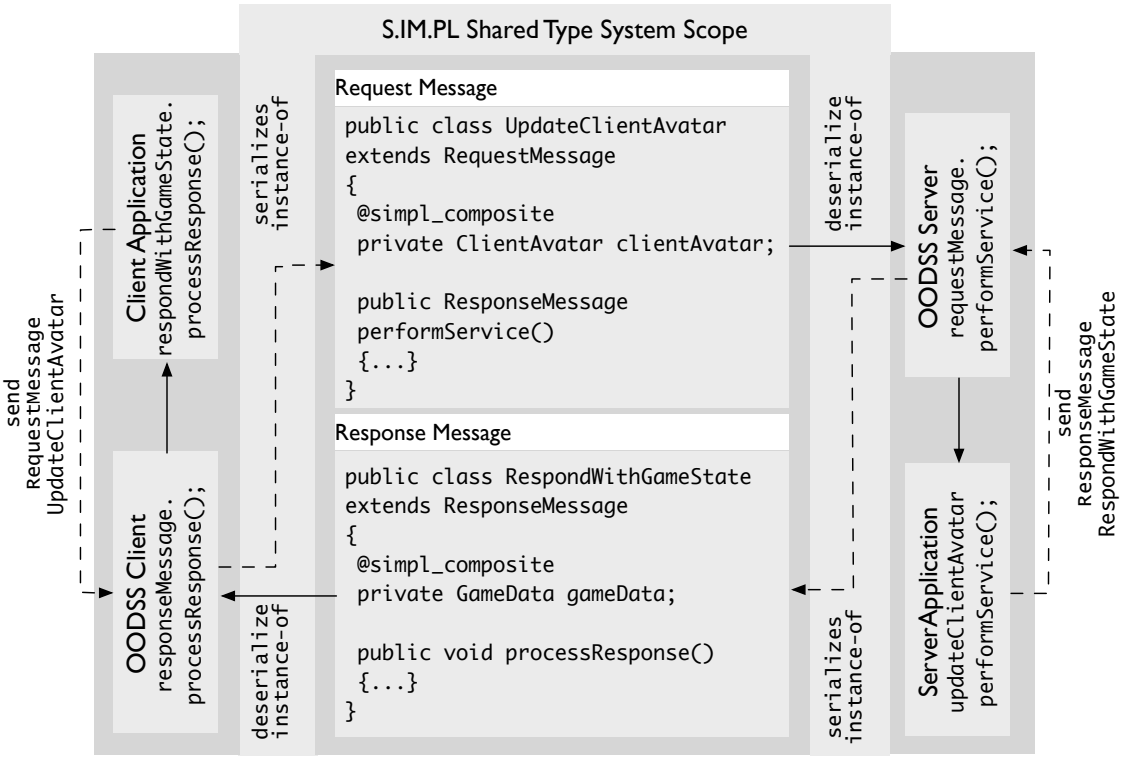

Fig. 25.: The communication flow and de/serialization of messages in OODSS. A shared type system scope encapsulates the subtypes of request and response message. Based on the polymorphic subtype of the message, invocation of `performService()` and `processResponse()` methods are dynamically dispatched.

Therefore, the subtype of a request or a response message specifies an implementation of a service or response method.

Figure 25 shows a request/response communication architecture in the multi-player Team Coordination (TeC) game. The client updates a player's location on the server, and the server responds by sending an updated state of the game object back to the client application. The client creates an instance of the `UpdateClientAvatar` request message, which is serialized and transported over the network to the server where S.IM.PL Serialization deserializes it. The server calls the `performService()` method on the object. Request messages extend a common `RequestMessage` base

class, therefore, the implementation of `performService()` is dynamically dispatched and executed on the server. Similarly, the service method creates an instance of `RespondWithGameState` response message, which is transported to the client. The client executes the `processResponse()` method, which is dynamically dispatched, as instances of response messages inherit from a common `ResponseMessage` base class.

The example shows that OODSS services rely on polymorphism to execute the overridden implementations of `performService` and `processResponse` methods. Therefore, instantiation of the correct subtype is crucial for executing the right implementation.

Type system scopes in S.IM.PL Serialization ensure the instantiation of the correct subtype from a serialized representation. In Figure 25, the service provider and consumer share a common type system scope. Sharing a common type system scope means that the provider and consumer can understand the messages exchanged between them. On a lower-level, this means that the service provider can deserialize messages, which are serialized and transmitted by service consumer and vice-versa.

Therefore, type system scopes function as objects of mutual understanding between the server and client applications. Serialized messages that do not conform to the bindings specified in the shared type system scope are rejected, with appropriate error reporting by OODSS.

2. Flexible Data Binding

An important facility that promotes polymorphic message architecture is the ability to bind only the required fields and specify how they are represented. In OODSS, message classes can contain local state variables, which dispatched methods utilize for executing application specific logic. These fields are omitted from serialization and sent over the network.

The ability to flexibly specify data bindings enables software engineers to specify their own message structures, instead of conforming to lengthy specifications enforced by message representation protocols such as SOAP [25] and XML-RPC [26]. Flexibility, fine-grained control over message representation, and the specification of the messages defined by a type system scope result in concise messages, as shown in Figure 26. An instance of `ClientAvatar`, from the `UpdateClientAvatar` request message, is serialized by OODSS using S.IM.PL Serialization and SOAP and XML-RPC protocols. The XML message produced by OODSS is significantly smaller in size and more readable as compared to SOAP or XML-RPC messages.

3. Platform-Independent and Multi-Format Approach

S.IM.PL Serialization enables OODSS services to work across platforms. The OODSS approach to cross-language distributed services has advantages over prior Interface Description Language (IDL)-based approaches. In IDL-based frameworks, such as CORBA [20], DCOM [27], and Mockingbird [28], programmers define message structures by writing specifications in IDL. Later, from IDL definitions, programmers can generate platform specific code for de/serialization of messages. Software development with this approach is cumbersome to maintain, as IDL-definitions are maintained in external files. In OODSS, message structure is specified by augmenting data type definitions with DBAL. This is easier to maintain through refactoring. The DBAL declarations also serve as means of documentation of source code. Software programmers can easily identify which fields are de/serialized and how they are represented.

Due to the inherent support of multiple-formats (XML, JSON, and TLV) in S.IM.PL Serialization, OODSS services can utilize these formats. For developing applications that integrate with third party software systems, the readability of messages is crucial in expediting processes of integration and debugging. Therefore, during de-

Java class definition

```
public class ClientAvatar
extends ElementState
{
 @simpl_scalar
 private String id;

 @simpl_composite
 private Vector2d pos;
}
```

S.IM.PL Serialization

```
<client_avatar id="vbush">
 <pos x="42.2" y="10.12" />
</client_avatar>
```

SOAP

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
   <performService xmlns="http://service">
    <clientAvatar>
     <ns1:id xmlns:ns1="http://entity">vbush</ns1:id>
     <ns2:pos xmlns:ns2="http://entity">
     <ns2:x>10.12</ns2:x>
     <ns2:y>42.42</ns2:y></ns2:pos>
   </clientAvatar>
   </performService>
  </soapenv:Body>
</soapenv:Envelope>
```

XML-RPC

```
<struct> <member>
  <name>id</name>
  <value>vbush</value></member>
 <member>
  <name>pos</name>
  <value><struct><member>
      <name>y</name>
      <value><double>42.42</double></value>
    </member>
    <member>
     <name>x</name>
     <value><double>10.12</double></value>
    </member>
   </struct></value>
</member></struct>
```

Fig. 26.: An instance of `ClientAvatar` as serialized through S.IM.PL Serialization, SOAP, and XML-RPC. Serialized representation through OODSS and S.IM.PL Serialization is significantly smaller in size as compared to other protocols.

velopment software engineers use more human-readable but verbose format such as XML. Later, when integration is complete and debugging is no longer required, support for multiple-formats allow software engineers to switch to a concise format, such as TLV to reduce bandwidth requirements.

B. Meta-Metadata Language and Architecture

Metadata is used to describe published information resources in a structured way. Kerne et al. conceived of information semantics as the integration of a document and its metadata, including data structures for representing metadata internally, rules for extracting instances of metadata types from particular published information sources, presentation to users, and operations supported by applications [29].

The meta-metadata language and architecture builds on S.IM.PL Serialization, comprising a type-system for describing wrappers that represent data derived from information sources. These wrappers are authored in metadata definition language and can be re-used to describe multiple information resources that publish data with the same structure.

Information sources are heterogeneous, as there are many types of metadata needed to describe real world entities ranging from scholarly articles to commercial products. By an information source, we mean a data source that publishes information with a consistent structure, such as an RSS feed, set of search engine results, or templated web pages. An information resource is an instance of a particular information source, typically associated with a specific URL.

Next, we demonstrate the use of S.IM.PL Serialization in developing the metadata type-system that forms the basis of metadata definition language for authoring metadata wrappers. Then, we examine the role of S.IM.PL Serialization in

facilitating meta-metadata compile-time and run-time modules. The compile-time module extends S.IM.PL Serialization's cross-language code generation facilities to derive language-specific metadata class definitions; instances of which function as typed-containers of data extracted from information resources. The run-time module extracts data from information resources to populate instances of metadata classes according to the rules specified by the developer and executes semantic actions that specify how information is acted-on by software tools and presented to users. Instances of meta-metadata wrappers and metadata associated with particular information resources are of course conveniently de/serializable using S.IM.PL Serialization.

1. Meta-Metadata Type System

Consider a `flickr.com` author-tagged photos web page as an information resource (Figure 27). It shows the collection of all photos uploaded by a user, associated with a particular tag. The Hypertext Markup Language (HTML) of the web page contains a list of hyper links of image thumbnails and image titles.

We want to extract metadata from information resources into abstract data type in multiple programming languages, such that collection representation applications can operate-on metadata and present information to the users. Therefore, information extraction can be considered as a sophisticated process of deserializing data, with certain rules. For example, we are interested in specific data within particular tags in HTML instead of the whole web page, which also contains presentation-specific data. A standard method of accessing particular information from within a specific tag in an XML document is to use XPath path expressions [30], which are a syntax for defining parts of XML document, enabling navigation and selection of nodes or sets of nodes in the document.

In some cases, such as web services, the data is provided for applications to

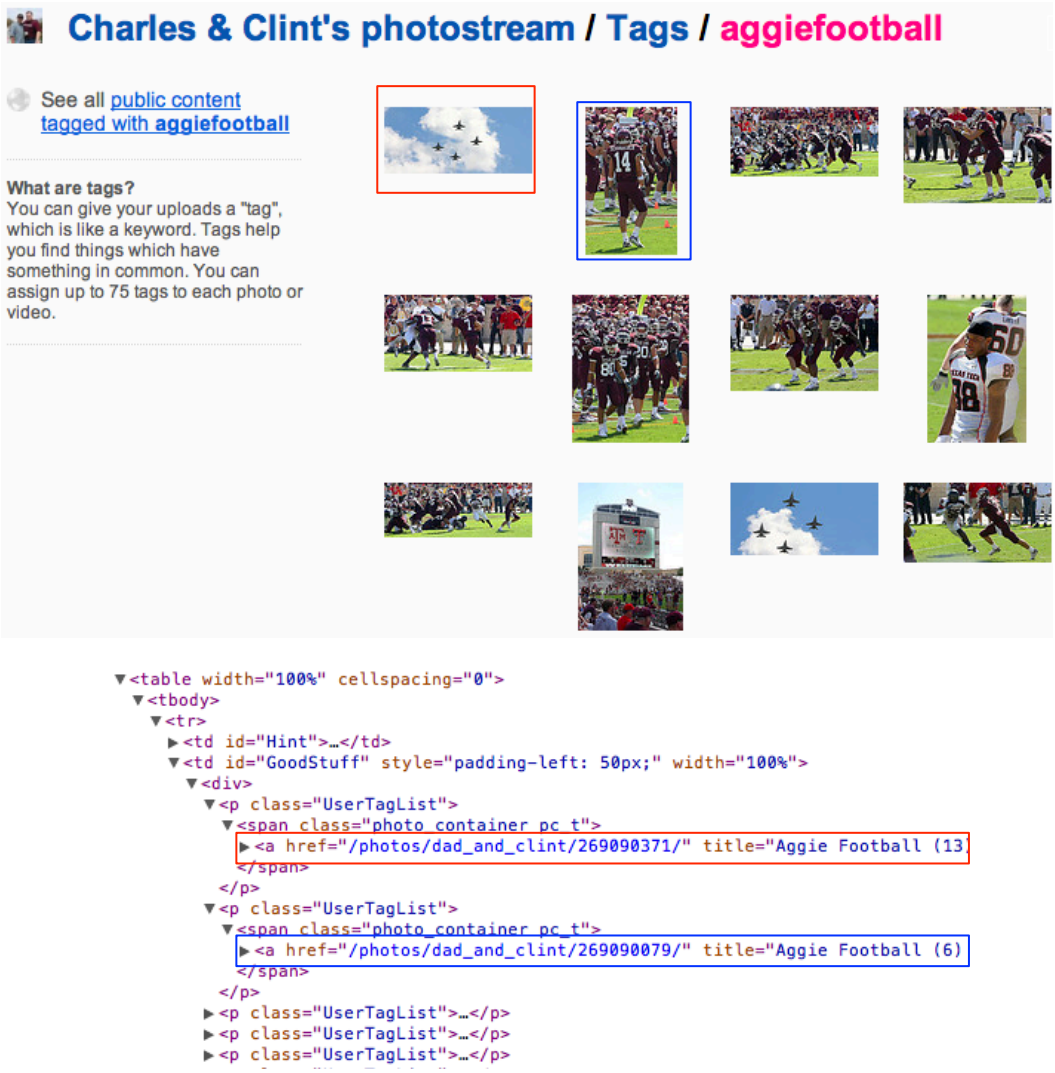

Fig. 27.: A `flickr.com` author tagged photos web page and its corresponding HTML. The web page contains a collection of thumbnails of the images uploaded by the user. Images are hyper-links that refer to more details of the particular image.

consume rather than for presentation. Such information sources provide data as formatted XML or JSON. We provide a mechanism called *direct binding* that directly deserializes data into metadata instances using S.IM.PL deserialization. Direct-binding is computationally less expensive in comparison to XPath information extraction, as it does not require creation of a Document Object Model (DOM) to support executing

XPath queries.

The meta-metadata type system provides a mechanism for describing the structure of metadata contained in information resources. For example, the structure of metadata in Figure 27 is a collection of links. Links are composed of scalar fields: image detail URL and image title. The image detail URL refers to another information resource that provides more specific information about the particular image. The ability to describe the structure of metadata enables software engineers to organize information sources as metadata types and promotes re-usable code in applications that work with information resources.

The structure of metadata in meta-metadata type system is represented by S.IM.PL basic types: scalar, composite, and collection. A composite type further contains scalars, composites, or collection types. Furthermore, a collection is either of type scalar or composite. These basic types are identical to the S.IM.PL types specified through type system scopes, which we presented earlier in Chapter II. The meta-metadata type system is an augmented super-set of type system scopes, providing a platform-independent, alternative syntax for specifying types in the S.IM.PL type system. The augmentation adds functionalities that are relevant to specifying metadata types, information extraction, presentation, and how software applications will operate-on metadata.

Declarations in the meta-metadata type system can be translated to *extended* type system scopes. The extended type system scopes are composed of `MetadataClassDescriptor`s and `MetadataFieldDescriptor`s that extend the `ClassDescriptor`s and `FieldDescriptor`s types to store additional information, derived from metadata-specific annotations. For example, the `@mm_name` annotation on a metadata field, binds it to its corresponding meta-metadata field declaration. The interpretation of the `@mm_name` annotation is stored in the

`MetadataFieldDescriptor` type, which is utilized by compile-time (VII, B, 3) and runtime (VII, B, 3) modules.

Declarations in the meta-metadata type system form the metadata type system, comprising of authored metadata types that describe information sources, and built-in types that function as base types for authored metadata types. Base classes, such as `metadata`, `document`, `compound_document`, `image`, `gps_location`, and `scholarly_article` are used to organize sets of polymorphic metadata subtypes.

Through meta-metadata type system, users have defined atleast 148 types in the metadata type system. These metadata types describe 85 different information sources, which includes scholarly articles, publications, search engine results, RSS feeds, and online photo albums, newspapers, and product stores.

2. Metadata Definition Language

The metadata type-system forms the basis of the metadata definition language, S.IM.PL Serialized language that software engineers use to write wrappers that specify types in the metadata type system. Figure 28 shows two wrappers for the `flickr.com` author-tagged photos web page. Utilizing the metadata type-system, the wrapper specifies the structure of metadata objects and how information is extracted through XPaths and regular expressions.

Information resources from a particular information source publishes information with consistent structure, a meta-metadata wrapper binds with an information source, specified by the `selector` element. In this case, we specify URL pattern for `flickr.com` author web pages. Multiple author web pages or information resources will utilize the same wrapper for information extraction. Meta-metadata types are extensible. Wrapper authors can re-use or extend existing wrappers. For example, the `flickr_link` wrapper is re-used in `flick_author` to describe a collection of links in

```
<meta_metadata name="flickr_link" extends="metadata">

 <scalar name="link" hide="true" scalar_type="ParsedURL" />
 <scalar name="title" navigates_to="link" scalar_type="String" />

</meta_metadata>
<meta_metadata name="flickr_author" extends="document" parser="xpath">
 <selector url_regex="http://www.flickr.com/photos/(?!tags)[A-z0-9_@-]+/$" />

 <semantic_actions>
  <get_field name="flickr_link_set" />
   <for_each collection="flickr_link_set" as="result">
   <get_field object="result" name="link" />
   <get_field object="result" name="title" />
  <parse_document now="true" >
    <arg value="title" name="anchor_text" />
    <arg value="link" name="container_link" />
  </parse_document>
 </for_each>
 </semantic_actions>

 <collection name="flickr_link_set" xpath="//div [@class=&#39;Photo&#39;]"
             child_type="flickr_link">
  <scalar name="link" xpath="./span/a/@href" />
  <scalar name="title" xpath="./span/a/@title">
    <filter regex="by \w*" replace="" />
  </scalar>
 </collection>

</meta_metadata>
```

Fig. 28.: The definition of meta-metadata wrapper for `flickr.com` author web pages. The `flickr_link` wrapper specifies the structure of links, while `flickr_author` wrapper specifies a collection of `flickr_link`s and semantics actions.

the document. Similarly, further types can be declared, which extends `flick_author` or re-use `flickr_author` type to describe another type of another information source.

In addition to specifying metadata structure, wrapper authors also specify semantic actions and presentation rules on metadata and metadata fields. Semantic actions allows software engineers to procedurally specify how collection representation applications will operate on metadata objects. Semantic actions are of three

kinds: variable declaration, control flow statements, and bridge functions. Variable declarations are made through `get_field`, `def_var`, and loop statements. They are used to pass arguments to bridge functions. Control flow statements specify the order of execution of semantic actions. These structures include loops and conditional statements. Bridge functions enable flow of metadata from meta-metadata wrapper into specific application functions.

For example, in the wrapper in Figure 28, the semantic action iterates over each `flickr_link` in the collection and invokes `parse_document` function with arguments, title and link. The `parse_document` is a special kind of semantic action used by web crawlers that uses the provided web-link to fetch additional documents for metadata extraction.

Wrappers authored in metadata definition language are, of course, deserialized using S.IM.PL. Constructs of metadata definition language are represented by programming language classes annotated with DBAL. The semantics of abstract data types augmented with DBAL provides a paradigm for structuring the meta-metadata definition language. Figure 29 shows the data binding of the meta-metadata wrapper with Java classes.

Polymorphism plays a crucial role in de/serializing meta-metadata wrappers. For example, a nested type, such as collection or composite can contain a list of scalars or more nested elements. Therefore, an abstract base class `MetadataNestedField` contains a polymorphic list (`kids`) of `MetaMetadataField` type; an abstract base class from which all meta-metadata types inherit. This enables nested structure in metadata definition language. The `@simpl_classes` annotation is used for databinding of the polymorphic list.

Similarly S.IM.PL de/serializatin is used to interpret selectors, as directives on the meta-metadata declarations. Further, the `MetaMetadata` type contains seman-

Fig. 29.: The data binding between a meta-metadata wrapper and DBAL-augmented class definitions in Java. S.IM.PL Serialization's support for polymorphic types enables the nested structure of metadata type declarations and semantic actions.

tic actions; represented by a polymorphic list of base class `SemanticAction`. The dynamic `@simpl_scope` annotation specifies the set of semantic actions that can be used. The `@simpl_scope` annotation facilitates addition of new semantic actions, as the metadata definition language is evolving. New semantic actions are added by extending `SemanticAction` base classes and added to the type system scope identified by the parameter (`SA_TYPES_SCOPE`) to `@simpl_scope` annotation. Therefore, less change in code is required to deserialize new semantic actions.

When deserializing a meta-metadata wrapper, S.IM.PL Serialization is also used to extend type system scopes to store additional information derived from meta-metadata specific annotations, such as the `@mm_dont_inherit` annotation, which is used for facilitating inheritance in metadata type declarations. A derived metadata type, automatically inherits all attributes from the parent metadata type. However, certain attributes are specific to the declared metadata type, which should not be inherited by derived types. For example, the `extends` attribute (declared in `flickr_author` wrapper in Figure 28) is specific to the declared metadata type. Derived types can override but not inherit the `extends` attribute.

3. Compile-time Module

Meta-metadata wrappers written by developers are stored in a repository on the users machine. The compile-time module generates metadata classes from meta-metadata wrappers as they specify the structure of metadata abstract data types.

Figure 30 shows the different components and use of S.IM.PL Serialization and type system scopes in generating metadata classes. The compile-time module iterates over each of the defined wrappers in the repository, deserializing it using S.IM.PL Serialization. Extended type system scopes consisting of `MetaMetadataFieldDescriptor`s and `MetaMetadataClassDescriptor`s encapsulate

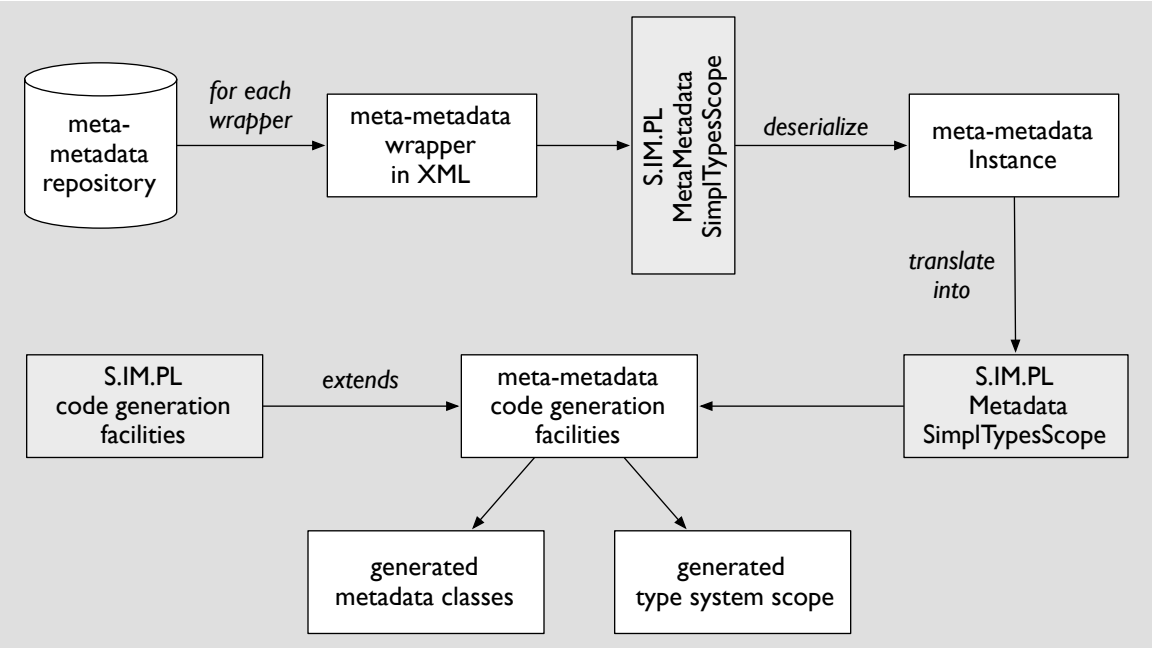

Fig. 30.: Different components involved in compiling a meta-metadata wrapper repository. S.IM.PL Serialization and type system scopes are used to deserialize wrappers and generate metadata classes.

additional information that facilitates creation of the correct meta-metadata instance. After deserialization, the meta-metadata definitions are translated to metadata type system scopes, which can be used to generate language-specific code through S.IM.PL Serialization code-generation facilities, which operate on type system scopes.

Since metadata type system scopes are an extension of S.IM.PL Serialization's type system scopes, which contain additional metadata-specific information, the compile-time module can extend S.IM.PL Serialization's code generation facilities to output additional metadata-specific annotations, such as the `@mm_name` annotation.

Figure 31 shows the generated metadata classes for the `flickr.com` author-tagged photos web page. The generated code is augmented with DBAL; therefore,

```
@simpl_inherit
public class FlickrAuthor
extends Document
{
 @simpl_collection("flickr_link")
 @simpl_tag("flickr_link_set")
 @mm_name("flickr_link_set")
 private ArrayList<FlickrLink>   flickrLinkSet;
 ..
}
```

```
@simpl_inherit
public class FlickrLink
extends Metadata
{
 @simpl_scalar
 private MetadataParsedURL link;

 @simpl_scalar
 private MetadataString    title;
}
```

Fig. 31.: Generated metadata classes for `flickr.com` author-tagged photos information source. `FlickrLink` is used as collection of links in `FlickrAuthor` class definition.

instances of metadata can be made persistent using S.IM.PL Serialization.

Generated metadata classes extend `Metadata`, built-in subtypes of `Metadata`, or other generated metadata classes. For example, in Figure 31 the `FlickrLink` class extends `Metadata` and `FlickrAuthor` extends `Document`, which is a sub-type of `Metadata`.

Collection representation applications that operate on different types of metadata use polymorphic fields and collections to store instances of metadata at runtime. De/serialization of such polymorphic fields and collections is handled automatically through type system scopes, definitions of which are also generated by meta-metadata's compile-time module.

4. Run-time Module

The runtime module populates metadata instances with information extracted from information resources. The information is populated either through XPaths and regular expressions or direct-binding. Figure 32 shows the flow of information from information resources into metadata objects and use of S.IM.PL Serialization and type system scopes.

In the case of direct binding, XML published through web services is automatically deserialized using S.IM.PL to instantiate metadata objects, as metadata class definitions contain DBAL declarations, which bind instances of metadata to serialized representations. In the case of population through information extraction rules, XPaths and regular expressions are used to populate particular fields in the metadata objects.

As shown in Figure 32, an application specifies a URL of the information resource from which it wants to extract metadata. Based on the mime-type and URL pattern, the runtime module creates an instance of the meta-metadata object, which was easily deserialized using S.IM.PL Serialization. The meta-metadata instance finds the associated `ClassDescriptor` in the generated type system scope, which was generated by the compile-time module and encapsulates all different type of metadata classes. The `ClassDescriptor` instantiates an empty instance of the metadata object.

The meta-metadata instance specifies the type of parser which should be used to populate the instance of metadata object. In case of direct-binding, the metadata instance is automatically populated through S.IM.PL Serialization. In case of information extraction rules, the framework utilizes `MetadataFieldDescriptor`s to gain access to each field. An XPath expression specifies the node in the document where the information is located; regular expressions specifies how that information

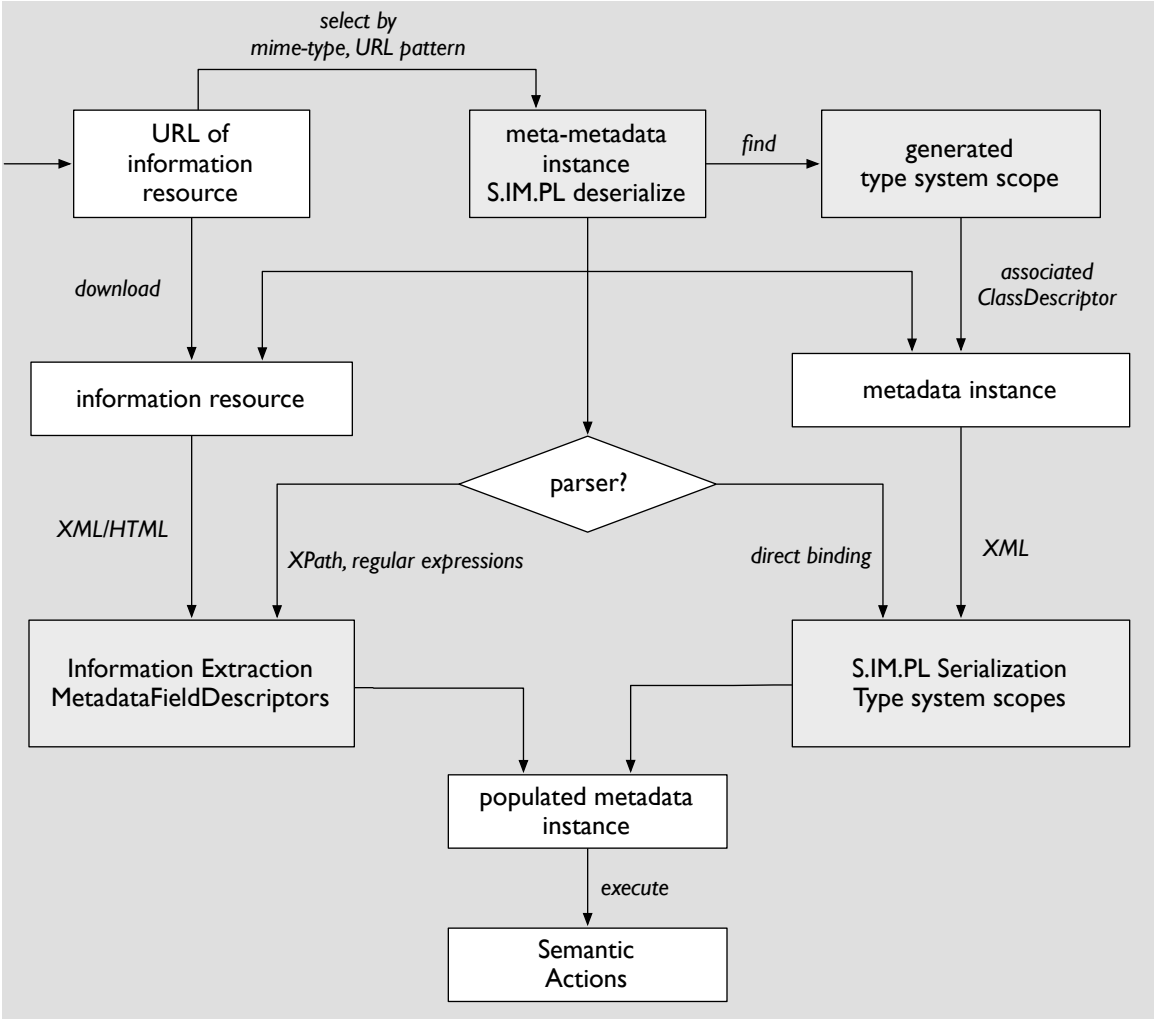

Fig. 32.: Runtime module: shows how metadata objects are populated from an information resource. S.IM.PL Serialization is used for deserializing meta-metadata wrappers, creating metadata instances, direct binding, and information extraction.

is filtered or formatted, such that it can correctly be assigned to the metadata field. Once the information is extracted as a string value, a `MetadataFieldDescriptors` provides functionality to marshall the scalar into the correct type

Populating scalar fields is managed through extension of S.IM.PL Serialization's scalar type system. The framework adds additional scalar types, such as

`MetadataString`, `MetadataFloat`, `MetadataInteger`, and other scalar types. This extension enables the framework to treat scalar fields as composite objects, such that additional information, like term vectors [31], is stored with scalar fields, which are used by software applications, but not required for de/serialization.

Finally, when a metadata instance is populated, the runtime executes semantic actions, which call bridge functions that enable scripting flows of control for specifying use of metadata in software applications, in case they need to perform further operations on the received metadata.

C. Preferences Management System

We developed an extensible Preferences Management System based on S.IM.PL Serialization. It introduces a type-system that enables users to specify different types of visual and functional configurations of softwares through an XML file. Later, the software application gains quick and easy access to these configuration settings. The framework is used in software applications, which we will examine in Chapter VIII.

A user specifies configuration settings through a *pref* XML file, as shown in Figure 33. The XML file binds to class definitions augmented with DBAL declarations. A generic container object `PrefSet`; maps preferences by their unique name. Preferences can be of different data types such as boolean, integer, float, string, or any other scalar or composite type. The `preferences` HashMap is declared with `@simpl_scope` annotation; therefore, it functions as a polymorphic collection that can contain preferences of any type abstracted by the type system scope. The identifier of which is specified through the argument `PREF_TYPE_SCOPE`.

The key feature of this system is typed-access to configuration settings, instead of a software application accessing string values and converting them to appropriate

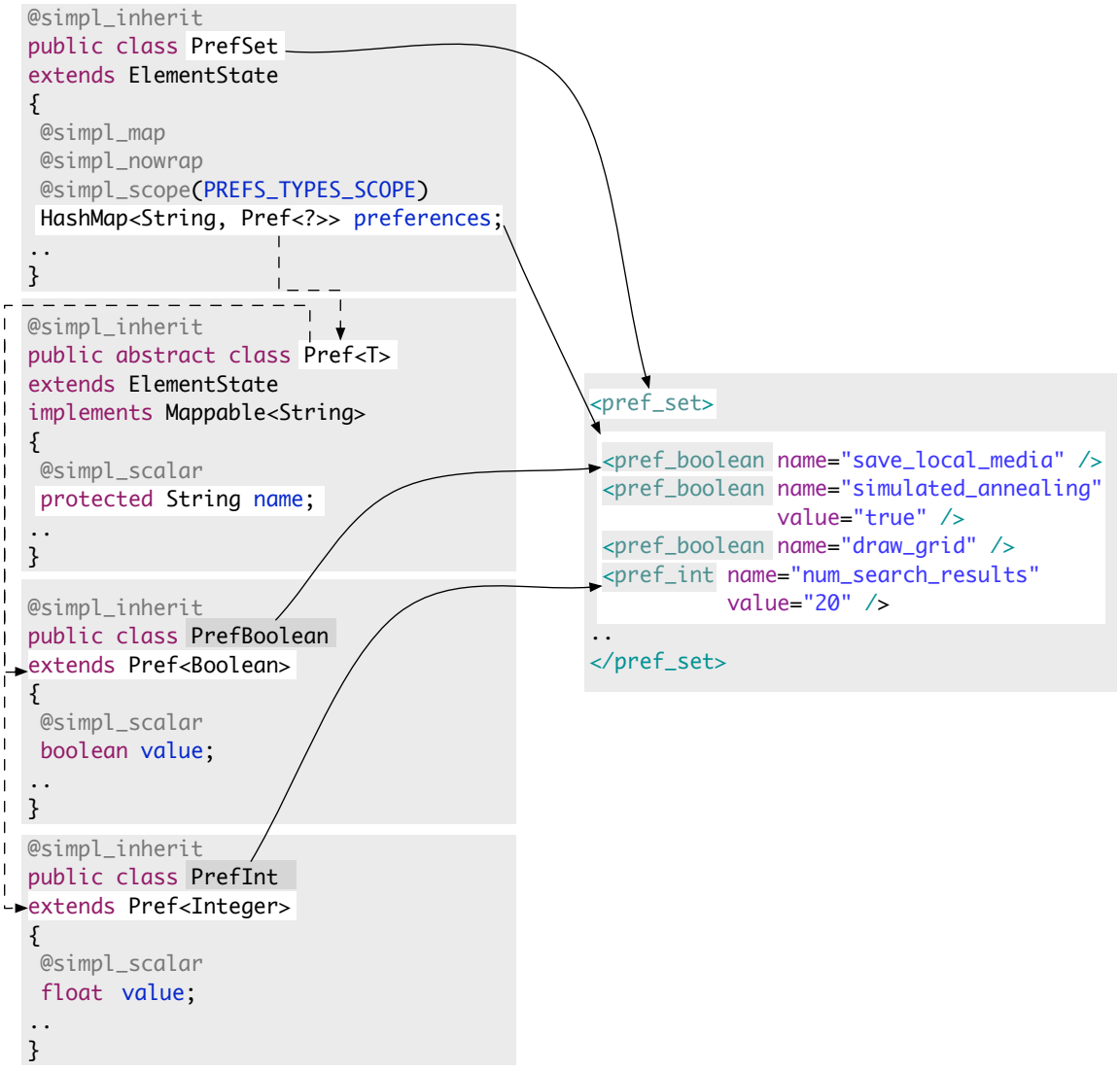

Fig. 33.: The data binding between a configuration XML file and Java classes. The `PrefSet` class contains a polymorphic map of preferences. The `@simpl_scope` annotation specifies the types of preferences, such as integer and boolean.

types. The type conversion is automatically handled by type system scopes through class-tag mappings with `ClassDescriptors`.

The use of `@simpl_scope` annotation enables developers to easily add new preference types to the type system scope and have the system automatically de/serialize

them to correct types. For simplicity, we used scalars in the example Figure 33. However, users can also add composite and collection objects as value for a preference.

1. Meta-Preferences System

When developing applications for users, the software requires a graphical user interface from where users can modify values of preferences by themselves without breaking the system through incorrect specifications. Maintaining the interface for preferences management can be difficult, as preferences are constantly added, removed, or modified based on the current version of system and features it supports.

To address this issue, we developed Meta-Preferences System. It extends Preferences Management System and based on S.IM.PL Serialization. The Meta-Preferences System enables software engineers to specify from a *meta-pref* XML file, the interface and error-handling related configuration of a particular type of preference. For example, an integer preference type has a corresponding integer meta-preference type that specifies additional information to the integer preference, such as the interface component that will modify this preference and the range of values the specified integer can contain.

The Meta-Preferences System reduces the burden on software engineers as interface for managing configuration is specified through an XML file, which is easier to maintain, in comparison to modifying the source code, which may result in errors.

Similar to Preferences Management System, the Meta-Preferences System utilizes S.IM.PL Serialization's support for data binding polymorphic instances. Analogous to `PrefSet`, a `MetaPrefSet` class encapsulates a generic collection of different types of `MetaPref`s that can be used. Therefore, software engineers can easily add new sub-types of `MetaPref`s in the system.

D. Conclusion

In this chapter, we examined software frameworks: Object-Oriented Distributed Semantics Services (OODSS), meta-metadata language and architecture, and Preferences Management System, to validate S.IM.PL Serialization's design and motivate its features. We demonstrated how S.IM.PL Serialization is used to support their architecture through its extensible design and flexible data binding. These frameworks are used in research software applications and used by students in undergraduate and graduate courses and Texas A&M University, Department of Computer Science and Engineering.

OODSS is a software framework designed to support real-time network communication and remote method invocation in distributed software applications. S.IM.PL Serialization is at the core of de/serialization of OODSS messages in an architecture that hinges on polymorphism. Remote methods are dynamically dispatched based on the sub-type of each message received. S.IM.PL Serialization's support for data binding polymorphic types facilitates dynamic dispatching through instantiation of the correct message sub-type on the remote machine. Type system scopes function as objects of mutual understanding between server and client applications, as they encapsulate types of messages exchanged. This, provides an automatic error-handling mechanism for unknown messages. Flexible data binding enables OODSS services to produce concise messages as compared to other commonly used technologies of representing information for remote method invocation, such as SOAP and XML-RPC. Multi-format and cross-language support in S.IM.PL Serialization enables OODSS services to function across platforms in multiple-formats.

The meta-metadata language and architecture comprise a framework that enables software engineers to consistently describe the structure of metadata, how it is

represented internally as program objects, acted-on by software tools, and presented to the users. The framework validates S.IM.PL Serialization's extensible design and motivate the value of its features. It extends type system scopes to encapsulate additional information related to metadata and meta-metadata objects. The extension of type system scopes facilitates code generation and information extraction. Data binding of polymorphic fields is extensively utilized, which facilitates re-usability and maintenance of the framework's source code. As of this writing, the framework describes 85 information sources, which extract information into 148 different types of metadata objects.

The Preferences Management System enables software engineers to gain quick and easy to configuration settings of a software application through external XML files. External XML files enables persistence of configuration settings on a user machine for later access. S.IM.PL Serialization's support for data binding polymorphic objects enables the Preferences Management System to deserialize configuration settings into efficient generic data structures in the runtime memory, enabling software applications to quickly access, modify, and store a particular value of a preference.

CHAPTER VIII

VALIDATION - RESEARCH APPLICATIONS

In this chapter, we examine S.IM.PL Serialization's use in two research software applications: the Team Coordination (TeC) game and combinFormation. TeC is a distributed software application, designed to teach and enhance team coordination skills through an engaging game experience. The TeC game has been played by university students, fire fighting students, and other emergency responders. combinFormation is a creativity support tool that connects searching, browsing, organizing, modelling, and visualizing information. combinFormation is used by university students and researchers to generate creative ideas.

By examining how S.IM.PL Serialization is used in various components of TeC and combinFormation, we validate the capability of S.IM.PL Serialization in supporting real-world software applications that require robustness, modularity, flexibility in supporting continuous enhancements, and ease in debugging and deployment. It also validates features of S.IM.PL Serialization as applicable to accommodate requirements of software engineers. As of this writing, atleast 2,100 users have used and benefited from TeC and combinFormation. S.IM.PL Serialization has performed without errors.

In the next section, we examine different components of combinFormation. S.IM.PL Serialization is used as part of meta-metadata language and architecture (VII, B) and independently for configuration and persistence of user-created compositions. Then, we examine different components of TeC. S.IM.PL Serialization is used as part of OODSS (VII, A) for network communication and independently for managing configuration settings, and recording and replaying game sessions. Finally, we examine TeC's cross-language implementation use-case that utilizes

cross-language code translation and data binding facilities. We examine how S.IM.PL Serialization facilitated migration of TeC client application from Java to Objective-C.

A. combinFormation

combinFormation is a creativity support tool that connects searching, browsing, organizing, modelling, and visualizing information [32][33][34]. combinFormation uses the integrative visual representation of information composition to represent collections, instead of lists or grids of separate elements. The composition (Figure 34) is formed using image and text clippings, derived from clippings from documents, to represent important ideas from the documents. combinFormation is integrated into the curriculum of the undergraduate coursework ENDS - 101 The Design Process and used by students in graduate course CSCE - 655 Human-Centered Computing. As of this writing, approximately 2,000 students have used combinFormation. Studies have shown that users benefit from combinFormation in generating creative ideas for course assignments.

Figure 35 shows stages of data flow in combinFormation, which use of S.IM.PL Serialization. A combinFormation user specifies a set of information resources and initial set of queries for the focused web crawler to find relevant web pages. The user then launches combinFormation with interface and application configurations loaded from XML files. Later, the web crawler extracts documents and image/text clippings through meta-metadata language and architecture, and ranks them based on the user's interest model to form an information composition. Finally, an information composition can be saved by the user as an XML file for later retrieval.

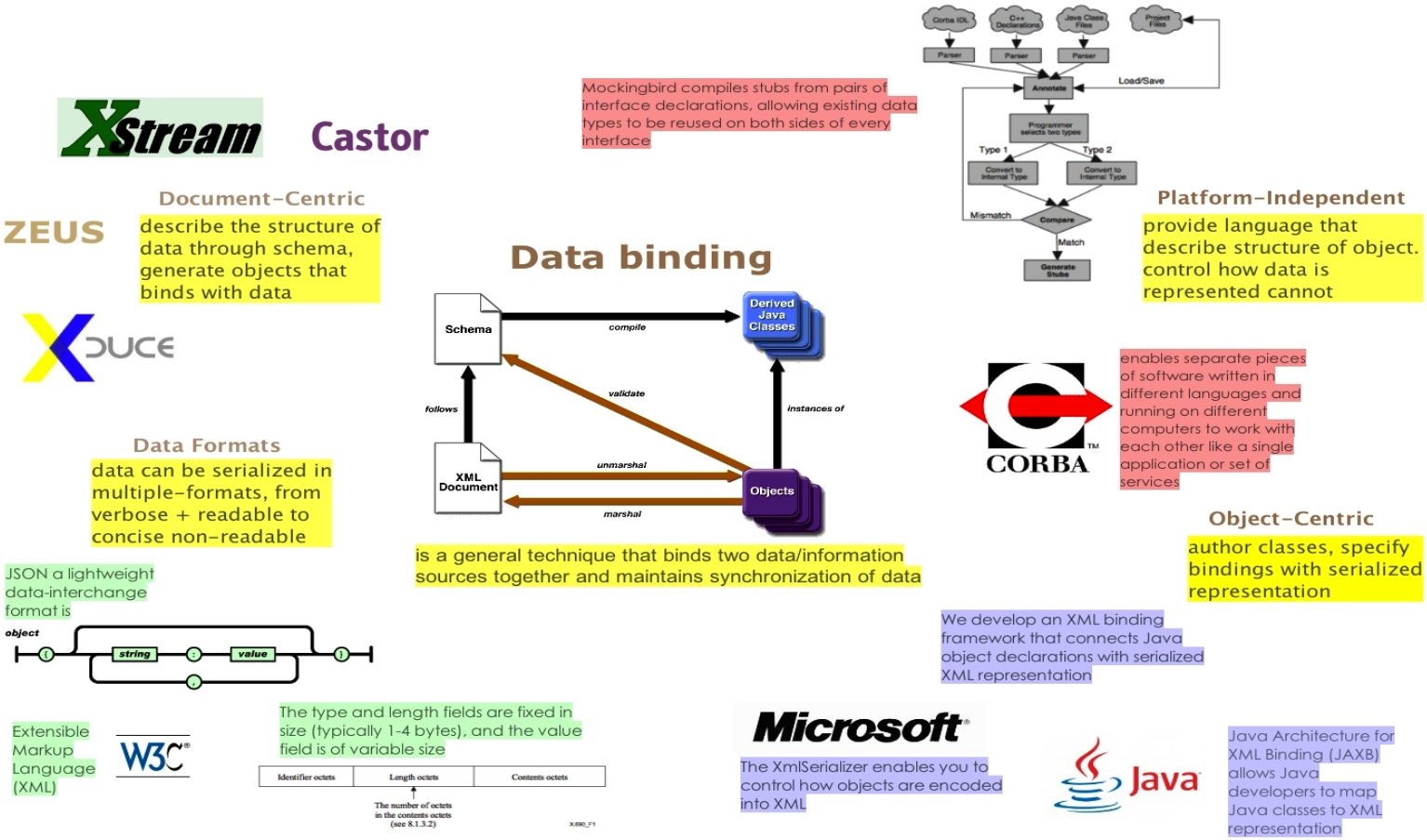

Fig. 34.: An information composition made from combinFormation, which presents software technologies and research related to S.IM.PL Serialization, in a form that provokes thinking. Image and text clippings are extracted from web resources and scholarly articles.

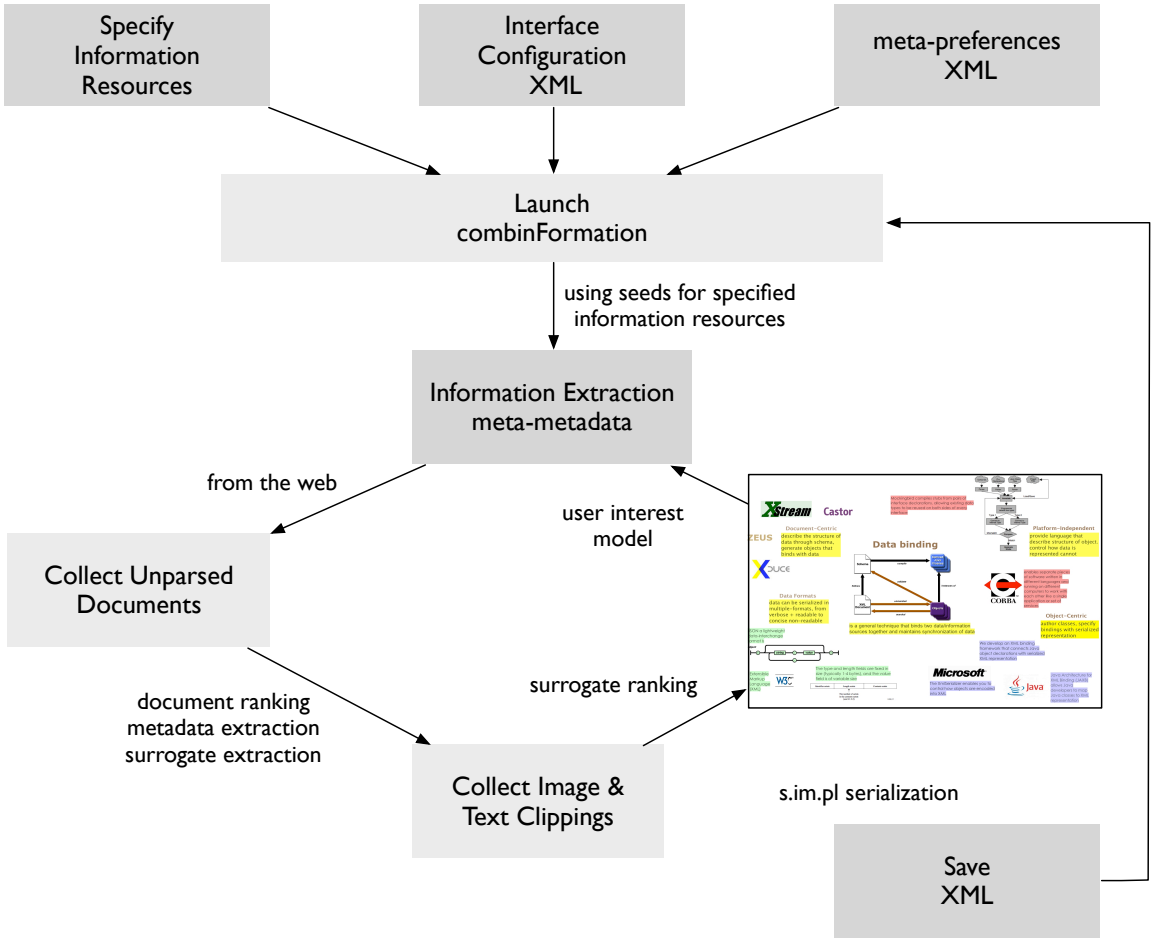

Fig. 35.: The data flow in combinFormation. S.IM.PL Serialization is used for configuring interface, specifying information sources, extracting information through meta-metadata, and persisting an information composition

1. Launch Configuration and User Settings

The combinFormation interface is managed through external XML files, which facilitates easy deployment and changes to the combinFormation software. For interface rendering, S.IM.PL Serialization is used to connect interface configuration XML files with typed-Java objects. The instances of Java objects, which contain interface settings, are used to render the Graphical User Interface (GUI). Through S.IM.PL

Serialization, the combinFormation GUI is configurable by external XML files, which avoids recompilation of the source code when an interface component is modified. Thus, facilitating deployment of the software in production, as only an XML file is replaced on the server. A total of 16 DBAL-augmented Java classes are used for encapsulating interface configuration settings, leveraging S.IM.PL Serialization's support for scalar, composite, collection, and polymorphic types.

Configuration settings for some visual and functional components of combinFormation are user-specific. They are handled by the Preferences Management System (VII, C), which internally uses S.IM.PL Serialization for management and persistence of preferences, providing quick and easy access to the combinFormation software. The meta-preferences (VII, C, 1) system is used to render an interface from where a user can modify preferences.

2. Information Extraction from the Web

In combinFormation, a focused web crawler uses the queries specified by the user to crawl and collect documents from the Internet. Meta-metadata, utilizing S.IM.PL Serialization, is used to extract document metadata and clippings that contain metadata from documents. Meta-metadata supports multiple information sources, such as Flickr, Wikipedia, IMDb, ACM Digital Library, RSS, and the Google and Bing search engines, enabling combinFormation to extract and present image and text clippings from these information sources.

The collected image/text clippings are presented to users, which they can manipulate by changing its size, location, font, and color. As the user hovers over a clipping, she can view, edit, or navigate to the source web document from in-context details on demand and tools (Figure 36). A user can also express interest in each clipping and specific terms inside a surrogate metadata, which changes the user's in-

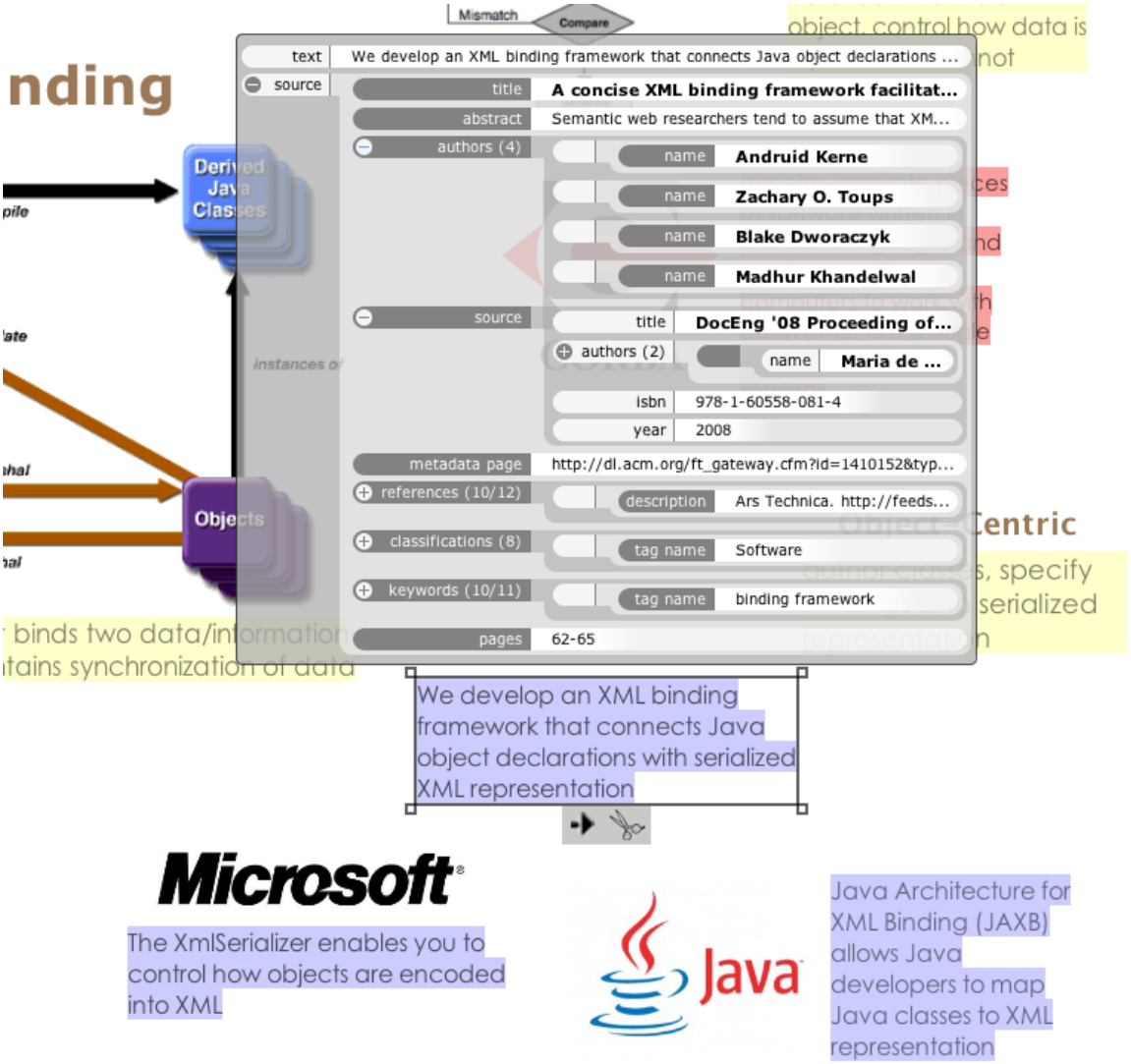

Fig. 36.: As a user hovers over a clipping, its nested metadata is displayed through in-context details on demand tool. A user can modify metadata, express interest in a specific term, or navigate to the actual resource.

terest model. The web crawler fetches more documents based on the user's interest model and presents additional clippings to the user. The crawler constantly fetches additional documents and clippings until the user stops further collection. Users once satisfied can save the information composition for later retrieval.

New information sources can be easily added in meta-metadata and seamlessly integrated in combinFormation through polymorphism. The `Document` class is a base type of all different kinds of metadata objects supported by meta-metadata. A declared field of type `Document` functions as a polymorphic type, which can be of any type of metadata specific to an information source. The information composition in Figure 36 is created from clippings from 13 different information sources. For the composition, meta-metadata facilitated metadata extraction from 136 information resources.

3. Saving an Information Composition

Image and text clippings, extracted metadata, and relationships between clippings and container documents is represented by DBAL-augmented Java classes. Therefore, an information composition can be easily persisted in the form of an XML document on the user's machine for later retrieval, enabling users to re-open a saved information composition to further modify/add/review information. S.IM.PL Serialization facilitates representation of relationships between different types of clippings and container documents from heterogeneous information sources as XML. Additional information specific to an information composition, such as visual and spatial attributes of clippings and information composition, are also serialized and persisted through S.IM.PL Serialization.

Figure 37 shows a serialized representation of an information composition and DBAL-augmented class definitions that encapsulate document and clipping metadata. The `InformationComposition` class encapsulates the structural and referential information about clippings and source document metadata, annotations, and media and composition elements.

Clippings can be of image or text types. The `ImageClipping` and `TextClipping`

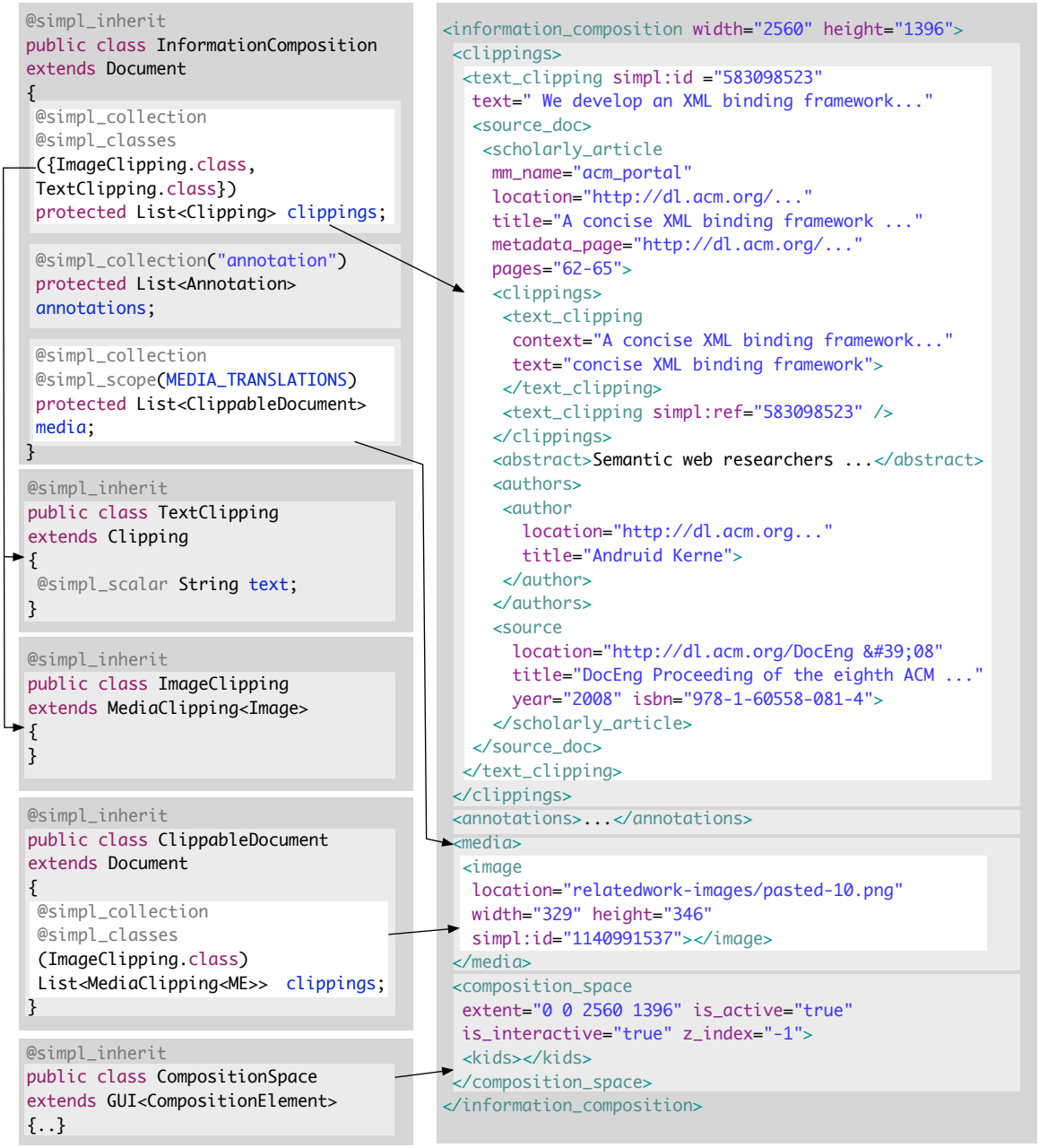

Fig. 37.: The mappings between Java classes and a serialized representation of an information composition. The `@simpl_scope` and `@simpl_classes` annotations are used to represent polymorphic types.

classes represent these different types. They extend the `Clipping` base class, which refers the source document metadata through a polymorphic composite type `Document`. The source document metadata types are dynamically specified through the `@simpl_scope` annotation, which contains the identifier of a generated type system scope from meta-metadata. The generated type system scope encapsulates all the different types of metadata documents supported by the meta-metadata language and architecture. Therefore, S.IM.PL Serialization's architecture of data binding polymorphic types enables combinFormation to easily persist diverse types of document metadata. Similarly, different types (image or text) of clippings are represented through a polymorphic list of `clippings` in `InformationComposition` class. Other elements, specified by users, such as text annotations, media elements, and visual attributes of text clippings are represented by S.IM.PL objects.

Graph serialization is utilized to encapsulate referential information between clippings and source documents. For example, a clipping references its source document for and particular document maintains references to all extracted clippings. In earlier versions of combinFormation, when graph serialization was not supported, the saved XML files were more verbose and the source code was less maintainable, as developers used alternate mechanisms to represent cyclic references.

As of now, an information composition is saved as XML, but support for multiple formats has enabled combinFormation developers to focus on future work related to exporting a composition in other formats to utilize third party tools. For example, support for BibTeX is planned for integration in combinFormation, which will enable users to export a BibTeX database, using this format's ability to compile bibliographies through third party tools.

B. Team Coordination Game

The Team Coordination (TeC) game is a multi-player game that focuses on teaching and enhancing team coordination skills through embodied interaction. The TeC project investigates real-life team coordination practices among fire emergency responders and integrates findings into game design. The focus of TeC is on human-centered aspects of distributed cognition and team coordination. TeC is played by university students, fire fighting students, and other emergency responders with the goal of improving team coordination skills. Studies have shown that the game is successful in this mission [35] [36]. As of this writing, 174 game sessions with 99 different users have been played. The Disaster Preparedness and Response (DPR) unit of the Texas Engineering Extension is in the process of deploying the game into multiple courses in the curriculum that it offers to the international community of emergency responders. DPR played an important role in responses to 9/11, Hurricane Katrina, Hurricane Rita, Hurricane Ike, and many other prominent disasters.

TeC is a multi-player game. Several players join a centralized game server. Players acquire roles in the game, through which they communicate and share different pieces of information to succeed in the game. TeC client applications, which are used by players, communicate with the game server to share, transfer, and update information about the game world. The communication, which is in the form of XML messages, is managed by OODSS (VIII, A), internally using S.IM.PL Serialization for de/serialization of messages. In all game sessions, S.IM.PL Serialization has performed without errors.

In the next section, we examine the message communication architecture of TeC and report on the complexity and frequency of de/serialization of game messages, handled correctly by S.IM.PL Serialization. Then, we further examine data types in

TeC that compose game messages and utilize various data binding features. Finally, we examine a logging and playback system that researchers use for replaying recorded games for measuring impact of game sessions on player skills.

1. Message Communication Architecture

In TeC, players join the game server from client applications. They are assigned the roles of *seeker* or *coordinator*. In a typical game setting, 4 players join the game server. One players takes the role of the coordinator, while the rest take the role of the seeker. Seekers see the local view of the game world as they move around to collect *goals* and avoid *threats*. They are guided by the coordinator, who sees the global view of the game world.

Seekers' actions in the game world change the state of the game, which is updated on the server. The server updates clients with new information at every game cycle. The communication between client and server applications is managed by OODSS. A game session is typically played for 15 minutes. It can end early if seekers are able to collect all 12 goals before the allotted time expires. Equation 8.1 calculates the quantity of messages de/serialized in a TeC game session; an approximation, assuming a game is played for complete its duration and all goals are collected. 9 different types of messages are used for communication, each of which, contains specific information and transmitted depending upon the state of the game.

$$\begin{aligned} \text{messages / game} &= 4 \text{ players} \times 900 \text{ seconds} \\ &\quad \times 30 \text{ messages / player / second} \\ &\quad + 1,456 \text{ conditional messages} \\ &= 109{,}456 \text{ XML messages (9 types)} \end{aligned} \tag{8.1}$$

Messages in TeC, play a crucial role in driving the game mechanics and changing the state of the TeC client. Figure 38 shows the communication architecture with types of messages exchanged between client and server application. A shared type system scope encapsulates these different kinds of messages. The client authenticates with the server by sending a *LoginWithRole* request message. If credentials are verified, the server replies with the *InitializeGame* message, which contains game configuration data, such as map, time, number of goals, number of threats, and number of seekers. Later, the client sends a *SetReadiness* message to notify that the player is ready to start the game. Finally, as the game starts, the client repeatedly sends the *UpdateClientAvatar* message to update a seeker's location on the server. The server replies with a *RespondWithGameState* message that updates the location of other seekers as wells a threats, on the map. *UpdateClientAvatar* and *RespondWithGameState* messages are repeatedly sent every 30 milliseconds to keep the game world synchronized across client applications. When seekers are collecting goal, a *CollectingGoal* message is sent to the server.

Table VIII shows the size, depth, and an approximate number of messages de/serialized during a single game session for each client. As of this writing, 174 TeC game sessions have been played by 99 users, which includes university students, fire fighting students, and other emergency responders. S.IM.PL Serialization performed without errors and performance issues.

2. Data Binding TeC Entities

TeC is a distributed application with client applications communicating with a centralized game server. Data exchanged is in the form of entities which are serialized

---

Conditional messages are sent based on certain conditions within the game, such as when a seeker is logging-in, confirming ready state, or collecting goal.

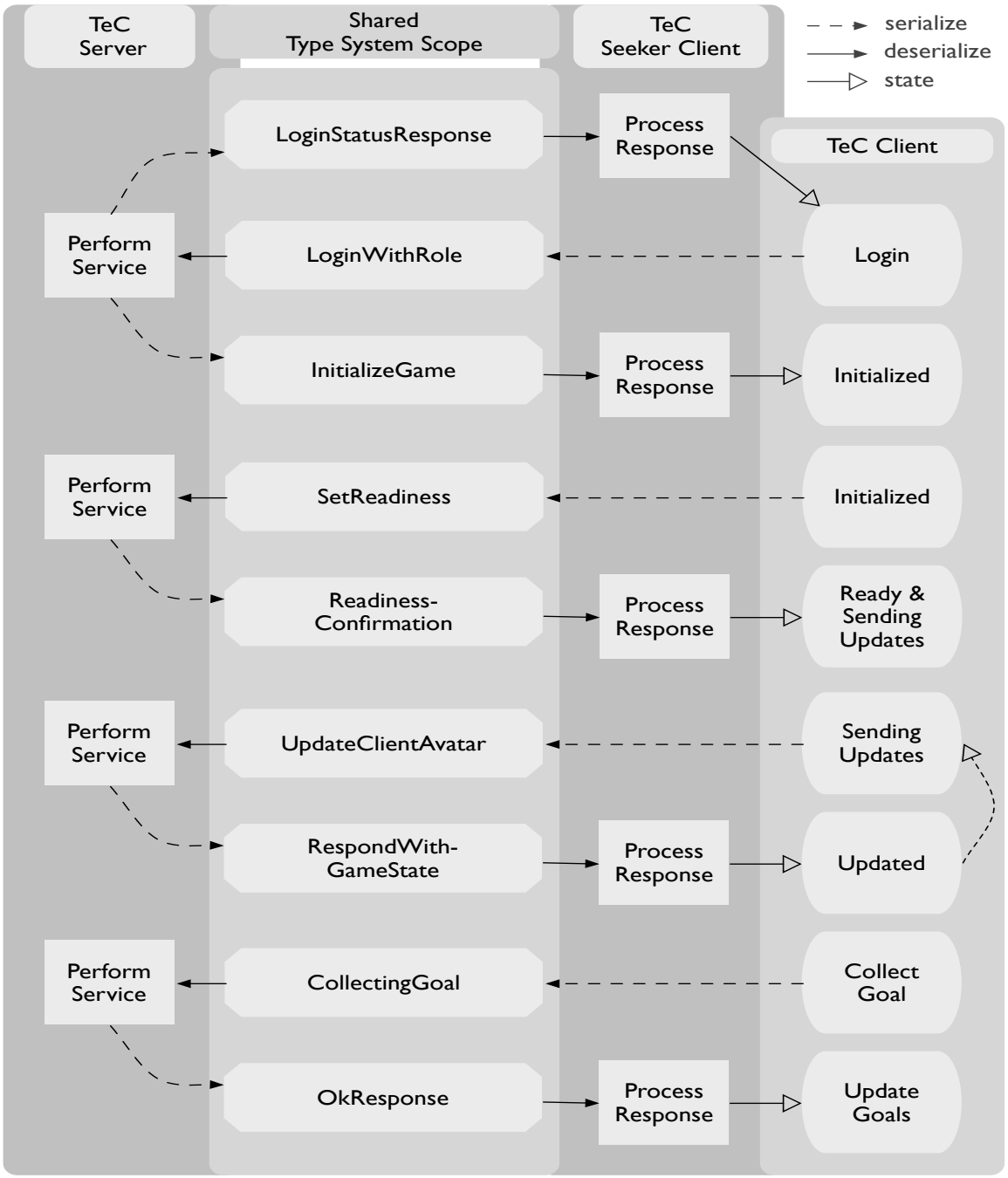

Fig. 38.: The communication architecture of TeC, utilizing OODSS. A shared type system scope encapsulates 9 different types of messages that modify client state.

Table VIII.: Types of TeC messages and their size, depth, and count. Size is calculated in bytes, which denotes the amount of data contained in the message. Depth is the level of nesting when serialized. Count is an approximate measure of the number of times the messages are sent in a single game session.

Messages	Size / Bytes	Depth	Count
LoginWithRole	125	2	1
LoginStatusResponse	125	2	1
InitializeGame	3,668	6	1
SetReadiness	60	1	1
ReadinessConfirmation	2,040	1	1
UpdateClientAvatar	227	3	13,500
RespondWithGameState	1,168	5	13,500
CollectingGoal	51	1	180
OkResponse	45	1	180

and transmitted for use by the remote application. These entities are complex data structures represented by deeply nested composites, collections, and polymorphic objects. They also function as objects that drive game mechanics. Flexibility to bind only the required fields for de/serialization in entities is crucial, as entity objects contain local application specific logic and state variables. Using S.IM.PL Serialization, software engineers can easily omit de/serialization of fields that are not required by the remote application. This improves performance and reduces message size.

Although, TeC's communication architecture involves exchange of more types of messages, we will examine the *InitializeGame*, *RespondWithGameState*, and *Collect-*

ingGoal messages, as these messages are crucial in the functioning of the game. All different types of entities are encapsulated by these messages.

Entities in TeC, can be categorized as *static* or *dynamic*. Static game entities do not change their runtime state as the game progresses. For example the game map and sub-entities that constitute a map are static entities. The states of dynamic game dynamic game entities are constantly in flux as a game progresses. For example, as a seeker moves around the map to collect goals, its location on the map changes. Static game entities are transmitted once, while dynamic game entities are transmitted every game cycle to update their current state across all applications.

a. Static Game Data

Static game entities are encapsulated in the `StaticGameData` class. An instance is transmitted through the *InitializeGame* message. Values in `StaticGameData` are populated through an external configuration file on the server. The Preferences Management System (VII, C) is used for access to these configurations. Initial locations of seekers and goals is also considered as static game data. The game map is loaded through an external XML file; the URL is sent as part of static game data. Figure 39 shows the data binding for the `InitializeGame` class, with its XML representation. Polymorphic collections are used to implement generic containers for different sub-types of seekers and goals. For example, goals can be collaborative and non-collaborative. Collaborative goals require more than one seeker to collect them, while non-collaborative goals require only a single seeker to collect. Collaborative goals implement different game mechanics as compared to non-collaborative goals, therefore, they are represented as different sub-types of common base class `Goal`.

The game map is stored as an external XML file. The XML contains information specific to the map, such as locations of walls and safe zones, gps and wifi signal

Fig. 39.: Data binding of an `InitializeGame` message class with its XML representation. Seekers and goals are entities that can of different sub-types. they are represented by polymorphic collections in `StaticGameData`.

strength areas, seeker spawn points, and the size of the map. Using S.IM.PL Serialization, the XML is deserialized into instances of static game entities, which are used by the server and client applications to drive game mechanics, such as collision detection with walls. Figure 40 shows the data binding of map XML with Java classes. The map is represented by a hierarchy of class definitions. The game maintains separate versions of the software that use different sub-types of maps. For example, gps and wifi signal strength areas are entities used for simulating the effect of signal strength on the game, when the game is played without these sensors. When the game is played with gps and wifi sensors, these entities are not required, hence a sub-type

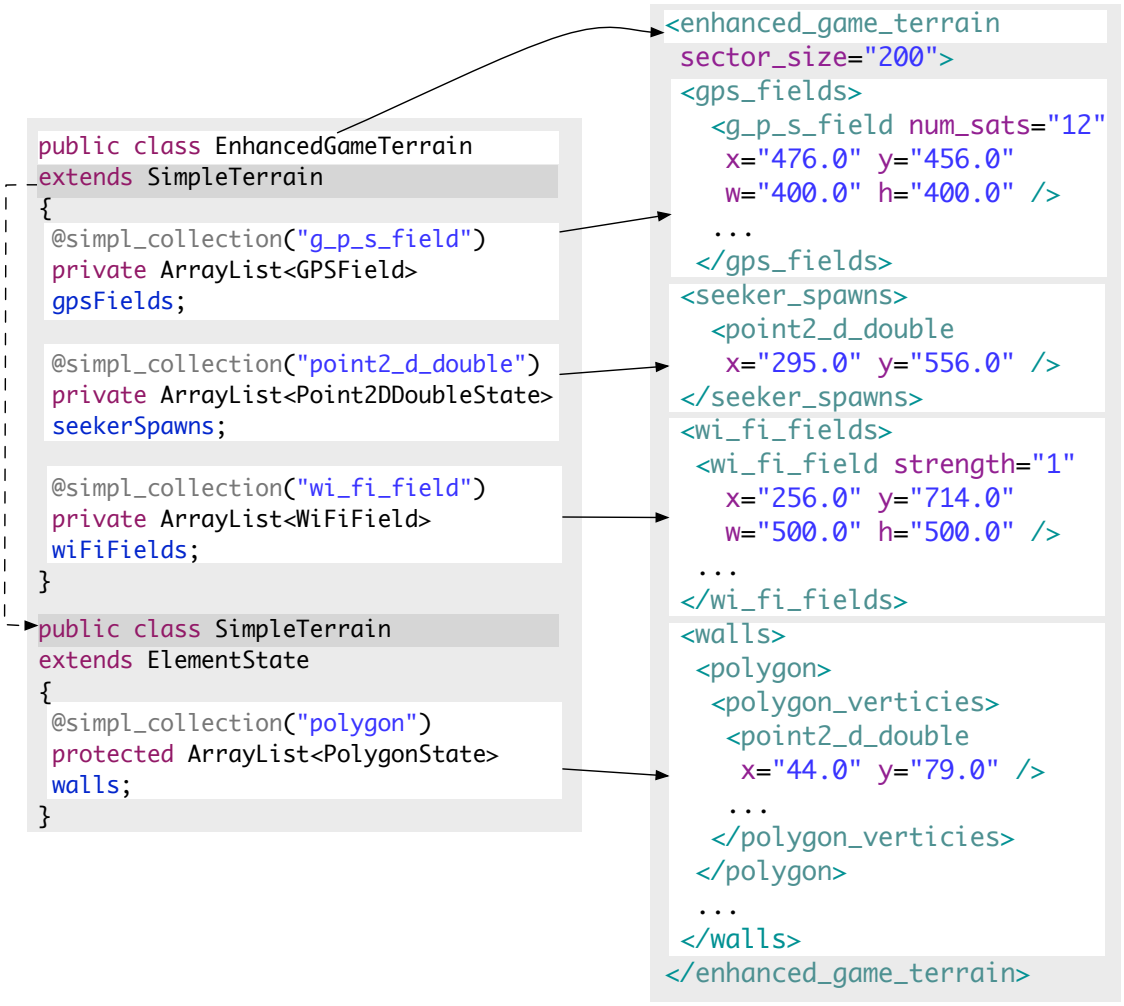

Fig. 40.: Data binding of an `EnhancedGameTerrain` map class with its XML representation. It extends `SimpleTerrain`; a base class for different types of maps.

other than `EnhancedGameTerrain` is used.

b. Dynamic Game Data

Dynamic entities, such as seekers, threats, and goals are updated on clients through *RespondWithGameSate* and *CollectingGoal* messages. The *RespondWithGameState* message updates the location of seekers and threats on TeC client. Figure 41 shows the data binding of Java classes with *RespondWithGameState* message. The `GameData`

class contains collections of seekers and threats. As introduced earlier in example walkthrough (II, A), threats can be of different types depending on their behaviour in the game. They derive from the common base class `Threat`. To represent such scenarios, software engineers usually declare a single polymorphic collection, instead of declaring separate collections for each sub-type, which are difficult to maintain. S.IM.PL Serialization's support for data binding polymorphic instances enable software engineers to adhere to object-oriented principals of software design that facilitates code maintenance and re-usability.

As mentioned earlier, goals in TeC can be of different sub-types: collaborative

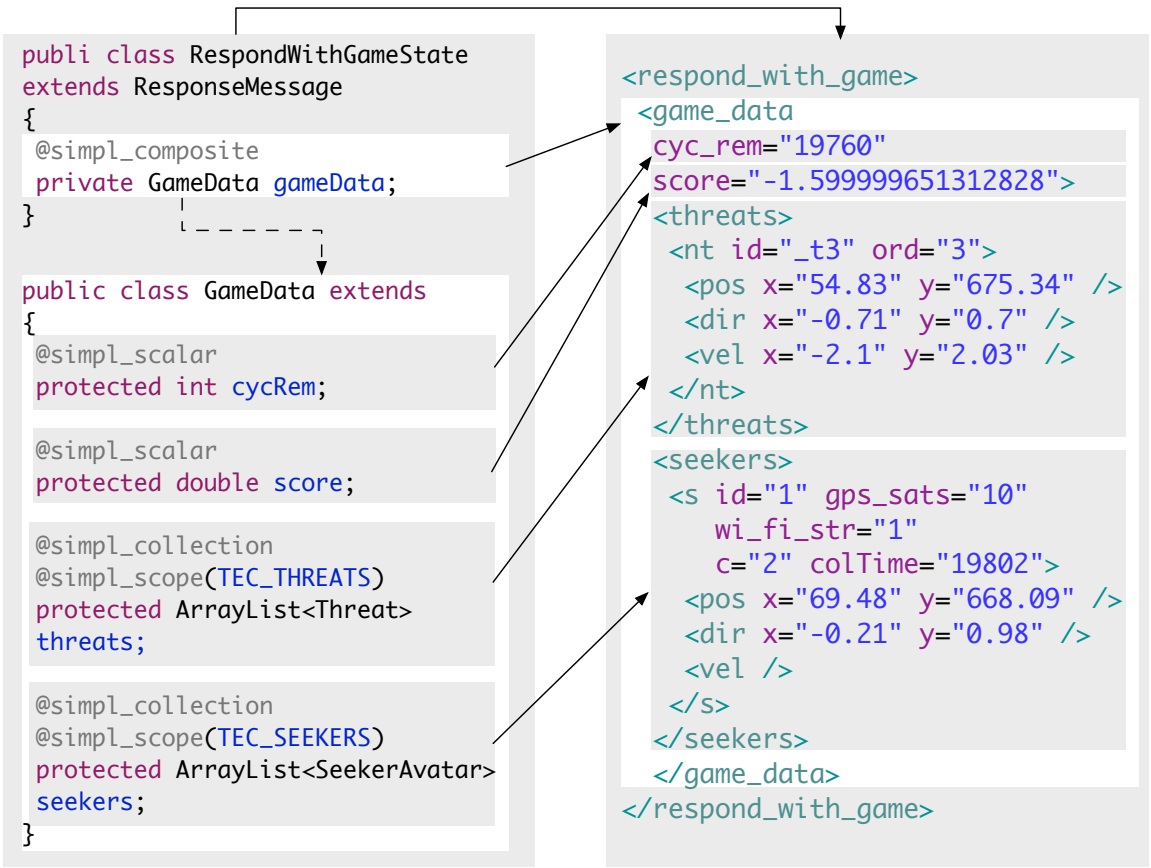

Fig. 41.: Data binding of an `EnhancedGameTerrain` map class with its XML representation. It extends `SimpleTerrain`; a base class for different types of maps.

and non-collaborative, deriving from a common base class `Goal`. Collaborative goals require the presence of more than one seeker at the goal's location to be collected. To synchronize goal collection, the goal entity maintains a map of goal collectors; seekers attempting to collect goal. The map is updated through the *CollectingGoal* message sent to the server. The server computes the requirements for goal collection; if satisfied, goal collection begins. Seekers must remain at goal location to complete the collection process. Usage of this map structure is necessary for synchronizing goal collection events in the game. S.IM.PL Serialization's support for binding the map data structure facilitates software engineers in handling such cases.

3. Recording and Replaying Game Sessions

During TeC user study sessions, researchers are interested in finding and collecting data based on how users interact and play the game. It is essential for researchers to record and playback a game session for later analysis, data collection, and reporting of findings. The TeC game session is recorded through S.IM.PL Serialization. The game server logs game states in real-time, which are later used by the Coordinated Log and Playback System (CLAPS) [37] to replay the game for analysis. All 174 game sessions that have been played by users are recorded through S.IM.PL Serialization and made available for researchers to analyse. CLAPS synchronizes game replay with audio data of players' communication, providing visual components that enables researchers to examine how communication between players affects their actions in the game. S.IM.PL Serialization is used to deserialize game states, making instances available for CLAPS to render game states. The rendering code is same as that originally written for TeC.

Flexible constructs in S.IM.PL Serialization played a crucial role in enabling CLAPS software to work with old XML files, generated by previous versions of the

TeC game. For example, serialized log files in earlier versions of TeC were verbose with lengthy tag names. Developers modified `@simpl_tag` declarations in class definitions with shorter tag names. The produced log files from the later version were concise. However, to be able to replay games from old XML files, developers used `@simpl_other_tags` annotation to specify the old tag for deserialization. Hence, less code modification was required for backward compatibility.

C. Cross-Language Use Case: TeC iPhone Client

We presented the application of S.IM.PL Serialization in the Team Coordination Game (TeC) (VIII, B), to validate the ability of the framework to support real time distributed applications. We also examined TeC's message communication architecture to present details of message complexity and frequency of de/serialization requests handled by the framework. In this case study, we examine cross-language implementation of TeC's client application to present how S.IM.PL Serialization facilitated software engineering principles of code reuse, maintenance, documentation, and porting existing software to other platforms and programming languages.

In the next section we describe the motivation behind the TeC project to port the existing Java-based client application to Objective-C programming language utilizing the Apple iOS platform. During this process, we ported 130 Java class definitions to Objective-C through S.IM.PL Serialization's cross-language code translation utilities. The generated code produced no compile-time or runtime errors in de/serialization of messages across platforms, reducing development, testing, and debugging time. Later, we briefly refer back to Section B, 2 to examine the complexity of entity representation in the TeC game, utilizing deeply nested composite, polymorphic and collection objects, thus validating S.IM.PL Serialization's cross-language capabilities

to handle complex data structures.

We examine how the flexibility to bind only the required fields for serialization facilitated TeC client's development in Objective-C. We also discuss issues in cross-language code generation such as, differences in keywords, memory size of data types, and integration of generated code.

1. Why Migrate TeC Client?

Migrating an application from one platform to another is a common scenario, as advancements in hardware technologies and programming languages offer new ways to build interactive applications. The TeC game's client and server applications were originally developed in Java. Java is a platform independent programming language and offers high-level programming language features that software developers benefit from during development. However, with the introduction and increased popularity of Apple iOS platform and iPhones, we have seen migration of many existing applications to run on iPhones. As of now, the Apple iOS platform does not support the Java virtual machine, therefore, code-bases for such applications must be ported to Objective-C programming language for iOS development. Similarly for TeC, iPhones offers new ways the game can be played and made available to the users.

TeC is a location-aware game that uses GPS and compass sensors to determine a player's location and heading, which is used to control their avatars in the virtual game world. iPhones provide a mobile platform and built-in sensors that can be easily utilized. Therefore, we wanted to port the mobile part of the TeC game, which is the seeker client application, to Objective-C.

2. Cross-Language Implementation

S.IM.PL Serialization facilitated migration of existing TeC client's Java source code to Objective-C through cross-language code translation utilities (VI, B). During this process, 130 Java class definitions were translated to Objective-C. The generated code did not produce compile-time errors. The Java code included embedded documentation, which was also ported into Objective-C, facilitating development in Objective-C. Since Objective-C does not support annotations, data bindings are specified through serialized type system scope XML file. The generated code also includes comments that described how a particular field is annotated in Java, facilitating development in Objective-C. Later, runtime cross-language de/serialization of messages functioned without errors and performance issues.

In Section B, 2, we examined the complexity of messages and briefly described the structure of entities in TeC. Declarations of entities are structured in the form of deeply nested class hierarchies, that utilizes inheritance to maximize code reuse. Polymorphism is extensively used for implementing different behaviours of similar entities in the game. A fairly large number of classes are used to encapsulate runtime behaviours of entities. Instances of entities are de/serializable through S.IM.PL Serialization. Therefore, in addition to driving the game mechanics, entities are also used for transmitting information to the remote application. Writing these classes in Objective-C by hand is a tedious and time consuming task. Later, debugging due to human-errors in hand coding can also increase the burden on software engineers and increase development time.

S.IM.PL Serialization's cross-language utilities resolved the issue of time required in hand-coding message specific classes and debugging runtime errors in de/serialization. The generated code was documented, functioned without errors, and produced

Fig. 42.: The TeC seeker client running on iPhone. Entities shown, such as seeker, threat, goal, and map are represented by S.IM.PL-objects in program.

a skeleton project that developers were able to easily modify and add application and platform-specific logic, such as graphic handlers. Figure 42 shows a screen shot of TeC seeker client running on iPhone, utilizing the message communication architecture presented earlier in Section B, 1.

Few issues were encountered during cross-language code translation and data binding, which helped us improve the cross-language support in S.IM.PL Serialization. For example, certain variable names, such as *in* and *id* were used in Java TeC client. Therefore, the generated code in Objective-C also contained variables with names *in* and *id*, which are keywords in Objective-C and cannot be used as variables names. We were able to resolve this issue by maintaining a map of keywords in the code

translation utility and notifying the user by generating a warning if variable names in Java conflict with keywords in Objective-C. A user can easily resolve this issue by refactoring the variable name to a value that does not conflict with keywords in Objective-C.

D. Conclusion

In this chapter, we presented applications of S.IM.PL Serialization in research software, combinFormation and Team Coordination (TeC) game, to validate the ability of the framework to support application development. As of this writing, approximately 2,100 users, which include students, researchers, and emergency responders, have used TeC and combinFormation. To support real-world software applications, S.IM.PL Serialization must be robust and flexible, and facilitate maintenance, debugging, deployment, and documentation. We examined how these requirements are satisfied in different components of these software applications.

combinFormation (VIII, A) is a creativity support tool, that presents information as image and text clippings to the users to form an information composition. These clippings are extracted from documents from the Internet, using the meta-metadata language and architecture (VII, B), which builds on S.IM.PL Serialization. Later, an information composition can be saved as an XML file using S.IM.PL Serialization. Configuration of combinFormation software is also managed through external XML files using the Preferences Management System (VII, C), which is based on S.IM.PL Serialization. S.IM.PL Serialization has functioned without errors in these software components. Extensive use of data binding support for polymorphic instances has facilitated software engineers in maintenance and re-usability of the source code. The ability to seamlessly access and manipulate data from external XML files have facili-

tated configuration management and deployment of the combinFormation software.

The Team Coordination (TeC) (VIII, B) game is a multi-player, distributed software application. Network communication during the game is performed in real-time using Object-Oriented Distributed Semantics Services (OODSS) (VII, A), which is based on S.IM.PL Serialization. S.IM.PL Serialization is also used for recording game sessions for later analysis. As of this writing, 99 users have played and benefited from TeC. Their game sessions have been recorded and analysed by researchers. Since, the communication is performed in real-time, performance of de/serialization is crucial. We have not experienced runtime errors and performance issues with S.IM.PL Serialization's use in TeC. Extensive use of data binding support for polymorphic instances has facilitated maintenance and re-usability of the source code.

Migration of software applications from one platform to another is a common real-world problem, as development in hardware and software technologies leads to newer ways a software application can be used and made available to the users. We examined a cross-language use-case of S.IM.PL Serialization in TeC game's client application, which involved migration of an existing Java-based client application to Objective-C programming language. S.IM.PL Serialization facilitated the process by generating code in Objective-C for 130 class definitions. The generated code produced no compile-time and runtime errors in de/serializing messages from Java-server. Software engineers were able to easily modify and add application-specific code to the generated class files, which facilitated development in Objective-C.

CHAPTER IX

PERFORMANCE BENEFITS AND DEVELOPER EXPERIENCE

Performance and usability are key components for the real-life application of a software framework. In Chapter VIII, we presented application of S.IM.PL Serialization in such real-life software systems whose users include students, researchers, and emergency responders. S.IM.PL Serialization performed well for this. To formally validate performance benefits of S.IM.PL Serialization we conducted benchmark tests to compare S.IM.PL Serialization's runtime performance of de/serialization with other XML data binding frameworks: JiBX, JAXB, Castor, and XStream. Our results showed that S.IM.PL Serialization outperformed these frameworks in nearly all de/serialization scenarios that included small (1.42KB), medium (123KB), and large (1,003 KB) size XML files.

S.IM.PL Serialization's prime goal is to relieve software engineers from the burdens associated with developing information-centric applications. Therefore, usability by software engineers is crucial validating goal for S.IM.PL Serialization. We conducted a small-scale study with semi-structured interviews of 10 student developers that used S.IM.PL Serialization in undergraduate and graduate coursework at Texas A&M University, Department of Computer Science and Engineering. Students used S.IM.PL Serialization in completing course assignments. Feedback from them indicated that S.IM.PL Serialization eased the development process, facilitated code re-usability, and documentation. Students also identified useful features and shortcomings in S.IM.PL Serialization that helped us improve our framework through refinement/addition of data binding features and improvement in documentation and error handling.

In the next section, we present benchmark results and analyse performance benefits of S.IM.PL Serialization. Later, we summarize data obtained from feedback of student developers and analyse two notable software applications developed by students: a sketch recognition application and a multi-modal rummy game.

A. Performance Evaluation

We compared the performance of S.IM.PL Serialization with the best XML data binding frameworks. Performance was measured using the Bindmark [38] benchmark. We ran serialization and deserialization benchmarks on an Intel Core i7 2.8 GHz processor with 8 GB 1067 Hz RAM. Often, the relative performance of XML frameworks differs greatly depending on the size of the XML representation. Therefore, we included varying sizes of XML files in our benchmark: small (1.42 KB), medium (123 KB), and large (1,003 KB) size XML files. The depth of XML data was kept constant at 2, 20, and 45 for small, medium, and large files respectively.

The benchmark results compared performance of S.IM.PL Serialization with JiBX, JAXB, XStream, and Castor. These frameworks are commonly used XML data binding frameworks for commercial and non-commercial applications. We ran multiple iterations of de/serialization across different XML file sizes: 1,000 small, 100 medium, and 10 large XML files. The choice for the number of iterations was arbitrary, but kept constant for a particular size across all frameworks. The data contained in XML files was also kept constant across all frameworks. The benchmark of S.IM.PL Serialization is unbiased, containing no code specific to Bindmark. Results showed that S.IM.PL Serialization outperformed other frameworks.

Table IX.: Serialization time required using different serialization. S.IM.PL Serialization outperforms other frameworks in all scenarios.

framework /	serialization time (nanoseconds)			runtime
(# runs)	small (1000)	medium (100)	large (10)	size (KB)
S.IM.PL	18,091	432,094	2,327,055	176
JiBX	18,400	460,726	2,663,215	141
JAXB	97,648	3,021,568	20,423,433	3,800
XStream	132,825	6,451,624	35,615,057	368
Castor	363,927	3,488,719	17,101,195	3,000
XML file size (KB)	1.42	123	1,003	
XML file lines	55	1,817	8,567	
XML file depth	2	20	45	

1. Serialization Benchmarks

Serialization in XML data binding frameworks is the process of generating XML representation of an object in memory. Table IX shows the benchmark results for serializing objects into small, medium, and large size XML files. Results indicate that S.IM.PL Serialization's runtime performance is better than all the frameworks used in Bindmark.

S.IM.PL Serialization is approximately 9 times faster than JAXB; a Java supported XML data binding framework. XStream and Castor are further down below in the performance chart. S.IM.PL Serialization is 15 times faster than XStream and 8 times faster than Castor. JiBX is closest in performance with S.IM.PL Serialization.

JiBX uses an offline compile-step, which is excluded by Bindmark. The compile-step augments Java byte-code by adding de/serialization methods to compiled classes.

Table X.: Deserialization time required using different serialization frameworks. S.IM.PL Serialization outperforms other frameworks in all scenarios, except small XML files compared to JiBX.

framework /	deserialization time (nanoseconds)			runtime
(# runs)	small (1000)	medium (100)	large (10)	size (KB)
S.IM.PL	202,671	1,852,291	11,659,359	176
JiBX	77,510	2,277,680	17,276,832	141
JAXB	216,060	3,290,398	18,903,100	3,800
XStream	260,395	6,506,639	40,732,718	368
Castor	766,221	6,112,564	35,433,526	3,000
XML file size (KB)	1.42	123	1,003	
XML file lines	55	1,817	8,567	
XML file depth	2	20	45	

2. Deserialization Benchmarks

Deserialization in XML data binding frameworks is the process of creating objects in memory from XML representation. Table X shows the benchmark results for deserializing small, medium, and large size XML files. Results indicate that S.IM.PL Serialization's runtime performance is better than all the frameworks used in Bindmark, except for small size files in comparison to JiBX.

S.IM.PL Serialization is approximately twice as fast as JAXB and approximately 3 times faster than XStream and Castor. Again, JiBX is closest in performance to S.IM.PL Serialization.

JiBX uses an offline compile-step, which is excluded by Bindmark. The compile-step augments Java byte-code by adding de/serialization methods to compiled classes.

3. Discussion on Benchmark Reports

When a framework needs to get data into and out of the runtime objects, a common way of doing this is through reflection. Reflection is a powerful language feature that enables runtime access to an object's encapsulated data, methods, and type information. However, reflection suffers from a performance disadvantage when compared to calling a method and accessing fields directly from the source code.

Through close examination of XML data binding frameworks used in benchmark results, we found that all other frameworks directly use reflection operations, except JiBX. JiBX takes a different approach: Java byte-code augmentation from an offline compile-step, which adds de/serialization code in compiled classes. Since, a part of processing required at runtime is moved to the compile time, JiBX yields better runtime performance than JAXB, XStream, and Castor. However, the compile-step is difficult to manage as it is handled by the programmer, using a separate utility software. Changes in source code requires that programmers re-run the compile-step.

Table XI.: De/serialization time normalized with the smallest value for each size of XML document.

	Serialization (normalize time)			Deserialization (normalized time)		
Framework	Small	Medium	Large	Small	Medium	Large
S.IM.PL	1	1	1	2.62	1	1
JiBX	1	1.07	1.15	1	1.23	1.48
JAXB	5.40	6.99	8.78	2.79	1.78	1.62
XStream	7.34	14.93	15.31	3.36	3.51	3.49
Castor	20.1	8.07	7.35	9.74	3.3	3.04

S.IM.PL Serialization also utilizes reflection features of the programming language, but performs better than JiBX and significantly better than rest of the frameworks Table XI. From the benchmark results, we draw the conclusion that the performance benefits of S.IM.PL Serialization are achieved through type system scopes. Type system scopes utilize tag-based mappings, which provide quick access to constituent instances of `ClassDescriptor`s and `FieldDescriptor`s that cache reflection accessor objects and data binding semantics, resulting in better runtime performance. However, further tests and benchmarks reports are required to formally verify and measure the impact of type system scopes on de/serialization performance.

Thus, in conclusion, the performance of JiBX is close to but inferior to that of S.IM.PL Serialization. Further, JiBX requires offline compilation and performs bytecode modification. It is thus harder to use. Further, JiBX is XML only, lacking cross-language and multi-format support. Developers are better served by using S.IM.PL Serialization.

B. Development Experience

We wanted to validate the usability of S.IM.PL Serialization by software engineers and verify that features supported by S.IM.PL Serialization are effectively used and sufficient in different scenarios of software development. We conducted semi-structured interviews with 10 students developers that used S.IM.PL Serialization. Initial data indicated that S.IM.PL Serialization is easy to understand and integrate with software applications that require de/serialization of runtime objects. Students also reported that S.IM.PL Serialization eased the development process, reduced development and testing time, and facilitated documentation of the source code. Students were able to spent more time concentrating on broader goals of their projects rather than writ-

ing tedious parsing code. We intend to conduct further studies with more student and expert developers to draw substantial conclusions about the usability of S.IM.PL Serialization.

Table XII.: Interview groups of student developers. Each group developed a research grade software application using S.IM.PL Serilialization independently or as part of other research frameworks, such as OODSS and meta-metadata.

identifier	# students	year
B1	2	2009
B2	1	2009
B3	1	2010
B4	2	2010
C1	2	2010
C2	2	2010

S.IM.PL Serialization was introduced to students at the start of the semester in undergraduate course CSCE-482 Capstone Design and graduate course CSCE-455 Human Centered Computing. Students were encouraged but not required to use S.IM.PL Serialization for assignments. At the end of the semester, we conducted semi-structured interviews with student developers who were actively involved in using S.IM.PL Serialization (Table XII). We mainly asked students about their experience and impact of S.IM.PL Serialization on their assignments.

1. Human Centered Computing

Students in the graduate computer science course, CSCE-455 Human Centered Computing, develop software applications that helps them understand principles of writing

usable softwares. At the end of the semester students are required to form groups of upto five students to complete a research project. They develop variety of distributed applications using location sensors and mobile devices and later, conduct user studies to validate the design, usability, and contribution of their software applications. Below we summarize qualitative data from semi-structured interviews with student developers and report issues they faced in using S.IM.PL Serialization.

Students in group [B1] liked the ability to seamlessly de/serialize data from XML. They reported that generated XML was human-readable, interoperable and suggested that they would use S.IM.PL Serialization again for future projects. [B2] started development with their own serialization implementation. However, they later switched to S.IM.PL Serialization as they thought it was more convenient to use. [B3] liked the ability of data binding with other sources and reported that S.IM.PL Serialization was more expressive than C#'s XmlSerializer. [B4] reported that S.IM.PL Serialization was easy to use and they could focus on broader goals of their projects rather than writing tedious code for parsing and de/serialization.

Students in groups [B1] and [B2] reported problems in understanding the library and use of Data Binding Annotation Language (DBAL) (II, D). [B1], [B2], and [B4] reported that they had trouble in debugging erroneous use of DBAL constructs. We later, developed an online guide for S.IM.PL Serialization with examples and tutorials [12]. We also implemented error handling and reporting code in S.IM.PL Serialization that helps developers in debugging. [B3] reported a special case of cyclic references in his class definitions, which was not supported by S.IM.PL Serialization. We implemented graph serialization (III, E) to resolve this shortcoming. Later, [B3] used S.IM.PL Serialization and OODSS to develop a distributed application for his research in Sketch Recognition (IX, C, 1).

2. Senior Capstone Design

CSCE-482 Capstone Design is an undergraduate course in which students develop variety of novel and interactive software systems. Students form groups of upto six students to complete a research project in phases during the course of the semester. They use mobile devices, location sensors, and multi-touch displays to develop interactive software systems. Below we summarize qualitative data obtained through semi-structured interviews of student developers.

[C1] developed a distributed application for composing music using a Java-based server and iPhones. iPhone users select music notes, which they send across to the server application. A music composition is created from contributions by iPhone users. Three users simultaneously connect with the server to play the game in real-time. S.IM.PL Serialization and OODSS were used for de/serialization and network communication. Students in group [C1] reported that they did not find any problems with S.IM.PL Serialization. They reported that online tutorials were sufficient for understanding the basic use of S.IM.PL Serialization. Students were able to directly use the example code from the guide, after making few changes.

Students in group [C2] developed a Multi-modal Rummy Game (IX, C, 2) using S.IM.PL Serialization and OODSS. The game was developed in Objective-C using Apple iOS platform on iPhones and iPad. They reported a minor bug of memory leak during deserialization of reference type objects, which was fixed during development. They also reported online documentation was helpful in understanding S.IM.PL Serialization and its supported features were sufficient in implementing their application. [C2] wanted to serialize enumerated types, which are not supported by S.IM.PL Serialization in Objective-C. An alternate mechanism was suggested, but we intend to implement support for enumerated types.

C. Applications Developed by Students

We demonstrate the use of S.IM.PL Serialization in two notable software applications developed by student developers. A Sketch Recognition application was developed by a graduate student, which is used by undergraduate students to submit hand-drawn sketches and have them automatically graded by the system. Multi-modal Rummy Game was developed by a team of undergraduate students that teaches old people to use mobile devices and technology through an engaging card game.

The Sketch Recognition application was used by Fall 2010 and Spring 2011 undergraduate students. It is publicly deployed for use by future students. The Multi-modal Rummy Game was developed in Spring 2010. Since then, the game is under active development by researchers and it is in the process of approval of publishing rights at Apple's App Store.

1. Sketch Recognition Application

A researcher [R1] from group [B3] used S.IM.PL Serialization and OODSS for his research application on sketch recognition. Sketch Recognition is a sub domain of Human Computer Interaction (HCI). Research in sketch recognition develops algorithms and software programs based on data from hand drawn sketches. The sketch data is usually represented as an XML document containing location and timestamps of points. Points are further categorized into strokes, such that a stroke can contain one or more points.

The application was developed for undergraduate students to submit assignments using a computer to draw diagrams and have them automatically graded by the system. S.IM.PL Serialization is used to serialize sketch data into XML files for transport and storage on the server. In sketch application, a stroke can belong to

multiple shapes. A change in data of a stroke must reflect a change in each of the containing shape objects, thus resulting in backward edges in the object model. [R1] used graph serialization feature to correctly represent such object models.

In Fall 2010 and Spring 2011, 33 and 20 students respectively used the system and submitted a total of 11 assignments, each consisting of 5 problems. A few students did not submit all problems. As a result a total of 2,370 submissions of sketch data were made; de/serialized and represented correctly by S.IM.PL Serialization. Each submission message on average contained 100 KB of data in XML, upto 6 levels deep. [R1] reported that he did not had issues with S.IM.PL Serialization and error reporting was sufficient for debugging incorrect bindings.

[R1] reported that using S.IM.PL Serialization instead of built-in binary representation helped in de/serializing old XML files with very less modification to the source code. He also reported that de/serializing polymorphic instances and graph serialization support were very important in implementing the desired functionality. [R1] identified minor bugs in S.IM.PL Serialization, which we were able to fix immediately. A few design issues were also raised. [R1] did not like the use of maps in S.IM.PL Serialization as it required implementation of `IMappable` interface. Also, data binding of maps in S.IM.PL Serialization requires keys as scalars and values as composite objects, which [R1] reported was a constraint for his application. [R1] requested support for using scalar as values in maps. We plan to implement this support in S.IM.PL Serialization.

2. Multi-Modal Rummy Game

[C2] developed Multi-modal Rummy Game game using iPhones and iPad. iPhones act as private displays that presents a user's hand of cards, while the iPad acts as a public display that users can interact-with and see which cards are played during

the game. A user selects a card on their iPhone. The iPhone application serializes data and sends it across to the iPad application, which deserializes it and renders the played card. Similarly, a user can also draw cards from the public display onto their iPhone. S.IM.PL Serialization is used for de/serialization of game objects represented by 43 class definitions.

Students in group [C2] continued working on their application to further investigate the social impact of their application. They conducted 6 user study sessions. Each session required 4 players and lasted for approximately 1.5 hours. The number of messages exchanged between the server and client application varies based on what is happening in the game. Also, the complexity of messages depends upon the nature of information being exchanged. For example, lay-down request for cards is represented with a data structure encapsulating an array of card objects, while drawing a card request message contains data about a single card. On average, 1.6 messages per second are exchanged between the server and client application. Therefore, approximately 34,560 XML messages were correctly de/serialized across all user-study sessions and client applications. [C2] reported that they did not experience runtime errors and performance issues with S.IM.PL Serialization during user study sessions.

D. Conclusion

In this chapter, we presented benchmark results and student developer experience reports. The benchmark results validate that S.IM.PL Serialization outperforms other XML data binding frameworks. JiBX, which is Java-based XML data binding framework, is a close alternative in performance. However, maintenance of the source code is difficult using JiBX, as data bindings are external to the source code and managed through an offline compile-step by the programmer. In comparison, S.IM.PL Serial-

ization uses annotations, which are easier to maintain, as they are specified within the source code. Interpretation of annotations is handled automatically at runtime. Frameworks that use annotations are significantly slower than S.IM.PL Serialization as they extensively use reflection, which is computationally expensive. S.IM.PL Serialization reduces the amount of reflection operations required through type system scopes (III) that cache data binding semantics and reflection accessor objects, providing better performance than prior frameworks.

To validate usability of S.IM.PL Serialization by software engineers, we conducted semi-structured with student developers who used S.IM.PL Serialization in course assignments. Initial data from 10 student developers is encouraging. We intend to conduct further studies to draw substantial conclusions about the development experience with S.IM.PL Serialization. Feedback from students helped us improve S.IM.PL Serialization's features, error handling, and documentation. A core feature that was added from feedback of a student was graph serialization. A complete tutorial set and documentation were developed to help students understand the use of the framework and Data Binding Annotation Language (DBAL) (II, D). Error handling code was added to facilitate debugging of incorrect use of DBAL constructs. Students suggested that S.IM.PL Serialization reduced development and testing time and facilitated documentation of the source code. They also reported that S.IM.PL Serialization was more flexible and supported more features and development requirements than other XML data binding frameworks, such as Microsoft's XmlSerializer. Students showed interest in using S.IM.PL Serialization for future projects. They were able to develop complex software applications that are publicly deployed or in the process of public deployment, such as the Sketch Recognition Application (IX, C, 1), and Multi-modal Rummy Game (IX, C, 2).

CHAPTER X

RELATED WORK

Systems that communicate, transfer, and exchange information extensively rely on serializing and deserializing structured information. Data binding frameworks facilitates translation between typed object models with serialized representations. Flexible mechanisms of data binding have been discussed and implemented before to reduce cognitive load on software engineers. Solutions such as JAXB [39], JiBX [7], Castor [40] are Java-specific data binding frameworks. Although, these frameworks are flexible in representing information, software programmers are restricted to platform-specific code, and to only use XML format. S.IM.PL Serialization enables data binding across platforms with multiple serialization formats.

The following sections categorizes the different types of data binding approaches and discusses them in comparison to S.IM.PL Serialization. Subsequently, cross-platform data-binding and XML transformation approaches are discussed.

A. Document-Centric Approaches

Many XML data binding frameworks focus on code generation from XML Schema files. Castor, Zeus [41], and JBind [42] provide tools that take an XML grammar as the starting point for serializing and deserializing information. These frameworks generate code in a target programming language, which binds to structured information representations through external mapping files. We categorize these approaches as *document-centric*, where programmers start by defining an information schema and later generate an object model from it.

Document-centric approaches are based on external specifications outside of the

program. These external files are difficult to maintain and debug, as they reside outside of source code. Changes to information structure require changes to these external files and regeneration of code.

B. Object-Centric Approaches

An alternative object-centric approach to data binding is to design the object model first and later bind fields for serialization. Object-centric approaches are more convenient for developers as they can focus on the design of the application.

S.IM.PL Serialization addresses object-centric data binding principles: (1) Write class definitions that define object structures. (2) Integrate algorithms, application logic, and state variables with class definitions. (3) Bind required fields to serialized representations. S.IM.PL Serialization supports these principles by defining the semantics of translation through an annotation language. Using annotations to augment class definitions is convenient for developers because of their closeness to the source code. They are easier to write through auto-completion features in Integrated Development Environments (IDE)s such as Visual Studio [16] for .NET and Eclipse [15] for Java. These IDEs also support refactoring, enabling easy maintenance of augmented class definitions.

In comparison to language constructs in prior frameworks, such as JAXB [39] and XmlSerializer [43], S.IM.PL Serialization's annotation language is more concise for handling complex and polymorphic type objects. Flexible constructs support de/serialization to multiple formats such as XML, JSON, and TLV (type-length-value) [8], providing fine-grained control over the representation.

C. Cross-Language Information Binding

When writing applications across multiple platforms, developers are required to write platform specific code to read and write structured data. Software tools such as Google Protocol Buffers [44] have addressed this issue by generating serialization code in multiple programming languages from definition of message structures specified through external files. External files can be difficult to maintain. In S.IM.PL Serialization, software engineers define message structure by augmenting class definition with DBAL.

Most of the prior research in supporting cross-language computing can be classified as Interface Description Language (IDL)-based approaches such as Distributed Component Object Model (DCOM) [27], Common Object Request Broker Architecture (CORBA) [20], and MockingBird [28]. These approaches require that the engineer define interfaces through external specification in an external language (the IDL). Although IDL-based approaches enable object passing across platforms, they are not data binding frameworks. The focus of IDL-based approaches is to support communication across platforms in distributed applications. Such systems ignore the value of control over serialized representations, as only deserialized information is important from an application's perspective.

In certain contexts, fine-grained control over the serialized representation is crucial. For example, reading data from the wild, such as an RSS [45] feed. To write an application that consumes RSS, developers require control over how serialized representation bind with program objects. To be able to easily write such applications, we need flexible data binding mechanisms across platforms. S.IM.PL Serialization's platform and format independent approach enables software engineers to easily bind data with objects in different platforms and seamlessly transition from one data format to

another, as required.

D. XML Programming Languages

XML is a powerful and widely used format for tree-structured data. To support the need for working with XML in programming languages, prior research led to the definition of XML programming languages, such as XSLT [46], XDuce [47], CDuce [48], Xtatic [49], and XCentric [50]. These projects presents XML programming languages to statically specify types of XML data, conforming to a specific structure. In contrast, S.IM.PL Serialization is based on data binding principles where data structure and types are driven by the structure and types of objects in an object-oriented programming language; eliminating the need to define a new external programming language. The model that S.IM.PL Serialization presents for data binding also supports other tree-structured representations as opposed to only XML.

CHAPTER XI

CONCLUSION

Development of cross-language and multi-format distributed applications is common in real-world software engineering, as different programming languages and data formats offer features that are suitable for a particular application. To support communication between applications in such scenarios, we need data binding frameworks that enable cross-language and multi-format de/serialization of data, reduce burdens on software engineers, and promote software engineering principles of code-reuse, maintenance, and documentation.

We developed S.IM.PL Serialization to facilitate software development in such scenarios. S.IM.PL Serialization supports cross-language data binding across Java, C#, and Objective-C programming languages, with XML, JSON, and TLV data formats. S.IM.PL Serialization provides cross-language code generation utilities that ease development of distributed software systems, and facilitate migration of existing applications.

Cross-language and multi-format data binding are facilitated through type system scopes that encapsulate data binding semantics as abstract semantic graphs. S.IM.PL Serialization utilize type system scopes to efficiently de/serialize data across programming languages in multiple formats. Type system scopes provide a platform-independent mechanism for describing the structure of data and representing data binding semantics, forming the basis of a cross-language type system for specifying data bindings.

Type system scopes are extensible. A software framework, meta-metadata language and architecture (VIII, B), extends type system scopes to develop the meta-metadata type system for describing the structure and flow of metadata from infor-

mation resources to software applications that operate on metadata. Other software frameworks, such as OODSS, and Preferences Management System, utilize type system scopes for data binding of polymorphic instances, which facilitates their architectures for network communication and configuration management.

S.IM.PL Serialization is efficient. Performance benchmarks indicate that S.IM.PL Serialization's run-time performance is better than other, widely used, data binding frameworks. S.IM.PL Serialization is robust. It has performed effectively and without errors in research and educational software used by thousands of users, who are not computer science students or researchers.

Student programmers have used S.IM.PL Serialization in development of software applications for course assignments and research applications. They found it helpful and easy to use. We have made S.IM.PL Serialization available to the open-source community. Its source-code and documentation is publicly available on the Internet [12].

A. S.IM.PL Serialization

S.IM.PL Serialization is a data binding framework, built with object-oriented design principles that facilitates development of correct information-centric software applications. It promotes software engineering principles of code re-usability, maintenance, documentation, and enables flexibility in data and object representation.

The framework presents a Data Binding Annotation Language (DBAL) to specify data binding semantics. In comparison to specifying data binding semantics through external files, DBAL declarations reside within the source-code, which are easier to maintain and facilitates documentation of the source code.

Prior data binding frameworks are language and format-specific, which limits

their usability. In comparison, S.IM.PL Serialization provides cross-language data binding in multiple formats, which facilitates development of distributed applications, integration with third party software systems, and migration of existing software applications to other programming languages.

B. Type System Scopes

S.IM.PL Serialization's data binding architecture is based on type system scopes; a key contribution of this research. Type system scopes provide abstraction over the semantics of de/serialization. They encapsulate data binding semantics as abstract semantics graphs. In programming languages that support annotations, type system scopes are automatically derived from interpretation of declarations in DBAL.

At runtime, type system scopes and their constituent data structures are immutable objects, which are cached for re-use. They cache reflection accessor objects and are organized in efficient data structures; mapped with associated tags in serialized representations. Therefore, type system scopes enable better runtime performance in comparison to other data binding frameworks.

The abstractions provided by type system scopes are utilized by programmers through specification of DBAL constructs, enabling data binding of different types of data structures, such as deeply nested composites, collections, map, graphs, and polymorphic types.

1. Cross-Language Type System

During the course of this research, we are formalizing type system scopes as a cross-language type system. This includes language independent representation of abstract data types, generation of language specific type declarations, and automatic, com-

patible, multi-format de/serialization of object graphs.

In the cross-language type system, `ClassDescriptor`s and `FieldDescriptor`s are language-independent basic types, declarations of which describe the structure of classes and constituent fields; they also specify how an instance of a class is serialized and represented in multiple formats. They enable specification of data structures in programming languages as abstract types: scalars, composite, collection, or polymorphic. Furthermore, these abstract types are mapped with concrete types in a particular programming language, such as Java's Integer, Float, String, ArrayList, and HashMap, or C#'s Int32, Single, String, List, and Dictionary data types.

C. Multi-Format Support

Type system scopes enable multi-format support in S.IM.PL Serialization, as they abstract data binding semantics, which are independent of any particular format. Utilizing the abstract data binding semantics encapsulated in type system scopes, S.IM.PL Serialization implements de/serialization functionalities in different formats.

Fine-grained control over a particular format is provided through DBAL constructs that further specifies how a particular field is represented in a particular format.

Switching between formats is seamless. Software engineers can specify the data format through parametrized de/serialization functions. This enables applications to easily translate data from one format to another, integrate with third party software, and facilitates debugging, as software engineers can switch from non-readable format to readable format and identify errors in data.

S.IM.PL Serialization comes with built-in support for XML, JSON, TLV, and BibTeX formats of data representation. We plan to integrate support for other for-

mats, such as YAML, an alternative to XML that is gaining popularity in web applications.

D. Cross-Language Support

Cross-language support in S.IM.PL Serialization is facilitated through type system scopes, as they describe data structures in a language-independent type system and specify how objects bind with serialized representations.

Equivalent specification of type system scopes in particular programming languages, either through DBAL augmented class definitions or serialized type system scopes, enables cross-language data binding.

We provide code generation utilities that facilitate cross-language data binding. The code generation utility uses type system scopes to generate code in any supported target programming language. Code comments used for documentation of the source code are part of the type system scope enabling parsing by third party documentation utilities.

Presently, cross-language support is provided between Java, C#, and Objective-C programming languages. Support for JavaScript as target programming language is currently under development. Support for further programming languages, such as C++ and Python, is planned.

E. Validation

1. Data Binding Architecture and Extensibility

We examined how S.IM.PL Serialization's data binding architecture is utilized and extended to explore different areas of research through software frameworks. Research frameworks that utilize and/or extend S.IM.PL Serialization were developed

to promote software engineering principles, reduce burden on software engineers, and facilitate development of different types and components of software systems. These frameworks are used in research applications and by students developing software applications for course and research assignments.

OODSS implements novel semantics of service call and return in distributed applications that promote object-oriented design principals. S.IM.PL Serialization's support for polymorphism is being used for OODSS message parsing and invocation. OODSS utilizes type system scopes and data binding architecture for polymorphic types to dynamically dispatch invocation of remote methods. S.IM.PL Serialization enables OODSS services to function across different programming languages and support communication in multiple formats.

The Preferences Management system utilizes the data binding architecture of polymorphic types to provide quick and typed access to configuration settings specified in external files. It facilitates configuration management and deployment of software applications.

The meta-metadata language and architecture validates the extensibility of type system scopes. It develops a language-independent meta-metadata type system that extends type system scopes for specifying the structure of metadata, how data is extracted from information resources, presented to users, and acted-on by software tools. As of this writing, software engineers have specified 148 metadata types through the meta-metadata type system describing 85 different information sources. These include scholarly articles, books, patents, search engine results, RSS feeds, social media, newspapers, and product stores.

2. Real-World Software Applications and Robustness

S.IM.PL Serialization is used in research and educational software applications, developed by researcher and student developers. The research applications Team Coordination (TeC) game and combinFormation are designed to teach and enhance team coordination skills and promote creativity.

TeC is a multi-player game that has been played by students, fire-fighters, and other emergency responders. Studies have shown that through TeC's engaging game experience, players learned and improved team coordination skills. As of this writing, 99 different users have played and benefited from TeC.

TeC client application was migrated from Java to Objective-C using the cross-language code generation facilities that translated 130 Java classes to Objective-C. This newer portable version of TeC uses cross-language data binding, as the game server is still maintained in Java.

combinFormation is a creativity support tool has been used by students in undergraduate and graduate coursework at Texas A&M University. A majority of combinFormation users are not computer science students. Studies have shown that combinFormation is successful in promoting creativity. At least 2,000 users have used an benefited from combinFormation.

A multi-modal rummy game was developed by undergraduate students. Its goal is to teach use of technology to old people through an engaging card game. S.IM.PL Serialization is used for de/serialization of data for network communication. At least, 24 different users have played multi-modal rummy across multiple user study sessions.

A graduate student developed a research application for undergraduate students to submit assignments through hand drawn sketches and have the system automatically grade them. S.IM.PL Serialization is used for de/serialization of sketch data.

At least 53 undergraduate students have used the software to develop and submit assignments.

S.IM.PL Serialization has performed without errors and performance issues in these software applications. Extensive use of data de/serialization in these software applications validates S.IM.PL Serialization's capability to accommodate requirements of software engineers, developing complex software systems.

3. Performance Benefits

We measured de/serialization performance of S.IM.PL Serialization with other widely used data binding frameworks: JiBX, JAXB, Caster, and XStream. Our benchmark results showed that S.IM.PL Serialization's runtime performance is better than these frameworks. S.IM.PL Serialization is 9 times faster in serialization and twice as fast in deserialization than JAXB, which is a Java supported XML data binding framework.

The performance benefits of S.IM.PL Serialization are achieved through type system scopes, as they efficiently cache data binding semantics derived from annotations and reflection accessor objects, which facilitates quick access to data binding semantics and reduces the number of reflection operations required for de/serialization.

4. Development Experience

We conducted semi-structured interviews with undergraduate and graduate student developers that used S.IM.PL Serialization for developing research projects and course assignments. Initial data obtained from 10 student developers is encouraging, as they suggested that S.IM.PL Serialization was easy to use and facilitated development and documentation of the source code. Data binding architecture accommodated their software design requirements.

Developers suggested some shortcomings, which helped us improve S.IM.PL Se-

rialization through addition/refinement of data binding features and error reporting and handling.

F. Ongoing Work

S.IM.PL Serialization is under active development. We are consistently working towards improving the design and data binding architecture that may further improve runtime performance and facilitate software engineers in application development.

In the next sections, we present the ongoing work in S.IM.PL Serialization and describe how it affects the framework and software engineers.

1. Pull Parsers

We presented earlier in Chapter III Section C, 2 that deserialization is performed using type system scopes with SAX parsers. To validate that type system scopes can be easily integrated with other types of parsers, we are integrating *pull* parsers for deserialization. Pull parsers parse data as a stream rather than pushing events out to the client code (SAX). The client code drives the parser rather than a parser driving the client code.

Pull parsers are generally shown to be faster in performance as compared to SAX parsers, and result in simpler code. We hypothesize that integrating pull parsers will improve runtime performance and maintainability of the code-base.

2. Extend `ElementState`?

In the current implementation, user-defined classes must extend `ElementState` base class to enable de/serialization of its instances. The `ElementState` base class stores information that is utilized by deserialization algorithms, code-generation utilities,

and other software frameworks that extend S.IM.PL Serialization, such as the meta-metadata language and architecture.

Although this mechanism of extending `ElementState` simplifies various functionalities that the framework provides, it imposes an unnecessary restriction on software engineers. For example, a software engineer wants to author a S.IM.PL de/serializable class that extends a class is provided by a third party framework, which of course does not extend `ElementState`. As multiple inheritance is not supported by many programming languages (for its issues), software engineers are forced to work around this limitation.

We are removing this limitation, so that software engineers will not be required to extend `ElementState` for de/serialization. This will provide more flexibility and improve effectiveness of S.IM.PL Serialization in real-world software applications.

G. Future Work

S.IM.PL Serialization is a project that started with focus on relieving software engineers from burdens associated with present day intricacies of writing information-centric applications. We implemented, presented, and validated a framework that addresses our present objectives.

We have identified areas of future research that will further facilitate development of information-centric applications, cater the growing requirements of software engineers, and validate efficacy of S.IM.PL Serialization.

1. Languages without Reflection

S.IM.PL Serialization utilizes reflection capabilities of a programming language to implement generic de/serialization functionalities. Since reflection is computationally

expensive, some programming languages such as C++, do not support this functionality. Thus, a generic mechanism for de/serialization cannot be implemented.

Future work in S.IM.PL Serialization will investigate support for programming languages that do not support reflection. We plan to implement a cross-language code generator for C++ that generates class definitions as well as parsing and serialization code from type system scopes. Objects serialized from supported programming languages will seamlessly deserialize into objects of C++ and vice versa.

Generation of parsing code will be difficult to manage as changes in object structure will require re-generation of C++ classes. However, serialization and deserialization processes will be significantly faster in C++, as runtime introspection is not required. Using this methodology we can support object-oriented programming languages that do not support reflection.

2. Generic Parsers

Addition of a new format presently requires changes to the S.IM.PL Serialization code base. The amount of change required can vary depending on the relative difference of the new format to the currently supported formats. The change may include modifications in DBAL and addition of new serialization and deserialization methods.

We plan to improve the architecture of S.IM.PL Serialization so that minimal or ideally no change in code base is required to incorporate a new format. We plan to investigate parser generators and the possibility of incorporating a generic parser. A generic parser can parse data in any format from specifications in BNF or EBNF grammar. Generic parsers are an area of active research. Difficulties arise from ambiguous grammar and efficiency in parsing. However, research in this domain presents methods to work around these difficulties.

Type system scopes are independent data structures and not integrated with

specific parsers. Therefore, through further research, we can incorporate a generic parser with type system scopes and implement de/serialization methods that can work with any format, provided its BNF or EBNF grammar.

3. Data Representation Protocols

Data representation protocols for remote method invocation follows a specific format such as XML but further defines their own syntax of how information is represented. For example, SOAP encapsulates data in a soap envelope containing soap-header and soap-body tags in XML. Similarly, XML-RPC also uses XML but represents information as key-value pairs. We plan to implement inherent support for widely used data representation protocols, enabling software developers using S.IM.PL Serialization to write applications that communicate with third party software systems, such as typical Web Services.

REFERENCES

[1] W3C, "Extensible markup language (xml) 1.0 (fifth edition)," 2009. http://www.w3.org/TR/REC-xml/.

[2] D. Crockford, "JSON: The fat-free alternative to XML," December 2006. http://www.json.org/fatfree.

[3] O. Ben-Kiki, C. Evans, and I. döt Net, "YAML Ain't Markup Language (YAML™) Version 1.2," 2009. http://www.yaml.org/spec/1.2/spec.html.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification*. Mountain View, CA: Addison-Wesley Professional, 3rd ed., 2005.

[5] T. L. Thai and H. Lam, *.NET Framework Essentials*. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

[6] Apple Inc., "The Objective-C Programming Language," October 2009. http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf.

[7] D. M. Sosnoski, "JiBX: Binding XML to Java Code," 2011. http://jibx.sourceforge.net/.

[8] International Telecommunication Union, "ITU-T X.690," 2002. http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf.

[9] A. Feder, "Bibtex format description," 2006. http://www.bibtex.org/Format/.

[10] A. Kerne, Z. O. Toups, B. Dworaczyk, and M. Khandelwal, "A concise XML binding framework facilitates practical object-oriented document engineering," in *Proc. ACM DocEng*, pp. 62–65, 2008.

[11] A. Kerne, Z. O. Toups, B. Dworaczyk, and A. Khandelwal, "Expressive, efficient, embedded, and componenet-based xml-java data binding framework," tech. rep., Texas A&M University, 2008.

[12] Interface Ecology Lab, "Support for information mapping in programming languages," July 2010. `http://ecologylab.net/research/simpl/index.html`.

[13] D. Flanagan, *Javascript: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, Inc., 4th ed., 2002.

[14] B. Stroustrup, *The C++ Programming Language, Third Edition*. Boston, MA: Addison-Wesley Longman Publishing Co, 3rd ed., 1997.

[15] Eclipse Foundation, "Eclipse integrated development environment," 2010. `http://eclipse.org`.

[16] Microsoft Corporation, "Microsoft Visual Studio," 2010. `http://www.microsoft.com/visualstudio/en-us/`.

[17] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, pp. 61 –72, May 1988.

[18] S. S. Skeina, *The Algorithm Design Manual*, ch. 15, pp. 495–496. Stony Brook, NY: Springer, 2nd ed., August 21, 2008.

[19] M. McCracken, "BibDesk Mac bibliography manager," 2010. `http://bibdesk.sourceforge.net/`.

[20] Object Management Group, "CORBA: Core specification," 2010. http://omg.org/docs/04-03-12.pdf.

[21] NDoc, "NDoc Code Documentation Generator for .NET," 2005. http://ndoc.sourceforge.net/.

[22] Apple Inc., "HeaderDoc User Guide: Introduction," 2010. http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/HeaderDoc/intro/intro.html.

[23] Z. O. Toups, A. Kerne, W. A. Hamilton, and N. Shahzad, "Object-oriented distributed semantic services: A S.IM.PL Approach," *Submitted to ICSE*, 2010.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[25] W3C, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," April 2007. http://w3.org/TR/soap.

[26] D. Winer, "XML-RPC specification," 1999. http://xmlrpc.com/spec.

[27] R. Grimes and D. R. Grimes, *Professional Dcom Programming*. Birmingham, UK: Wrox Press Ltd., 1997.

[28] J. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari, "Mockingbird: Flexible stub compilation from pairs of declarations," in *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 393–402, IEEE Computer Society, 1999.

[29] A. Kerne, Y. Qu, A. Webb, S. Damaraju, N. Lupfer, and A. Mathur, "Meta-metadata: A metadata semantics language for collection representation applications," in *Proc. of 2010 ACM Conference on Information and Knowledge Management*, (Toronto, Ontario), pp. 26–30, October 2010.

[30] W3C, "XML Path Language," 2009. `http://www.w3.org/TR/xpath/`.

[31] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, pp. 513–523, August 1988.

[32] Interface Ecology Lab, "combinformation," `http://ecologylab.net/combinFormation/`.

[33] A. Kerne, E. Koh, B. Dworaczyk, J. M. Mistrot, H. Choi, S. M. Smith, R. Graeber, D. Caruso, A. Webb, R. Hill, and J. Albea, "combinformation: a mixed-initiative system for representing collections as compositions of image and text surrogates," in *Proc. JCDL*, pp. 11–20, 2006.

[34] A. Kerne, E. Koh, S. M. Smith, A. Webb, and B. Dworaczyk, "combinFormation: Mixed-initiative composition of image and text surrogates promotes information discovery," *ACM Trans. Information Systems*, vol. 27, no. 1, pp. 1–45, 2008.

[35] Z. O. Toups, W. A. Hamilton, A. Kerne, and N. Shahzad, "Zero-fidelity simulation of fire emergency response: Improving team coordination," in *Proc. ACM SIGCHI Conference on Human-Computer Interaction*, (Vancouver, BC, Canada), May 2011.

[36] Z. O. Toups, A. Kerne, and W. A. Hamilton, "The team coordination game: A zero-fidelity simulation abstracted from fire emergency response work practice,"

ACM Transactions on Computer-Human Interaction, vol. 18, no. 4, 2011. in press.

[37] W. A. Hamilton, Z. O. Toups, and A. Kerne, "Synchronized communication and coordinated views: qualitative data discovery for team game user studies," in *Proc. Ext Abs ACM Computer Human Interaction*, pp. 4573–4578, 2009.

[38] Bindmark, "bindmark: BindMark home page," 2005. `https://bindmark.dev.java.net/old-index.html`.

[39] J. Fialli and S. Vajjhala, "The Java architecture for XML binding (JAXB)," *JSR Specification*, January 2003. `http://jaxb.java.net/`.

[40] E. Group, "The Castor Project," 2005. `http://www.castor.org/`.

[41] C. Contreras, C. Ney, and F. Letellier, "Enhydra Zeus Project," 2008. `http://zeus.ow2.org/`.

[42] S. Wachter, "JBind - A Java-XML Data Binding Framework," 2009. `http://jbind.sourceforge.net/jBind.html`.

[43] Microsoft Corporation, "XmlSerializer Class (System.Xml.Serialization)," 2010. `http://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer.aspx`.

[44] Google Inc., "Protocol buffers : Google's data interchange format," 2011. `http://code.google.com/p/protobuf`.

[45] W3C, "RSS 2.0 Specification (RSS 2.0 at Harvard Law)," 2003. `http://cyber.law.harvard.edu/rss/rss.html`.

[46] J. Clark, "XSL Transformations (XSLT), Version 1.0," November 1999. `http://w3.org/TR/xslt`.

[47] H. Hosoya and B. C. Pierce, "Xduce: A statically typed xml processing language," *ACM Trans. Internet Technol.*, vol. 3, pp. 117–148, May 2003.

[48] V. Benzaken, G. Castagna, and A. Frisch, "CDuce: An XML-centric general-purpose language," in *Proc. ACM SIGPLAN*, pp. 51–63, 2003.

[49] V. Gapeyev, F. Garillot, and B. C. Pierce, "Statically typed document transformation: An Xtatic experience," in *Workshop on Programming Language Technologies for XML*, Jan. 2006.

[50] J. Coelho and M. Florido, "xCentric: Logic programming for XML processing," in *Proc. ACM Int'l Workshop on Web Information and Data Management*, pp. 1–8, 2007.

APPENDIX A

NOMENCLATURE: ABBREVIATIONS

DBAL	Data Binding Annotation Language
XML	Extensible Markup Language
JSON	Java Script Object Notation
TLV	Type Length Value
YAML	Y A'int Markup Language
OODSS	Object Oriented Disctribute Semantics Services
TeC	Team Coordination
S.IM.PL	Support for Information Mapping in Programming Languages
CORBA	Componenet Object Request Broker Architecture
DCOM	Distributed Component Object Model
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
HTML	HyperText Markup Language
RPC	Remote Procedure Call
W3C	World Wide Web Consortium
URL	Uniform Resource Locator

APPENDIX B

DBAL - REFERENCE AND EXAMPLES

Annotation	Production Rule	Description
`@simpl_scalar`	*SIMPLIndividualAugmentation*: `@simpl_composite` | `@simpl_scalar` | `@simpl_hints`({**HintName** , {**HintName**}}) | `@simpl_filter` (**regex** = "***Expression***")	defines a scalar class attribute to be translated as XML attribute. **Example**: `@simpl_scalar String item;`
`@simpl_hints`	*SIMPLIndividualAugmentation*: `@simpl_composite` | `@simpl_scalar` | `@simpl_hints`({**HintName** , {**HintName**}}) | `@simpl_filter` (**regex** = "***Expression***")	precisely define the syntactic structure of serialization by using XML_ATTRIBUTE, XML_LEAF, XML_TEXT, XML_TEXT_CDATA, XML_LEAF_CDATA **Example**: `@simpl_hints (Hint.XML_ATTRIBUTE) String item;`

Annotation	Production Rule	Description
`@simpl_composite`	*SIMPLIndividualAugmentation*: `@simpl_composite` | `@simpl_scalar` | `@simpl_hints`({**HintName** , {**HintName**}}) | `@simpl_filter` (**regex** = "***Expression***")	must be a subclass of `ElementState`, meaning that it has further annotated fields. **Example**: `@simpl_composite CopmositeObject item;`
`@simpl_filter`	*SIMPLIndividualAugmentation*: `@simpl_composite` | `@simpl_scalar` | `@simpl_hints`({**HintName** , {**HintName**}}) | `@simpl_filter`(**regex** = "***Expression***")	takes a valid regular expression to filter out the data when translating from serialized representation. **Example**: `@simpl_filter (regex = "") String item;`
`@simpl_collection`	*SIMPLCollectionAugmentation*: `@simpl_wrap` | `@simpl_nowrap` | `@simpl_collection` (***["TagName"]***) | `@simpl_map` (***["TagName"]***) ***[SIMPLClasses]*** *Identifier TypeArguments*	describes a collection of scalar/composite/polymorphic types; by default collections are *wrapped* **Example**: `@simpl_collection ("item") ArrayList<Item> items;`

Annotation	Production Rule	Description
`@simpl_map`	*SIMPLCollectionAugmentation*: `@simpl_wrap` | `@simpl_nowrap` | `@simpl_collection` (***["TagName"]***) | `@simpl_map` (***["TagName"]***) ***[SIMPLClasses]*** *Identifier TypeArguments*	describes a collection of scalar/composite/polymorphic types; by default collections are *wrapped* **Example**: `@simpl_map ("item") HashMap<String, IMappableObj> items;`
`@simpl_wrap`	*SIMPLCollectionAugmentation*: `@simpl_wrap` | `@simpl_nowrap` | `@simpl_collection` (***["TagName"]***) | `@simpl_map` (***["TagName"]***) ***[SIMPLClasses]*** *Identifier TypeArguments*	used with collections and maps; wraps the serialized representation with tag name **Example**: `@simpl_wrap @simpl_collection ("item") ArrayList<String> items;`

Annotation	Production Rule	Description

`@simpl_nowrap`	*SIMPLCollectionAugmentation*:
	`@simpl_wrap`
	| `@simpl_nowrap`
	| `@simpl_collection`
	(***["TagName"]***)
	| `@simpl_map`
	(***["TagName"]***)
	[SIMPLClasses]
	Identifier TypeArguments
		this directive defines if the resultant leaf nodes in serialized representation be wrapped with class or attribute name
		Example:
		`@simpl_nowrap`
		`@simpl_collection`
		`("item")`
		`ArrayList<String>`
		`items;`

`@simpl_classes`	*SIMPLClasses*:
	`@simpl_classes`(
	{***ClassName.class***,
	{***ClassName.class***}})
		for polymorphic type definitions; takes input classes that the defined type can to
		Example:
		`@simpl_classes {`
		`{Base.class,`
		`Sub.classs}}`
		`@simpl_collection`
		`("item")`
		`ArrayList<String>`
		`items;`

Annotation	Production Rule	Description

`@simpl_scope`	*SIMPLScope*:
	`@simpl_scope`(
	["SimplTypesScope"])
		for polymorphic type definitions; takes input classes that the defined type can to
		Example:
		`@simpl_scope{"tScope"}`
		`@simpl_map`
		`("item")`
		`HashMap<String,`
		`IMappableObj> items;`

`@simpl_inherit`	*ClassDeclaration*
	class [`@simpl_inherit`]
	[`@simpl_tag`(" ***TagName***")]
	[SIMPLOtherTags]
	Identifier [extends *Type*]
	[implements *TypeList*]
	ClassBody
		indicates that the fields of superclass should be translated.
		Example:
		`@simpl_inherit`
		`public class`
		`SomeClass extends`
		`AnotherClass`

Annotation	Production Rule	Description

`@simpl_tag`	***FieldDecl***:
	[`@simpl_tag`(" ***TagName***")]
	[SIMPLOtherTags]
	[SIMPLClasses]
	SIMPLIndividualAugmentation
	Type Identifier
	MethodOrFieldRest |
	SIMPLCollectionAugmentation
	Identifier MethodORFieldRest
		allows programmer to explicitly declare the tag name for a given field or class.
		Example:
		`@simpl_tag("tag_name")`
		`String item;`

`@simpl_other_tags`	***SIMPLOtherTags*** :
	`@simpl_other_tags(`
	{" ***TagName***",
	" ***TagName***" }})
		for backward compatibility. allows programs to read serialized formats with a different tag name
		Example:
		`@simpl_other_tags(`
		`"old_tag")`
		`String item;`

APPENDIX C

QUESTIONNAIRE - SEMI-STRUCTURED INTERVIEW

1) Which software tool(s) did you employ in creating your application?

2) What was your experience using the software tools(s) in creating your applications?

   a. Did the software tools(s) impact the way you constructed your application?

   b. Did they make any aspect simpler?

   c. More difficult?

   d. How did using the software tools(s) impact your development process?

3) How would you compare working with the software tools to other methods you could have used to solve the same problem?

4) Have you used any technologies similar to the software tools previously?

   a. Which technologies?

   b. What applications did you develop?

   c. Did the other technologies make development simpler?

   e. How did the other technologies compare to the software tools?

5) Did you modify the software tools(s) in any way?

   a. What was the purpose of the modifications?

   6) What did you like about developing with the software tools?

7) What did you dislike about developing wit the software tools?

8) Is your application for one-time use, or will you re-use it?

9) Would you use the software tools(s) again?

10) What would make the software tools(s) better?

VITA

Nabeel Shahzad received his Bachelor of Science in computer science from National University of Computer and Emerging Sciences in Karachi, Pakistan in 2006. After working in the industry for 2 years, he entered MS program at Texas A&M University in College Station, Texas in 2008. In 2009, he joined the Interface Ecology Lab with Prof. Andruid Kerne as his advisor.

Nabeel can be reached by email (nabeel@ecologylab.net) or by mail:

Department of Computer Science and Engineering

c/o Dr. Andruid Kerne

Texas A&M University

College Station, Texas, USA 77843-3112