7 Related Work

There are a number of proposals and systems which are similar to CoffeeStrainer. The main part of this chapter, from Section 7.1 to Section 7.5 consists of a detailed comparison of five such systems to CoffeeStrainer. Section 7.6 summarizes the comparison. Other related work is discussed in Section 7.7.

The criteria for including a system in the detailed comparison are:

- *implemented system*: We only include systems that have been implemented, and exclude proposals or formal models which have not been validated by an implementation.
- *useable for checking realistic constraints*: Many of the systems support not only the checking of constraints, but also general queries on source code. Some of these are based on restricted models, such as regular expressions, or relational database queries, for efficiency reasons. We only include systems with more expressive query languages.
- *sufficiently documented constraint language*: To perform the detailed comparison, we use example constraint implementations using the constraint or query languages of each compared system. A system could not be included if its constraint or query language was only partly documented.

The five systems which are compared in detail to CoffeeStrainer are:

- The *C++ Constraint Expression Language - CCEL* - [Chowdhury, Meyers 1993] allows to specify statically checked constraints on programs written in C++. Constraints are specified in a special-purpose, C++-like language and can be checked globally, for a single file, for a class, or for a function.
- *Law-Governed Architecture - LGA* - [Minsky 1996] supports constraints specified on a language-independent object model, using Prolog as the constraint language. A mapping of the abstract object model to Eiffel has been defined and implemented [Minsky, Pal 1997]. LGA can specify more than constraints on object-oriented programs, it also addresses constraints that can be defined on the process of software development. In this comparison, we will only consider the subset of LGA that applies to the software itself.
- The *Category Description Language - CDL* - which has been proposed for specifying Formal Design Constraints [Klarlund et al. 1996], is a constraint language based on a theory of logics on parse trees. CDL is a restricted formalism that allows to check automatically whether a set of constraints is consistent.

- *GENOA* [Devanbu 1992], a customizable code analyzer which can be interfaced to existing language front-ends, provides a LISP-like query language that applies to the complete ASG of a program under examination.
- *ASTLOG* [Crew 1997], a language for examining abstract syntax trees, is a variant of Prolog in which the clauses have access to a complete AST of C++ programs. In ASTLOG, constraints are written in an inside-out functional style that is particularly suited for analyzing tree structures.

The first three of these systems are constraint checking systems; the last two are more general systems for querying source code.

We will use a running example for comparing the five systems. The context of the example is a hypothetical framework that provides an implementation of a certain form of persistent objects, i.e., objects that are stored in a database system in order to survive several invocations of an application across time. Although the example constraints have been made up purposely to cover many facets of constraint-checking systems, we think that it still is realistic and could be part of a real-world framework that requires the example constraints to be satisfied. In the following, we will motivate and explain the set of constraints we have chosen, and we will present how these constraints can be expressed using CoffeeStrainer.

The persistency framework, which is shown – without constraints – in Figure 7.1, provides automatic storing and retrieving of objects. It frees the programmer from the tedious task of writing hand-crafted code for storing objects in the database and retrieving objects from the database. The main abstraction of the framework is the interface `Storable`: objects of classes that implement `Storable` can be stored in the database. The persistency framework stores in the database all public fields of `Storable` objects that either have a type which is `Storable`, or the type `String`, or a primitive type.

Each class that implements `Storable` must define an empty no-arg constructor to enable the persistency framework to instantiate objects when retrieving data from the database. The method `notifyRetrieved` is invoked by the framework upon successful retrieval of an object from the database; it can be used to re-initialize non-public fields of `Storable` objects.

The implementation of the framework is in the class `Database` with three methods: The method `makePersistent`, when called on an object of a class that implements `Storable`, stores this object in the database together with all `Storable` objects which are transitively reachable from this object by `public` fields. The return value of this method is an integer ID which identifies the persistent object uniquely. Using this ID, an object can be retrieved from the database by calling `retrieve`. The third method, `update`, is to be called by classes that implement `DatabaseObject` from all methods that change public instance fields in order to notify the framework of changes. To be able to check this statically, we require all field accesses to be enclosed in a `try` statement which has a `finally` clause with a call to `update` (see below).

Figure 7.2 shows an example for using the framework. It consists of two classes, `Account` and `Customer`, which implement `Storable`. The class `Account` has a `public` field `balance`, which is stored in the database, and a `protected` field `owner`, which is a back reference to the `Customer` object which owns the `Account` and is not stored. The method

```
package database;

// can be stored in the database
// only public fields of primitive type or type Storable are stored
// call Database.update(this) in methods that change fields
protected interface Storable {
  // this method should be called by the framework only:
  public void notifyRetrieved();
}

public class Database {
  public static int makePersistent(Storable s) { ... }
  public static Storable retrieve(int id) { ... }
  public static void update(Storable s) { ... }
}

public class DatabaseThread extends Thread { ... }
```

Figure 7.1: The example framework

`deposit`, which changes the field `balance`, calls the `update` method of `Database` in a `finally` clause of a `try` statement which wraps the field access. The two fields `name` and `account` of class `Customer` are stored in the database. Additionall, the class Customer is responsible for maintaining the back reference in its `Account` object; this is reflected in the constructor and in the implementation of `notifyRetrieved`.

The correct functioning of the persistency framework depends on assumptions that require the user of the framework to obey a number of rules on the application-specific classes he writes, i.e., there is a set of constraints which need to be checked on client code. The constraints, each of which demonstrates a certain aspect of CoffeeStrainer, are numbered from P1 to P4:

P1 When retrieving an object from the database, the persistency framework uses Java reflection to instantiate objects of subtypes of `Storable`. To make this possible, all implementations of `Storable` must provide an empty public no-arg constructor so that the persistency framework can create new instances of the class when retrieving objects from the database. The object's `public` fields can then be set to their stored values by using reflection.
This constraint is a simple type definition constraint. Part of it can be checked by only looking at properties of classes and signatures of constructors in these classes. This part should be possible to express in other systems that only provide structural information about classes and signatures of methods rather than a full ASG. For determining whether or not the constructor is empty, a system has to provide access to method bodies and not only to class definitions and method signatures.

P2 The method `notifyRetrieved` should not be called by application code.
This constraint is a simple (method) usage constraint. It will help to find out how usage constraints can be expressed in the other systems.

```
public class Account implements Storable {
  public Account() {} // called by the persistency framework
  public int balance;
  protected Customer owner; // to be set by the owner
  public void deposit(int amount) {
    try { balance += amount; }
    finally { database.Database.update(this); }
  }
  public void notifyRetrieved() { }
}

public class Customer implements Storable {
  public Customer() {} // called by the persistency framework
  public String name;
  public Account account;
  public Customer(String name_, Account account_) {
    name = name_; account = account_; account.owner = this;
  }
  public void notifyRetrieved() {
    account.owner = this;
  }
}
```

Figure 7.2: Example of using the persistency framework

P3 The method `notifyRetrieved`, although direct uses of it are disallowed due to P2, can still be invoked indirectly, by using reflection. Such invocations cannot be prevented statically except by disallowing all reflective features. However, the persistency framework's implementation uses reflection, so disallowing the use of reflection would be too drastic. What we would like to ensure is that, at run-time, only the persistency framework invokes the method `notifyRetrieved`. The solution in this case is to require that invocations of `notifyRetrieved` be performed only by special threads of type `DatabaseThread` which the persistency framework creates.
This constraint is a dynamic constraint for which a run-time check should be inserted in all implementations of `notifyRetrieved`.

P4 Finally, all `Storable` objects need to wrap write accesses to fields in a `try-finally` construct which ensures that the method `update` of the class `Database` is called whenever a field has been updated. We require that all write accesses to fields of Storable objects be performed on the current instance (`this`) and that they be enclosed by a `try` statement with a `finally` clause whose first statement is the invocation "`Database.update(this)`".
With this constraint, we examine how the different systems can handle constraints that require pattern matching in ASG structures.

Figure 7.3 shows the CoffeeStrainer implementation of the constraints P1–P4.

Most constraint methods are straightforward translations of the constraints P1–P4. The only constraint that cannot be implemented by one or two lines of Coffeestrainer code is con-

```
/**
 *@constraints
 * public boolean checkClass(Class c) {
 *   rationale = "Storable class needs public empty no-arg constructor (P1)";
 *   Constructor ctor = c.getConstructor(new AType[]);
 *   return ctor!=null && ctor.isPublic()
 *                  && ctor.getBody().getStatements().size()!=0;
 * }
 * private AMethod updateMethod = Barat.getClass("database.Database")
 *            .getStaticMethod("update", new AType[1]{thisInterface});
 * public boolean checkUseAtInstanceFieldAccess(InstanceFieldAccess a) {
 *   if(!a.getAccessedField().isPublic()) return true;
 *   if(!a.containmentAspect().equals("lvalue")) return true;
 *   rationale = "fields of Storable object should only be written to using this (P4)";
 *   if(!(a.getInstance() instanceof This)) return false;
 *   rationale = "field acess should be wrapped in try..finally { Database.update(this); } (P4)";
 *   Try t = (Try)a.containing(Try.class);
 *   if(t==null || t.getFinally()==null) return false;
 *   AStatementList sl = t.getFinally().getBody().getStatements();
 *   if(sl.size()==0
 *       || !(sl.get(0) instanceof ExpressionStatement)) return false;
 *   ExpressionStatement es = (ExpressionStatement)sl.get(0);
 *   if(!(es.getExpression instanceof StaticMethodCall)) return false;
 *   StaticMethodCall smc = (StaticMethodCall)es.getExpression();
 *   return smc.getCalledMethod==updateMethod
 *          && (smc.getArguments().get(0) instanceof This);
 * }
 */
public interface Storable extends Storable {
  /**
   *@constraints
   * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall c) {
   *   rationale = "notifyRetrieved should not be called by application code (P2)";
   *   return c.containingPackage() != Barat.getPackage("database");
   * }
   * public boolean checkConcreteMethod(ConcreteMethod m) {
   *   rationale = "notifyRetrieved can only be called from DatabaseThread (P3)";
   *   return preRuntime(m,
   *      "Thread.currentThread() instanceof database.DatabaseThread");
   * }
   */
  public void notifyRetrieved();
}
```

Figure 7.3: Implementation of constraints P1-P4 using CoffeeStrainer

straint P4 in which one needs to traverse the ASG, making sure that it has the required structure. Note that a substantial portion of the code is due to `instanceof` testing and downcasting.

In the next sections, each of the other systems will be addressed in turn. Some of the systems have been implemented for a specific programming language, while others claim to be language-independent. To be able to compare all systems to CoffeeStrainer using the same example, we present them as if they had been implemented for Java.

The next sections, from Section 7.1 to Section 7.5, explain each of the other systems. Section 7.6 compares these systems with CoffeeStrainer. Finally, Section 7.7 at the end of this chapter discusses other related work which could not be integrated into the comparison of constraint checking systems which makes up the main part of this chapter.

7.1 CCEL

CCEL is a system for checking static constraints on C++ programs. As already noted, for the purposes of this comparison, we treat it as if it was a system for checking constraints on Java.

CCEL does not support checking complete ASGs. Rather, constraints can only refer to definitions of classes and methods, i.e., constraints cannot be applied to method bodies with the contained statements and expressions. This limitation is quite severe as all of the example constraints require checking on the level of statements or expressions. Constraint P3 cannot be expressed with CCEL because only static constraints are supported. In effect, only the part of constraint P1 which deals with types and signatures and not with method bodies can be expressed with CCEL. The implementation of this part is shown in Figure 7.4.

```
P1_StorableNeedsPublicNoArgConstructor {
  Interface I | I.name() == "database.Storable";
  Class C | C.implements(I);     // for all classes C that implement I
  Assert(Constructor C::ctor; |  // there must be a constructor in C
          ctor.parameters().size() == 0  // with no parameters
          && ctor.is_public());          // which is declared public.
}
```

Figure 7.4: Implementation of P1 using CCEL

In CCEL, constraints have a name which is reported for each violation of the constraint. It is not possible to define violation messages directly, such as the `rationale` in CoffeeStrainer. The constraint implementation, which is enclosed in curly braces, consist of a *context declaration* and an *assertion*. The declaration part defines the context in which the assertion is checked. In our example, the context consist of an interface `I` which must be equal to the interface `Storable`, and an arbitrary class `C` which implements `I`. Note that unlike in CoffeeStrainer, constraints are declared in separate files and thus have to refer to program elements by name. Moreover, it has to be declared explicitly that a constraint applies to all subtypes of a type.

The context declaration can be read as declaration of *universally* quantified variables for which certain properties must hold ("for all interfaces `I` equal to `Storable`, and all classes `C` that implement `I`"), and the assertion part of a constraint may include *existentially* quantified variables. In our case, the assertion requires the existence of a constructor of `C` ("there exists a constructor of `C`"). The boolean expression following the vertical bar specifies a property that must hold for all variables that have been declared. Thus, the assertion part of our example can be read as "there exists a constructor of C that has no parameters and is declared public".

Violations reported by CCEL include the constraint name and all bindings for the universally quantified variables. Thus, it is not necessary to base constraints on single program elements, as is the case in CoffeeStrainer, making CCEL's constraints more expressive than CoffeeStrainer constraints. For example, the constraint "for every pair of classes `C` and `D`, there must be a field with the same name in both `C` and `D`" can be specified in CCEL but not in CoffeeStrainer. However, the added expressiveness comes at a price, because CCEL constraints potentially require whole-program analysis. It seems that the authors of CCEL are not aware of this, because the implementation of CCEL – like CoffeeStrainer – applies checks modularily, one compilation unit at a time.

In another aspect, CCEL constraints are less expressive than CoffeeStrainer's: It is not possible to traverse the ASG in a bottom-up way because CCEL does not provide methods like `container` or `containingClass` etc., and even when traversing from top to bottom, it is not possible to specify constraints that require a certain order of child nodes for a certain ASG node. In the implemented constraint P1, this does not occur; however, constraint P4, even if CCEL provided access to method bodies and statements, could not be specified because it requires the call to `update` to be the *first* statement of the `finally` clause. Probably, this lack of specification mechanisms for ordering is due to the fact that CCEL does not address statements and expressions where ordering is crucial.

7.2 Law-Governed Architecture

LGA, Law-Governed Architecture, is based on an object-oriented model of a software development project. The objects it applies to can be quite diverse, ranging from classes that have been developed to the developers themselves. It allows to specify constraints on relations between objects, called *interactions*. Possible interactions are the creation of a class by a programmer, the inheritance relation between two classes, or the usage relationship between a client class and a supplier class. All of these interactions can be constrained, i.e., allowed or disallowed depending on properties of the objects that play a role in an interaction.

A specific implementation of LGA, called Darwin/E, allows to specify constraints on interactions between program elements for Eiffel. For the purposes of this comparison, we present Darwin/E as if it had been implemented for Java. The formalism used for expressing constraints is the language Prolog. Rules for constraining a certain type `t` of interactions are specified as Prolog clauses that are called `cannot_t(A1, A2, ..., An)` and `can_t(A1, A2, ..., An)` which specify conditions for disallowing or allowing interactions of type `t` with participating objects `A1, A2, ..., An`. Overall, an interaction `t` may take place if the built-in clause `t(A1, A2, ..., An)` succeeds. This built-in clause is defined in terms of `cannot_t` and `can_t` as follows:

```
t(A1, A2, ..., An) :-
          cannot_t(A1, A2, ..., An) ->
            $do(error(['interaction prohibited']))
          | (can_t(A1, A2, ..., An) -> true
          | $do(error(['interaction not permitted'])).
```

It specifies that an interaction can take place *only* if it is not prohibited explicitly by means of `cannot_t`, *and* if it is permitted explicitly by means of `can_t`. Technically, checking whether an interaction is allowed means evaluation the Prolog goal `t(a1, a2, ..., an)` where `a1, a2, ..., an` represent the actual objects that participate in the interaction. Using this structure, one can define rules based only on prohibitions, by including a rule `can_t(A1, A2, ..., An) :- true`, or based only on permissions, by not including any rules for `cannot_t`. Of course, it is also possible to combine both approaches. In contrast, only prohibitions can be specified in CoffeeStrainer.

The constraint P1 that requires a public empty no-arg constructor in classes that implement `Storable` can be specified as follows, based on the interaction `implement(C, I)` which states that class `C` directly or indirectly implements the interface `I`:

```
can_implement(C, database_object_interface) :-
          provides_ctor(C, Ctor),
          is_public(Ctor),
          is_empty(Ctor),
          not has_parameter(Ctor, _).
```

Note that in Prolog, variables start with an uppercase letter and constants with a lowercase letter. The implementation of P1 makes use of the Prolog rules `provides`, `is_public`, `is_empty`, and `has_parameter` which we assume to be built-in. The goal `provides_ctor(C, Ctor)` succeeds if class C contains the constructor `Ctor`. It is used to bind the variable `Ctor` repeatedly to all contructors of C until a public constructor with no arguments is found.

LGA is the only system besides CoffeeStrainer which directly supports usage constraints. The constraint P2 which prohibits calling the method `notifyRetrieved` can be expressed as a rule on the interaction `call(C1, M1, C2, M2)` which states that method `M1` in class `C1` calls method `M2` of class `C2` . We assume that the goal `contains(P, C)` succeeds if class `C` is contained in package P.

```
can_call(_, _, _, _)       :- true.
cannot_call(C, _, _, notifyRetrieve_method) :-
          contains(database_package, C).
```

The constraint P4, however, cannot be specified using LGA. It states that field accesses to `public` fields of classes implementing `Storable` need to be wrapped in a `try` statement with a `finally` clause in which the `update` method is called on the class `Database`. Because this is a constraint on the structure of the abstract syntax tree, and not a constraint based on an interaction between program elements such as classes, methods, and constructors, there are no `can_t` or `cannot_t` clauses which apply in this case.

Although Darwin/E is a purely static framework, LGA's general nature of constraining arbitrary interaction would allow to express the dynamic constraint P3 as well. This constraint requires that run-time invocations (which can only be caused using Java reflection) be issued from special threads of type `DatabaseThread` which are created by the persistency framework. If we call this interaction `reflection_call(T, M)`, denoting a run-time invocation of the method M by thread `T`, the constraint can be expressed as follows, using `has_type(Thread, Type)` which succeeds if `Thread`'s runtime type is `Type`.

```
can_reflection_call(T, notifyRetrieve_method) :-
          has_type(T, database_thread).
```

LGA stands out from the other systems because, like CoffeeStrainer, it supports usage constraints and dynamic constraints. However, LGA does not provide access to the ASG. Instead, it is based on an object-oriented model where interactions between objects can be constrained. As has been explained, constraints on the structure of the ASG therefore cannot be expressed using LGA. It should also be noted that it is not easy to state what set of external rules for accessing important information about a program, such as `provides_ctor`, `is_public`, `has_type`, etc. would be needed.

Although LGA is based on an object-oriented model, the modeled domain of object-oriented programs is not reflected appropriately, because constraints do not automatically apply to subtypes of a type to which they apply. This can be seen in the example code, where we had to include an explicit query for those classes that implement a specific interface.

There is no support for including explanations or rationales for the constraints, and it is unclear how a constraint violation is reported; in particular, it is probably not possible in the LGA framework to exactly state which part of the program caused the violation. Like in CCEL, it is possible to express constraints that require whole-program analysis.

7.3 Category Description Language

The Category Description Language – CDL – can be used to express constraints on parse trees. Its distinguishing feature is that constraints expressed in CDL can be checked for consistency, i.e., it can be determined automatically if two constraints contradict each other. This requires, however, a restricted constraint language in which not all constraints can be expressed.

CDL is based on predicate logic. The first-order terms denote nodes of an abstract syntax tree. Formulas can include the usual logical connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, ... and three tree-specific operators: $x \leq y$ holds true if y is a descendant tree node of x, $x \lhd y$ holds true if y is a direct child node of x, and $x.i$ denotes the i^{th} child node of x. Furthermore, CDL supports the quantifiers $\forall$ and $\exists$ which range over typed nodes of the syntax tree.

For implementing arbitrary constraints, CDL includes an escape mechanism, namely, externally computed unary predicates, which, when used, prevent consistency of constraints to be checked. As will be seen below, our examples do include such external predicates.

Constraint specifications in CDL are called *categories*. A category applies to a program element if the program element is annotated with the category's name — CDL thus requires

changes to the base language's syntax. As a first example, consider the following definition of category *storable*, which captures P1 (public empty no-arg constructor needed):

CATEGORY storable FOR Class IS

$$\begin{aligned}\exists c : \text{Constructor}.\quad & \text{root} \leq c \wedge \\ & (\exists m : \text{Modifier}.\ \text{c.1} \triangleleft m \wedge m = \text{public}) \wedge \\ & \neg(\exists p : \text{Parameter}.\ c.2 \triangleleft p) \wedge \\ & \neg(\exists s : \text{Statement}.\ c \leq s)\end{aligned}$$

For this example, we assume that the first child of a constructor node is a list containing nodes for the constructor's modifiers, and that the second child is a list of nodes for the parameter declarations. In English words, the constraint requires the existence of a constructor node that is a tree descendant of the root node c of type `Class` for which there exists a "`public`" modifier m but no parameter node p and no statement node s.

To let this constraint (category) apply to all classes that implement the interface `Storable`, each of these *classes* (not only the interface `Storable` itself) needs to be annotated with the category name "storable" — a requirement that itself would necessitate a static constraint which ensures that this annotation is not forgotten in some class that indirectly implements the interface.

To express constraint P2 (the method `notifyRetrieved` should not be called), we need an externally defined predicate callsNotifyRetrieved(a) that determines if a tree node a of type `InstanceMethodCall` calls a method "`notifyRetrieved`" of a class that implements the interface `Storable`:

CATEGORY global1 FOR Class IS

$$\begin{aligned}\forall a : \text{InstanceMethodCall}.\quad & \text{root} \leq a \Rightarrow \\ & \neg\text{callsNotifyRetrieved}(a)\end{aligned}$$

As can be seen, CDL does not offer much to express this constraint — most of it needs to be implemented externally. Moreover, the category name "global1" hints at another problem: This constraint should be checked on all usages of types derived from `DatabaseObject`, which could occur in any class. Thus, every class would need to be annotated with the category name "global1".

Because CDL does not support dynamic constraints, the constraint P3 cannot be specified.

The constraint P4, which requires that write accesses to `public` fields of classes implementing `Storable` should be wrapped in a try-finally, again must employ externally-defined predicates (accessesPublicFieldOfStorable and callsUpdate):

CATEGORY global2 FOR Class IS

$$\begin{aligned}\forall c : \text{InstanceFieldAccess}.\quad & (\text{root} \leq c \wedge \\ & \text{accessesPublicFieldOfStorable}(c)) \Rightarrow \\ & (\exists t : \text{Try}.\ t \leq c \wedge \\ & \exists t : \text{Finally}.\ t \triangleleft f \wedge \\ & \text{callsUpdate}(f.1.1.1))\end{aligned}$$

It can be seen that the theoretically interesting possibility to automatically check constraint consistency cannot be used in many cases, because externally-defined predicates are often needed for practical constraints. Moreover, implementing such predicates requires using another, probably low-level language to implement the predicate, making it difficult to understand constraints because their semantics cannot be derived from looking at the CDL part alone. The operators for accessing the abstract syntax tree are very low-level as well, using indices to refer to children nodes. In our example, the expression $f.1.1.1$ refers to the `InstanceMethodCall` contained as the first child in an `ExpressionStatement`, which in turn is the first child of the `finally` clause.

7.4 GENOA

GENOA is a system for performing many kinds of analysis tasks on ASGs. Besides supporting static checking of constraints, it can be used for other purposes as well, such as for example, collecting metrics, control flow analysis, program querying and understanding, etc. Although implemented for C++, its concepts are language-independent and apply to the examination of arbitrary trees. In fact, GENOA is a LISP-like language tailored for expressing tree traversals.

A GENOA program is a list of procedures, each defining a traversal of the tree part of the ASG starting at a certain ASG node type. Some of the procedures can be declared to be *root procedures* that are applied automatically to each compilation unit. To express the example constraints, we have chosen to implement a top-level procedure, called `CheckConstraints`, which defines a traversal starting at ASG nodes of type `CompilationUnit`. It is declared as a root procedure (`ROOTPROC`) and will be called implicitly for each compilation unit under examination by GENOA.

The main part of a GENOA procedure consists of a series of *constructs*, each of which is an operation on an implicit *current node*. A construct is either a *statement* or a *traversal*. A statement can be an assignment, a conditional statement, a print statement, a procedure call, or a call to an externally defined operation written in C++. Statements can contain *expressions*, including LISP-like list operations and the special expressions `$token` for the current ASG node and `$location` for the current node's location.

GENOA does not support dynamic constraints; therefore, only constraints P1, P2, and P4 can be implemented with GENOA.

Figure 7.5 shows the top-level procedure, whose current node on invocation is defined to be a `CompilationUnit`. The procedure implementation — enclosed in curly braces — performs a subtree traversal, denoted by the square brackets, which applies the contained constructs to every node of the tree whose root is the `CompilationUnit`. Within the subtree-traversal, the procedure contains three procedure calls (`CALL`), each enclosed in a test-traversal denoted by "`(?`". Each test-traversal construct applies the contained construct (in our example, the contained statement) only if the type of the current node matches the test. For example, the construct `[(?Class x)]` applies `x` to all nodes of type `Class` which are part of the subtree whose root is the current node.

A GENOA traversal usually contains other traversals or statements. Overall, there are four kinds of traversals, distinguished by the kinds of parentheses used. Using BNF notation,

```
ROOTPROC CheckConstraints;
PROC CheckConstraints
ROOT CompilationUnit;
{
  [
    (?Class
      (CALL HasPublicEmptyNoArgCtor $token)
    )
    (?InstanceMethodCall
      (CALL DontCallNotifyRetrieved $token)
    )
    (?InstanceFieldAccess
      (CALL WrapWriteAccessInTryFinally $token)
    )
  ]
}
```

Figure 7.5: Top-level GENOA procedure

where *constructs* represents zero or more traversals or statements, the four kinds of traversals are:

- "`[`" *constructs* "`]`" (subtree-traverse) applies the contained constructs to all ASG nodes that are contained in the subtree which has the current node as its root.
- "`(?`" *typename constructs* "`)`" (test-traverse) applies the contained constructs to the current node if the current node is of type *typename*. A common idiom in GENOA is to combine subtree-traverse and test-traverse for traversing all nodes of a certain type in the subtree rooted at the current node. For example, `[(?Assignment x)]` applies `x` to all `Assignment` nodes in the subtree which has the current node as its root.
- "`<`" *childname constructs* "`>`" (child-traverse) applies the contained constructs to the current node's child node with name *childname*. For example, `[(?Assignment <lvalue x>)]` applies `x` to the `lvalue` children of all `Assignment` nodes in the subtree which has the current node as its root.
- "`{`" *constructs* "`}`" (list-traverse) applies the contained constructs to each element of the list which is represented by the current node. For example, `[(?Class <constructors { x }>)]` applies `x` to all constructors of classes in the subtree which has the current node as its root.

Note that the idiom of combining subtree-traverse with test-traverse is very similar to the basic structuring mechanism of CoffeeStrainer, namely, the Visitor-like constraint methods. When used as provided by CoffeeStrainer, the constraint methods provide only one such traversal as opposed to arbitrary nesting which is possible in GENOA. However, by nesting visitors, nested traversals can be specified in Coffeestrainer as well. Furthermore, the traversal in CoffeeStrainer is implicit, whereas it is explicit in GENOA, which requires changing the top-level procedure whenever a new constraint procedure is added.

The implementation of P1, called `HasPublicEmptyNoArgCtor`, is shown in Figure 7.6. A GENOA procedure body can contain declarations of local variables. In the implementation of P1, we define two integer variables: `has_public_empty_ctor` is needed for determining whether a class contains a public empty no-arg constructor, and `param_count` is used for counting the number of parameters of constructors. GENOA is not statically typed; all variables are of type GNODE and may, at run-time, contain integer values, string values, references to ASG nodes, or homogeneous lists of these.

```
PROC HasPublicEmptyNoArgCtor
ROOT Class
{
  LOCAL GNODE has_public_ctor;
  LOCAL GNODE param_count;
  (IF (EVAL Implements "database.Storable" $token)
    (THEN
      (ASSIGN has_public_empty_ctor 0)
      <constructors {
        <parameters
          (ASSIGN param_count (LENGTH $token))
        >
        (IF (EQUAL param_count 0)
          (THEN
            <modifiers
              (IF (MEMBER-EQUAL "public" $token))
                (THEN
                  (ASSIGN has_public_empty_ctor 1))
                (ELSE
                  (PRINT stdout
                    "no-arg constructor at %s is not public (P1)" $location)))>
            <body
              <statements
                (IF (NOT (EMPTY $token))
                  (THEN
                    (PRINT stdout
                      "no-arg constructor at %s must be empty (P1)" $location)))>>
          )
        )
      >
      (IF (EQUAL has_public_empty_ctor 0)
        (THEN
          (PRINT stdout "class %s needs no-arg constructor (P1)" $token)
        )
      )
    )
  )
}
```

Figure 7.6: Checking P1 with GENOA: procedure `HasPublicEmptyNoArgCtor`

The procedure `HasPublicEmptyNoArgCtor` first checks whether the current node is a class that implements the interface `Storable` (we assume the existence of an external pro-

cedure `Implements` that determines whether a given class implements a named interface). If this is the case, the variable `has_public_ctor` is initialized to `0`, and an iteration over all constructor nodes for the class is performed, using a combination of child-traverse and list-traverse. Then, the implementation examines the list of all parameter nodes for each constructor and stores the number of parameters in `param_count`. If this number is zero, we have arrived at a no-arg constructor. Note that the variable `param_count` is needed for making the number of parameters available in an outer context; the syntax tree can only be accessed using the traversal constructs.

If a no-arg constructor is found, the procedure checks that it is declared public, i.e., its modifiers must contain a node equal to "`public`". If this is the case, we set the variable `has_public_ctor` to 1; otherwise, we output a constraint violation message. Furthermore, if the constructor's body is not empty, we output another constraint violation message. Compared to the CoffeeStrainer implementation of the same constraint, the GENOA implementation is much longer; apparently due to the two local variables which had to be introduced to pass information from one branch of the traversal to another, and because GENOA constraint implementations seem to be more verbose.

In Figure 7.7, the GENOA implementation of P2 is shown. The procedure `DontCallNotifyRetrieved`, when invoked on an `InstanceMethodCall` node, checks whether it is a call to `notifyRetrieved` which is not contained in the package `database`, and outputs a message if this is the case. We have used three helper functions:

- The function `CallsNotifyRetrieved` returns `TRUE`[1] if its InstanceMethodCall argument represents a call to the method `notifyRetrieved`. From the GENOA language specification, it is unclear whether the procedure `CallsNotifyRetrieved` could be implemented without resorting to a primitive function written in GENOA's implementation language, C++.
- The function `PackageName` returns, for a node of type `Package`, the package name. Like in the implementation of P1, this additional code is needed because the syntax tree can only be accessed by traversal constructs and not using expressions.
- The recursive function `ContainingPackage` returns, for an arbitrary `GNODE`, the first containing node of type `Package`.

Finally, Figures 7.8 and 7.9 show the GENOA implementation of P4. The top-level procedure for checking the constraint, `WrapWriteAccessInTryFinally`, operates on a node of type `InstanceFieldAccess` and checks whether a public field of a class implementing `Storable` is accessed as the l-value in the context of an assignment. The expression "`$slot`" denotes the name of the parent node's slot in which the current node is contained. It is equivalent to CoffeeStrainer's `containmentAspect()`. The function `AccessesPublicFieldOfStorable`, whose implementation is shown in Figure 7.9, probably needs to be implemented in C++ like `CallsNotifyRetrieved` above. Again, a local variable `access_using_this` is needed because the test-traverse construct "`(?This`" cannot be used directly in the `IF` statement which either prints out a message or not.

[1] The GENOA language definition does not contain a definition for boolean constants. We consider this to be an omission in the paper and not in the language, and use `TRUE` and `FALSE` as the names of the two boolean values.

```
PROC DontCallNotifyRetrieved
ROOT InstanceMethodCall
{
  (IF
    (AND
      (EVAL CallsNotifyRetrieved $token)
      (NOT (EQUAL (EVAL (PackageName (ContainingPackage $token)))
                  "database")))
    (THEN
      (PRINT stdout
        "calling notifyRetrieved at %s not allowed (P2)" $location)
      )
    )
  )
}
FUNCTION CallsNotifyRetrieved
ROOT InstanceMethodCall
{
  ...
}
FUNCTION ContainingPackage
ROOT GNODE
{
  (?Package
    (RETURN $token)
  )
  (IF (NULL $parent) (THEN (RETURN $parent)))
  (RETURN (EVAL ContainingPackage $parent))
}
FUNCTION PackageName
ROOT Package
{
  <name (RETURN $token)>
}
```

Figure 7.7: Checking P2 with GENOA: procedure `DontCallNotifyRetrieved`

The function `TryHasRequiredFinally` returns `TRUE` if it is invoked on a `Try` which has a `Finally` clause containing, as its first statement, a call to the `update` method of `Database`. Note that it is not possible to factor out the common code of the function `ContainingTry` and the function `ContainingPackage` above. This is because GENOA lacks the necessary higher-order features, in this example, it lacks the possibility of parameterizing a test-traverse construct. The function `CAR`, which is used in the implementation of `TryHasRequiredFinally`, returns the first element of a list; in this case, the first statement in the list of statements of the `Finally` clause.

The function `StatementCallsUpdate` returns `TRUE` if it is invoked on an `ExpressionStatement` which contains a `StaticMethodCall` to the `update` method. The expression "`$parent`" denotes the current node's parent node and is equivalent to CoffeeStrainer's method `container`.

The function `IsThis`, used in the implementation of StatementCallsUpdate, shows that by defining an additional function, local variables for storing traversal results can be avoided. Note that in `StatementCallsUpdate`, only by passing `$parent` to `CallsUpdate`, we could avoid declaring a local variable either for the method call node or the result of `IsThis`.

GENOA is interesting because of its concept of traversals that operate on a current node, which is language independent and can be applied to many other tree traversal problems. However, this strength also has a downside: Accessing the ASG with GENOA is more low-level than with CoffeeStrainer; for example, the number of parameters had to be assigned to a local variable for later use because the parameter count could not be computed in the context where this number was needed. This hints at a severe restriction of GENOA: It does not allow traversals to be used where expressions are expected, leading to the definition of otherwise unneeded helper functions or local variables.

Like CoffeeStrainer, GENOA allows to specify what messages are printed when reporting violations, and the printed location depends on the current ASG node. Because GENOA is used for more than just checking constraints, it is more flexible than CoffeeStrainer in that arbitrary print commands can be issued.

A subset of GENOA, which is essentially GENOA as described so far, but without recursion and externally-defined functions, allows to specify only algorithms that are of polynomial complexity. At the same time, all polynomial-time analyses of ASGs can be expressed with this subset of GENOA, which can be proved by encoding a polynomial-time turing machine in an ASG together with a GENOA program. However, even in our simple example, we had to use recursion, and external C++ functions had to be assumed for implementing the constraints P2 and P4, because it was important to examine non-tree properties of the ASG, such as the method called by a method call expression, which are not accessible from within GENOA.

GENOA code is not very easy to read because of the many different types of parentheses used. Moreover, it seems that when using GENOA, more lines of code than in Coffeestrainer are needed for implementing the same set of constraints. Some of these lines of code are due to local variables which had to be introduced for purposes that one would like to achieve without local variables. One such example is the use of `has_public_ctor`, which simply records if there is at least one element in a tree or list that satisfies certain properties. It would be very useful if Genoa included higher-order features which would allow to specify such

```
PROC WrapWriteAccessInTryFinally
ROOT InstanceFieldAccess
{ LOCAL GNODE access_using_this;
  (IF (AND
        (EVAL AccessesPublicFieldOfStorable $token)
        (EQUAL "lvalue" $slot)
      )
    (THEN
      (ASSIGN access_using_this 0)
      <instance
        (?This
          (ASSIGN access_using_this 1)
        )
      >
      (IF (EQUAL access_using_this 0)
          (PRINT stdout
            "write access to fields of Storable object should only be accessed using this (P4)")
      )
      (IF (NOT (EVAL TryHasRequiredFinally
                     (EVAL ContainingTry $token)))
        (THEN
          (PRINT stdout
            "field acess should be wrapped in try..finally { Database.update(this); } (P4)")
  ) ) ) )
}
FUNCTION TryHasRequiredFinally
ROOT Try
{ (IF (NULL $token) (THEN (RETURN FALSE)))
  <finally <statements
      (RETURN (EVAL StatementCallsUpdate (CAR $token)))
  > >
  (RETURN FALSE)
}
FUNCTION StatementCallsUpdate
ROOT Statement
{ (?ExpressionStatement
    <expression
      (?StaticMethodCall
        <arguments
          (IF (EVAL IsThis (CAR $token))
            (THEN (RETURN (EVAL CallsUpdate $parent))))>)>)
  (RETURN FALSE)
}
```

Figure 7.8: Checking P4 with GENOA: `WrapWriteAccessInTryFinally`

```
FUNCTION ContainingTry
ROOT GNODE
{
  ... like "ContainingPackage"
}
FUNCTION IsThis
ROOT Expression
{
  (?This (RETURN TRUE))
  (RETURN FALSE)
}
FUNCTION AccessesPublicFieldOfStorable
ROOT InstanceFieldAccess
{
  ...
}
FUNCTION CallsUpdate
ROOT StaticMethodCall
{
  ...
}
```

Figure 7.9: Helper functions for implementing P4

constraints in a straightforward way, or if a primitive was included with which one could express such constraints directly, like the existentially qualified variables in CCEL.

7.5 ASTLOG

ASTLOG, which has been implemented for C++, is a language for examining abstract syntax trees. It is similar to Prolog; the language is used to express matching conditions for certain tree structures. Static constraints can be expressed as ASTLOG programs that match violating constructs.

Like GENOA, ASTLOG has a notion of a "current object" — in fact, every ASTLOG term operates on a current object, which can be a node of the abstract syntax tree, a list, or an integer. For example, consider the problem of detemining the length of a string. In a functional language, one would write `strlen(string)` to calculate the length of the variable string. In Prolog, the same calculation would look like `strlen(string, length)`, which can be used to make sure that the length of `string` is indeed `length`. Using unification, an unbound variable can be bound to the actual string length. In ASTLOG, one would write `strlen(`length-pred`)`. Here, the string is the implicit current object, and the string length is supplied as the implicit current object of "length-pred". This programming style, which the author calls "functional inside-out", is claimed to be particularly well-suited to the matching of tree structures.

Like for GENOA, it is possible to define a top-level ASTLOG predicate which applies the constraint implementations for P1, P2, and P4 to a complete abstract syntax tree. The re-

cursive predicate `check`, shown in Figure 7.10, can be evaluated in the context of an abstract syntax tree root node. It tries to match one of the predicates `match_p1_violation`, `match_p2_violation`, or `match_p4_violation` to the current node and emits an error message for each such match. The top-level term `or` matches the current object if one of its comma-separated operands match the current object; and analogously, the term `and` matches the current object if all of its comma-separated operands match the current object. The term `emit` matches for all current nodes and prints its argument string. The predicate `check` is applied recursively to all child nodes of the current node. The term `kid(`int-pred`,` node-pred`)` matches if the "int-pred" matches an index of a child node of the current node, and the term "node-pred" matches the corresponding child node. As in Prolog, the special variable "_" is an anonymous variable which unifies with anything.

As already noted for GENOA, this structure is similar to CoffeeStrainer's built-in visitor-like ASG traversal. The implicit traversal in CoffeeStrainer, although not as flexible as a custom traversal, has the advantage that there is no top-level procedure or predicate which has to be changed whenever a new constraint is added.

```
check()
  <- or(
       and(
         match_p1_violation(),
         emit("violation of p1")
       ),
       and(
         match_p2_violation(),
         emit("violation of p2")
       ),
       and(
         match_p4_violation(),
         emit("violation of p4")
       ),
       kid(_, check())
     )
```

Figure 7.10: Implementing P1, P2, and P4 as a top-level ASTLOG predicate

We will now explain the ASTLOG implementation of constraint P1 (public empty no-arg ctor needed) in detail. The constraint, shown in Figure 7.11, is formulated as an ASTLOG query so that all matches to this query that are reported by ASTLOG are violations of the constraint. The term `op(`int-pred`)` matches if the current object is a tree node, and the "int-pred" matches the opcode of that tree node. The integer constant `#CLASS` is the opcode for tree nodes representing classes. An integer constant matches the current object if the current object is an integer of the same value. The term `implements(`node-pred`)` matches if the current object is a tree node representing a class and if the contained "node-pred" matches an interface implemented by that class. We assume the constant `storable_interface` to refer to the tree node representing the interface `Storable`.

The term `not(`obj-pred`)` matches if "obj-pred" does *not* match the current object. Note that matching in ASTLOG is performed using backtracking as in Prolog. The constant `#CON-`

STRUCTOR/PARAMETERS stands for the index of the child of a constructor node that contains the nodes representing parameters.

```
match_p1_violation()
  <- and(
       op(#CLASS), // all classes that implement Storable
       implements(storable_interface),
       not(
         kid(_,  // match if there is no kid
           and(
             op(#CONSTRUCTOR), // which is a constructor
             not(              // with no args
               kid(#CONSTRUCTOR/PARAMETERS,
                 kid(_, _)
               )
             ),
             is_public(),      // declared as public
             kid(#CONSTRUCTOR/BODY,
               not(            // and empty
                 kid(_, _)
               )
             )
           )
         )
       )
     );
```

Figure 7.11: Implementing P1 with ASTLOG

This query matches all nodes of type `Class` which represent classes that implement the interface `Storable` for which there exists no match for a child node of type constructor that has no arguments, is declared `public`, and has no containing statements. The term `is_public()` is defined in Figure 7.12; it matches the current node if it has a `public` modifier as one of its child nodes.

In ASTLOG, it is easier to define general abstractions than in GENOA. For example, the predicate `containing`, which is equivalent to CoffeeStrainer's `containing(Class c)`, as shown in Figure 7.12, cannot be implemented in GENOA. The term `containing(Optype, Pred)` matches the current node if it has a containing node of type `Optype` for which `Pred` matches. It makes use of `parent`(`node_pred`), which matches the current node if its parent node matches `node_pred`.

Constraint P2 (no calls to `notifyRetrieved`) can be implemented as a query which matches nodes of type `INSTANCE_METHOD_CALL` (see Figure 7.13). The term `calls_method`(`node_pred`) matches if the current object is a node representing a method call to a method whose representing node matches `node_pred`. The term `asym(sname`(`string_pred`)) matches the current node if it is a symbol whose name matches `string_pred`. As with numbers, string constants match the current node if it is a number of equal value.

Figure 7.14 shows the implementation of P4 (write accesses to `public` fields of `Storable`

```
is_public()
  <- kid(#MODIFIERS,
       kid(_,
         op(#PUBLIC)
       )
     );

containing(optype, pred)
  <- parent(
       if(
         op(optype),
         pred,
         containing(optype, pred)
       )
     );
```

Figure 7.12: ASTLOG utility predicates

```
match_p2_violation()
  <- and(
       op(#INSTANCE_METHOD_CALL),
       calls_method(
         and(
           asym(
             sname("notifyRetrieved")
           ),
           containing(#CLASS,
             implements(storable_interface)
           )
         )
       )
     )
```

Figure 7.13: Implementing P2 with ASTLOG

objects should be wrapped in try-finally).

```
match_p4_violation()
  <- and(
       op(#INSTANCE_FIELD_ACCESS), // all instance field accesses
       accesses_field(
         and(
           parent(
             implements(storable_interface)
           ),
           is_public() // public
         )
       ),
       not(
         containing(#TRY,
           kid(#TRY/FINALLY,
             kid(#FINALLY/BODY,
               kid(0,
                 and(                          // first statement
                   op(#EXPRESSION_STATEMENT), // of type expr_statement
                   kid(0,                      // its expression
                     and(
                       op(#STATIC_METHOD_CALL), // is static method call
                       calls_method(database_update) // to db.update
                       kid(#STATIC_METHOD_CALL/ARGUMENTS,
                         kid(0,
                           op(#THIS)
                         )    // with arg "this"
                       )
                     )
                   )
                 )
               )
             )
           )
         )
       )
     )
```

Figure 7.14: Implementation of P4 with ASTLOG

With ASTLOG, one cannot implement the dynamic constraint P3.

ASTLOG is a well-designed language that applies the idea of a current object to every language construct (every term). It is difficult to say whether ASTLOG programs inherently are difficult to understand or whether this is due to the unusual programming style which seems inside-out to conventional programmers. The language is based on only a few primitives for accessing abstract syntax trees, which keeps the language small but also makes accessing the AST lower-level than necessary. Unlike GENOA, ASTLOG includes higher-order features which enable better reuse of ASTLOG code.

The author notes that arithmetic expressions are extremely difficult to read. For example,

the term `divide(minus(prime(),1),2)` matches if the current object is an integer that, when *multiplied* by two, yields an integer that when *incremented* by one is prime. For example, the term matches if the current object is `5`, because $5 * 2 + 1 = 11$ is prime.

It is interesting to see, when comparing the implementation of constraint P4 with the corresponding CoffeeStrainer constraint method, that using a special-purpose language tailored for tree matching does not necessarily lead to more compact or more readable code for matching trees.

7.6 Summary

While CoffeeStrainer, GENOA, CDL, and ASTLOG operate on complete abstract syntax trees, i.e. including method bodies with statements and expressions, both CCEL and LGA provide only partial representations of the parsed program. In CCEL, constraints have access only to top-level declarations, namely to class declarations, signatures of methods, and field declarations. Constraints in LGA are restricted to operations defined on objects in the abstract object model - object creation and deletion, reading and writing object properties, and invoking methods on objects. Thus, constraints that refer to the control flow, to declarations of parameters and variables, and constraints that take language-specific features into account are not possible. Additionally, it should be noted that CDL provides only a simple parse tree without name and type analysis information, and that it is not well-documented to what extent GENOA and ASTLOG support accessing information obtained from name and type analysis.

GENOA, CCEL, LGA, and ASTLOG allow more concise constraint specifications because they use declarative constraint languages. To assess CDL in this respect is not easy, because it is a restricted language in which consistency of a set of constraints is statically decidable, but implementing certain constraints requires the implementation of predicates that must be implemented outside CDL. For CoffeeStrainer, much care has been taken to make constraints as declarative as possible, but the fact that constraint methods are written in Java makes them less declarative and concise than they should be. However, CoffeeStrainer has the advantage that the programmer need not learn a new language or syntax for specifying constraints — as in the other systems — and thus makes it much more accessible for practitioners. For example, the language used for specifying constraints in CCEL, although very similar to the base-level language C++, needs more than three pages of grammar description [Duby et al. 1992]. Moreover, constraints defined in ASTLOG are very difficult to read because of their higher-order and inside-out style. GENOA constraints would be easier to understand if the language did not rely on four different kinds of parentheses.

For implementing the example constraints, externally-defined functions or predicates were needed for GENOA, LGA, and CDL. This usually means dealing with yet another language — usually, C or C++, the particular language used for implementing the constraint checking system. CCEL is difficult to assess in this respect, as it covers only small parts of a program's ASG. Only for ASTLOG and CoffeeStrainer, it seems to be sufficient to learn one language — in the case of CoffeeStrainer, because the constraint language is the same as the base-level language and the language used for implementing the system, and in the case of ASTLOG because it appears to be powerful enough to express many different kinds of (static) example constraints.

CoffeeStrainer allows to specify constraints that are modular, customizable and composable. Constraints can be extended and refined in subtypes, and well-known object-oriented structuring techniques can be used for constraints. By using interface inheritance, marker interfaces can be defined that combine several constraints. Because multiple inheritance is allowed for interfaces, this idiom is quite flexible. In CoffeeStrainer, constraints are associated with classes and interfaces, so that different compilation units can be checked separately. Most of these possibilities are not present at all, or only limited in the other systems. GENOA, CCEL, LGA, and ASTLOG check all constraints globally, and a list of all constraints has to be searched when the programmer wants to find out what constraints apply to a given class. In CCEL, there are no provisions for composing or extending constraints. GENOA, LGA and ASTLOG support composition of constraints, by means of user-defined abstractions like procedures, functions, or predicates. Similarly, common constraint code can be factored out, but in contrast to CoffeeStrainer, this issue has to be considered in advance — it is not possible to reuse constraint code without changing it. ASTLOG includes powerful higher-order features for better reusability. In CDL, it is as easy as in CoffeeStrainer to find constraints that apply to a type, because language constructs are annotated with constraint names. However, this requires the base language's syntax to be extended, which is not an option in most software development projects.

Usage constraints cannot be specified directly in CDL at all, because constraints are only applied to annotated constructs. CCEL deals only with restricted sets of usage constraints; it only supports usage constraints based on explicitly naming other types in class definitions, field declarations, and method signatures. LGA supports usage constraints directly, based on interactions that take place at run-time, based, e.g., on method calls and object allocations. However, it would be difficult to constrain, say, downcasts to a specific type using LGA, because it is not clear what interaction takes place when casting. Although the other systems do not support usage constraints directly, they can be written as definition constraints that apply globally.

Regarding efficiency, it seems that CDL is best, because it is based on a restricted formalism, followed by GENOA, which allows only polynomial-time constraints. CCEL and CoffeeStrainer follow; they traverse the AST of the program to be checked exactly once and cannot apply the optimizations that are possible in CDL. LGA and ASTLOG, which are based on Prolog, support backtracking and unification and thus trade expressiveness for efficiency.

An issue that has not been discussed so far is that of static typing of the constraint code. Because CoffeeStrainer uses Java for constraint methods, the constraint code is statically type-checked. For all other systems, no static type checking is performed, potentially leading to more errors in constraint code that will only detected at check-time. Furthermore, the language-independent systems GENOA, CDL, and ASTLOG all provide a very low-level access to the ASG, for example, by referring to child nodes by index rather than by name, and they do not distinguish between different types of ASG nodes, although some of their primitives are defined only for certain node types.

Figure 7.15 shows, for each example constraint and each system, the number of lines of code needed for implementing the constraint. As always, such metrics should be taken with a grain of salt. However, we think that the information is nevertheless interesting. In particular, one can observe the following:

- For GENOA, LGA, and CDL, an unknown number of lines of code, written in a dif-

ferent language, is needed *additionally* to implement externally-defined functions or predicates.

- With ASTLOG, it was possible to use a predicate with 7 lines of code, the equivalent of CoffeeStrainer's `containing(Class c)`, in two constraint implementations.
- One of the systems that define a special-purpose language for traversing and examining trees, GENOA, has by far the longest implementations. As already noted, this seems to be due to the fact that GENOA distinguishes between expressions and traversal constructs, making it cumbersome to use results from certain traversal in expressions, e.g., in conditional expressions of `IF` statements.
- Interestingly, the CoffeeStrainer constraint implementation are the most concise, even in the case of P4 which requires matching a certain tree structure and looks not as concise as it could be if Java (or the constraint language) had pattern matching features. Even ASTLOG, which has the most powerful constraint language, requires more lines of code for implementing P4.

	P1 def. constr.	P2 usage constr.	P3 dynamic constr.	P4 tree matching
CCEL	5 (incomplete)	-	-	-
LGA	5 + ext. pred.	2 + ext. pred.	2 + ext. pred.	-
CDL	4	2 + ext. pred.	-	5 + ext. pred.
GENOA	35	17 + ext. func.	-	44 + ext. func.
ASTLOG	13	9 + 7 reused	-	33 + 7 reused
CoffeeStrainer	3	1	2	15

Figure 7.15: Lines of code needed for implementing the example constraints

Figure 7.16 compares the systems that have been discussed on an abstract level. A "complete AST" is given if the AST covers all language constructs and includes semantic information from name and type analysis. One one side of the expressiveness scale, ASTLOG's scores very high because it has higher-order features. In CDL, on the other side, certain constraints cannot be specified at all, but consistency of a set of constraints can be checked automatically. CoffeeStrainer is the only system that allows modular checking and modular composition of constraints, and no other system has support for usage constraints comparable to CoffeeStrainer. The potential for optimizations is highest for CDL with its restricted formalism and lowest for ASTLOG with its higher-order capabilities.

7.7 Other related work

There are a number of other systems similar to CoffeeStrainer which have not been included in the detailed comparison. They are discussed in Section 7.7.1. Section 7.7.2 presents related work which is concerned with certain kinds of constraints and not with support for checking general constraints. In Section 7.7.3, we briefly discuss techniques which might help to avoid constraints in complex object-oriented systems.

System	language	complete AST	expressiveness	modular	usage constraints	efficiency
CCEL	C++	no	medium	no	no support	high
LGA	generic / Eiffel	no	high	no	restricted support	medium
CDL	generic	no	low	no	not possible	very high
GENOA	generic / C++	yes	medium / low	no	no support	high
ASTLOG	C++	yes	very high	no	no support	medium
CoffeeStrainer	Java	yes	high	yes	yes	high

Figure 7.16: Summarized comparison of systems for static constraint checking

7.7.1 Other systems similar to CoffeeStrainer

We are aware of one other system for checking constraints, called PATTERN-LINT [Sefika et al. 1996], which addresses both static and dynamic constraints. Constraints can either be low-level rules, architectural rules, or heuristic rules. Only low-level rules are meant to be specified by a programmer. Architectural rules are predefined for a number of typical architectural styles [Garlan et al. 1994] and design patterns [Gamma et al. 1995] and need only be applied to a given program. Heuristic rules are quantitative rules that are checked by a human; PATTERN-LINT supports these kind of constraints by providing graphical visualizations. The system could not be included in the detailed comparison because the language used for implementing constraints is not documented.

There are a number of systems for querying source code which might be used for checking constraints.

The experimental language A* [Ladd, Ramming 1995] is AWK-like [Aho et al. 1988] except that it operates on concrete syntax trees instead of on file records. It allows to specify arbitrary traversals of the concrete syntax tree, and traversal code can be separated from the code which operates on specific nodes in the tree. A* has been applied for implementing a number of source code tools for various languages. The authors note that A* is a good choice for implementing tools which operate on concrete syntax and which need no or little semantic information. Thus, although the constraints P1–P4 could, in principle, be implemented in A*, we consider this a task too tedious to be practical.

Also based on AWK, the tool TAWK [Griswold et al. 1996] allows to specify source code queries based on abstract syntax trees. For efficiency reasons, a programmer needs to specify the granularity of the queries he writes, i.e., the size of the program fragments which may match the query. TAWK performs a predefined traversal of the AST and executes actions written in the programming language C upon matches. We have not included TAWK in our detailed comparison because it does not provide semantic information, and because the language for writing queries is not described in sufficient detail to enable a reader to write TAWK programs on his own.

The source code search tool SCRUPLE [Paul, Prakash 1994] is a based on a finite state machine model. Queries are written in an extended version of the underlying programming

language which incorporates special wildcard symbols for matching certain syntactic entities or collections of syntactic entities. The primary focus of SCRUPLE is supporting efficient syntax-based queries. The constraints P1–P4 could not be implemented using SCRUPLE because it does not provide semantic information, nor is its computation model powerful enough to calculate the needed semantic information. The authors of SCRUPLE, in a later paper [Paul, Prakash 1996], have also proposed a theoretical model for querying source code, called the Source Code Algebra (SCA), which is much more expressive and allows accessing semantical information.

Horwitz and Teitelbaum propose a model for querying source code which combines an attributed abstract syntax tree with an associated relational database [Horwitz, Teitelbaum 1986]. The authors show how the weaknesses of relational queries (e.g., it is impossible to compute the transitive closure of a relation) are covered by the strengths of attribute grammars and vice versa. Their model is appealing because it incorporates an incremental update algorithm which can be used in a language-aware editor to check constraints efficiently while writing a program. However, the model is not easily applied to a complex language like Java, because the definition of attributes and relations are interspersed in the grammar definition. Implementing the constraints P1–P4, would require several additions to a Java grammar. Because the issue of modularity of attribute and relation definitions is not addressed, it is unclear whether these additions would be local or global changes to the grammar.

7.7.2 Related work on constraints

In [Helm et al. 1990], *contracts* as a high-level formalism for specifying mutual usage constraints are proposed. Unfortunately, the formalism is too expressive to be statically checkable. Type definition constraints appear in [Lamping 1993] as the *specialization interface* between a class and its subclasses. This notion is refined in [Steyaert et al. 1996] which introduces the notion of *reuse contracts* for constraints regarding assumptions of a class with respect to its subclasses. We consider CoffeeStrainer an ideal platform for implementing reuse contracts for Java. [Gil, Eckel 1997] presents formal definitions of *traits*, a notion very similar to structural constraints. Again, CoffeeStrainer would be a good candidate for implementing a system that can enforce traits.

An interesting method for extending standard static type systems with *type qualifiers* is proposed in [Foster et al. 1999]. Type qualifiers can be used to express static properties of variables; a well-known example of a type qualifier is the qualifier `const` from the programming language C++ [Stroustrup 1991]. Another example is the qualifier `nonnull`, described in the paper, which is similar to `NullValueInvalid` of Section 4.3.3. The advantage of using type qualifiers to express such properties is that they can be checked within the type system, thus guaranteeing soundness and termination of the checking process, and that types can be polymorphic in the type qualifiers, unlike `NullValueInvalid`, which is tied to a particular type.

7.7.3 Avoiding constraints

Constraints occur in complex object-oriented software because design-specific consistency requirements cannot be expressed in general-purpose programming languages. Rather than checking constraints, other research is concerned with techniques for building complex object-oriented software while avoiding the need for constraints.

Aspect-oriented programming [Kiczales et al. 1997][Walker et al. 1999] separates the functional core of a software system from its non-functional *aspects* such as concurrency, distribution, efficieny, etc., which usually are the reason for much of the complexity of software. The core of the system is written in a general-purpose programming language, and each of the aspects is specified in a separate *aspect language*. Then, a tool, called the *aspect weaver*, generates a resulting program by interleaving the functional core with code that handles the non-functional aspects. Aspect-oriented programming is based on the assumption that there is a finite number of aspects for which an aspect weaver can be provided. Because of the complex interdependencies between different aspects, adding a new aspect is a difficult task which cannot be handled by normal programmers. Currently, it is unclear how much of the underlying problem can be solved using the techniques of aspect-oriented programming. Moreover, design-specific consistency requirements might appear in the functional core of a system, requiring constraints which cannot be replaced by carefully-chosen new aspects.

Another possibility to avoid constraints would be to use domain-specific programming languages [USENIX 1997] instead of general-purpose programming languages. However, constraints from the general-purpose programming language would most likely appear in the type system of the domain-specific programming language. Thus, to provide the same safety as with constraints added to a general-purpose programming language, one ends up with more work since all aspects of the domain-specific language have to be worked out and not just the type system, which pays off only if the domain-specific language can be used for writing a large number of programs.