Available online at www.sciencedirect.com
SCIENCE DIRECT
Computer Languages, Systems & Structures 32 (2006) 109–124
COMPUTER LANGUAGES SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

Inter-language reflection: A conceptual model and its implementation

Kris Gybels[a,*], Roel Wuyts[b], Stéphane Ducasse[c], Maja D'Hondt[d]

[a] *Vrije Universiteit Brussel, Brussels, Belgium*
[b] *Université Libre de Bruxelles, Brussels, Belgium*
[c] *LISTIC, Université de Savoie, France*
[d] *Université des Sciences et Technologies de Lille, Villeneuve D'Ascq, France*

Received 20 September 2005

Abstract

Meta programming is the act of reasoning about a computational system. For example, a program in Prolog can reason about a program written in Smalltalk. Reflection is a more powerful form of meta programming where the same language is used to reason about, and act upon, itself in a causally connected way. Thus on the one hand we have meta programming that allows different languages or paradigms to be used, but without causal connection, while on the other hand we have reflection that offers causal connection but only for a single language. This paper combines both and presents *inter-language reflection* that allows one language to reason about and change in a causally connected way another language and vice versa. The fundamental aspects of inter-language reflection and the language symbiosis used therein, are discussed. Moreover the implementation of two symbiotic reflective languages is discussed: Agora/Java and SOUL/Smalltalk.

Keywords: Meta programming; Reflection; Linguistic symbiosis; Inter-language reflection

1. Introduction

Software engineering practices often require tools that incorporate a form of *meta programming* for extracting information from programs, checking them against a certain specification or generating them. Examples abound such as tools for detecting bad smells in programs to be refactored [1], checking the program's conformance with architectural restrictions [2], generating skeleton code for the implementation of design patterns or user interface elements [3], detecting the possible types a variable can hold in programs written in a dynamically typed language [4] and so on. In a meta programming context, the *base language* is the language of the computational system under analysis, whereas the *meta language* is the language in which a representation of the base program is made available and which is used to

* Corresponding author. Tel.: +32 2 629 35 81.
E-mail addresses: Kris.Gybels@vub.ac.be (K. Gybels), Roel.Wuyts@ulb.ac.be (R. Wuyts), Stephane.Ducasse@univ-savoie.fr (S. Ducasse), Maja.D'Hondt@vub.ac.be (M. D'Hondt).

1477-8424/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2005.10.003

perform the analysis. The meta language does not necessarily have to be the same as the base language. In fact, it has often been found that different programming paradigms are more suitable for certain meta programming activities than others. Several approaches exist where the base language is for example a procedural or object-oriented one, while the meta language is based on a different paradigm: logic programming languages such as Prolog have been found to be especially suitable for extracting information from programs [1] and expressing constraints over a program's structure [5,1,6,7], the well-known tool Lint that incorporates a regular-expressions-based language for encoding patterns of typically problematic C code [8].

In the particular form of meta programming known as *reflection*, the base and meta languages are the same [9–11]. This contrasts with the aforementioned observation that using different programming languages as base and meta languages can be beneficial. A second crucial difference between reflection and meta programming is that in the former the base program and its representation as data in the meta program are *causally connected* [12,13]. This allows the meta program to also manipulate the elements of the base program that are made available to it, which could be the base program's source text, runtime stack, data structures in memory and so on. When the meta program only inspects the base level elements, the system is called introspective; when it also modifies them, it is called intercessory [11]. Since in either case the base and meta programs are represented in the same language, the base and meta programs can also be the same, allowing a program to manipulate itself while it is running. Reflection is heavily relied upon in the self-extensible software development systems of Smalltalk [14,15], Self [16] and CLOS [11].

This paper introduces *inter-language reflection*, a form of reflection between two different languages, possibly of a different programming paradigm. Inter-language reflection extends the causal connection property of reflection to hold between these different languages. In addition to reflection, another fundamental ingredient of inter-language reflection is *linguistic symbiosis* between the two languages. Linguistic symbiosis enables the representation of data of one language in the other, as well as the activation of behavior described in one language from the other. In order to achieve these two elements, a *data mapping* and a *protocol mapping* need to be devised between the two languages. A clean linguistic symbiosis between two languages that each have traditional reflection, results in inter-language reflection, where a program can be implemented in one language and still use the reflection interface of the other language.

Inter-language reflection as presented in this paper is implemented in *Agora* [17] and *SOUL* [18]. Agora is a prototype-based object-oriented language that has inter-language reflection with Java, a class-based object-oriented language. This allows a very dynamic programming style in Agora, while providing access to the extensive libraries of Java. SOUL is a variant of Prolog, a logic programming language, and has inter-language reflection with Smalltalk, an object-oriented language. It has been successfully used as a tool to support several software engineering activities, such as detection of design patterns, software architectures or bad smells in source code [19], as a basis for aspect-oriented programming [20], or to reason about run-time execution traces [21]. While previous papers have demonstrated the usefulness of Agora and SOUL, this paper's purpose is to illustrate concrete instances of inter-language reflection and the specific implementation strategy. The reason why we introduce these two instances, is that Agora shows inter-language reflection of two different languages of the object-oriented programming paradigm, whereas SOUL demonstrates the more complex inter-language reflection between a language of the logic programming paradigm and a language of the object-oriented programming paradigm.

The contributions of this paper are:

- the identification of the limits of traditional reflection and meta programming,
- the definition of inter-language reflection and its use of linguistic symbiosis,
- the presentation of a concrete implementation strategy of inter-language reflection.

The outline of the paper is as follows. Section 2 defines inter-language reflection and shows that it can be achieved through traditional reflection and linguistic symbiosis. Section 3 introduces the concept of linguistic symbiosis, and its key elements of data mapping and protocol mapping. Sections 4 and 5 show the linguistic symbiosis in Agora and SOUL, respectively. In both cases, the data mappings between the languages involved are explained. Section 6 explains how the data mapping at the base level is achieved by protocol mapping at the meta level. Section 7 introduces the concrete implementation strategy of linguistic symbiosis and inter-language reflection that is employed in both Agora and SOUL. Section 8 presents an example of inter-language reflection. Section 9 discusses related work, and Section 10 concludes the paper.

2. Inter-language reflection = linguistic symbiosis + reflection

This paper introduces *inter-language reflection*, a form of reflection between two different languages, possibly of a different programming paradigm. Inter-language reflection extends the causal connection property of traditional reflection to hold between these different languages. As such, a program that uses the reflection interface of the one language can be described in the other language. Therefore, inter-language reflection provides all the aforementioned benefits of being able to represent a meta program in a different programming language (or even paradigm) than the base language. Additionally, a program implemented in one language, is able to make changes to elements of a program implemented in another language.

The key idea exposed in this paper is that inter-language reflection between languages *A* and *B* can be achieved by combining the following two ingredients (this is shown in Fig. 1):

- *traditional reflection* of both language *A* and language *B*. Languages with traditional reflection allow programs to observe and manipulate the data of their own execution process just as if that data were regular data in the language.
- *linguistic symbiosis* between language *A* and language *B*. Two languages are in linguistic symbiosis when they can transparently exchange data and invoke each other's behavior. In order to achieve this, data of both languages needs to be mapped to one another, as well as the protocols for invoking behavior.

It is important to note that by allowing linguistic symbiosis between the two reflective languages, they cannot only access each other's basic data and behavior at the base level, but also data and behavior through the reflective interfaces. Linguistic symbiosis is presented in the next section.

Note that we assume languages in which there's an explicit meta representation of the language's operations in the evaluation process. This typically holds for interpreted and bytecode interpreted languages.

3. Linguistic symbiosis model overview

As illustrated in Fig. 2 this section focuses on linguistic symbiosis in itself. Two languages are in linguistic symbiosis when they can transparently invoke each other's behaviour and exchange data. Linguistic symbiosis enables the representation of data of one language in the other, as well as the activation of behavior described in one language from the other. Our model for achieving linguistic symbiosis consists of two elements, a *data mapping* and a *protocol*

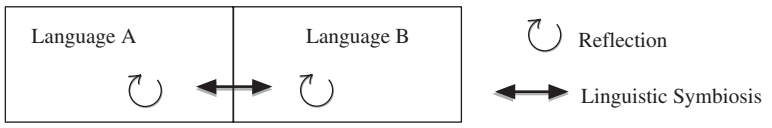

Fig. 1. Conceptual overview of inter-language reflection between two languages A and B: A and B are reflective languages that are in linguistic symbiosis.

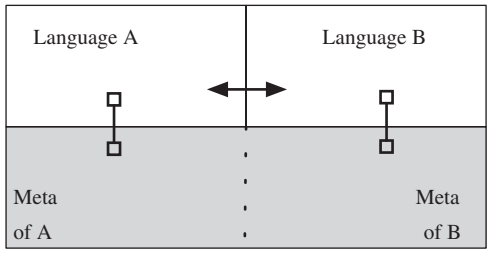

Fig. 2. Conceptual overview of linguistic symbiosis between two languages A and B, showing both base and meta levels.

mapping that is needed between the two languages:

Data mapping: To achieve a tight integration at the syntactic level when passing data between the programs in the different languages, the data should "appear" in each language as seemingly native data. This means that it should be possible for programs in B to apply operations on data of A as though it was native data of B, and vice versa. Therefore, operations invoked in B on data of A somehow need to be translated to operations of A, and vice versa.

Protocol mapping: The key point in linguistic symbiosis is that the data mapping at the syntactic level comes down to a protocol mapping at the language implementation level: making the data of one language "appear" in the other is achieved on the language implementation level by ensuring that the meta representations of that data can be passed between the interpreters. To do so the protocols of the representations of both languages must be explicitly considered: to allow a meta representation to be passed to another interpreter requires making the meta operations of that interpreter applicable to that meta representation as well. Explicitly considering the protocol mapping gives a clear picture of the differences between the languages that need to be resolved in order to integrate them syntactically at the base level.

This linguistic symbiosis model does not constitute a general definition of how to construct protocol mappings for concrete languages, but serves as a conceptual framework that needs to be instantiated. The following sections therefore discuss the data mapping and protocol mapping for two concrete cases: Agora and Java, and SOUL and Smalltalk. Section 4 discusses the data mapping for Agora and Java, Section 5 discusses this for SOUL and Smalltalk, and Section 6 discusses the protocol mapping for both cases.

4. Linguistic symbiosis between Agora and Java

Defining a linguistic symbiosis between Agora and Java requires transparent ways for exchanging data and invoking behavior. Exchanging data means that it should be possible to pass an Agora object to a Java program, and vice versa to pass a Java object to an Agora program. Invoking behavior means that from Agora it should be possible to send messages to these Java objects, and vice versa to send messages to Java objects from within Agora programs. For these exchanges and invocations to be *transparent*, the Java object should appear as an Agora object in the Agora program: an Agora program should be able to send messages to a Java object in the same way as it sends messages to native Agora objects. The same should hold for Agora objects in Java programs.

4.1. An example

Fig. 3 shows a concrete example of an Agora program that uses linguistic symbiosis with Java:

- The first expression defines the variable `frame` to which we assign a newly created instance of the Java class `Frame`. Note that here the message `JAVA` is sent to a string that contains the name of a Java class, the message returns this Java class as an Agora object to which then the message `new` is sent.
- The second expression similarly defines a variable `ok` to hold an instance of the Java class `Button`.
- The third expression sends the message `addComponent:` to the frame with the button as argument. Note that this message is sent using the regular Agora syntax for message sending, behavior is effectively invoked on the Java object contained in the `frame` variable as if it were an Agora object.
- The fourth expression defines a variable `okListener` to hold a new Agora object to act as the button's listener.
- The fifth expression sends the message `addActionListenerActionListener:` to the button with the Agora object as argument to install this listener. Note that since the `ok` variable contains a Java object, the message and its arguments are passed to Java which in this case means an Agora object is passed to Java.

4.2. Data mapping

Accessing Java objects from Agora: In Agora it is possible to access Java classes as *regular* Agora objects. The constructors are invoked through Agora messages to create new instances. Fig. 3 shows a number of examples where a Java class is accessed from Agora using the `JAVA` message. The `JAVA` message is sent to a string, which interprets the string as the name of a class in Java and returns that class. Thus the first objects that can be "grabbed" from Agora are

```
frame VARIABLE:
  ("java.awt.Frame" JAVA) new;

ok VARIABLE:
  ("java.awt.Button" JAVA) newString: "OK";

frame addComponent: ok;

okListener VARIABLE: [
  implements METHOD:
    (1 ARRAY: ("java.awt.event.ActionListener" JAVA));
  replaces METHOD:
    ("java.lang.Object" JAVA);
  actionPerformedActionEvent: e METHOD: {
    ("java.lang.System" JAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];

ok addActionListenerActionListener: okListener
```

Fig. 3. Example of the language symbiosis between Agora and Java.

classes by using their name. Using Agora messages like `new` and `newString:` new instances are created. Another way for Java objects to wind up in Agora is of course by having them passed as arguments to messages to Agora objects.

Passing Agora objects to Java: Agora objects are only passed to Java when they are used as arguments in messages to Java objects from within Agora. In the last expression, the message `addActionListenerActionListener:` is sent to the Java `Button` object, with the Agora object `okListener` as argument. Because Java is a class-based language and Agora is not, the Agora object needs to appear as the instance of a Java class when it is passed to Java. Which class it is made an instance of is determined by the Agora object itself: Agora objects that are passed to Java are expected to implement the message `implements` and the message `replaces` which should, respectively, return an array of Java interfaces and a single Java class. From Java, the Agora object then appears as an instance of a class that implements the interfaces as given by the `implements` message and that is a subclass of the class as given by the `replaces` message. Note however that in order to preserve the dynamic nature of Agora, it is not checked whether the object actually supports the messages as declared by the Java interfaces.

5. Linguistic symbiosis between SOUL and Smalltalk

SOUL and Smalltalk differ more fundamentally in their underlying paradigm than Java and Agora: Smalltalk is an object-oriented language while SOUL is a logic language. Achieving the linguistic symbiosis is therefore much more complicated, since the basic building blocks of each language differ: Smalltalk uses objects and messages, while SOUL uses logic terms and backtracking.

5.1. Issues and an example

To achieve the linguistic symbiosis we again need to show how the data is mapped, and how protocol differences are resolved. These issues are illustrated in Fig. 4 with an example application of symbiosis where logic rules are used to implement business rules about an object-oriented business application [22]: an object *product* calculates its discounted price for another object *customer*. A first issue to be solved is that the discount is inferred by the logic rules, which somehow have to be triggered. This is depicted by the black arrow starting from *product*. Secondly, when triggered, the logic rules need information from the objects in order to infer information. For example, one of the rules needs to invoke a method of the customer object that establishes whether *customer* has a charge card or not (denoted by the black arrows). Therefore, a third issue is that the logic rules need to access the object to begin with (denoted by the white arrows). Lastly, the result of the inference of the logic rule needs to be accessible in Smalltalk again. For example,

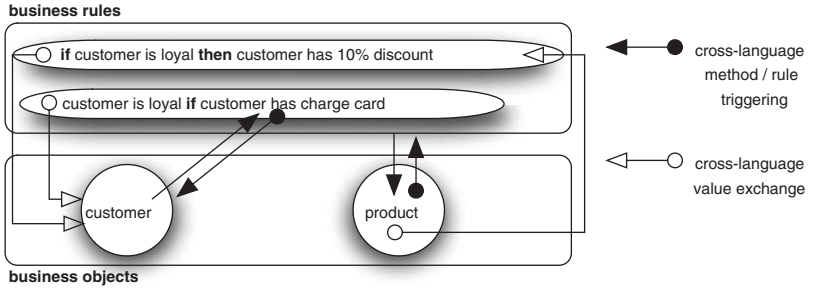

Fig. 4. Illustration of issues in defining a symbiosis between a logic and object-oriented language.

SOUL ⟷ Smalltalk

```
discount(?customer, ?product, 10) if
  loyal(?customer).
discount(?customer, ?product, 15) if
  student(?customer),
  educationalProduct(?product).

loyal(?customer) if
  [ ?customer hasChargeCard ].

educationalProduct(?product) if
  equals(?sectionitems, [ Section education products ]),
  [ ?sectionitems includes: ?product ]

...
```

```
Product>>priceFor: aCustomer
 | discounts |

 discounts :=
  (SOULEvaluator
      evaluate: 'discount(?customer, ?product, ?discount)'
      withArguments:
        (Array with: #customer -> aCustomer
                  with: #product -> self))
      valuesForVariable: #discount.
 ^ price * (100 - discounts max) / 100
```

Fig. 5. Actual implementation of the business rule example in Smalltalk and SOUL.

the object *product* needs to refer to the inferred *discount* of *customer*, as depicted by the white arrow starting from product.

Fig. 5 shows the code for the example in SOUL and Smalltalk. In SOUL (left part of the figure), two rules are implemented for a predicate `discount`, and a third rule implements the `loyal` predicate used in the first rule for `discount`. The discount rules are triggered from the Smalltalk class `Product`'s method `priceFor:` by sending a message `evaluate:withArguments:` to the class `SOULEvaluator`. The rules get access to the *customer* and *product* objects by having them passed to the message to the `SOULEvaluator` together with an array that specifies to which logic variables the objects should be bound. The result is made accessible in Smalltalk as an object with all the possible results for the logic variable `?discount`, these solutions are accessed by sending the message `valuesForVariable:` which returns a collection with all the solutions, this collection is assigned to the variable `discounts`. The rule for the `loyal` predicate illustrates how rules can access information from the objects. This is done using a Smalltalk term, an expression that is syntactically like a Smalltalk message expression enclosed in square brackets, but it is not just a regular Smalltalk expression as it can contain logic variables to pass values from logic rules to Smalltalk methods. The only condition in the `loyal` rule is a Smalltalk term, specifying that the message `hasChargeCard` sent to the `?customer` object should return the boolean true. The `educationalProduct` rule illustrates more advanced accessing of information from the objects: in the first condition of the rule, the logic variable `?sectionitems` is bound to the result of a Smalltalk message which gets a collection of all the products of the educational section, the second condition specifies that this collection should include the product.

5.2. Data mapping

Passing objects from Smalltalk to SOUL: Objects are generally passed to SOUL by invoking logic queries from Smalltalk and passing the objects as arguments to the query. While it is possible for logic rules to create new Smalltalk objects by sending instance creation messages to classes, this is not generally used as a logic query should normally not have any side effects.

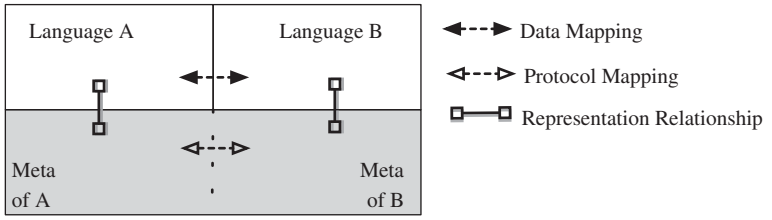

Fig. 6. Linguistic symbiosis between two languages A and B at the meta level: A and B have meta-level representations that have different protocols that need to be bridged.

Passing logic data from SOUL to Smalltalk: SOUL logic data can be passed to Smalltalk as arguments in Smalltalk terms, and appears in Smalltalk as objects. Message sends on these objects are simply mapped to data access operations on the logic data, thus mostly only accessor messages can be sent to these objects.

6. Linguistic symbiosis at the meta level

As explained in Section 3, linguistic symbiosis involves a data mapping at the base level which is implemented as a protocol mapping at the meta level: to ensure that the interpreter of one language can apply its meta operations to the meta representations coming from the other interpreter, the protocols of these representations need to be mapped to each other. This is illustrated in Fig. 6: data of languages A and B is represented at the meta level, and on this meta level the differences of protocol between the representations need to be resolved.

As before we now show how this is accomplished in Agora and Java on the one hand, and in SOUL and Smalltalk at the other hand. The choice of the meta language in which these interpreters are written does not really matter for demonstrating how the protocol mapping at the meta level works. We will however also show in the next section how the conceptual model we explain here is used in actual implementations. In that case one of the two languages is actually implemented in the other and there is not a clear separation between the meta level and base level. To clearly show the difference in how the mappings occur then, we already in this section use one of the two base languages on the meta level as well: Java in the first case, Smalltalk in the second. The important point is that there is a clear separation of the base and meta levels, and that a common language is used on the meta level for the two base level languages. This is explained in further detail at the start of the next section.

6.1. Protocol mapping for Agora and Java

On the meta level, there are two interpreters, one for Agora and one for Java. As we have assumed Java to be the meta level language for the implementation of these interpreters as well, these interpreters are written as a number of cooperating objects of different classes. Two important classes are the ones that implement the base level objects themselves: a class `JavaObject` and a class `AgoraObject`. Instances of these classes are thus meta level objects that represent base level objects.

Each of the two classes of meta objects understands a fairly similar protocol that implements the message sending of the base level. Both `JavaObject` and `AgoraObject` have methods that are the implementations of base level message sending. Of course, this protocol is similar but not entirely the same: the class `JavaObject` supports the meta operation `send(JavaMessageName, Array[JavaType], Array[JavaObject])` while the class `AgoraObject` supports the meta operation `send(AgoraMessageName, Array[AgoraObject])`.[1]

As shown in Fig. 7, the data mapping of the base level can be split in *left* and *right* appearance relationships which allow base language data of one language to appear in the other language. On the meta level, there are meta representations for this base data, and the *left* and *right* relationships of the base level require equivalent protocol mapping relationships at the meta level. A clean equivalent relationship and way of implementing the symbiosis is to introduce wrapper classes that take care of mapping the protocol differences: in the case of Agora and Java, a class

[1] Of course, other implementations are possible as well, the ones chosen here simply illustrate the point that there is an inherent difference that requires a mapping from one to the other to enable symbiosis between the two languages.

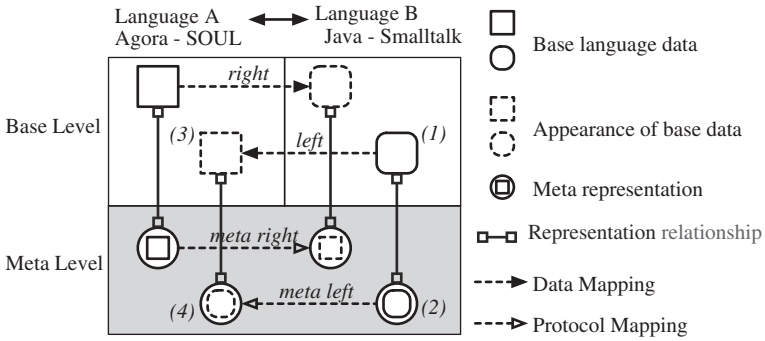

Fig. 7. Linguistic Symbiosis in more detail, focussing on the left and right appearance relationships and their equivalent relationships on the meta level.

`JavaWrappedAgoraObject` and a class `AgoraWrappedJavaObject` can be introduced. Instances of these classes, respectively, wrap around an `AgoraObject` instance and support the `JavaObject` protocol, or wrap around a `JavaObject` instance and support the `AgoraObject` protocol. So for example in the figure, the base level Java object labeled (1) is represented by the meta object labeled (2) and appears in Agora as an Agora object (3), which is implemented as an `AgoraWrappedJavaObject` wrapper around the `JavaObject` instance (2). One desirable property of the *left* and *right* relationships is that they cancel each other out: applying the *right* relationship to wrapped meta object produced by the *left* relationship should yield the original meta object, and vice versa.

The protocol mappings used on the meta level between Agora and Java meta objects should have the following effect on the base level for messages between Agora and Java objects:

Sending messages from Agora to Java objects: In Agora variables, Java objects appear as regular Agora objects, and can be sent messages in the same way as other Agora objects. For this to work a mapping is needed that maps messages sent to Java objects in Agora to Java messages, taking into account the different syntax and semantics used for messages in the two languages. The semantic difference is that the name of a message in Agora uniquely identifies a method for a specific receiver object, while in Java it does not due to the possibility for overloading. The syntactic difference is that Agora messages consist of multiple keywords after the fashion of Smalltalk, while Java messages consist of a single name. The solution adopted is to construct an Agora message from a Java message by using the type of each argument as the name for the corresponding keyword, with the exception of the first keyword which consists of the name of the Java message together with the type of the first argument. The solution for the semantic differences thus at once also provides one for the syntactic difference.

Sending `addComponent:` to `frame` is an example of an Agora message constructed from a Java message. The Java class `Frame` has at least two `add` methods: one which takes a single argument with static type `Component` and one which takes a single argument with static type `PopupMenu`. As we wish to add a "Component" in the example, the resulting mapped message that is sent from Agora to the Java object in the `frame` variable is `addComponent:`, a message consisting of a single keyword which is the concatenation of the Java message name `add:` with the name of the type of the first argument.

The same mapping is used for invoking constructors on classes, with the difference that the special name "new" is used for the constructors as they are nameless in Java. Thus to create an instance of the Java `Button` class from within Agora using the constructor that takes a `String` as an argument, the message `newString:` can be sent to the `Button` class from within Agora.

Sending messages from Java to Agora objects: When contained in Java variables, Agora objects can be sent messages by Java objects in the same way as the latter would send messages to other Java objects. A Java message sent to an Agora object is constructed from an Agora message in the inverse way as described above.

A critical point in the mappings performed by the protocol wrappers is to ensure that the appropriate left and right relationships are applied when mapping arguments of one protocol to the other. When a `JavaWrappedAgoraObject` maps a Java `send` operation to the Agora `send` operation, the arguments involved in the message send are Java meta objects which also need to be converted to Agora ones. Thus, the mapping done by this wrapper semi-formally comes down to what is shown in Fig. 8.

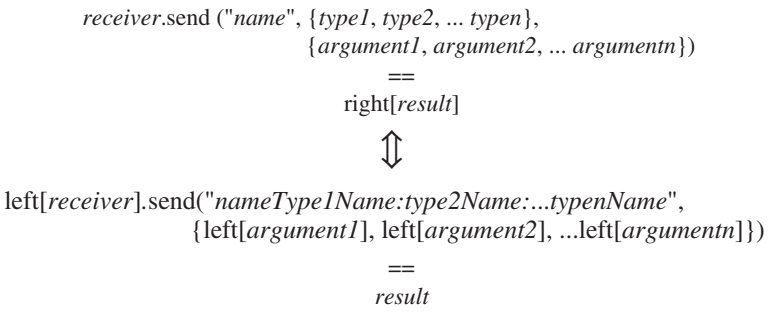

Fig. 8. Semi-formal description of meta operation mapping for the send operation from Java to Agora.

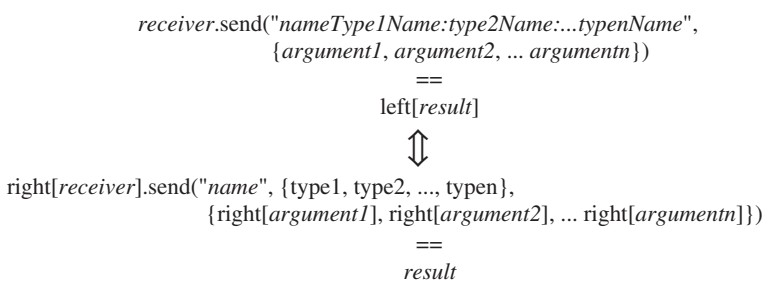

Fig. 9. Semi-formal description of meta operation mapping for the send operation from Agora to Java.

The rule simply describes the same protocol mapping solution for sending Java messages to Agora objects as described above, but illustrates the point of needing to convert the receiver and arguments to Agora objects. Applying the left relationship on the receiver, which in this case is the `JavaWrappedAgoraObject` wrapper, simply results in the unwrapped Agora meta object. Similarly, the left relationship applied to the arguments either wraps them or unwraps them, depending on whether they were wrapped Agora meta objects produced by the right relationship, or plain Java meta objects in the first place. As also illustrated, the result of the mapped message also needs to be mapped back using the *right* relationship to turn it from an Agora object into a Java object.

The converse rule for mapping Agora messages to Java messages is very similar and can be given without further explanation as illustrated in Fig. 9, note that this rule is easily derived from the rule above using the fact that the *left* and *right* relationships cancel each other out (i.e. $left[right[x]] = x$).

6.2. Protocol mapping for SOUL and Smalltalk

Unifying objects: In an interpreter for a logic language like SOUL, unification is a particularly important operation that is applied on the meta representations for logic rules, logic functors and other logic terms. In the actual implementation of SOUL, there is a class `AbstractTerm` from which all classes for representing the different kinds of logic terms—functors, variables and lists—inherit. The `AbstractTerm` class defines an abstract method `unifyWith:inEnvironment:` which the other classes implement accordingly. The method is passed another object that is the meta representation of another logic functor or list etc. and an environment of logic variable bindings. The method on meta objects representing logic functors for example checks if the other object also represents a logic functor, and then recursively sends the same unification messages to the different objects representing the arguments of each of the two functors. A meta object representing a logic variable responds to the message by checking whether the environment already holds a binding for it. If not it simply adds a new binding with the meta object it is being unified with as the binding's value. If there already is a binding, the logic variable object again recursively sends the unification meta message to the existing binding's value with the same arguments it received for the message itself, which are the environment and the logic meta object it should unify with. Thus unification is implemented by a protocol of recursive

message sending, and the meta objects passed as arguments to the unification message are expected to support this protocol.

An interpreter for Smalltalk on the other hand has meta representations for objects and classes. The meta objects representing Smalltalk objects need to support a protocol for sending messages, accessing instance variables etc. The meta objects representing classes need to support a protocol for looking up methods, defining instance variables etc. Thus, in an interpreter for Smalltalk, there would be a class `AnObject` whose instances would represent objects at the base level. The `AnObject` meta objects would understand messages such as `sendSelector:withArguments:`.

Thus, the base-level *left* appearance relationship which allows a base-level Smalltalk object to appear in SOUL, needs an equivalent on the meta level which maps the unification protocol to the message send protocol. The particular solution chosen in SOUL for this issue is to allow objects to unify when they are equivalent according to the equivalency message `=`. One way to implement the left relationship on the meta level then, is to put a wrapper around Smalltalk meta objects with the wrapper mapping the unification operation to the message send operation. In SOUL, this wrapper is `SmalltalkObject`, a subclass of `AbstractTerm`.

As with similar mappings in Java and Agora, care must again be taken to perform the appropriate *right* relationship on any meta objects involved in the mapping. Thus, when a `SmalltalkObject` wrapper receives a `unify:` message, it maps this operation to the message send operation `send:withArguments:` where the message that is sent is `=`. Both the receiver of this operation and the argument that it is passed need to support the message send protocol of Smalltalk meta objects, thus the *right* relation needs to be applied before applying the message send operation. Semi-formally, the protocol difference mapping that happens on the meta level is:

lo1 `unify:` *lo2*

⇕

right[l*o1*] `send: #=withArguments:` { right[*lo2*] }

Sending messages from SOUL to Smalltalk objects: Message sending is again not an operation native to logic programming, but depending on the type of the message it can be mapped to either of two operations of logic programming. Boolean messages can be naturally mapped to proving of conditions in logic rules, while accessing data from objects by means of accessor or other side-effect-less methods can be mapped to a part of unification. Invoking mutator and other methods with side-effects does not make sense from the logic paradigm.

The mapping of the first two message types is handled in SOUL with the same linguistic construct: the Smalltalk term. A Smalltalk term is a novel linguistic construct that was added to SOUL specifically for symbiosis: as shown in the rule for the predicate `loyal` in the example, a Smalltalk term is denoted by square brackets ("`[`" and "`]`") and contains a message send in Smalltalk syntax but involving logic variables. A smalltalk term can be used as a condition in a rule, as also shown in the rule for the `loyal` predicate, in which case the condition proving operation of logic programming maps it to the evaluation of the message of the Smalltalk term; the message is expected to return a boolean which is then mapped to the success or failure of proving the condition. A Smalltalk term can also be used as an argument in a condition, or as part of other compound data structures of logic programming such as lists, in which case the unification operation maps it to execution of the message: when a Smalltalk term is unified with another logic data construct, this is first mapped to evaluating the message in Smalltalk and the resulting object is then again unified with the other logic data construct using the process described in the previous point.

Invoking SOUL logic queries from Smalltalk: The invocation of logic queries is mapped to a message send to a `SOULEvaluator` class in Smalltalk. The `SOULEvaluator` class supports a message `evaluate:withArguments:` which can be passed a query as a string. The second argument of this message is an array which is used to specify which objects are passed to the query for which logic variable: the array should contain associations of names of logic variables to objects. The message finds all solutions for the given query in SOUL.

The result of the `evaluate:withArguments:` message requires another mapping because of a particular difference between SOUL and Smalltalk: logic queries can have several "output" variables and furthermore can result in multiple different results for these variables, while Smalltalk messages can only return a single object. The results of the query are therefore mapped to a single object that understands a message `valuesForVariable:` which expects as argument the name of a logic variable used in the query. The result of this message is the solutions of the query for that variable, mapped to a collection object.

Accessing SOUL data from Smalltalk: In logic programming, data are accessed from compound structures such as functors and lists through unification, this needs to be mapped to Smalltalk's accessor messages for accessing data.

When a SOUL value is passed to Smalltalk, it appears in Smalltalk as an instance of the equivalent Smalltalk class for that type of SOUL value: lists appear as instances of `OrderedCollection`, numbers as `Number` instances etc. Functors are mapped to instances of the SOUL-specific class `CompoundTerm`, which supports messages for accessing the functor's name and its arguments.

7. Inter-language reflection in actual implementations

Our conceptual model for inter-language reflection and linguistic symbiosis is readily applicable to actual implementation schemes where the two languages in symbiosis are implemented as interpreters in a third common implementation language. There are however two differing schemes possible, and in this section we explain how the conceptual model maps to these schemes. The first possible variation is that the interpreters are not written in a common implementation language. The second variation is that the interpreter of one language is written in the other language, and that a linguistic symbiosis is defined between the first language and its implementation language rather than with a language that is also implemented in that implementation language.

As noted earlier, the first variation simply shifts the problem of achieving a linguistic symbiosis one level down. A key point in our conceptual model is to explicitly take into account the meta level for both of the two base level languages, and to assume that at the meta level there is a common implementation language. This allows us to clearly show how data mapping at the base level comes down to protocol mapping at the meta level: while the meta representations of each language can already be exchanged between the two interpreters because of the common language, they support a different protocol of meta representations which needs to be mapped. In the actual implementation variation where the meta representations of one language are written in language X and those of the other in language Y, there needs to be a linguistic symbiosis between X and Y to allow the meta representations to be interchanged in the first place.

The second variation essentially entails that the meta level of one language is made to overlap the base level of the other language. As illustrated in Fig. 10, the interpreter for the one language is written in the other language, and there is no interpreter for the other language at this level. One reason for this variation is that in practice it is typically easier to implement a new language in the one with which it should be in an inter-language reflection relationship, rather than in a common language, or that such an implementation already exists and there is a need to allow for inter-language reflection. Note that while we already used one of the base languages as meta language as well in the explanation of the conceptual model in Section 3, we still made a distinction between the meta level and the base level. The variation we are referring to here is that, as illustrated in Fig. 11, the meta representations of one language—SOUL in the figure—exist on the same level as the values of the other language. This deviation of the conceptual model has an effect on how the linguistic symbiosis and inter-language reflection are actually implemented, as we will discuss in more detail in the remainder of this section.

7.1. Linguistic symbiosis implementation

One effect of the base and meta level overlap is that the *right* relationship maps a SOUL (or Agora) value *directly* to a wrapper. Contrast this with the pure conceptual model of Fig. 7, where the *right* relationship allows a SOUL value to appear in Smalltalk, and this appearance is implemented as a wrapper around the meta object representing the value. Here, the *right* relationship maps directly to the wrapper. Furthermore, this wrapper is a base level Smalltalk object,

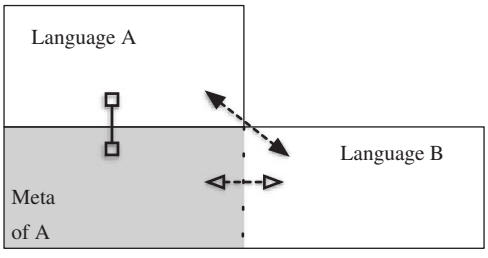

Fig. 10. Implementation of inter-language reflection and language symbiosis.

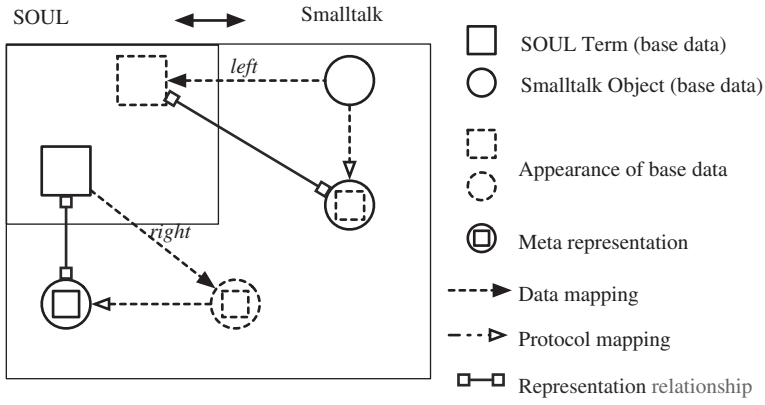

Fig. 11. Folding of inter-language reflection and language symbiosis in the actual implementation of SOUL in Smalltalk.

receiver.name(*argument1, argument2, ... argumentn*)
==
right[*result*]
⇕
left[*receiver*].send("*nameType1Name:type2Name:...typenName*",
{left[*argument1*], left[*argument2*], ...left[*argumentn*]})
==
result

(where {type1 ... typen} are the static types of the arguments)

Fig. 12. Semi-formal description of base and meta level operation mapping in actual implementations with overlap of base and meta levels.

rather than a meta level object as in the pure conceptual model. Thus the wrapper translates *base* level Smalltalk (or Java) messages to *meta* operations on the SOUL (or Agora) meta object.

The important point to note about this difference in how the mappings work is that the mappings in the conceptual model more clearly show the protocol difference that is being solved. This is the reason for clearly separating the base level and meta level in the conceptual model. For example, the mappings in the conceptual model show that the Java message send operation takes the static type of the arguments into account, while the one for Agora does not. Because in the actual implementation, wrappers map between base level and meta level operations, this difference is no longer as obvious. The mapping performed by the *right* wrappers in the actual implementation for Agora and Java is for example the one given in Fig. 12. Note again that the *base* level Java message with name `name` is mapped to the Agora meta operation `send`, while previously it was the Java *meta* operation `send` that was mapped to the Agora meta operation `send`.

The *left* relationship is similarly affected. Making a Smalltalk object appear in SOUL for example involves wrapping the Smalltalk object in a wrapper that maps the SOUL meta protocol to the Smalltalk base level, instead of as in the conceptual model where it maps it to the Smalltalk meta protocol.

7.2. Inter-language reflection implementation

The effect of the overlap in the actual implementations is also of folding the inter-language reflection of one direction. To reflect about one language from within the language that implements it does not require getting a reflective representation in the one language and passing it through symbiosis to the other. So in the case of SOUL in Smalltalk, there is no need for the (base level) Smalltalk to access a reflective representation of SOUL by having SOUL reflect on itself and passing the reflective representation by linguistic symbiosis to Smalltalk. Instead, for Smalltalk objects to reflect on SOUL only requires them to directly access the Smalltalk objects representing SOUL *without* applying the *right* relationship. This can actually be another reason for deviating from the conceptual model in the actual implementation.

As it is thus for one language not necessary to be reflective. Note however, that in the other direction where SOUL is used for inter-language reflection about Smalltalk the combination of linguistic symbiosis and traditional reflection is still used to achieve this inter-language reflection.

8. Examples

A very simple example of using inter-language reflection in SOUL about Smalltalk is one for generating accessor methods on a class. While simple, the example nevertheless clearly shows the use of linguistic symbiosis for inter-language reflection. More extensive examples can be found in previous publications on the use of SOUL [1,2,6,21,23]. The following rule defines what the source for an accessor method for a variable of a class should be:

```
accessorMethod(?class, ?varname, ?source) if
      hasInstanceVariable(?class, ?varname),
      equals(?source, [ ?varname , '^', ?varname ])
```

The first condition of the above rule specifies that `?varname` should be the name of an instance variable of the class `?class`. The predicate `hasInstanceVariable` is one from a library that comes with SOUL with several such predicates, the rules for this predicate are not shown here, but we can note they make use of linguistic symbiosis to retrieve the instance variables from the class. The second condition of the above rule specifies that the `?source` variable should contain a string which is the concatenation of the variable name, with a return expression that returns the variable's value. Note that this concatenation is defined using an argument to the `equals` predicate where linguistic symbiosis is used: the value in `?varname` is *right* mapped to Smalltalk, which in this case will result in a string, to which then the concatenation message "," is sent, the resulting string is *left* mapped back to SOUL.

This rule can then be used in a query to actually generate all the accessor methods for a class, for example by finding all solutions to the query below one can generate all accessor methods on the `BankAccount` class:

```
if accessorMethod([BankAccount], ?variable, ?body),
   [?class compile: ?body. true]
```

The above query shows two things. First of all, it clearly shows the use of the combination of traditional reflection and linguistic symbiosis for doing inter-language reflection: the `compile:` message that is sent is a reflective message in Smalltalk and it is sent from SOUL using linguistic symbiosis.

Secondly, the query shows that both introspection and intercession using inter-language reflection about Smalltalk is possible from SOUL. While doing side-effects breaks the declarative nature of logic programming and is therefore usually not recommended, there is no inherent restriction in SOUL that prevents one from accessing Smalltalk's intercessory reflection interface. Thus it is possible to send such messages as `compile:` and other messages that change the Smalltalk program, as well as all the messages for simply querying the Smalltalk program for lists of instance variables of a class, or its subclasses etc.

9. Related work

The term "linguistic symbiosis" was previously defined in the work on RBCL [24]. RBCL is a language implemented in C++ which also has linguistic symbiosis with C++. This is used to allow RBCL base level objects to interact with, and take the place of the RBCL meta objects defined in C++, thus achieving what we have dubbed traditional reflection. There are three important differences with our work. Firstly, the use of linguistic symbiosis as a mechanism for achieving inter-language reflection was not considered. Secondly, the languages between which symbiosis was defined were not fundamentally paradigmatically different as in our case for SOUL and Smalltalk. Lastly, in RBCL the symbiosis was directly defined in the implementation with overlap of the RBCL meta level and C++ base level, there was no consideration of the conceptual model we introduced where the linguistic symbiosis is defined through a common meta level for the two languages which allows a clearer modeling of which meta operations need to be mapped and how.

While Agora was one of the first languages to be implemented and integrated in Java, the popularity of the platform has brought about numerous languages which are hosted in Java. In most cases, linguistic symbiosis and our model for it was not explicitly considered. In several cases, the integration is asymmetric: from the hosted language, Java classes can be instantiated and the instances can be sent messages, but only Java objects can be passed as arguments, not objects implemented in the hosted language (i.e., the integration is parasitic instead of symbiotic). In other cases, the integration is not fully asymmetric, but the use of values from the hosted language in Java is not transparent. For logic languages integrated in Java, an extensive survey illustrating these problems is given by D'Hondt [25].

The integration of Piccola in Java (JPiccola) and also in Smalltalk (SPiccola) is very interesting in this respect, as it is an exceptional case in which a concept similar to linguistic symbiosis, "inter-language bridging", was considered for achieving the integration [26]. However, while in the definition of "inter-language bridging" the meta level of both languages is also considered, the meta and base levels are not clearly separated which has resulted in an asymmetric integration. The meta operations applied by the interpreters are not explicitly considered: the definitions of the *up* and *down* operations, which are, respectively, equivalent to our *left* and *right* operations, is only given in terms of base level data mapping rather than as protocol mapping at the meta level. The *up* operation maps a Smalltalk object to a Piccola "form" with a "service"—the behavior of the form—for each method of the object. The *down* operation is described as a simple mapping in which "the form itself is passed down to the host language". This actually has the semantics of mapping a Piccola form to its meta representation in Smalltalk, the services of the form are thus not mapped to Smalltalk methods and a Piccola form cannot be sent messages from Smalltalk to invoke the services as if the form is a regular Smalltalk object. The *down* operation thus confuses the base and meta levels and actually is a folding of the linguistic symbiosis and inter-language reflection operations as we have described in Section 7. The integration is thus asymmetric as it is not possible to implement a form in Piccola which when passed "down" behaves as an existing Smalltalk object, while in the other direction it is possible to implement a Smalltalk object which when "upped" behaves the same as a native Piccola form. An interesting aspect of Piccola is that it allows reflective control of the *up* operation: instead of simply returning the "upped" object, the *up* operation can be modified to return a Piccola form which has a field "peer" with the actual "upped" object. This can be used to give the "upped" object a Piccola-specific interface by implementing services that appropriately forward to the peer. However, the *up* and *down* operations behave differently on such forms and do not have the desirable property of cancelling each other out (cf. Section 6.1): applying *down* on such a form results in the "downed" peer, when that is again "upped" it is not the original "downed" form. This resulted in the need for a reflective control over the *down* operation as well, so forms can be explicitly "protected" in certain cases when they are downed [26].

A specific goal of the .NET platform is "language inter-operability". For this, a common intermediate language was defined, to which all languages supported on the platform are compiled. The common intermediate language is actually a more primitive form of the major language of the platform, C#. Compiling a language to .NET is thus in part a base level data mapping to integrate with C#, which does not clearly expose the meta level protocol differences of C# and the implemented language.

One can hardly talk about reflection without discussing the work that has been done in the LISP community, even though the goals of our approach are quite different. One of the very nice features of LISP is that it has a built-in mechanism to represent its language constructs: the quotation form. This provides a core meta-reasoning structure, since parts of programs can be assembled, passed around and then evaluated at will. This basic LISP functionality was extended in the well-known work on *procedural reflection* by Smith [27]. In this work, 2 languages were introduced. The first, 2-LISP, deals with quotation issues by providing two explicit user primitives to switch between representations of structures and the structures themselves. These primitives were called *up* and *down*, the equivalents of the *left* and *right* relationships used in this paper. We want to stress two problems with 2-LISP, which is that *up* and *down* needed to be called by the user whenever necessary and that *down* is not the inverse of *up*. 2-LISP was actually meant as the basis for the better-known 3-LISP, a reflective language with an implementation based on *reflective towers*.

Muller [28] has argued that the quotation form in the original definition of LISP is essentially flawed, and hence that the apply in LISP and descendants like 2-LISP and 3-LISP or even Scheme *crosses levels*. Moreover, it is this level-crossing that allows much of the meta-circular capabilities of LISP. This was remarked by Muller, and was addressed in his LISP flavor, called M-LISP. In M-LISP, the apply function does not cross levels, which removes a lot of the awkward constructions needed in 2-LISP. Moreover, up is represented by a relation R, and down by R^{-1}, where down is the inverse of up. Reification has to be introduced at the cost of equational reasoning (which is done with extended M-LISP), and, even extended M-Lisp corresponds to only a restricted 3-Lisp.

The approach taken by inter-language reflection is comparable with the approach taken by M-Lisp, except that the goals are quite different. The goal of M-Lisp (as with the other research in reflection in general) is to study 'self-extensibility' of programs, and provide formal language semantics to that end. The approach in our paper is driven from a software engineering problem, to study a symbiosis between two languages from different paradigms. In our case, *left* and *right* therefore not only bridge levels, but also language paradigm boundaries. As in M-Lisp (and contrary to 2-Lisp and 3-Lisp), *left* and *right* in SOUL are symmetric and are called automatically. Note however that in SOUL this relation is still kept fairly simple (only objects are reified as logic terms), however this is not necessarily the case. One of the important parts of future work is to have a tighter integration between languages by reifying more object-oriented concepts, which leads to a more difficult relation.

10. Conclusion

In this paper we introduced inter-language reflection, an extension of traditional single-language reflection to two languages. We introduced a scheme for achieving inter-language reflection where traditional reflection is combined with linguistic symbiosis. Linguistic symbiosis allows programs in two languages to transparently interchange values and invoke behavior defined on these values in the other program. As traditional reflection already allows programs to invoke behavior and access values that are causally connected to its own execution process, the combination with linguistic symbiosis automatically allows programs in each language to access the traditional reflective interface of the other language.

We introduced a conceptual model for linguistic symbiosis where, in contrast with previous work on linguistic symbiosis, we made the meta level of both base level languages explicit. This model thus allows us to clearly show how achieving linguistic symbiosis on the base level comes down to solving protocol differences between the meta operations applied on the meta level. On the meta level, different meta operations are applied to the meta representations for each language, and allowing the values on the base level to be interchanged requires making the meta operations for one language applicable to the meta representations of the other. We illustrated this for two cases of linguistic symbiosis, that of SOUL with Smalltalk, and of Agora with Java.

We showed how in actual implementations of inter-language reflection and linguistic symbiosis, the clear separation made in the conceptual model between base level and meta level is typically abandoned. This simplifies the actual implementation of inter-language reflection as the base level of one of the two languages can directly access the meta representations of the other. We showed how this affects actual implementations of linguistic symbiosis to involve wrappers that map base and meta operations rather than meta operations as in the conceptual model. But this folding of one meta level makes the linguistic symbiosis mechanism harder to understand. The simpler conceptual model helps to see the mechanisms in their pure form and subsequently to understand the 'shortcut' taken in typical implementations.

References

[1] Wuyts R. Declarative reasoning about the structure object-oriented systems. In: Proceedings of the TOOLS USA '98 conference. Silver Spring, MD: IEEE Computer Society Press; 1998. p. 112–24.

[2] Mens K, Wuyts R, D'Hondt T. Declaratively codifying software architectures using virtual software classifications. In: Proceedings of TOOLS-Europe 99; 1999. p. 33–45.

[3] Florijn G, Meijers M, van Winsen, P. Tool support for object-oriented patterns. In: Aksit M, Matsuoka S, editors. Proceedings ECOOP '97, Lecture notes in computer science, vol. 1241. Jyvaskyla, Finland: Springer; 1997. p. 472–95.

[4] Spoon SA, Shivers O. Demand-driven type inference with subgoal pruning: trading precision for scalability. In: Proceedings of ECOOP'04; 2004. p. 51–74.

[5] Minsky NH. Law governed regularities in software systems; 1994.

[6] Wuyts R, Mens K. Declaratively codifying software architectures using virtual software classifications. In: Proceedings of TOOLS-Europe 1999; 1999.

[7] Crew RF. Astlog: a language for examining abstract syntax trees. In: Proceedings of the USENIX conference on domain-specific languages; 1997.

[8] Johnson SC. Lint, a C program checker. Computing Science TR 1977;65.

[9] Ferber J. Conceptual reflection and actor languages. In: North-Holland PM, Nardi D, editors. Meta-level architectures and reflection; 1988. p. 177–93.

[10] Bobrow D, Gabriel R, White J. Clos in context—the shape of the design. In: Paepcke A, editor. Object-oriented programming: the CLOS perspective. Cambridge, MA: MIT Press; 1993. p. 29–61.
[11] Kiczales G, des Rivières J, Bobrow DG. The art of the metaobject protocol. Cambridge, MA: MIT Press; 1991.
[12] Maes P. Concepts and experiments in computational reflection. In: Proceedings OOPSLA '87, ACM SIGPLAN notices, vol. 22; 1987. p. 147–55.
[13] Maes P. Computational reflection. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium; 1987.
[14] Rivard F. Reflective Facilities in Smalltalk, Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février; 1996.
[15] Ducasse S. Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) 1999;12:39–44.
[16] Ungar D, Smith RB. Self: the power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN notices, vol. 22; 1987. p. 227–42.
[17] Meuter WD. Agora: the story of the simplest mop in the world. In: Prototype-based programming. Berlin: Springer; 1998.
[18] Wuyts R. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. PhD thesis, Vrije Universiteit Brussel; 2001.
[19] Wuyts R, Ducasse S. Symbiotic reflection between an object-oriented and a logic programming language. In: ECOOP 2001 international workshop on multiparadigm programming with object-oriented languages; 2001.
[20] Gybels K. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. Licentiate's thesis, Vrije Universiteit Brussel; 2001.
[21] Roover CD, Gybels K, D'Hondt T. Towards abstract interpretation for recovering design information. Electronic notes in theoretical computer science, vol. 131; 2005. p. 15–25.
[22] D'Hondt M, Gybels K. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In: Proceedings of the 19th annual ACM symposium on applied computing (SAC 2004), special track on object-oriented programming, languages and systems. New York: ACM Press; 2004.
[23] Fabry J, Mens T. Language-independent detection of object-oriented design patterns. Computer Languages, Systems and Structures 2004;30(1–2):21–33.
[24] Ichisugi Y, Matsuoka S, Yonezawa A. Rbcl: a reflective object-oriented concurrent language without a runtime kernel. In: IMSA'92 international workshop on reflection and meta-level architectures; 1992.
[25] D'Hondt M. Hybrid aspects for integrating rule-based knowledge and object-oriented functionality. PhD thesis, Vrije Universiteit Brussel; 2004.
[26] Schärli N. Supporting pure composition by inter-language bridging on the meta-level. Master's thesis, Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern; 2001.
[27] Smith B. Reflection and semantics in lisp. In: Proceedings of the 11th symposium on principles of programming languages; 1984. p. 23–35.
[28] Muller R. M-LISP: a representation-independent dialect of LISP with reduction semantics. ACM Transactions on Programming Languages and Systems 1992;14:589–616.