A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information

Patrick Lam Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{plam, rinard}@lcs.mit.edu

Abstract. We present a new type system and associated type checker, analysis, and model extraction algorithms for automatically extracting models that capture aspects of a program's design. Our type system enables the developer to place a *token* on each object; this token serves as the object's representative during the analysis and model extraction. The polymorphism in our type system enables the use of general-purpose classes whose instances may serve different purposes in the computation; programmers may also hide the details of internal data structures by placing the same token on all of the objects in these data structures.
Our combined type system and analysis provide the model extraction algorithms with sound heap aliasing information. Our algorithms can therefore extract both structural models that characterize object referencing relationships and behavioral models that capture indirect interactions mediated by objects in the heap. Previous approaches, in contrast, limited by an absence of aliasing information, have focused on control-flow interactions that take place at procedure call boundaries. We have implemented our type checker, analysis, and model extraction algorithms and used them to automatically extract design models. Our experience indicates that it is straightforward to produce the token annotations and that the extracted models provide useful insight into the structure and behavior of the program.

1 Introduction

Design abstractions such as object models [12] and module dependency diagrams are a central feature of many software development processes. In this capacity they provide a way to quickly and easily explore design alternatives and give the members of the design team a common and effective language for communicating important aspects of the design.

* This research was supported in part by a fellowship from Canada's Natural Sciences and Engineering Research Council, DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR-0086154, NSF Grant CCR-0073513, NSF Grant CCR-0209075, an Eclipse Innovation Grant, and the Singapore-MIT Alliance.

In principle, the design abstractions should remain a primary source of information about the program for its entire lifetime. But the standard practice is for programmers to manually implement the design once it has been finalized, raising the possibility of the implementation diverging from the design. This divergence becomes ever more likely over the lifetime of the program, limiting the credibility of the original design and therefore its utility as a source of information about the program. In most cases, the design is eventually discarded and the code becomes the primary source of information about the program.

This paper presents a new type system and an associated analysis that together support the automatic extraction of design-level information from the source code. The goal is to establish a guaranteed connection between the program and its design, restore the credibility of the design as a reliable source of information about the program, and enable developers to use design abstractions effectively throughout the entire lifetime of the program.

We focus on abstractions that involve the structure of the heap and the information flow (or lack of such flow) between different subsystems. One particularly novel aspect of our technique is that it accurately captures even indirect interactions mediated by objects in the heap. Existing approaches, in contrast, focus only on the direct interactions that take place at procedure or method calls.

The key idea behind our approach is to allow the developer to use the type system to place a *token* (chosen from a finite set of tokens fixed at program analysis time) on each object in the program; this token serves as the object's representative during the analysis that extracts the design information from the program. This approach addresses several common problems that complicate the effective automatic extraction of design information:

- **Multiple Design Elements, Single Code Element:** Well-structured programs factor common behavior and structure into a single, general-purpose code element (for example, a container class or object factory). Different instantiations of such an element often have distinct conceptual purposes in the computation and should therefore correspond to different elements in the design. But standard analysis approaches treat each code element as a unit, conflating the attributes of its different instantiations and failing to capture important design-level distinctions.

  The polymorphism in our type system eliminates this problem. It allows the developer to place different tokens on different instantiations of the same class so that the analysis separates objects with different conceptual purposes even if the objects happen to be instances of the same general-purpose class.
- **Single Design Element, Multiple Code Elements:** Because the design captures aspects of the computation at a higher level of abstraction than the code, multiple code elements are often required to implement a single design element. For example, a primary object may maintain complex internal data structures that the design abstracts as conceptually part of the object. Any approach that fails to abstract these internal data structures will deliver an overly detailed model that obscures key aspects of the design.

Our type system addresses this problem by allowing the developer to place the same token on both the primary object and all of the objects that implement its internal data structures. The analysis then treats the entire collection of objects as a unit and appropriately coalesces the combined information from all of the objects into a single design element.
Consider, for example, a set object with an internal linked list of references to items in the set. Our system allows the developer to place the same token on both the set object and all of the linked list objects, with a separate token on the items that the list nodes reference. In the extracted models, the set and all of its internal linked list nodes comprise a single abstraction. Because the items in the set have a different token, they correspond to a separate abstraction.

– **Aliasing:** To accurately extract structural information (for example, referencing relationships between objects) and behavioral information (for example, how information flows between subsystems), the analysis needs to have information about the aliasing relationships in the heap. An expensive whole-program pointer analysis is the standard way to obtain this information. Pointer analyses typically use the creation site of each object to represent the object during the analysis, in which case the analysis results conflate all objects allocated at the same site and fail to appropriately coalesce internal objects.
In our type system, the type of each object completely characterizes the referencing relationships (at the granularity of tokens) in the part of the heap reachable from that object. Instead of processing all of the load and store statements to construct a model of the heap, our analysis can simply propagate token information across procedure boundaries to substitute out the token variables in the polymorphic types. The resulting ground types provide the required aliasing information.

We present the extracted design information to the developer via a set of models. Each model is designed to capture a specific kind of design information; together, the models provide a comprehensive summary of the relationships between the structural and behavioral aspects of the design. In particular, our models help the developer visualize referencing relationships between objects in the heap and understand the full range of interaction relationships between subsystems.

1.1 Object Models

An object model identifies the kinds of objects in the heap and characterizes the relationships between these different kinds of objects [12]. We model the objects and relationships at the granularity of tokens. Specifically, there is a node in the model for each token. There is a labelled edge between two tokens if the heap may contain two objects represented by the tokens and one object may contain a reference to the other. The label identifies the field containing the reference.

Building the model at the granularity of the tokens separates conceptually distinct instances of the same class and enables the model to appropriately

capture the different structural relationships associated with these different instances.The standard approach, in contrast, operates at the granularity of classes and fails to capture these distinctions [18].

1.2 Subsystem Access Models

These models characterize how subsystems access objects. Each of these models is a bipartite graph. There is a node for each token and a node for each subsystem, with an edge from a subsystem to a token if the subsystem may access an object represented by the token.

1.3 Interaction Models

Interaction models characterize interactions between subsystems at the granularity of tokens. We support two kinds of models:

- **Call/Return Interaction Model:** This model characterizes the direct interactions that take place at method calls and returns. The nodes in the call/return model are subsystems. There is a solid directed edge from subsystem s_1 to s_2 if a method in s_1 invokes a method in s_2. The edge is labelled with the tokens that represent the objects passed as parameters in any s_1 method calling s_2. There is a dashed directed edge from s_2 to s_1 if some method in the s_2 subsystem returns a result to a method in s_1. The edge is labelled with all tokens representing objects returned from s_2 to s_1.
- **Heap Interaction Model:** This model characterizes the indirect interactions that take place at reads and writes to and from objects in the heap. The nodes in this model are tokens. There is a solid directed edge between two tokens if a subsystem writes a reference to an object represented by the first token into an object represented by the second token. The label on the edge identifies the subsystem that performed the write. There is a dashed directed edge between two tokens if a subsystem reads a field in an object represented by the first token and obtains a reference to an object represented by the second token. The label on the edge is the subsystem that performed the read.
  This model smoothly generalizes to support higher-level actions (such as insertions and removals) on abstract data types (such as hashtables and lists).

Together, these models enable the developer to trace all of the dependences between and flow of information through the subsystems in the program. They also support useful projection operations — to focus on a particular aspect of the interactions, the developer selects the relevant subsystems or tokens, then hides those parts of the model that do not involve these subsystems or tokens. The resulting projected models clearly expose the properties of interest.

Our enhanced subsystem models succinctly capture all of the information in standard subsystem interaction models (which focus on aspects of the control flow; in particular, on how methods in one subsystem invoke methods in

other subystems). But the availability of a sound, relevant model of the heap also enables the analysis to characterize not only the control flow but also the information flow that occurs at method calls. Perhaps more significantly, it can also characterize how subsystems access data and capture indirect subsystem interactions mediated by objects in the heap.

1.4 Contributions

This paper makes the following contributions:

- **Polymorphic Token Type System:** It presents a polymorphic type system that allows developers to place a token on each object. This type system is structured as an extension to Java, and includes a type checking algorithm that determines if the type declarations are correct.
- **Analysis and Model Extraction Algorithms:** It presents an analysis algorithm and model extraction algorithms that, together, use the type system to extract models that capture aspects of the design of the program. This extraction-based approach ensures that the models correctly reflect the design of the program. In contrast with many previous approaches, the presence of sound heap aliasing information enables the extraction of both structural models that characterize object referencing relationships and behavioral models that capture indirect interactions mediated by objects in the heap.
- **Experience:** We have implemented our type system, analysis, and model extraction algorithms. We have used these algorithms to produce design models. Our experience indicates that it is straightforward to produce the token annotations and that the extracted models provide useful insight into the structure and behavior of the program.

2 Example

We next present an example that illustrates how our analysis produces interaction models. Figure 1 presents a program in which a driver coordinates the activities of a producer and a consumer. The producer and consumer interact via a stack of objects; the driver creates the stack, then repeatedly invokes the producer (which pushes some `Int` items on to the stack) and the consumer (which pops the `Int` items off of the stack). There are two kinds of interactions: *call/return interactions* in which the stack flows between the driver, the producer, and the consumer, and *heap interactions* in which the produced items flow from the producer through the stack to the consumer. We next discuss how our analysis produces models that present information about this program.

2.1 Subsystems

Our analysis describes the behavior of the system at the granularity of *subsystems*. Each subsystem corresponds to a set of method invocations that serve

```
token     P, C, D, PCS, PCI;
subsys    EP, EC, ED;

class Int<i> {
    int v;
    Int(int v) { this.v = v; }
}
class Node<s,i> {
  Node<s,i> next;
  Int<i> data;
}
class Stack<s,i> {
  private Node <s,i> first;
  public void push (Int<i> k) {
    Node <s,i> n =
      new Node<s,i>();
    n.data = k;
    n.next = first;
    first = n;
  }
  public Int<i> pop() {
    Int<i> r = first.data;
    first = first.next;
    return r;
  }
}
class Producer<p,s,i> enter EP {
  int n = 0;
  public void produce
         (Stack<s,i> s) {
    s.push(new Int<i>(n++));
  }
}
class Consumer<c,s,i> enter EC {
  Int<i> r;
  public void consume
         (Stack<s,i> s) {
    r = s.pop();
  }
}
class Driver<d> enter ED {
  public void enter() {
    Stack<PCS,PCI> s =
      new Stack<PCS,PCI>();
    Producer<P,PCS,PCI> p =
      new Producer<PT,PCS,PCI>();
    Consumer<C,PCS,PCI> c =
      new Consumer<C,PCS,PCI>();
    while (true) {
      p.produce(s);
      c.consume(s);
    }
  }
}
class ProducerConsumer {
  public static void main
           (String[] argv) {
    new Driver<D>().enter();
  }
}
```

Fig. 1. Example Producer/Consumer Program

the same conceptual purpose in the computation. Our example contains four subsystems: the `MAIN` subsytem that executes the `main` method, the `EP` (Event Producer) subsystem that produces the data, the `EC` (Event Consumer) subsystem that consumes the data, and the `ED` (Event Driver) subsystem that invokes the `EP` and `EC` subsystems.[1]

The program identifies some of the classes as subsystem entry points. In our example, the program uses the `enter EP` clause to identify all of the methods in the `Producer` class as entry points to the `EP` subsystem, and similarly for the `EC` and `ED` subsystems. In particular, any call to a static or instance method on

[1] In practice, we would expect the subsystems to be much larger. We adopt this fine subsystem granularity in our example for expository purposes.

a class which is a subsystem entry point triggers a subsystem change; we define an entry method to be any method on an entry class.

Once the program enters a subsystem, it remains within that subsystem until it invokes a method in a class that is an entry point for a different subsystem. So in our example, execution starts within the `MAIN` subsystem, then moves into the `ED` subsystem when the `main` method invokes the `enter` method. The `ED` subsystem then invokes the `EP` and `EC` subsystems to produce and consume the data.

Note that because the `push` and `pop` methods are not subsystem entry points, invocations of these methods are part of the same subsystem that invoked them. This approach enables the construction of general-purpose classes that may be used for different purposes in different subsystems.

2.2 Polymorphic Token Types

Each class has a set of token parameters. The first parameter identifies the token placed on the class; the other parameters are used to declare the types of the reference fields of instances of the class. In our example, the `Stack <s, i>` class has two parameters: the token variable `s` identifies the token placed on stack instances and the token variable `i` identifies the token placed on items in the stack. The class can use these token variables to declare the types of its reference fields and the types of the parameters of its methods.

The program specifies values for the token parameters at object creation sites. In our example, the `enter` method uses the statement

```
Stack<PCS,PCI> s = new stack<PCS,PCI>();
```

to create a new instance `s` of the `Stack` class with tokens `PCS` (producer/consumer stack) and `PCI` (producer/consumer item). This object creation site uses concrete token values (`PCS` and `PCI`). It is possible, however, for the program to use token variables to specify the tokens at object creation sites. Consider, for example, the object creation site `new Int<i>(n++);` inside the `produce` method. This site uses the token variable `i` to identify the token placed in the newly created `Int` object.

As our example illustrates, token variables support a form of polymorphism in which different instantiations of the same class can have different tokens. This mechanism supports general classes whose instances serve different conceptual purposes in the computation.

2.3 Analysis

The goal of our analysis is to compute, at the granularity of tokens, the referencing relationships within the program. This information allows the analysis to characterize structural relationships in the heap. It also serves as a foundation for computing behavioral information about how subsystems access and share information.

Our analysis processes the object creation and method call statements to propagate token variable binding information from callers to callees. In effect,

the analysis substitutes out all of the token variables from all of the types, replacing the variables with the concrete tokens on objects that actually appear when the program runs.

In our example, the analysis propagates token bindings from the `enter` method to the `produce` and `consume` methods as follows. At the call to the `produce` method, the analysis uses the declared types of `p` and `s` to generate the binding $[\texttt{p} \mapsto \texttt{P}, \texttt{s} \mapsto \texttt{PCS}, \texttt{i} \mapsto \texttt{PCI}]$ for the token variables in the `produce` method. It then propagates these bindings to generate the binding $[\texttt{s} \mapsto \texttt{PCS}, \texttt{i} \mapsto \texttt{PCI}]$ for the token variables in the `push` method. In a similar way, the analysis can substitute out the token variables in the `consume` and `pop` methods to obtain a complete set of bindings for all of the token variables in the program.

The token propagation algorithm also propagates the current subsystem identifier between invoked methods. The combined analysis result contains both the token variable bindings and a binding that indicates the subsystems that may execute each method. So, in our example, the analysis computes that the `push` method may execute as part of the `EP` subsystem, and that the `pop` method may execute as part of the `EC` subsystem.

At this point, the analysis can use the bindings to compute, for each local variable, the set of tokens that represent the objects to which the variable may refer. As described below in Sections 3.4, 3.5, and 3.6, this information enables the analysis to produce models that characterize the objects that each subsystem may access and the ways that information may flow between subsystems.

As described below in Section 3.3, the bindings at object creation sites, when combined with the type declarations for object fields, enable the analysis to produce an object model that characterizes the referencing relationships between objects at the granularity of tokens.

Finally, the question may arise how to combine binding information when different invocations of a single method may have different token variable bindings. Our framework supports both context sensitive approaches (which provide a separate result for each different combination of the values of the token variables and subsystems in each method) and context-insensitive approaches (which combine the different contexts to generate a single mapping of token variables to possible values valid for all executions). An intermediate approach combines contexts from the same subsystem but keeps contexts from different subsystems apart. Our implementation uses a context sensitive approach, which keeps distinct sets of token variable bindings for each distinct invocation of a method.

2.4 Object Models

In our system, the concrete type of each object, in combination with the types of the objects that it (transitively) references, characterizes the structure of the heap reachable from the object. Once our analysis has computed the bindings for the token variables at each object allocation site, it can use the type declarations for the fields of the object to build an object model that characterizes the referencing relationships in the part of the heap reachable from that object. This

object model is a labelled, directed graph. The nodes in the graph correspond to tokens; there is an edge between two tokens if one of the objects represented by the first token may contain a reference to an object represented by the second token. The label on the edge is the name of the field that may contain the reference.

By combining the object models from each of the object creation sites, the analysis can produce a single object model that characterizes, at the granularity of tokens, all of the referencing relationships in the entire heap. In some cases it is also desirable to summarize local variable referencing relationships in the object model. Our tool can therefore process the local variable declarations to insert an unlabelled edge between two tokens if a method of an object represented by the first token has a local variable that may refer to an object represented by the second token. Figure 2 presents the object model from our example; this object model contains the unlabelled edges from local variables.[2]

2.5 Subsystem Access Models

Our analysis processes the statements in each method in the context of the token variable binding information to extract a subsystem access model. This model characterizes how subsystems access objects at the granularity of tokens. Each subsystem access model is a bipartite graph. The nodes in the graph correspond to subsystems and tokens; there is an edge connecting a subsystem and a token if the subsystem may access objects represented by the token.

Figure 3 presents the subsystem access model from our example program. The square nodes represent subsystems; the ellipse nodes represent tokens. The edge between `EP` and `PCS`, for example, indicates that `EP` may access the stack used to pass values between the producer and consumer.

Note that this model is not designed to reflect object creation relationships. For example, the `ED` (example driver) subsystem creates the `Stack` object (represented by the token `PCS`), the `Producer` object (represented by the token `P`), and the `Consumer` object (represented by the token `C`). The subsystem interaction model is not intended to present these relationships — our analysis does have enough information to present this object creation information, but we believe it would be best presented in a separate model that deals only with object creation relationships. Note also that this model is not designed to present relationships involving primitive fields — the `EP` subsystem accesses the primitive field `n` in the `Producer` object, but the model does not contain this information. Once again, the analysis has the information required to present such relationships, but we believe it would be better presented elsewhere.

[2] We have implemented our type system, analysis, and model extraction algorithms. To ease the construction of the parser, it accepts a language whose surface syntactic details differ a bit from those in our example. For example, our implemented system encloses token parameters in `*<` and `*>` instead of `<` and `>`. We use the dot graph presentation system [13] to automatically produce graphical representations of our extracted models. All of the pictures in this paper were automatically produced using our implemented system.

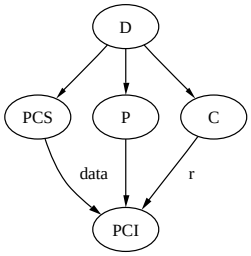

Fig. 2. Object Model

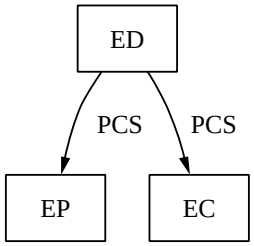

Fig. 4. Call/Return Interaction Model

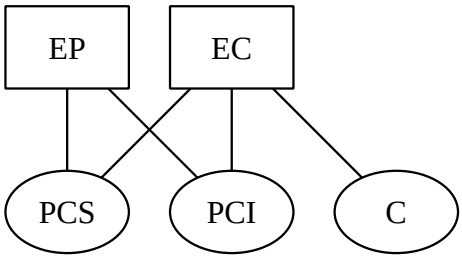

Fig. 3. Subsystem Access Model

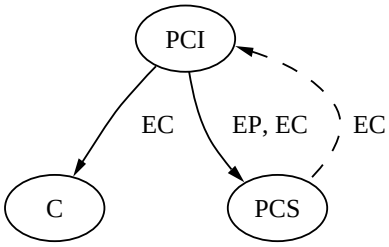

Fig. 5. Heap Interaction Model

2.6 Call/Return Interaction Models

Call/return interaction models characterize the control and data flow transfers that take place when a method in one subsystem invokes a method in a different subsystem. The model itself is a labelled, directed graph. The nodes correspond to subsystems; there is a solid edge between two subsystems if a method in the first subsystem may invoke a method in the second subsystem. There is a dashed edge if the second method may return an object to the first subsystem. The labels on the edges are the tokens that represent the objects passed as parameters or returned as values.

We use the analysis results to extract the call/return interaction model as follows. At each method call site, we retrieve the bindings that the analysis has computed for each of the token variables in the types of the parameters. These bindings identify the tokens that represent the objects passed as parameters from the caller to the callee. We also extract the subsystems for the caller and the callee.

If the callee is an entry method, the analysis generates a solid edge between the two subsystems and labels the edge with the set of tokens that represent the parameters. If the invoked method returns an object, it also generates a return edge, using the analysis results at the return statement(s) in the callee to extract the tokens on the label of the return edge. Figure 4 presents the call/return interaction model in our example.

Note that the call/return model treats a callback as just another method call in the program code. Like all other method calls, we draw the call and return edges in the call/return model for callbacks.

2.7 Heap Interaction Models

Heap interaction models capture the indirect interactions that take place via objects in the heap. The nodes in this model correspond to tokens. There is a solid edge between two tokens if a subsytem may write a reference to an object represented by the first token into an object represented by the second token; there is a dashed edge (in the opposite direction) if a subsystem may read that reference. The label on each edge is the subsystem that performed the write or the read.

We use the analysis results to compute the heap interaction model as follows. At each statement that reads or writes a reference from one object to another object, we retrieve the subsystems that may execute the statement, and, for each subsystem, the tokens that represent the two objects. There is an edge between each possible pair of tokens that represent the source and target objects. The label on each such edge is the corresponding retrieved subsystem.

Figure 5 presents the heap interaction model for our example. The solid lines indicate that the `EP` subsystem may write a `PCI` object into a `PCS` object and that the `EC` subsystem may write a `PCI` object into a `C` object. The dashed line indicates that the `EC` subsystem may read the `PCI` object back out of the `PCS` object.

Note that we have placed the `PCS` token on both the `Stack` object and the `Node` objects that implement the `Stack`'s internal state, in effect collapsing all of the objects into a single abstraction in the heap interaction model (and other models as well). This is an example of how tokens allow the developer to hide irrelevant detail in the generated models.

2.8 Discussion

As this example illustrates, extracting and using pointer analysis information is relatively straightforward given the polymorphic token declarations. This information allows us to create a broad range of models that characterize the heap structure of the program, its information access behavior, and both the direct and the indirect information flow between its subsystems.

We note that our analysis has more information about the program than it presents in the extracted models. We have chosen our specific set of models based on our expectations of what developers would find most useful. We

envision, however, a much richer interactive program exploration system that would allow developers to customize the models to include more or less detail depending on their current needs. To cite just one example, the developer could choose to display the name and method of each local variable that generated a given unlabelled edge in the object model. Such a system would give developers appropriate access to all of the information that the analysis extracts.

3 Analysis and Model Extraction

We next present the analysis and model extraction algorithms. The purpose of the analysis is to determine all of the possible token variable bindings for each method. The model extraction algorithms use the bindings to produce the models.

3.1 Preliminaries and Notation

The program defines a set of tokens $\mathsf{t} \in \mathsf{T}$, token variables $\mathsf{p}, \mathsf{v} \in \mathsf{V} \cup \mathsf{T}$, a set of methods $\mathsf{m} \in \mathsf{M}$, a set of subsystem identifiers $\mathsf{s} \in \mathsf{S}$, a set of classes $\mathsf{k} \in \mathsf{K}$, a set of call sites $\mathsf{c} \in \mathsf{C}$, and a set of object creation sites $\mathsf{o} \in \mathsf{O}$. Each class k has a set of object reference fields $\mathsf{f} \in \mathsf{fields}(\mathsf{k})$. Each call site c may invoke a set of methods $\mathsf{m} \in \mathsf{callees}(\mathsf{c})$; we compute the call graph information using a variant of class hierarchy analysis. Each call site c is contained in a method $\mathsf{method}(\mathsf{c})$ and each object creation site o is contained in a method $\mathsf{method}(\mathsf{o})$. If a method m is an entry method, then $\mathsf{entry}(\mathsf{m})$ is its subsystem identifier s, otherwise $\mathsf{entry}(\mathsf{m}) = \mathsf{same}$, where same is a special identifier indicating that each invocation of m is part of the same subsystem as its caller. The type of an object created at an object creation site o is $\mathsf{k}\ \langle \mathsf{v}_1, \ldots, \mathsf{v}_l \rangle = \mathsf{type}(\mathsf{o})$, where k is the class of the new object and $\mathsf{v}_1, \ldots, \mathsf{v}_l$ are the actual token parameters of the new object. Each local variable $\mathsf{lv} \in \mathsf{LV}$ has a type $\mathsf{k}\ \langle \mathsf{v}_1, \ldots, \mathsf{v}_l \rangle = \mathsf{type}(\mathsf{lv})$. Each class k has a set of formal token parameters $\langle \mathsf{p}_1, \ldots, \mathsf{p}_l \rangle = \mathsf{parms}(\mathsf{k})$ and a set of object references $\mathsf{f}\ \mathsf{k}\langle \mathsf{v}_1, \ldots, \mathsf{v}_l \rangle$, where $\mathsf{k}\langle \mathsf{v}_1, \ldots, \mathsf{v}_l \rangle$ is the type of the object which field f references.

The analysis produces bindings $\mathsf{b} \in \mathsf{B} = \mathsf{T} \cup \mathsf{V} \rightarrow \mathsf{T}$; we require that $\mathsf{b}(\mathsf{t}) = \mathsf{t}$ for all $\mathsf{t} \in \mathsf{T}$. The identity function on tokens is $\mathsf{Id} = \lambda \mathsf{t}.\mathsf{t}$.

3.2 Analysis

The analysis propagates binding information from caller to callee to compute a set of calling contexts for each method. More specifically, for each method m, it produces a set of tuples $\langle \mathsf{s}, \mathsf{b} \rangle \in \mathsf{contexts}(\mathsf{m})$. This set of tuples satisfies the following context soundness condition:[3]

[3] Note that constructors are treated just like any other method in this analysis.

If:

- c is a call site with $l+1$ actual parameters[4] whose types are
  $$\mathsf{k}_0\ \langle \mathsf{v}^0_1, \ldots, \mathsf{v}^0_{n_0}\rangle, \ldots, \mathsf{k}_l\ \langle \mathsf{v}^l_1, \ldots, \mathsf{v}^l_{n_l}\rangle,$$
- c is inside a method $\mathsf{m_c} = \mathsf{method}(\mathsf{c})$,
- $\langle \mathsf{s}, \mathsf{b}\rangle \in \mathsf{contexts}(\mathsf{m_c})$, $\mathsf{m} \in \mathsf{callees}(\mathsf{c})$, and
- m has $l+1$ formal parameters whose types are
  $$\mathsf{k}_0\ \langle \mathsf{p}^0_1, \ldots, \mathsf{p}^0_{n_0}\rangle, \ldots, \mathsf{k}_l\ \langle \mathsf{p}^l_1, \ldots, \mathsf{p}^l_{n_l}\rangle,$$

then $\langle s', [\mathsf{p}^j_i \mapsto \mathsf{b}(\mathsf{v}^j_i).0 \le j \le l, 1 \le i \le n_j] \cup \mathsf{ld}\rangle \in \mathsf{contexts}(\mathsf{m})$, where $s' = s$ if $\mathsf{entry}(\mathsf{m}) = \mathsf{same}$, otherwise $s' = \mathsf{entry}(\mathsf{m})$.

The analysis produces an analysis result that satisfies this condition by propagating token bindings in a top-down fashion from callers to callees starting with the `main` method. It initializes the analysis by setting $\mathsf{contexts}(\texttt{main}) = \{\langle \texttt{MAIN}, \mathsf{ld}\rangle\}$. It uses a fixed-point computation within strongly connected components of the call graph to ensure that the final result satisfies the context soundness condition. Note that this algorithm produces a completely context-sensitive solution in that it records each context separately in the analysis result. It is also possible to adjust the algorithm to merge contexts and produce a less context-sensitive result.

3.3 Object Model Extraction

Figure 6 presents the object model extraction algorithm. This algorithm produces a set of nodes $N \subseteq \mathsf{T}$ and a set of labelled edges E of the form $\langle \mathsf{t}_1, \mathsf{f}, \mathsf{t}_2\rangle$; each such edge indicates that the field f in an object represented by token t_1 may contain a reference to an object represented by token t_2. The algorithm processes all of the object creation sites o in the program; for each site, it uses the token variable bindings produced by the analysis to determine the potential token instantiations for objects created at that site. It then uses the bindings to trace out the part of the heap reachable from objects created at that site. The visit algorithm uses a set V of visited class/binding pairs to ensure that it terminates in the presence of recursive data structures.

Note that this algorithm produces only the labelled edges for the heap references. Our implemented algorithm also processes the local variable declarations to add the unlabelled edges that summarize potential referencing relationships associated with the local variables in each class.

3.4 Subsystem Access Model Extraction

Figure 7 presents the subsystem access model extraction algorithm. It produces a set of nodes $N \subseteq \mathsf{S} \cup \mathsf{T}$ and a set of edges E of the form $\langle \mathsf{s}, \mathsf{t}\rangle$; each such edge indicates that the subsystem s may access an object represented by token t. The

[4] By convention, the receiver is parameter 0.

```
set N = ∅, E = ∅, V = ∅
for all object creation sites o ∈ O
  let m = method(o)
  let k⟨v_1, ... , v_l⟩ = type(o)
  let ⟨p_1, ... , p_l⟩ = parms(k)
  for all ⟨s, b⟩ ∈ contexts(m)
    visit(k, [p_i ↦ b(v_i).1 ≤ i ≤ l] ∪ ld)
```

```
visit(k, b)
  if ⟨k, b⟩ ∉ V then
    let ⟨v_1, ... , v_l⟩ = parms(k)
    set N = N ∪ {b(v_1)}
    set V = V ∪ {⟨k, b⟩}
    for all f k'⟨v'_1, ... v'_j⟩ ∈ refs(k)
      set E = E ∪ {⟨b(v_1), f, b(v'_1)⟩}
      let ⟨p_1, ... , p_j⟩ = parms(k')
      visit(k', [p_i ↦ b(v'_i).1 ≤ i ≤ j] ∪ ld)
```

Fig. 6. Object Model Extraction Algorithm

algorithm processes all of the accesses in the program, retrieving the binding information produced by the analysis to determine 1) the subsystems that can execute the access and 2) the tokens that represent the accessed objects. Note that, as described earlier in Section 2, this model is not designed to capture accesses to primitive fields.

```
set N = ∅, E = ∅
for each method m
  for each access lv.f in m
    let k ⟨v_1, ... , v_l⟩ = type(lv)
    for each ⟨s, b⟩ ∈ contexts(m)
      set N = N ∪ {s, b(v_1)}
      set E = E ∪ {⟨s, b(v_1)⟩}
```

Fig. 7. Subsystem Access Model Extraction Algorithm

3.5 Call/Return Interaction Model Extraction

Figure 8 presents the call/return model extraction algorithm. It produces a set of nodes $N \subseteq \mathsf{S}$ and a set of edges E of the form $\langle \mathsf{s}_1, \mathsf{t}, \mathsf{s}_2 \rangle$. The algorithm processes all of the call sites in the program, retrieving the binding information produced by the analysis to determine 1) if the call site may invoke an entry method of a different subsystem, and 2) if so, the tokens that represent the objects passed as parameters between the subsystems. Note that there is an edge for each such token. To eliminate visual clutter, our model display algorithm coalesces all edges between the same two subsystems, producing a single edge with a list of the tokens passed as parameters between the subsystems.

The algorithm in Figure 8 does not generate the return edges. Our implemented algorithm generates these edges by similarly processing the return statements of entry methods.

```
set N = ∅, E = ∅
for each call site c
  for each ⟨s, b⟩ ∈ contexts(method(c))
    for each m ∈ callees(c)
      let s' = entry(m)
      if s' ≠ same and s' ≠ s then
        set N = N ∪ {s, s'}
        let k_0 ⟨v^0_1, ..., v^0_{n_0}⟩, ..., k_l ⟨v^l_1, ..., v^l_{n_l}⟩ be the types
          of the actual parameters at the call site c
        set E = E ∪ {⟨s, b(v^i_1), s'⟩.1 ≤ i ≤ l}
```

Fig. 8. Call/Return Model Extraction Algorithm

3.6 Heap Interaction Model Extraction

The heap interaction model extraction algorithm produces a set of nodes $N \subseteq \mathsf{T}$ and two sets of edges. The write edges $W \subseteq \mathsf{T} \times \mathsf{S} \times \mathsf{T}$ summarize the write interactions; an edge $\langle \mathsf{t}_1, \mathsf{s}, \mathsf{t}_2 \rangle \in W$ indicates that the subsytem s may write a reference to an object represented by token t_1 into an object represented by token t_2. The read edges $R \subseteq \mathsf{T} \times \mathsf{S} \times \mathsf{T}$ summarize the read interactions; an edge $\langle \mathsf{t}_1, \mathsf{s}, \mathsf{t}_2 \rangle \in R$ indicates that the subsytem s may read a reference to an object represented by token t_2 from an object represented by token t_1.

```
set W = ∅
for each method m
  for each write access |v_1.f = |v_2 in m
    let k^1 ⟨v^1_1, ..., v^1_{l_1}⟩ = type(|v_1)
    let k_2 ⟨v^2_1, ..., v^2_{l_2}⟩ = type(|v_2)
    for each ⟨s, b⟩ ∈ contexts(m)
      if (b(v^1_1) ≠ b(v^2_1) then
        set N = N ∪ {b(v^1_1), b(v^2_1)}
        set W = W ∪ {⟨b(v^2_1), s, b(v^1_1)⟩}
```

Fig. 9. Heap Interaction Model Extraction Algorithm

Figure 9 presents the algorithm that extracts the write interactions W. The algorithm processes all of the write accesses in the program, retrieving the binding information produced by the analysis to determine 1) the subsystems that may perform the write and 2) the tokens that represent the accessed objects. The algorithm that extracts the read interactions is similar. The set of nodes N is initialized to $\emptyset$ before the read and write interaction algorithms execute.

4 Type System

We next present a formal treatment of the type system. The type system is used to check token consistency constraints. Its primary purpose is to verify that the token declarations match at assignment and method invocation statements. These checks help ensure that our models are sound; in particular, they ensure that the type declarations in object fields correctly reflect the structure of the heap. We realize our type system as a set of typing rules for a simplified core language, whose grammar is in Figure 10. To simplify the presentation, we omit subsystems from the formal treatment.

$$
\begin{aligned}
P &::= \mathit{token}^*\ \mathit{defn}^* \\
\mathit{defn} &::= \texttt{class}\ cn\langle t^*\rangle\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{field} &::= \tau\ fd \\
\tau &::= cn\langle t^+\rangle \mid \texttt{Object}\langle t\rangle \\
\mathit{meth} &::= pn\langle t^*\rangle(\mathit{arg}^*)\ \{\mathit{local}^*\ s^*\} \\
\mathit{token} &::= tn \\
t &::= \mathsf{formal} \mid tn \\
\mathit{arg} &::= \tau\ x \\
\mathit{local} &::= \tau\ y \\
s &::= x = e \mid x.fd = y \mid x = \texttt{new}\ c\langle t^+\rangle \mid \\
&\quad\ e.pn\langle t^*\rangle(e^*) \mid \\
&\quad\ l: \mid \texttt{goto}\ l \mid \texttt{if}\ \mathit{cond}\ \texttt{then}\ l\ \texttt{else}\ l \\
e &::= y \mid y.fd \\
\mathit{cond} &::= e\texttt{==}e \mid e\ \texttt{!=}e
\end{aligned}
$$

$\mathsf{formal} \in$ formal token names
$cn \in$ class names
$fd \in$ field names
$mn \in$ method names
$tn \in$ token names
$x, y \in$ variable names
$l \in$ statement labels

Fig. 10. Grammar for core language

Figure 16 presents the static type rules that define the type checker; their meaning is explained in Figure 15. Formally, a program consists of a sequence of class definitions, containing method, field and token definitions, as well as token definitions (see Rule [PROG] in Figure 16). The goal is to derive the type judgement $\vdash P$, indicating that the program satisfies the static type constraints.

The type system checks each method in turn by using the type declarations of its class in conjunction with the method parameter definitions to construct

an initial typing environment for the method (see Rule [METH]). The type system then checks each statement of the method in turn (Rules [STMT NEW] through [STMT INVOKE]). For each statement, it attempts to derive a typing judgement of the form $P; E \vdash s$, which indicates that the statement type-checks in the context of the program P and the typing environment E. The typing environment E binds variables to types and provides the list of formal token variables. The Rule [STMT INVOKE] ensures that a method call may only occur when the necessary conditions hold.

5 Experience

We have implemented a prototype version of our system by extending the Kopi Java compiler.[5] We tested our approach on Tagger, a text formatting system written by Daniel Jackson. Tagger consists of 1721 lines of Java code and 14 classes (not counting the standard Java libraries). It accepts a text file augmented with formatting commands as input and produces as output another text file in the Quark document definition language.

We first augmented Tagger with subsystem and token annotations. This augmentation increased the number of lines of code to 1755. We added token and/or subsystem annotations to a total of 201 lines of code. There was no perceptible compile-time overhead associated with analyzing the annotated code and producing models. Our system does not incur any run-time overhead because we exclusively use static techniques.

5.1 Subsystems of Tagger

We first discuss the subsystems we added to Tagger. Our augmented version has the following subsystems, with one subsystem entry point class per subsystem:

- **ParsS:** The parser subsystem, which contains code to read the input file, group characters into words, and recognize formatting commands.
- **PMapS:** The property management subsystem, which manages the data structures that control the translation between each Tagger formatting command and the corresponding Quark output.
- **ActS:** The action subsystem, which uses the property management subsystem to translate Tagger commands into Quark commands, then passes the output to the generation subsystem.
- **GenS:** The generation subystem, which produces the output Quark document. This subsystem manages the translation of the Quark commands into a flat stream of output symbols. It is responsible for generating the surface syntax of the Quark document and producing the output file.
- **EngS:** The engine subsystem, which processes the Tagger commands and serially dispatches each command to the Act subsystem.

[5] Available at `http://www.dms.at/kopi/`

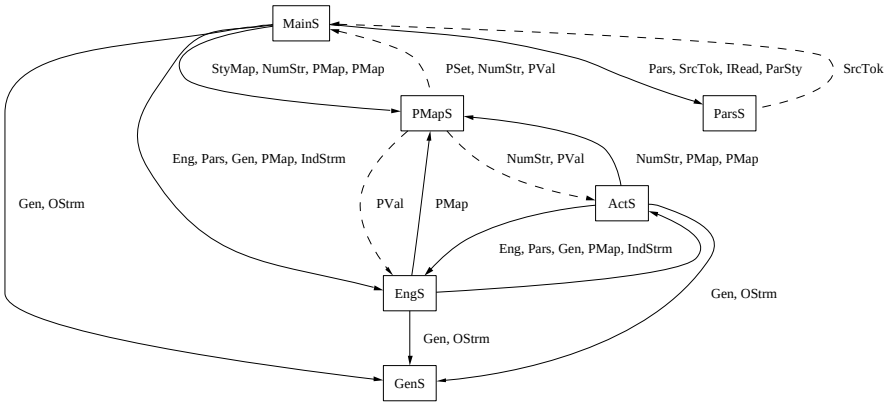

Fig. 11. Call/return model for Tagger

– **MainS:** The main subsystem, which initializes the system and implements the connection between the Pars subsystem, which reads the input file, and the Act subsystem, which processes the text and Tagger commands in the file.

Of the original 14 classes, six are subsystem entry point classes in the annotated version. Two more are abstract superclasses of subsystem entry point classes. Another two are used to transfer data between the Pars, Eng, Act, and Gen subsystems; their methods simply store and retrieve the transferred data. Another class reads in the configuration data that governs the translation from Tagger to Quark formatting commands; this class is encapsulated within the PMap subsystem. Another two store updatable processing state relating to the output document, for example the current position in an itemized list of paragraphs. These classes are encapsulated inside the Eng subsystem. The remaining class manages assertions.

In Figure 11, we present the call/return interaction model for Tagger. We can observe that the **GenS** subsystem is invoked by the **ActS**, **EngS** and **MainS** subsystems; our analysis guarantees that **GenS** is only called by these three subsystems. Note also that there is no edge between **PMapS** and **GenS**: the generator does not invoke the property management subsystem. In addition to the invocation relationships, this diagram presents the tokens that represent the objects that carry data between subsystems.

5.2 Tokens of Tagger

To present the other models, we need to discuss the token structure of Tagger. We have augmented Tagger with 16 tokens, some of which represent system classes, such as **EStrm**, the standard error output stream. The tokens are:

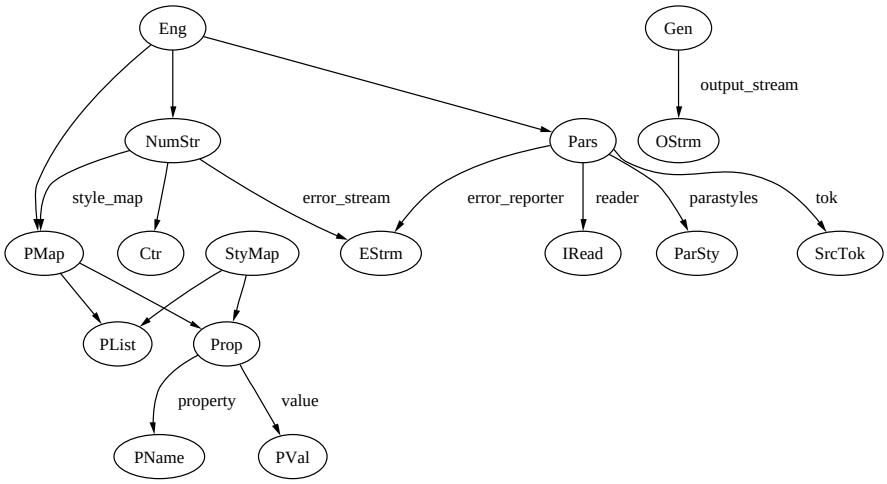

Fig. 12. Object model for Tagger

- **Gen, Eng, Pars:** These tokens represent objects that are required by the generator, engine, and parser, respectively.
- **NumStr, Ctr:** These tokens represent objects containing state for list counters used in the Tagger source.
- **SrcTok:** This token represents objects containing the input characters read by the parser, and passed to the engine.
- **PMap, PList, PName, PVal:** These tokens are owned by the property management subsystem **PMapS**, and represent objects containing text properties and maps thereof.
- **StyMap, ParSty:** These tokens represent objects containing particular property maps, used for storing character and paragraph style information.
- **IRead:** This token represents the input stream object, from which Tagger reads its input data.
- **EStrm, OStrm:** These tokens represent the system output and error output stream objects.

In Figure 12 we show the object model for Tagger. This model illustrates the reachability relations in the heap, and records what state is held by which tokens. Note that a **Gen** object refers only to an **OStrm** object, which holds the output stream.

5.3 Combining Subsystems and Tokens

Our analysis makes both subsystem and token information available. We can combine this information to give the subsystem access and heap interaction models.

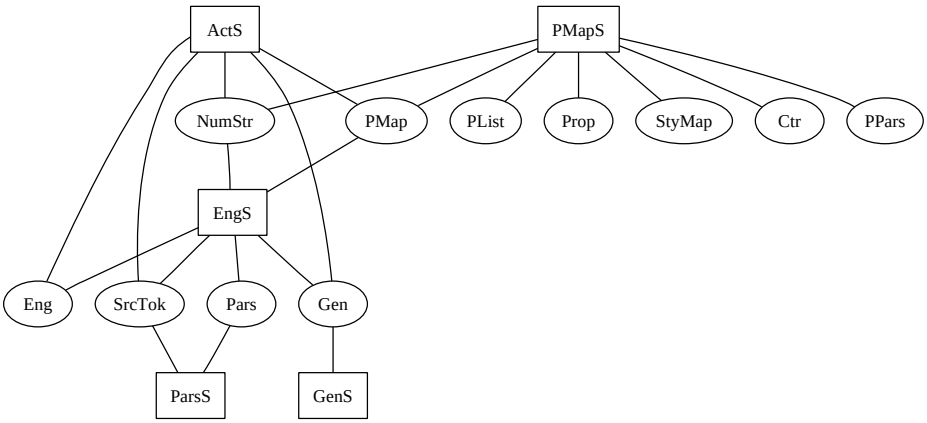

Fig. 13. Subsystem Access Model for Tagger

The subsystem access model is illustrated in Figure 13. We can observe, for instance, that the **EngS, ActS** and **ParsS** subsystems all use **SrcTok** tokens. This strongly suggests that they might use heap objects with **SrcTok** tokens to indirectly communicate between themselves: we expect that the parser subsystem must somehow share information with the action subsystem, and there are no direct interactions in the call/return model. In Figure 14 we present the heap interaction model for Tagger.

6 Related Work

We discuss related work in the areas of software model extraction, pointer analysis, and ownership types. We also briefly compare the models we extract with the models present in the Unified Modelling Language.

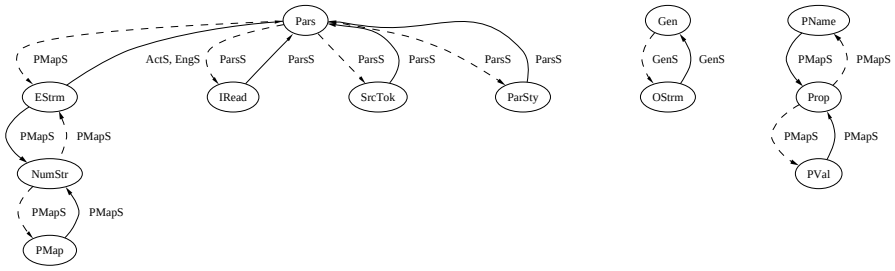

Fig. 14. Heap Interaction Model for Tagger

6.1 Modeling Extraction

Software models play a key role in most software development processes [28, 12]. Modeling is usually carried out during the design phase as a way of exploring and specifying the design. The design is then usually implemented by hand, opening up the possibility of inconsistencies between the design and the implementation. The software engineering community has long recognized the need for tools to help ensure that the software conforms to its design [17]. Automatic model extraction is a particularly appealing alternative, because it holds out the promise of delivering models that are guaranteed to correctly reflect the structure of the implementation.

There are tools currently on the market which can both automatically extract UML models from code and generate skeletal code from UML models, e.g. TogetherJ by TogetherSoft and Rational Rose. These tools use heuristics to extract (possibly unsound) UML design information from the source code. There is also a Java Specification Request underway[4] which would extend Java to permit developers to embed arbitrary metadata into their code, including for instance UML design information. While this embedding may facilitate the process of manually updating the design information to match the implementation, there is still no guaranteed connection.

Control-Flow Interactions Most previous model extraction systems have focused on control-flow interactions. The software reflexion system [21], for example, automatically extracts an abstraction of the call graph and enables the developer to compare this abstraction with a high-level module dependency diagram. Like our system, ArchJava [1] enables the extraction of software architecture information embedded directly into program code. Its approach augments Java with the software architecture concepts of components, connections, and ports; ArchJava then enforces the constraint that all inter-component control transfers must take place through ports (ensuring communication integrity). This enables the automatic extraction of communication diagrams similar to our call/return interaction models. Note in particular that ArchJava summarizes only control-flow interactions; it does not handle heap-mediated interaction between components; our system, on the other hand, was designed to capture the structure of heap-mediated interaction between subsystems.

Our use of polymorphic token types and the associated analysis enables us to capture a wider range of design issues; specifically structural issues associated with referencing relationships between objects in the heap and information flow issues associated with method invocations. Most importantly, we also capture indirect information flow between subsystems that takes place via objects in the heap. To the best of our knowledge, all previous systems do not attempt to perform the analysis that would enable them to capture these kinds of dependences. This raises the possibility that the extracted models fail to accurately capture all important interactions.

Object Model Extraction Standard approaches for extracting object models from code treat each class as a unit. In type-safe languages, it is even possible to extract a (relatively crude) object model directly from the type declarations of the fields in the objects. Problems with this approach include conflation of different instances of general-purpose classes and overly detailed object models because of a failure to abstract internal data structures. Womble [18] attacks the latter failure by treating collection classes separately as relations between objects. Womble is also unsound in that the extracted model may fail to accurately characterize the referencing relationships. In contrast, our extracted object models are sound and avoid both conflation of instances of general-purpose classes and excessive detail associated with failing to abstract internal data structures.

6.2 Comparison with UML

We next compare our extracted models to UML models. A primary difference underlying the two approaches is that UML was designed solely as a design abstraction with (at least in principle) no formally precise connection with the code that implements the design. One of the primary goals of our approach, on the other hand, is to establish such a connection and to ensure that our extracted models are sound (i.e., correctly reflect all potential implementation behaviors). We view this connection as necessary to ensure that the models remain consistent with the implementation and therefore a useful source of information about the design. We have found that realizing such a connection caused our models to differ, in some cases substantially, from standard UML models.

Class Diagrams and Object Models UML class diagrams are designed, in part, to characterize relationships between objects in the heap. The standard interpretation of a UML object model is that each box represents a class and each arrow represents a relationship between instances of the involved classes. Our object models also capture this kind of structural information. But each box corresponds to a token, not a class, and the arrows represent relationships derived from object referencing relationships in the heap.

We found that the lack of a connection between the design and code can actually help the designer to deliver a clear, effective design — the designer has great flexibility to adjust the design to present the most important and relevant aspects of the envisioned implementation. In particular, UML allows designers to elide instances of auxiliary classes such as the list nodes in a collection. It also allows designers to draw two distinct boxes that correspond to the same class in the implementation — a clear case of the designer using multiple design elements that correspond to a single code element. We designed tokens, in part, to enable the designer to adjust the granularity of the extracted design in similar ways.

UML also allows the developer to present arbitrary relationships that may be implemented in a variety of ways. All of the relationships in our models, on the other hand, are derived from referencing relationships in the implemented

data structures. While we support derived relationships that are implemented by multiple fields working together and enable the developer to hide irrelevant referencing relationships, our models do not capture relationships that do not have a concrete realization as references in the heap. As this discussion illustrates, we believe that any guaranteed connection will inevitably reduce the flexibility of expressing the design because of the constraints imposed by the enforcing the connection.

Interaction Diagrams and Call/Return Models UML interaction diagrams (sequence and collaboration diagrams) typically summarize the control-flow interactions which occur in one specific or several related executions of a program (a use case). Our call/return models also summarize these kinds of interactions, but because they are sound, they capture all potential interactions that may happen in *any* execution, not just the execution that corresponds to a given use case.

We note that this distinction reflects the different contexts in which UML and our system were developed. UML diagrams are primarily intended to be produced by designers. It is much easier and more productive for the designer to produce partial diagrams that capture important scenarios rather than tediously listing all possible interactions. An automated tool, however, has no problem enumerating all possible interactions. It is, of course, possible to eliminate any clutter by hiding interactions that may be considered irrelevant.

State Diagrams UML state diagrams capture the conceptual state transitions that objects take during their lifetimes in the computation. In our system, the state of an object can be represented by the token on the object and changed when there is a change in the object state. Currently, we support token changes only for objects which have at most one heap reference; in this case, the holder of the reference can change the token on the object. Because of problems associated with ensuring the soundness of such token changes in the presence of aliasing, we do not currently support token changes for objects that may have multiple references. Once we extend our system using roles [19] to allow controlled aliasing of objects, we will be able to generate transition diagrams for arbitrary objects as their tokens (and therefore their conceptual states) change throughout the computation. This discussion highlights another complication which arises because of our goal of establishing a sound connection between the design and the implementation.

Indirect Interaction Models We view our indirect interaction models as providing a substatial benefit missing in UML: they summarize the indirect interactions that may take place via objects allocated in the heap. We decided to support indirect interaction models both because of the importance of these kinds of interactions in many programs, and for completeness: we wished to allow developers to reason about the independence of different subsystems. Specifically,

our goal was to ensure that if none of our models indicated a potential interaction between two subsystems, one subsystem could not affect the other subsystem.[6] We believe that UML would benefit from the introduction of a model that captured indirect heap interactions.

6.3 Pointer Analysis

Pointer analysis has been an active area of research for well over 15 years. Approaches range from efficient flow- and context-insensitive approaches [3, 27, 26, 23, 15, 11, 16] to potentially more precise but less efficient flow- and context-sensitive approaches [24, 29, 14, 8, 20, 25]. These approaches vary in whether they create a result for each program point (flow-sensitive analyses) or one result for the entire program (flow-insensitive analyses). They also vary in whether they produce a result for each calling context (context-sensitive analyses) or one result that is valid for all calling contexts (context-insensitive analyses). There are also flow-insensitive but context-sensitive analyses that produce a single parameterized result for each procedure that can be specialized for each different calling context [22].

From our perspective, a primary difference between existing pointer analysis algorithms and our approach is the flexibility our approach offers in selecting object representatives. Specifically, our polymorphic type system enables the developer to separate objects allocated at the same object creation site in the generated model. We believe this separation is crucial to delivering models that accurately reflect the conceptual purposes of the different objects in the computation. Of course, obtaining this additional precision requires the developer to provide the polymorphic type declarations.

Another difference is that because the type declarations in our programs characterize the points-to relations in the reachable region of the heap, there is no need to analyze the individual store and load instructions to synthesize a points-to graph. Instead, the analysis can simply propagate tokens to substitute token variables out of the polymorphic types. The analysis needs to process the load and store instructions only to generate the heap interaction graph.

Our approach is quite flexible in the degree of context-sensitivity that it provides. It is possible to tune the analysis to produce a separate result for each combination of token variable and subsystem values, a result that separates subsystems but combines information within a single subsystem, or a single result for each method. Our implementation currently produces a separate result for each distinct instantiation of token variable and subsystem values.

6.4 Ownership Types

Ownership type systems are designed to enforce object encapsulation properties [10, 7, 6, 9, 2]. In this capacity, they can be used to ensure that objects from one instance of an abstraction are not used to inappropriately communicate with

[6] With the possible exception of timing channels.

other instances of the same abstraction [5, 2]. For example, one might use ownership types in a multithreaded web server to ensure that the sockets associated with one server thread do not escape to be used by another server thread.

Our system focuses on extracting communication patterns. Encapsulation violations in our system therefore show up as unexpected communication. We would attack the problem of verifying encapsulation properties by enabling the developer to state desired properties, then checking the appropriate extracted model to verify that the program did not violate these properties.

7 Conclusion

The software engineering community has long recognized the need for tools to help ensure that the software conforms to its design. Our implemented system, with its polymorphic type system, analysis, and automatic model extractors, takes an important step towards this goal of verified design. Our models capture important information about the program; because they are automatically generated, they are guaranteed to accurately reflect the program's structure and behavior. The sound heap aliasing information provided by our combined type system and analysis enables the extraction of both structural object referencing models and behavioral models that characterize not only direct interactions that take place at method and procedure calls, but also indirect interactions mediated by objects in the heap.

We believe our approach holds out the promise of integrating the design effectively into the entire lifecycle of the software. Today, in contrast, design models tend to become increasingly less reliable (and therefore less relevant) as development proceeds into the implementation and maintenance phases. The potential result would be a more powerful and pervasive notion of design, leading to more reliable systems and more economical development.

Acknowledgements

The authors would like to thank Derek Rayside for much useful feedback on the paper.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *24th International Conference on Software Engineering*, Orlando, FL, May 2002.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, Nov. 2002.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

4. J. Bloch et al. JSR175: A metadata facility for the Java^{TM} programming language, Apr 2002.
5. B. Bokowski and J. Vitek. Confined types. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.
6. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, Nov. 2002.
7. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Tampa Bay, Florida, Oct. 2001.
8. J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993. ACM.
9. D. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, Nov. 2002.
10. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, Oct. 1998.
11. M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
12. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: the catalysis approach*. Addison-Wesley, Reading, Mass., 1998.
13. J. Ellson, E. Ganser, E. Koutsofios, and S. North. Graphviz. Available from http://www.research.att.com/sw/tools/graphviz.
14. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.
15. M. Fahndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.
16. N. Heintze and O. Tardieu. Ultra-fast aliasing using CLA: A million lines of code in a second. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.
17. D. Jackson and M. Rinard. The future of software analysis. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM, New York, June 2000.
18. D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
19. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, Jan. 2002.
20. W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

21. G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering*, Washington, DC, Oct. 1995.
22. R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, Nov. 2000.
23. R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *1997 International Conference on Software Engineering*, Boston, MA, May 1997.
24. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.
25. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.
26. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, Jan. 1997.
27. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.
28. J. Warmer and A. Kieppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, Reading, Mass., Redwood City, CA, 1998.
29. R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.

Judgement	Meaning
$\vdash P$	program P is well-typed
$P \vdash \mathit{token}$	*token* is a well-formed token
$P \vdash \mathit{defn}$	*defn* is a well-formed class definition
$P;E \vdash \mathit{meth}$	*meth* is a well-formed method
$P;E \vdash \mathit{field}$	*field* is a well-formed field
$P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle$	class *cn* with formal parameters $f_{1..n}$ declares field *field*
$P;E \vdash \mathit{wf}$	E is a well-formed typing environment
$P;E \vdash_{\mathrm{token}} t$	t is a token defined in the program or the environment
$P;E \vdash \tau$	τ is a well-formed type
$P;E \vdash e{:}\tau$	expression e has type τ
$P;E \vdash \mathit{cond}$	condition *cond* is well-typed
$P;E \vdash s$	statement s is well-typed

Fig. 15. Meaning of Judgements in Type System

$\boxed{\vdash P}$

[PROG]

$$\frac{\begin{array}{c} ClassesOnce(P) \quad FieldsOnce(P) \\ MethodsOnce(P) \quad TokensOnce(P) \quad JumpsLocal(P) \\ P = token_{1..m}\, defn_{1..n} \quad P \vdash token_i \quad P \vdash defn_i \end{array}}{\vdash P}$$

$\boxed{P \vdash defn}$

[CLASS]

$$\frac{\begin{array}{c} g_i = \mathsf{token} f_i \quad E = g_{1..n} \\ P;E \vdash field_i \quad P;E \vdash meth_i \end{array}}{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \{field_{1..j}\quad meth_{1..m}\}}$$

$\boxed{P;E \vdash meth}$

[METH]

$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ c\langle f_{1..n}\rangle\ \{\cdots meth_{mn} \cdots\} \\ arg_i = cn_i\langle f_{i_1..i_{m_i}}\rangle vn_i \quad \mathsf{local}_j = cn_j\langle f_{j_1..j_{m_j}}\rangle ln_j \\ E = E_0, arg_{1..n}, local_{n+1..n+l} \quad \forall i \in [1..t].\ P;E \vdash s_i \\ \forall n, k.\ (\exists m.\ f_{n_k} = f_m \vee P \vdash_{\mathrm{token}} f_{n_k}) \quad P;E_0 \vdash wf \end{array}}{P;E \vdash mn\langle f_{1..r}\rangle(arg_{1..n})\ \{local_{n+1..n+l}\ s_{1..t}\}}$$

$\boxed{P;E \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P;\emptyset \vdash wf}$$

[ENV TOKEN FORMAL]

$$\frac{P;E \vdash wf \quad tn \notin Dom(E)}{P;E, \mathsf{token}\ tn \vdash wf}$$

[ENV X]

$$\frac{P;E \vdash \tau \quad x \notin Dom(E)}{P;E, \tau\ x \vdash wf}$$

$\boxed{P;E \vdash field}$

[FIELD INIT]

$$\frac{P;E \vdash \tau}{P;E \vdash \tau\ fd}$$

$\boxed{P;E \vdash field \in c}$

[FIELD DECLARED]

$$\frac{P;E \vdash \mathsf{class}\ c\langle f_{1..n}\rangle\ \{\cdots fd \cdots\}}{P;E \vdash fd \in c\langle f_{1..n}\rangle}$$

$\boxed{P \vdash token}$

[TOKEN]

$$\frac{}{P \vdash \mathsf{token}\ tn}$$

$\boxed{P;E \vdash_{\mathrm{token}} t}$

[TOKEN GB'L REF]

$$\frac{P = \cdots \mathsf{token}\ t \cdots}{P;E \vdash_{\mathrm{token}} t}$$

[TOKEN FORMAL]

$$\frac{E = E_1, \mathsf{token}\ t, E_2 \quad P;E \vdash wf}{P;E \vdash_{\mathrm{token}} t}$$

$\boxed{P;E \vdash \tau}$

[TYPE OBJECT]

$$\frac{P;E \vdash_{\mathrm{token}} t}{P;E \vdash \mathsf{Object}\langle t\rangle}$$

[TYPE C]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle \cdots \quad P \vdash_{\mathrm{token}} t_{1..n}}{P;E \vdash cn\langle t_{1..n}\rangle}$$

$\boxed{P;E \vdash cond}$

[COND EQ]

$$\frac{P;E \vdash e_1 \quad P;E \vdash e_2}{P;E \vdash e_1 \texttt{==} e_2}$$

[COND NEQ]

$$\frac{P;E \vdash e_1 \quad P;E \vdash e_2}{P;E \vdash e_1 \texttt{!=} e_2}$$

$\boxed{P;E \vdash e{:}\tau}$

[EXP VAR READ]

$$\frac{E = E_1, \tau\ y, E_2 \quad \tau = c\langle t_{1..n}\rangle}{P;E \vdash y{:}\tau}$$

[EXP FIELD READ]

$$\frac{\begin{array}{c} E = E_1, \tau_y\ y, E_2 \quad \tau_y = c_y\langle t^y_{1..m}\rangle \\ \tau_f = c_f\langle t^f_{1..n}\rangle \quad P \vdash (\tau_f\ fd) \in \tau_y \\ P;E \vdash_{\mathrm{token}} t^y_1 \quad P;E \vdash_{\mathrm{token}} t^f_1 \end{array}}{P;E \vdash y.fd{:}\tau_f[t^f_1/t^y_{f(1)}] \cdots [t^f_m/t^y_{f(m)}]}$$

$\boxed{P;E \vdash s}$

[STMT NEW]

$$\frac{E = E_1, \tau\ x, E_2 \quad \tau = c\langle t_{1..n}\rangle \quad P;E \vdash c\langle f_{1..n}\rangle}{P;E \vdash x = \mathsf{new}\ c\langle t_{1..n}\rangle}$$

[STMT READ/COPY]

$$\frac{E = E_1, \tau\ x, E_2 \quad P;E \vdash e{:}\tau}{P;E \vdash x = e}$$

[STMT LABEL]

$$\frac{P;E \vdash wf}{P;E \vdash \ell{:}}$$

[STMT GOTO]

$$\frac{P;E \vdash wf}{P;E \vdash \mathsf{goto}\ \ell}$$

[STMT IF]

$$\frac{P;E \vdash \ell_1{:} \quad P;E \vdash \ell_2{:} \quad P;E \vdash cond}{P;E \vdash \mathsf{if}\, cond\ \mathsf{then}\ \ell_1 \mathsf{else}\ \ell_2}$$

[STMT WRITE]

$$\frac{\begin{array}{c} E = E_1, \tau_x\ x, E_2 \quad E = E'_1, \tau_y\ y, E'_2 \\ \tau_x = c_x\langle t^x_{1..n}\rangle \quad \tau_y = c_y\langle t^y_{1..m}\rangle \\ P;E \vdash_{\mathrm{token}} t^x_1 \quad P;E \vdash_{\mathrm{token}} t^y_1 \\ P;E \vdash (\tau_y\ fd) \in \tau_x \end{array}}{P;E \vdash x.fd = y}$$

[STMT INVOKE]

$$\frac{\begin{array}{c} P;E \vdash mn\langle f_{1..r}\rangle(\tau_j\ y_j)_{j\in 1..n}\{\cdots\} \\ P;E \vdash_{\mathrm{token}} a_i \quad \tau_j = cn_j\langle f_{j_1..j_{m_j}}\rangle \quad \tau'_j = \tau_j[a_i/f_i] \quad a_{j_1} = f_{j_1}[a_i/f_i] \quad P;E \vdash_{\mathrm{token}} a_{j_1} \\ P;E \vdash e'_j{:}\tau'_j \quad P;E \vdash a_0{:}\tau_0 \quad \tau_0 = cn\langle f_{1..m_0}\rangle \quad meth_{mn} \in cn \end{array}}{P;E \vdash a_0.mn\langle a_{1..r}\rangle(e'_{1..n})}$$

Fig. 16. Type Rules