The OSUIF Library

`osuif@cs.ucsb.edu`
Department of Computer Science
University of California, Santa Barbara

November 23, 1999

Abstract

Object SUIF (OSUIF) is an extension to SUIF2, which provides support for the representation of object-oriented languages. It can be viewed as a layer on top of the SUIF2 system.

DRAFT NOTICE

This document and the implementation it describes are works in progress. We cordially invite suggestions on design, on implementation or documentation style, and on any other relevant issues, profound or picky.

Contents

1 Some Conventions

In general, we use Java's terminology to talk about (generic) object-oriented concepts.

1.1 The Family Way

We refer to classes as *parent* and *child* classes if they are immediate ancestors or descendants. We use the more general terms *ancestor* and *descendant* for the relationship of uncertain (and possibly null) degree. Thus, surprisingly, a class is considered its own ancestor of null degree but not its own parent. This convention is reflected in the names of fields and methods.

1.2 Names

We generally use verbs as names for boolean functions, methods and fields, and nouns as the names for everything else.

2 Motivation

SUIF is well suited to capture high-level information for code that has been translated from an imperative programming language, but it lacks support for object-oriented languages.

Object-oriented languages require additional—or completely different—high-level information than imperative languages. OSUIF defines a small number of specific object-oriented constructs (e.g., support for dynamic dispatch) that retain the high-level information of an object-oriented source language as long as necessary. Once this information is no longer needed (i.e., all optimization passes for object-oriented features have been run), OSUIF code is lowered to Standard SUIF code so that all generic SUIF passes can be used for the remaining compilation process.

While it is certainly possible to write a plain SUIF front end for an object-oriented language, such a compiler would not serve as a good vehicle for object-oriented compiler research. For example, if one wants to express a dynamic method call in SUIF, a concrete implementation technique has to be chosen and implemented with the available SUIF instructions. Thus, high-level information is lost that could have been used for code optimization (e.g., class hierarchy analysis or type inference). OSUIF overcomes this problem by introducing a new, high-level IR constructs that model dynamic method calls, classes, class hierarchies, etc. This approach is analogous to the high-level array instructions that SUIF already offers to make array accesses explicit.

3 Introduction

OSUIF's goal is to extend SUIF with object-oriented features, thus allowing to express object-oriented behavior in SUIF and to provide a framework that makes it possible to share passes that are specific to object-oriented languages. To give a first impression, we briefly list the features provided by OSUIF:

- classes with static and instance members.
- dynamic method calls.
- object creation and destruction.
- relationships between classes.
- exception handling.

- generic lowering and object-oriented optimization passes.
- accurate garbage collection (planned).
- type sets (planned).

Unfortunately, object-oriented languages are very diverse and hence it is hard to design a consistent, common framework that encompasses all languages. For example, different object-oriented languages usually have a very different notion of inheritance, subtyping, subclassing, and dynamic binding. In contrast, imperative languages have a fairly uniform semantic and type system. For example, SUIF accommodates both C and Fortran in a single type system (and support for Pascal and Modula-2 could probably be realized without any extensions to the existing type system). Thus, owing to the diverse nature of object-oriented languages compared to imperative ones, OSUIF takes a different approach.

Instead of trying to contain the union of all object-oriented languages, OSUIF provides an extensible framework, which abstracts away the details of a particular language. Each front end must extend the framework in order to represent its source language. A Java compiler, for example, would subclass from (abstract) OSUIF classes in order to implement front-end–specific functionality. The front end can also make use of certain functionality that OSUIF already provides, such as a generic pass that builds virtual function tables for single inheritance.

To summarize, OSUIF clients have to extend the framework in order to model the semantics of a specific source language, and then build a front end that depends on these extensions.

4 OSUIF hoof

To model object-oriented features, OSUIF introduces the (abstract) concept of classes, methods, and fields by defining new IR nodes.

The OSUIF IR nodes are described with hoof. Hoof is a domain-specific language to express SUIF IR nodes. This section describes the OSUIF hoof file.

4.1 Overview

OSUIF extends the SUIF2 IR nodes in the following areas:

- `Types`: to express types for classes, methods, and fields.
- `Symbols`: to express symbols for fields and methods.
- `SymbolTable`: to model class-specific behavior for member lookup.
- `Statements`: to express exception handling and object-oriented instructions.

The following sections discuss these extensions in more detail.

4.2 Module `osuif`

OSUIF subclasses from IR nodes defined in `basicnodes` and `suifnodes`.

5 ⟨* 5⟩≡

```
#include "basicnodes/basic.hoof"
#include "suifnodes/suif.hoof"

module osuif{
  include <iostream.h>;
  include "iokernel/cast.h";
  include "basicnodes/basic.h";
  include "suifnodes/suif.h";

  import basicnodes;
  import suifnodes;

⟨osuif class declarations 6⟩
}
```

OSUIF defines the following IR nodes:

6 ⟨osuif *class declarations* 6⟩≡ (5)

```
⟨class MethodType 8a⟩
⟨class StaticMethodType 8b⟩
⟨class InstanceMethodType 9a⟩
⟨class InstanceMethodSymbol 9c⟩
⟨class InstanceFieldSymbol 9b⟩
⟨class StaticFieldSymbol 10a⟩
⟨class StaticMethodSymbol 10b⟩
⟨class PerClassSymbolTable 11a⟩
⟨class InstanceFieldSymbolTable 11b⟩
⟨class InstanceMethodSymbolTable 11c⟩
⟨class ClassType 12⟩
⟨class InheritanceLink 14c⟩
⟨class InstanceMethodCallStatement 15a⟩
⟨class StaticMethodCallStatement 15b⟩
⟨class NewStatement 15c⟩
⟨class DeleteStatement 15d⟩
⟨class CheckStatement 16a⟩
⟨class CatchStatement 16b⟩
⟨class TryStatement 17a⟩
⟨class ThrowStatement 17b⟩
⟨class ThrowsAnnote 17c⟩
```

4.3 General Concepts

Several OSUIF classes share common concepts. Instead of repeating these concepts for every class individually we discuss them here up front.

4.3.1 Semantic Attributes

Several OSUIF IR nodes contain an *attribute list*. An attribute is a simple string[1] that can be used to attach front-end–specific information to the class type. For example, a Java front end can use attributes to distinguish between a Java `class` and `interface`.

[1] SUIF has a similar mechanism, which allows to add qualifications to types with the QualifiedType node. The important difference is that SUIF qualifications describe behavior of storage and are associated with types, whereas attributes describe arbitrary semantic properties and are associated with OSUIF symbols and the ClassType.

4.3.2 Owning Class Reference

Every OSUIF IR node derived from a Symbol contains a reference to its owning (or defining) class (`owning_class` field). This reference is useful if the symbol is moved out of its class scope. The standard OSUIF lowering, for example, moves static members out of the per-class symbol table. This reference makes it also possible to undo the lowering by moving a symbol back to its defining class scope ("raising").

It is not required to set this field — it can be NULL. Not setting this field means that a raising pass will not be able to move the symbol back to its owning class.

4.3.3 Type System

OSUIF provides type equivalence tests for MethodTypes and MethodSymbols with the binary method `is_equivalent()`. The standard implementation of *equivalence* is defined as follows:

Definition 4.1 (Type equivalence) *If not otherwise specified for a particular type, two types are equivalent if they are represented by the same type object (i.e., they are pointer-equivalent).*

Definition 4.2 (MethodType equivalence) *Two method types are equivalent if they have the same number of formal arguments, the same number of return types, type equivalence for all corresponding formal arguments except for the "this" reference (which is the first formal argument by convention), and type equivalence for all corresponding return types. No restriction is imposed upon the "this" reference. Note that qualifications are ignored for the equivalence test.*

This definition corresponds to non-variance of formal argument types and return types, such as specified by Java and C++.[2]

[OSUIF does not restrict the type of the "this" reference, but certain OSUIF dialects will impose restrictions on it (e.g., it must be a `void*` or a pointer to a class type).]

Qualifications are ignored for the equivalence test because they depend on the source language. In C++, these qualifications yield new types.[3] Java does not have qualifications on MethodTypes.

Definition 4.3 (MethodSymbol equivalence) *Two methods symbols are equivalent if they have identical names and if their types are MethodType equivalent.*

Note that attributes, such as `public`, `final`, and `abstract` are not part of the equivalence check. This seems to be a good default behavior because these attribute usually do not introduce a new type. In Java, a `native` method can be redefined with a non-`native` method (and vice versa) and a non-`final` method can be redefined with a `final` method (and vice versa). In C++, a `protected` method can be made `public` when it is redefined. The exceptions that a method can raise are ignored for the equivalence check. This means that both the `throw()` exception specification in C++ and the `throw`-clause in Java are not part of the method's type.

4.4 Are Constructors Static or Instance Methods?

Sooner or later a front-end has to face the question whether constructors should be treated as static or instance methods in OSUIF.

Usually constructors have the following properties:

[2] A more general scheme would allow contravariance of formal argument types and covariance of return types, but the current definition is simple and sufficient for Java, C++, and Ada-95.

[3] Actually, this is only true for method overriding, but not for `catch` handler matching.

- They have a "this"-reference like an instance method.
- They are not dispatched like a static method.

The front-end has to decide which property is more consistent with the the source language. For example, in Java an object is created by first allocating storage (which is initialized to '0') for the object and then (statically) calling the constructor of the class by passing the allocated memory as the "this"-reference. Constructors in Java are not entered in the virtual function table. Because of this behavior a front-end can treat a constructor as an InstanceMethodSymbol that is not dispatched (i.e., the `is_dispatched` field is set to false).

4.5 MethodType

The abstract MethodType class is used to represent common characteristics of both instance and static methods.

Currently no such common characteristic exits, but this might change.

This design allows us to easily add new functionality at every desired level in the OSUIF type hierarchy without breaking the interface.

The `is_equivalent()` method with two type arguments is used to compare two non-OSUIF types. Two non-OSUIF types are equivalent if they are pointer-equivalent.

8a ⟨*class* `MethodType` 8a⟩≡ (6)

```
abstract MethodType : CProcedureType {
CPP_DECLARE
public:
  virtual bool is_equivalent( Type* type1, Type* type2 );
  virtual bool is_equivalent( MethodType* mtype );
CPP_DECLARE
};
```

4.6 StaticMethodType

The type of a static method is represented with the class StaticMethodType, which is derived from MethodType.

Currently no new functionality is introduced by this class (and the MethodType superclass), which means that it is effectively a CProcedureType. This might change.

8b ⟨*class* `StaticMethodType` 8b⟩≡ (6)

```
concrete StaticMethodType : MethodType {
CPP_DECLARE
public:
  virtual bool is_equivalent( Type* type1, Type* type2 );
  virtual bool is_equivalent( MethodType* mtype );
CPP_DECLARE
};
```

4.7 InstanceMethodType

The type of an instance method is represented with the class InstanceMethodType, which is derived from MethodType. The first argument is distinguished as the receiver ("this"-reference) and associates the method with the class defining it.

This class provides a more convenient and type-safe interface for the receiver type with the `get_receiver_type()` and `set_receiver_type()` methods.

9a ⟨*class* `InstanceMethodType` 9a⟩≡ (6)

```
concrete InstanceMethodType : MethodType {
CPP_DECLARE
public:
  virtual QualifiedType* get_receiver_type();
  virtual void set_receiver_type(QualifiedType* receiver);

  virtual bool is_equivalent( Type* type1, Type* type2 );
  virtual bool is_equivalent( MethodType* mtype );
CPP_DECLARE
};
```

4.8 Instance Members

Instance fields and methods are represented with `InstanceFieldSymbol` and `InstanceMethodSymbol`, respectively.

4.8.1 `InstanceFieldSymbol`

InstanceFieldSymbol represents the instance fields of a class. They usually reside in the InstanceFieldSymbolTable of the class.

9b ⟨*class* `InstanceFieldSymbol` 9b⟩≡ (6)

```
concrete InstanceFieldSymbol : FieldSymbol{
  searchable_list<LString> attributes;
  ClassType * reference owning_class omitted;
};
```

4.8.2 `InstanceMethodSymbol`

The InstanceMethodSymbol subclasses from a ProcedureSymbol and allows the user to distinguish the receiver. Furthermore, it associates the method with its class type and provides an attribute list.

By convention, the first formal parameter of the method is its receiver (e.g., the `this` reference in Java and C++). However, the user is still responsible to explicitly define the formal parameter and type that represents the receiver. Typically, the type of the receiver is a pointer to its defining class type. The *defining class* of a method or field is the class that first introduces the method or field, respectively.

The type of an instance method is represented with a InstanceMethodType (see section 4.7).

The `is_dispatched` field is true for method symbols that are dispatched. Typically, only non-constructor, non-private, virtual methods are dispatched (e.g., C++ and Java).[4]

Each instance also contains a pointer `overridden_method` to the MethodSymbol of the method (if any) overridden by this method.

[4] Class hierarchy analysis can be used to obtain a more accurate result of the methods that actually require dispatch and change the `is_dispatched` field accordingly.

9c ⟨*class* `InstanceMethodSymbol` 9c⟩≡ (6)
```
concrete InstanceMethodSymbol : ProcedureSymbol{
  searchable_list<LString> attributes;
  ClassType * reference owning_class omitted;

  bool is_dispatched default {true};
  InstanceMethodSymbol * reference overridden_method omitted;

CPP_DECLARE
  public:
    bool is_equivalent( InstanceMethodSymbol* mysm );
    bool overrides( InstanceMethodSymbol* msym );
CPP_DECLARE
};
```

4.9 Static Members

Static fields and methods are represented with `StaticFieldSymbol` and `StaticMethodSymbol`, respectively. They usually reside in the PerClassSymbolTable of the class.

4.9.1 `StaticFieldSymbol`

Static field symbols (as opposed to `InstanceFieldSymbols`) represent only a single storage location (if any). Hence every StaticFieldSymbol is implicitly associated with a unique storage location. For this reason, all SUIF instructions that work for non-`GroupType`s are applicable to static fields. For example, a `SymbolAddressExpression` can be used to obtain the address of a static field.

10a ⟨*class* `StaticFieldSymbol` 10a⟩≡ (6)
```
concrete StaticFieldSymbol : VariableSymbol {
  searchable_list<LString> attributes;
  ClassType * reference owning_class omitted;
};
```

4.9.2 `StaticMethodSymbol`

A StaticMethodSymbols, which is derived from ProcedureSymbol, represents a static method of a class. StaticMethodSymbols are ProcedureSymbols that are owned (or defined) by a certain class. They neither have a this “this”-reference nor are they dispatched.

10b ⟨*class* `StaticMethodSymbol` 10b⟩≡ (6)
```
concrete StaticMethodSymbol : ProcedureSymbol {
  searchable_list<LString> attributes;
  ClassType * reference owning_class omitted;

CPP_DECLARE
  public:
    bool is_equivalent( StaticMethodSymbol *mysm );
CPP_DECLARE
};
```

4.10 ClassType

The central paradigm of OSUIF is the ability to represent classes. A OSUIF class is represented with a ClassType, which is derived from StructType. A ClassType can have

- static (i.e., per-class) members (i.e., static fields and methods).
- instance members (i.e., instance fields and methods).
- semantic attributes.
- links to its ancestor and descendant classes.

Moreover, a class introduces a new scope for its members. These properties seem general enough to apply to all object-oriented languages.

A ClassType has three different symbol tables. Each symbol table is used to hold symbol table objects with different semantical properties:

`PerClassSymbolTable`

The `per_class_symbol_table` field of the ClassType contains a PerClassSymbolTable, which holds symbols and types of the class scope. Typically, StaticFieldSymbols and StaticMethodSymbols are put here.

The PerClassSymbolTable does not put any restrictions on the symbol table objects that it can hold.

11a ⟨*class* `PerClassSymbolTable` 11a⟩≡ (6)
```
concrete PerClassSymbolTable : BasicSymbolTable {
};
```

`InstanceFieldSymbolTable`

The ClassType's `group_symbol_table` field contains a InstanceFieldSymbolTable. Note that this field is inherited from GroupType and usually contains a GroupSymbolTable.

Similarly to the GroupSymbolTable, which restricts its contents to FieldSymbols, the InstanceFieldSymbolTable restricts its contents to InstanceFieldSymbols.

Importantly, InstanceFieldSymbols are subclassed from FieldSymbols, which means that the instance fields of a ClassType can be viewed as the fields of a GroupType (via the interface provided by GroupType). See below for a further discussion of this property.

11b ⟨*class* `InstanceFieldSymbolTable` 11b⟩≡ (6)
```
concrete InstanceFieldSymbolTable : GroupSymbolTable {
  constrain symbol_table_objects to searchable_list<InstanceFieldSymbol *owner>;
};
```

`InstanceMethodSymbolTable`

The ClassType's `instance_method_symbol_table` field contains a InstanceMethodSymbolTable.

A InstanceMethodSymbolTable is a BasicSymbolTable restricted to contain only InstanceMethodSymbols.

11c ⟨*class* `InstanceMethodSymbolTable` 11c⟩≡ (6)
```
concrete InstanceMethodSymbolTable : BasicSymbolTable {
  constrain symbol_table_objects to searchable_list<InstanceMethodSymbol* owner>;
};
```

ClassType provides an interface to all of its symbol tables, which allow access to the methods and fields declared or defined in the class.

Furthermore, the ClassType can contain InheritanceLinks, which are used to represent direct ancestor and descendant classes:

childclasses a list of InheritanceLink owner pointers that lead to the ClassType entries for child classes.

parentclasses a list of InheritanceLink pointers that lead to the ClassType entries for any parent class(es).

Note that class members have no domain-specific semantics, because OSUIF does not know anything about the source language. For example, the front end must support/enforce language-specific behavior (e.g., access restrictions), and language-specific passes must be written in order to exploit language-specific features (e.g., `static final` fields in Java).

A ClassType keeps its instance fields in the symbol table it inherits from StructType. This design decision yields the most intuitive interface because the data layout of the instance fields are essentially a StructType. One important difference is that a ClassType usually holds only the instance fields that it defines (as opposed to the instance fields it can access via inherited fields), but this is merely an implementation choice.[5]

Furthermore, a variable symbol that has a class type represents only the instance fields of the class type, and a variable definition reserves storage only for these fields. Hence, declaring a variable and defining storage for a class type works the same way as for a StructType.

The `is_ancestor_class()` and `is_descendant_class() methods` test if the given ClassType is an ancestor or descendant class, respectively.

Incomplete Class Types Since classes are modules with well-defined interfaces, a compiler can easily generate separate code for a single class. We assume *separate compilation* of classes in the following.

Generated code for a certain class usually contains method calls and field accesses to other classes. In the following, we call the class we currently want to compile the *main class* and all other classes that this class references its *auxiliary classes*.

In order to generate code for method calls and field accesses to auxiliary classes, the class types of these classes must be included as part of the code. This is even the case if just a reference to an auxiliary class is used and can result in a large amount of superfluous information in the symbol table, because usually only a small fraction of the fields and methods of an auxiliary class are actually referenced in the generated code. For example, the private part of an auxiliary class is never visible to the main class and hence can always be omitted.

OSUIF allows to specify an *incomplete class type*, which is a subset of the complete class type. The given subset typically is only the interface that is actually referenced by the main class. A class type has two boolean flags to mark separately if instance fields or instance methods are incomplete. The boolean flag `is_complete` that specifies if fields are complete is actually inherited from GroupType. This scheme is more flexible than the “forward declaration” that SUIF2 offers.

For a typical compilation, the class type of the main class is complete, but the class types of the auxiliary classes are mainly incomplete.

[OSUIF will provide a linker that merges (incomplete) class types.]

[5] An OSUIF lowering pass can be run to copy fields down the inheritance hierarchy.

12 ⟨*class* ClassType 12⟩≡ (6) 13a ▷

```
  concrete ClassType : StructType{
    list<InheritanceLink * owner > child_classes;
    list<InheritanceLink * reference > parent_classes;
    searchable_list<LString> attributes;
    bool methods_are_complete build {false};
    DefinitionBlock * owner definition_block;
    PerClassSymbolTable * owner per_class_symbol_table
          build {::create_per_class_symbol_table(get_suif_environment())};
    InstanceMethodSymbolTable * owner instance_method_symbol_table
          build {::create_instance_method_symbol_table(get_suif_environment())};

    constrain group_symbol_table to InstanceFieldSymbolTable * owner
          build {::create_instance_field_symbol_table(get_suif_environment())};

  CPP_DECLARE
  public:
    void set_fields_are_complete( bool b ) {
      set_is_complete( b );
    }
    bool get_fields_are_complete() const {
      return get_is_complete();
    }

    virtual bool is_ancestor_class(ClassType* ancestor_class);
    virtual bool is_descendant_class(ClassType* descendant_class);
```

4.10.1 Static Member Accessors

These functions access the per-class symbol table objects. These are typically StaticMethodSymbols and StaticFieldSymbols.

13a ⟨*class* ClassType 12⟩+≡ (6) ◁ 12 13b ▷

```
      virtual void add_static_field_symbol( StaticFieldSymbol* sym );
      virtual bool has_static_field_symbol( const LString& name );
      virtual StaticFieldSymbol* lookup_static_field_symbol( const LString& name);
```

13b ⟨*class* ClassType 12⟩+≡ (6) ◁ 13a 13c ▷

```
      virtual void add_static_method_symbol( StaticMethodSymbol* sym );
      virtual bool has_static_method_symbol( const LString& name );
      virtual StaticMethodSymbol* lookup_static_method_symbol( const LString& name );
```

4.10.2 Instance Field Accessors

These functions access the symbol table for instance fields.

13c ⟨*class* ClassType 12⟩+≡ (6) ◁ 13b 14a ▷

```
    virtual void add_instance_field_symbol( InstanceFieldSymbol* fsym );
    virtual bool has_instance_field_symbol( const LString& name );
    virtual InstanceFieldSymbol* lookup_instance_field_symbol( const LString& name );
```

4.10.3 Instance Method Accessors

These functions access the symbol table for instance methods.

14a ⟨*class* ClassType 12⟩+≡ (6) ◁13c 14b▷
```
      virtual void add_instance_method_symbol( InstanceMethodSymbol* msym );
      virtual bool has_instance_method_symbol( const LString& name );
      virtual InstanceMethodSymbol* lookup_instance_method_symbol( const LString& name );
```

14b ⟨*class* ClassType 12⟩+≡ (6) ◁14a
```
  CPP_DECLARE
  };
```

4.11 InheritanceLink

A class hierarchy (i.e., subtype and/or subclass relationships between classes) can be modeled with InheritanceLinks. A class hierarchy is essentially a directed acyclic graph (DAG). The nodes of the DAG are classes and the edges are inheritance links, which describe relations between classes.

An InheritanceLink has a list of attributes (see section 4.3.1). An attribute is a simple string that can be used to express the meaning of inheritance. In C++, for instance, it could be a `public` vs. `private` and `virtual` vs. `non-virtual` inheritance link. Of course, one can also subclass from InheritanceLink to model specific behavior. For instance, as an alternative a C++ front end might want to subclass inheritance link with a VirtualLink and a NonVirtualLink instead of using attributes.

14c ⟨*class* InheritanceLink 14c⟩≡ (6)
```
  concrete InheritanceLink : AnnotableObject{
    searchable_list<LString> attributes;
    ClassType * reference parent_class_type;
    ClassType * reference child_class_type;

  CPP_DECLARE
  public:
    virtual bool is_ancestor_class(ClassType* the_ancestor_class) const;
    virtual bool is_descendant_class(ClassType* the_descendant_class) const;
  CPP_DECLARE
  };
```

4.12 InstanceMethodCallStatement

The most distinctive feature of object-oriented methodology is the dynamic binding of methods. A compiler may have considerable latitude in how to implement it. For this reason, OSUIF provides a statement to express message sends at a high level. The MethodCallStatement, which is derived from CallStatement, keeps information about the dispatched method (`m`), the receiver(`r`), and the actual parameters of the call:

```
r.m(...)
```

The method `m` is represented by an InstanceMethodSymbol, which states its name, signature and defining static class. A lowering pass resolves the method dispatch, by replacing `m` with code that calls the actual method. A statically bound call typically replaces `m` with the address of `m` or one of its redefinitions. A dynamically bound call could generate a virtual function table access through `m`'s slot number.

The receiver `r` can be, for example, a variable symbol whose type is a pointer to a class type (that is a subtype of `m`'s class type). Note that OSUIF poses no restriction on the receiver type.

[It is planned that these restrictions are introduced by OSUIF dialects.]

Subsequent passes have access to the method call's actual parameters through the inherited interface (which shows the receiver as the first parameter) and direct access to the receiver through the new interface. Note that OSUIF only supports a single receiver (i.e., no multiple dispatch).

The `is_dispatched` flag indicates if the method call is dispatched or not. If this flag is true, the `target_method` must be dispatchable (i.e, the `is_dispatched` flag of the InstanceMethodSymbol must be true). If this flag is false, the method call is static. This is needed if the `target_method` is not dispatchable or if the method has to be statically resolved, for example:

`super.m(...)` (Java)
`C::m(...)` (C++)

15a ⟨*class* `InstanceMethodCallStatement` 15a⟩≡ (6)

```
concrete InstanceMethodCallStatement : CallStatement {
  InstanceMethodSymbol * reference target_method;
  bool is_dispatched default {true};
};
```

4.13 StaticMethodCallStatement

Static method calls are provided by the StaticMethodCallStatement.

[...]

15b ⟨*class* `StaticMethodCallStatement` 15b⟩≡ (6)

```
concrete StaticMethodCallStatement : CallStatement {
  StaticMethodSymbol * reference target_method;
};
```

4.14 NewStatement

NewStatement is used to allocate storage for an object of a certain ClassType.

[Note that only storage is allocated, but no constructors are called. (This behavior is similar to the JVM `new` instruction.)]

15c ⟨*class* `NewStatement` 15c⟩≡ (6)

```
#concrete NewStatement : Statement {
#  VariableSymbol* reference destination in destination_vars;
#  ClassType * reference class_type;
#};
```

4.15 DeleteStatement

DeleteStatement represents a destructor call of a ClassType.

15d ⟨*class* `DeleteStatement` 15d⟩≡ (6)

```
#concrete DeleteStatement : CallStatement{
#};
```

4.16 Runtime Checks

A runtime check is performed during the execution of a program to make sure a certain condition is satisfied. If the condition is not satisfied, a certain action is taken. A typical runtime check is an array bounds checks. If the array bounds fails, an exception might be thrown (e.g., Java) or the program might be aborted (e.g., Modula-2).

16a ⟨*class* CheckStatement 16a⟩≡ (6)
```
concrete CheckStatement : Statement {
  # Expression must evaluate to true or false
  Expression * owner check in source_ops;

  # Code executed if "check" fails
  Statement * owner body in child_statements build {0};
};
```

4.17 Exception Handling

The current constructs support exceptions at the statement level and allow exceptions to be expressed in the following way:

`try {`
Statement
`}`
`catch(`*Type*$_1$ *[obj*$_1$*]* `) {` *Statement* `}`
...
`catch(`*Type*$_n$ *[obj*$_n$*]* `) {` *Statement* `}`

A `try`-block encapuslates a piece of code that can be protected by exception handlers. An exception handler specifies the exception type that the handler is willing to catch, optionally followed by an exception object. Thus, exceptions in OSUIF are very similar to the Java and C++ exception model.

4.17.1 `CatchStatement`

This class represents an exception handler.

The `catch_argument` specifies the exception object. The exeception object can be NULL. (This is useful for languges, that do not have an exception object, such as Ada and Eiffel.)

The `catch_type` specifies the type of exceptions that the exception handler is willing to handle. If `catch_argument` is not NULL, its type must be identical to `catch_type`. (The `catch_type` is needed, for example, for "anoymous" handlers in C++. An Ada front end could introduce artifical exception-handling types (subclassed from IntegerType?).)

A CatchStatement can only attached to a TryStatement.

16b ⟨*class* CatchStatement 16b⟩≡ (6)
```
concrete CatchStatement : Statement {
  Statement * owner body in child_statements;
  VariableSymbol * reference catch_argument;
  Type * reference catch_type;
};
```

4.17.2 TryStatement

This statement specifies a scope in which exception handlers are active. The exceptions handlers are attached at `catch_statements`.

17a ⟨*class* TryStatement 17a⟩≡ (6)
```
concrete TryStatement : Statement {
  Statement * owner body in child_statements;
  vector<CatchStatement * owner> catch_statements in child_statements;
};
```

4.17.3 ThrowStatement

This class expresses the actual throwing of an exception.

The ThrowStatement takes an `exception_object` as its only source operand. If `exception_object` is not NULL, its result type must be identical to `throw_type`. (The `throw_type` is needed for languages, that do not have (explicit) exception objects.)

17b ⟨*class* ThrowStatement 17b⟩≡ (6)
```
concrete ThrowStatement : Statement {
  Expression * owner exception_object in source_ops;
  Type * reference throw_type;
};
```

4.17.4 ThrowsAnnote

This annotation contains the exception types that a method might throw. It corresponds to the `throws` clause of C++ and Java.

This annotation is typically attached to a ProcedureDefinition.

17c ⟨*class* ThrowsAnnote 17c⟩≡ (6)
```
concrete ThrowsAnnote : GeneralAnnote {
  searchable_list<Type* reference> throw_types;
};
```