A Language for Role Specifications

Viktor Kuncak, Patrick Lam, and Martin Rinard

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

Abstract. This paper presents a new language for identifying the changing roles that objects play over the course of the computation. Each object's points-to relationships with other objects determine the role that it currently plays. Roles therefore reflect the object's membership in specific data structures, with the object's role changing as it moves between data structures. We provide a programming model which allows the developer to specify the roles of objects at different points in the computation. The model also allows the developer to specify the effect of each operation at the granularity of role changes that occur in identified regions of the heap.

1 Introduction

In standard type systems for object-oriented languages, each object is created as an instance of a specific class, with the object's type determined by that class. Because the object's class does not change, the object has the same type for its entire existence in the computation. This property limits the ability of the type system to capture dynamically changing object properties. Specifically, a given object may play many different roles during its lifetime in the computation, with the distinctions between these roles crucial to the computation's safety and correctness. The inability of the type system to model these changing roles prevents it from capturing these important distinctions.

This paper presents a new kind of type system, called a *role system*, which enables a developer to express the different roles that each object plays during its lifetime in the computation. The role of each object is determined by its points-to relationships with other objects. As these relationships change, the object's type changes to reflect its changing role in the computation. Our system can therefore capture important distinctions between objects of the same class as they play different roles in the computation.

Because roles are determined by the linking relationships, role changes often correspond to movements between data structures. Our role system is therefore

[*] This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and an NSERC graduate scholarship.

designed to capture the linking relationships at a level of precision that makes it possible to track the removals and insertions that implement movements between data structures. We realize this goal by providing three mechanisms:

1. **Role Definitions:** The role definitions specify the referencing relationships for each role. For all references to an object o playing a given role, the role definition specifies the field where the reference to o is stored and the role of the object containing this reference. On the other hand, for each reference originating at the object playing the role, the role definitions specify the roles of the objects to which it refers. The role definitions therefore provide complete heap aliasing information for each object at the granularity of roles.
2. **Role Declarations:** The programmer can declare the role of the object to which each local variable or parameter refers. In effect, these role declarations express additional application-specific safety properties not captured by standard type systems.
3. **Operation Effects:** The programmer can declare how operations change the roles of the objects that they access, providing useful information about the effect of each operation at the granularity of roles.

2 Examples

We next present several examples that illustrate the role specification language. The first example illustrates how roles capture distinctions that arise from the semantics of the underlying application domain. The second example illustrates how roles capture shape invariants of linked data structures at sufficient precision to capture removals (and corresponding insertions) from the data structure.

2.1 Aircraft Example

Our first example illustrates how roles can capture the distinction between aircraft that are parked at a gate, aircraft that are taxiing on the ground, and flying aircraft. Each parked or taxiing aircraft is associated with an airport, with the ground controllers at the airport responsible for its movements. Flying aircraft are not associated with a specific airport; instead, the controllers at a control center are responsible for its flight path.

Aircraft are represented in the system by instances of the `Aircraft` class from Figure 1. Each `Aircraft` object has two instance variables: `cc` is its control center when it is flying, and `ap` is its airport when it is parked or taxiing. Figure 1 also presents the definitions of the roles that `Aircraft` objects can play. The `Parked` and `Taxiing` role definitions specify that the `ap` field of each `Parked` and `Taxiing` aircraft refers to a specific, non-null `Airport` object where the aircraft is located. The `cc` field is null for objects playing these roles, as an airport is controlling these aircraft.

Conceptually, each role has a set of slots filled by incoming references from other objects; the role definitions specify the number of slots and the roles and fields of the references that may fill each slot. In our example, each `Parked` aircraft has an incoming slot filled by a reference from the `Gate` object where the aircraft is parked; this reference is the only heap reference to a `Parked` aircraft. `Taxiing` aircraft have a slot filled by a reference from the runway that the aircraft is on; this reference is the only heap reference to a `Taxiing` aircraft. The `cc` field of `Flying` aircraft refers to a non-null `ControlCenter` object that represents the control center responsible for the aircraft's flight plan; the `ap` field is null. `Flying` aircraft have a single slot, filled by a reference from the controlling center's list of aircraft.

In addition to the textual representation, Figure 1 presents a graphical representation of the roles and their referencing relationships. Each box in the picture represents either a role or a class. Arrows with closed heads represent references between objects, while arrows with open heads represent the partition of a class into the roles that objects from that class can play.

```
class Aircraft {
    ControlCenter cc;
    Airport ap;
}
role Parked of Aircraft {
    fields ap: Airport, cc: null;
    slots Gate.p;
}
role Taxiing of Aircraft {
    fields ap: Airport, cc: null;
    slots Runway.p;
}
role Flying of Aircraft {
    fields cc: ControlCenter, ap: null;
    slots ControlCenterListNode.aircraft;
}
```

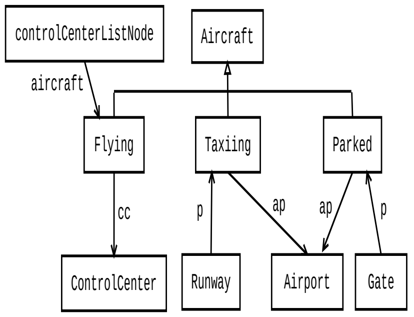

Fig. 1. Aircraft Example Role Definitions

The developer can use roles to improve the precision of operation interfaces. Figure 2 presents a sample operation on an aircraft. The `land` operation executes when an aircraft lands at an airport. The parameter declarations state that landing aircraft must be playing the `Flying` role. The effects declarations specify that control of the landing aircraft passes from the control center to the airport, with the aircraft's role changing from `Flying` to `Taxiing`. Other operations (`takeoff`, `pushback`, etc.) place similar requirements on the roles that their parameters play and have similar effects on these roles. From these operations, it is possible to automatically extract the role transition diagram for `Aircraft` objects, which is presented in Figure 3.

```
void land(Flying p, Runway r, Airport a)
effects { p.ap = a; p.cc = null; r.p = p;
          roleChange(p : Taxiing); }
{
    p.ap = a; p.cc = null; r.p = p;
    roleChange(p : Taxiing);
}
```

Fig. 2. The `land` Operation

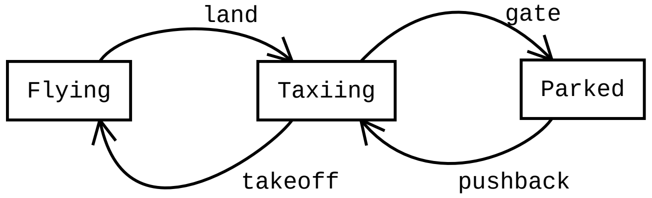

Fig. 3. Role Transition Diagram for `Aircraft` Objects

2.2 Doubly Linked List Example

This example illustrates the use of roles with a simple doubly-linked list data structure. The data structure has a dummy header node followed by some inner nodes which refer to the elements stored in the list. Figure 4 presents the role definitions for the list data structures. The nodes in the linked list are all objects of the `doubleNode` class, which has three roles. The dummy header plays the `doubleHeader` role, which requires the header to have a null `content` field. Inner nodes play the `doubleInner` role, which require the inner nodes to have a non-null `content` field. An object playing the `doubleNode` role can play either the `doubleHeader` or `doubleInner` role. Finally, the `soloNode` is a node that has been deleted from the list.

The role definitions for the `doubleHeader` and `doubleInner` roles require the `next` and `prev` fields of objects playing these roles to point to non-null objects playing either the `doubleHeader` or `doubleInner` role, and that `prev.next` and `next.prev` paths terminate at the object where they started, i.e., the two links are inverses. The `next` and `prev` fields of objects playing the `soloNode` role are null, and there are no heap references to `soloNode` roles.

We next discuss the removal of a node from a `doubleNode` list. The procedure navigates the list until it reaches the node to remove or returns back to the header node. If it finds the node to remove, it removes it, changing the node's role from `doubleInner` to `soloNode`. The effects statement of the `remove` operation states that the operation may set the `content` field of one of the nodes in the list to `null` and change the role of a list node to `soloNode`. The operation may also read and write some of the fields of the nodes in the list.

```
class doubleNode {
    doubleNode next, prev;
    Element content;
}
role doubleHeader of doubleNode {
    fields next: doubleHeader | doubleInner,
           prev: doubleHeader | doubleInner, content: null;
    slots (doubleHeader | doubleInner).prev,
          (doubleHeader | doubleInner).next;
    identities prev.next, next.prev;
}
role doubleInner of doubleNode {
    fields next: doubleHeader | doubleInner,
           prev: doubleHeader | doubleInner, content: stored;
    slots (doubleHeader | doubleInner).prev,
          (doubleHeader | doubleInner).next;
    identities prev.next, next.prev;
}
role soloNode of doubleNode {
    fields next: null, prev: null;
    slots ;
}
role stored of Element {
    slots doubleInter.content;
}
role soloElem of Element {
    slots ;
}
```

Fig. 4. Roles for the Circular Linked List

```
void remove(doubleHeader d, stored c)
effects {
   read(d.next);
   (([x, y : d.next*] x.* = y;
     [x : d.next*] x.content = null; changeRole(x : soloNode);
     [x : d.next*] read(x.*);
     changeRole(c : soloElem)
    ) | skip) }
{
    doubleNode n = d.next;
    do {
        if (n.content == c) {
            assert(n : doubleInner);
            doubleNode nn = n.next, np = n.prev;
            nn.prev = n.prev; np.next = n.next;
            n.next = null; n.prev = null;
            n.content = null;
            changeRole(n : soloNode);
            changeRole(c : soloElem);
            return;
        }
        n = n.next;
    } while (n != d);
}
```

Fig. 5. Code for Removing a Node from a Doubly Linked List

3 The Role Definition Language

We next present the full role definition language.

3.1 Basic Constraints

The heart of the role definition language is a set of basic constraints that the programmer can use to identify the relationships that define a role. There are several kinds of constraints:

Field constraints: Each field constraint is of the form

$$\mathsf{field} : \mathsf{role}_1 | \cdots | \mathsf{role}_k$$

where field is the name of a field in the object and role_1 through role_k are the names of roles. If this constraint appears in the definition of a given role, all objects playing the role have a field named field that refers to an object playing one of the roles role_1 through role_k.

Slot constraints: Each role has a number of *slots*, or incoming references. There is a slot constraint associated with each slot that defines the kinds of references that can fill the slot. Each slot constraint has the form

$$\mathsf{role}_1.\mathsf{field}_1 \mid \cdots \mid \mathsf{role}_k.\mathsf{field}_k$$

where role_1 through role_k are role names and field_1 through field_k are field names. If a given object is playing a role whose definition contains this constraint, there must exist an i such that the field field_i in some object playing role role_i contains a reference to the given object.

Identity constraints: Each identity constraint is of the form $\mathsf{field}_1.\mathsf{field}_2$, where field_1 and field_2 are two field names. If this constraint appears in the definition of a given role and an object o is playing the role, $o.\mathsf{field}_1.\mathsf{field}_2$ refers back to o. The standard example is a doubly linked list node `l`, where `l.next.prev` = `l`.

Property constraints: Each property constraint consists of a predicate over the primitive fields (integers, booleans, doubles, etc.) of the object. When an object satisfies a role which contains some property constraint p, then p must evaluate to true on this object. In this way, properties allow the specification of user-defined abstractions of object state.

Acyclicity constraints: Each acyclicity constraint is a specification of the form regExp, where regExp is a regular expression over the field names. Given an object playing the role, this constraint states that there are no cycles in the subgraph obtained by following paths that 1) start from the given object and 2) conform to the regular expression.

Figure 6 summarizes the syntax for basic role definitions.

```
role r{
     fields       $f_1$ :$r_{11}$ | $r_{12}$ | ⋯ | $r_{1p_1}$,
                  ...
                  $f_n$ :$r_{n1}$ | $r_{n2}$ | ⋯ | $r_{np_n}$;

     slots        $r'_{11}.f'_{11}$ | ⋯ | $r'_{1q_1}.f'_{1q_1}$,
                  ...
                  $r'_{m1}.f'_{m1}$ | ⋯ | $r'_{mq_m}.f'_{mq_m}$

     identities $f_1.g_1, \ldots, f_k.g_k$;
     properties $p_1, \ldots, p_l$;
     acyclic      $\mathrm{regExp}_1, \ldots, \mathrm{regExp}_t$
}
```

Fig. 6. General Form of Basic Role Specification

3.2 Multislots

An object of basic role with k slots requires exactly k references from other objects. In some cases an object may be referred to by a statically undetermined number of other objects. This possibility can be specified using *multislots.*

$$\texttt{multislots}\ \mathsf{role}_1.\mathsf{field}_1, \ldots, \mathsf{role}_k.\mathsf{field}_k;$$

A multislot allows arbitrary number of references of types $\mathsf{role}_1.\mathsf{field}_1$ through $\mathsf{role}_k.\mathsf{field}_k$. All references must be distinct and they must also to be distinct from all references mentioned in the `slots` declaration.

3.3 Compound Roles

As described so far, each object plays a single role at any given time, with its role changing over time as it moves between data structures and its relationships with other objects change. It is also sometimes useful for an object to play multiple roles at the same time. For example, an object may participate in both a linked list and a tree, playing the linked list and tree roles at the same time. We support this concept by allowing the programmer to define compound roles, which combine multiple roles into a single new role. Syntactically, the programmer declares a compound role as follows.

$$\texttt{role}\ \mathsf{r} = \mathsf{r}_1 + \ldots + \mathsf{r}_n\ ;$$

The fields and slots of the role r are the disjoint union of the fields and slots of roles r_1 through r_n. A object of role r satisfies all identity, property, and acyclicity constraints of roles r_1 through r_n.

3.4 Parameterized Roles

It is useful to parameterize roles with respect to other roles or with respect to individual references.

Role Parameters allow the definition of a role to be parametrized by names of other roles. This is a form of parametric polymorphism for role definitions. For example, a list can be parametrized by the role of its elements. Role parameters are introduced by `< >` brackets in the role definition. Once introduced, role parameters can be used inside the role definition in all places where a fixed role name is expected. In order to be used as an ordinary role, a parametrized role needs to be supplied with actual role arguments, written in `< >` brackets.

```
role List<T> {
  fields  first : ListNode<T>;
}
role ListNode<T> {
  fields next : ListNode<T> | null,
         elem : T;
  slots  ListNode<T>.next | List<T>.first;
}
role Airport {
  fields landed : List<Aircraft>;
}
```

Fig. 7. Parametrization by Roles

Reference Parameters allow role definitions to be parametrized by individual references from some object, where the identity of the object may not be known until run-time. This allows very fine-grained role definitions, suitable even for descriptions of nested data structures. Reference parameters are introduced into role definitions using `[ ]` brackets after the role name. Reference parameters can be used in slots or to instantiate other reference-parametrized roles. Every reference-parametrized role must be instantiated with an appropriate number of reference arguments supplied in `[ ]` brackets. Arguments can be field names or other role parameters.

The example in figure 8 illustrates the use of reference parameters. The `GraphList` role represents a list of disjoint graphs: there are no edges between nodes of graphs reachable from different nodes of the `GraphListNode` role. Nodes of the list are represented by objects with `GraphListNode` role. Each graph is made up of `GraphNode`s. The disjointness of the graphs is ensured by parametrizing the `GraphNode` role.

Reference parameters for roles, unlike role parameters, cannot in general be eliminated by source-to-source transformation at the role definition time. Note,

```
role GraphList {
  fields first : GraphListNode;
}
role GraphListNode {
  fields next  : GraphListNode | null,
         graph : GraphNode[graph];
  slots  GraphListNode.next | GraphList.first;
}
role GraphNode[f] {
  fields succ : List<GraphNode[f]>
  slots f | ListNode<GraphNode[f]>.elem;
}
```

Fig. 8. Parametrization by References: List of Disjoint Graphs

however, that parametrization by individual references does not prevent the static analysis of data structures. If, for references $o_1.f_1$ and $o_2.f_2$, either $f_1 \neq f_2$ or the alias analysis implies $o_1 \neq o_2$, then two data structures parametrized by roles $o_1.f_1$ and $o_2.f_2$ are known to be disjoint.

3.5 Roles and Classes

Our role system can be realized as a refinement of a static class system where each role is a refinement of some class, with one class being refined into multiple roles. To indicate that a role rl refines a class cl, we write the name of c after the definition of role r.

role rl of cl {. . . }

We note that it is also possible to use roles as a stand-alone type system.

4 The Role Programming Model

Due to the fine-grained nature of load statements `x.f=y` and store statements `x=y.f`, role constraints tend to be temporarily violated at certain program points. In this section we provide a programming model that gives the minimal requirements for a program to be role correct. A static program analysis can enforce a stronger role checking policy; it may not accept a weaker role checking policy. We assume a type safe programming language with a clean memory model, such as Java [5]. In the absence of acyclicity constraints, at any given program point, the set of all objects on the heap can be partitioned into:

1. **onstage objects** which are referenced by at least one local variable of the currently executing procedure;
2. **offstage objects** which are not referenced by any of the local variables of the currently executing procedure.

Only onstage objects can have their role constraints temporarily violated. More precisely, we have the following invariant.

```
     roleDef ::= "role" roleName
                 ("<" roleParams ">")? ("[" refParams "]")?   "of" ClassName "{"
                 ("fields" fieldDecls ";")?
                 ("slots" slotDecls ";")?
                 ("multislots" multislotDecl ";")?
                 ("identities" identDecls ";")?
                 ("acyclic" acyclicDecls ";")?   "}"
               | "role" roleName "=" disjRole ";"
               | "role" roleName "=" roleSum ";"
  fieldDecls ::= fieldDecl | fieldDecls "," fieldDecl
   fieldDecl ::= field ":" disjRole
    disjRole ::= role | disjRole "|" role
   slotDecls ::= slotDecl | slotDecls "," slotDecl
    slotDecl ::= reference | slotDecl "|" reference
multislotDecl ::= reference | multislotDecl "," reference
  identDecls ::= identDecl | identDecls "," identDecl
   identDecl ::= field "." field
acyclicDecls ::= acyclicDecl | acyclicDecls "," acyclicDecl
 acyclicDecl ::= regExp
   reference ::= role "." field
     roleSum ::= role | roleSum "+" role
        role ::= roleName("<" roleArg ">")? ("[" refArg "]")?
     roleArg ::= role | roleArg "," role
      refArg ::= field | refParam
  roleParams ::= ID | roleParams "," ID
   refParams ::= ID | refParams "," ID
```

Fig. 9. Syntax of Role Specifications

Local Role Consistency Invariant: At *every program point*, there exists an assignment of roles to all objects of the heap such that the constraints for all *offstage* objects are satisfied.

Next, we introduce the notion of program checkpoints. The checkpoints include at least procedure entry, procedure exit, and procedure call points. They may also include additional program points specified by the programmer using the `roleCheck()` command.

Global Role Consistency Invariant: At *every program checkpoint*, there exists an assignment of roles to all objects of the heap such that the constraints for *all* objects are satisfied.

Note that it is not neccessary to have a checkpoint in every loop of the control-flow graph. This allows the verification of nonlocal changes to the heap without complex global loop invariants.

Role changes to onstage objects are specified using the `changeRole` construct. The statement `changeRole(x : r)` changes the current role of the onstage object o, referenced by local variable `x`, to role `r`. Local role consistency implies that the constraints of all offstage objects adjacent to o must be consistent with the new role of o. (The consistency of the object o with its own role is not checked until o goes offstage, or until a checkpoint is reached.)

In some cases it is useful to change roles of multiple offstage objects, without bringing them onstage first. The `changeRoles` statement is used for this purpose. It specifies a regular expression denoting a region of the heap, and a set of role transitions to be performed on this region. Every role transition specifies an initial role and a final role. As in Section 5, a regular expression denotes all objects reachable from the given variable without passing though other variables in the current scope.

```
statement ::= ...
            | "changeRole" "(" var ":" role ")"                       onstage role change
            | "changeRoles" "(" regExp "," "{"roleTrans"}" ")"   offstage role changes
            | "roleCheck()"                                              global role check
roleTrans ::= roleTrans "," roleTran                                role transition list
 roleTran ::= role1 "->" role2                                          role transition
```

Fig. 10. Role Changing Statements

Acyclicity Constraints: Acyclicity constraints introduce the need to take into account the reachability of one onstage object from another to ensure that the program does not introduce a cycle into a region of the heap that the roles require to be free of cycles.

5 The Invariant Specification Language

Invariants allow the programmer to specify properties that hold at a given program point. The `assert` statement is used to enforce invariants. An invariant is a propositional formula over atomic properties. Atomic properties allow stating 1) roles of objects at a given program point; 2) aliasing between two given references; and 3) an upper bound on the set of paths between two objects in the current heap.

statement ::=	...	
|	"`assert`" "(" prop ")"	assertion
prop ::=	atomic	atomic proposition
|	"!" prop	negation
|	prop_1 "||" prop_2	disjunction
|	prop_1 "&&" prop_2	conjunction
atomic ::=	obj_1 ":" role	role assertion
|	obj_1 "==" obj_2	aliasing
|	obj_1 "*>" obj_2 ":" regExp	reachability
obj ::=	var | obj "." field	object reference
regExp ::=	"`empty`" | "`none`"	empty path, empty set
|	field	single field
|	regExp_1 "." regExp_2	concatenation
|	regExp_1 "|" regExp_2	union
|	regExp "*"	Kleene star
|	regExp "&"	Infinite paths

Fig. 11. Invariant Specifications

Roles at a given program point are asserted by the proposition obj : role. This allows the developer to verify the statically computable role of an object at a given program point.

To specify that objects obj_1 and obj_2 must be aliased, the assertion $\text{obj}_1 == \text{obj}_2$ is used. Objects are referred to using a sequence of fields starting from program variables. To specify that obj_1 and obj_2 must not be aliased, the assertion $!(\text{obj}_1 == \text{obj}_2)$ is used. Omitting the aliasing relation between these two objects allows for both cases. Note that both must and may aliasing information can be specified.

The expression obj_1 *> obj_2 denotes the set of all finite and infinite directed paths in the heap that lead from obj_1 to obj_2 without going through any onstage objects. (Paths that may pass through onstage objects can be split into segments that pass only through offstage objects.) Concrete sets of paths are specified using regular expressions, extended with the & operator, which allows the specification of infinite paths. This allows the precise specification of cyclicity. The symbol ":" denotes path subset, so obj_1 *> obj_2 : regExp gives may-reachability information in the form of an upper bound on the set of paths between two objects. For example, in the context of variables x and y, the constraint `x *> y : none` implies that any sequence of operations that has access only to x cannot perform structural modifications on the object pointed to by y. Using negation, it is possible to express must reachability information, which is a poweful tool when combined with aliasing constraints in role definitions, allowing the analysis of nonlocal operations on tree-like data structures.

6 Procedure Specifications

In this section we present the sublanguage for specifying procedure effects. It is designed to convey detailed yet concise procedure summaries which can be used as a basis for a compositional flow-sensitive interprocedural analysis with strong updates. The use of procedure summaries allows the use of precise analysis techniques inside procedures while retaining an overall linear analysis complexity in the total size of the program. Among our design goals for procedure specifications were:

1. the ability to approximate incremental changes to the regions of the heap;
2. easy instantiation of procedure specification in the context of the caller;
3. the ability to precisely specify the effects of simple procedures that perform local transformations on the heap;
4. the ability to specify aliasing contexts among procedure parameters as well as regions of heap reachable from parameters.

The first goal led us to a language whose primitives are local effects similar to loads and stores. These effects can be combined using nondeterministic choice, and their location specified using regular expressions. The second goal implied the decision to interpret all regular expressions in a procedure specification with respect to the state of the heap at procedure invocation time. The effects of simple procedures can be easily specified as a combination of elementary effects. The procedure contexts can be specified in a flexible way using the conditional effect construction. The syntax of procedure effects is given in figure 12.

Formally, an effect with no free variables is a binary relation between the initial heap and the final heap. We define the following hierarchy of effects: 1) primitive effects 2) simple effects 3) effects. Each class includes the previous ones.

Primitive effects are the building blocks of all effects. A *write* effect corresponds to a store statement or modification of a primitive field. A *read* effect specifies a load statement or read of a primitive field of a given object, without specifying which local variable receives the value. The *role change* effects specify a change of roles for one or more objects and correspond to the `changeRole` and `changeRoles` statements in the procedure. The `skip` statement denotes an empty effect; it does nothing. The expression `fail` denotes the effect which always fails. It is allowed to call the procedure only from contexts for which the procedure effect does not fail.

effect	::=	simpleEffect	simple effect
	|	effect$_1$ "|" effect$_2$	nondeterministic choice
	|	effect$_1$ ";" effect$_2$	sequence
	|	prop "->" effect	conditional effect
	|	"[" bindings "]" simpleEffect	variable bindings
	|	simpleEffect "*"	iteration
simpleEffect	::=	primitive	primitive effect
	|	simpleEffect$_1$ "|" simpleEffect$_2$	nondeterministic choice
	|	simpleEffect$_1$ ";" simpleEffect$_2$	sequence
	|	prop "->" simpleEffect	conditional simple effect
primitive	::=	obj "." fieldSpec "=" valSpec	write
	|	"read" "(" obj "." fieldSpec ")"	read
	|	"changeRole" "(" var ":" role ")"	onstage role change
	|	"changeRoles" "(" regExp "," {roleTrans} ")"	offstage role changes
	|	"skip"	empty effect
	|	"fail"	failure
bindings	::=	binding | bindings "," binding	binding sequence
binding	::=	var ":" "regExp"	existing object binding
	|	var ":" "new" role	new object binding
valSpec	::=	obj | "null" | "any"	value specification
fieldSpec	::=	field | "any"	field specification
obj	::=	var | paramRef	object
paramRef	::=	param | global | paramRef "." field	object at fixed path

Fig. 12. Procedure Effects

Objects can be referred to via a fixed sequence of field names starting from parameters (and global variables), or via a variable bound to a region of the heap or a new object identifier. In both write and read effects it is possible to abstract away from the value written or the field name by using the `any` keyword.

Simple effects are built out of primitive effects using nondeterministic choice, sequence, and conditional. The *nondeterministic choice* operator "|" specifies the union of the effect relations. In the expression $\mathsf{effect}_1 \mid \mathsf{effect}_2$, both effect_1 and effect_2 can occur; the called precedure is free to choose either one of them. The *sequence* of the effects $\mathsf{effect}_1; \mathsf{effect}_2$ denotes execution of effect_1 followed by the execution of effect_2. This corresponds to the composition of the effect relations. The *conditional* effect $\mathsf{prop} \rightarrow\ \mathsf{effect}$ is the restriction of effect to the states which satisfy the proposition prop. The effect relation acts as the identity on all states for which prop predicate is not satisfied. The syntax for propositions is the same as in Section 5.

Effects are built out of simple effects using variable binding and iteration in addition to nondeterministic choice, sequence, and conditional. A *variable binding* specifies a list of bindings for the free variables of an effect expression. A variable can be bound either to a nondeterministically chosen object in the region of the initial heap specified by a given regular expression (notation $\mathsf{var} : \mathsf{regExp}$), or to a newly allocated object of a given role (notation $\mathsf{var} : \texttt{new}\ \mathsf{role}$). The first form summarizes structural changes of a given region of the heap. The second form allows naming of objects created inside the procedure. This is important since new objects are often incorporated into existing data structures, so that effects that involve them determine the reachability properties of the heap after the procedure execution. The *iteration* operator `*` denotes repetition of the effect an unspecified number of times. It can be used to summarize the effect of loops in the procedure.

7 Parallelization with Roles

It is possible to use the role definitions and operation effect statements as a basis for the automatic parallelization of programs that manipulate linked data structures. Because the role definitions characterize the aliasing relationships in which objects engage, the compiler can use the role definitions to discover computations that access disjoint regions of recursive data structures. The role definitions of the objects in a tree data structure, for example, enable the compiler to determine that different subtrees rooted at the same node are disjoint. The combination of this information with the operation effects information enables the compiler to, for example, parallelize standard recursively-defined computations that update tree nodes but do not change the structure of the tree. Similar transformations are possible for other computations that access linked data structures.

8 Related Work

The concept of role models as a generalization of the static class system has been present in the object modelling community for some time [13], but usually with no formal relationship with program code. The idea of static analysis of types which change at run-time was explored in [17], but without any treatment of relationships between objects in the heap. A system for object reclassification is presented in [3], but the class changes are designed to be transparent to aliasing; in our approach, the roles change when the aliases change, which is a requirement for reasoning about the role changes that take place when objects move between data structures. Our system also associates a set of invariants with the current role of every object, allowing stronger structural properties to be expressed. Another difference is that implementation of the language in [3] is based on performing additional run-time checks whereas our language is primarily an interface to a static program analysis system.

The sublanguage we use for specifying context-specific invariants is similar to the logic described in [1] which also explains the relationship with [8]. A more general system used for dependence testing is described in [9].

There appears to be surprisingly little work on languages for describing precise effects procedures with respect to the heap. The importance of procedure specifications for pointer analysis was indicated in [14]. A language for annotating software libraries is described in [6]. Effects systems in general were used in functional languages with side effects [10]. Our specification language bears some similarities to propositional dynamic logic [7]. Similarly to [4], our effect language specifies operations on heap. Unlike graph rewrite rules, our effect specifications are based on primitive effects which correspond to statements in imperative programs. The effects of complex procedures also tend to be more nondeterministic than graph rewrite rules due to their approximate nature.

Although the focus in this paper is on the specification language, the analyzability of the language was our major concern. The techniques useful for role analysis are discussed in [15], [12], [4]. More restrictive approaches rely on the extensions of linear type systems or on ownership types [16], [11], [2].

9 Conclusion

We have proposed a language for specifying invariants of objects which move between dynamically changing data structures. We have given the syntax and semantics of the language and illustrated its use on several examples.

The role definition sublanguage enables the classification of objects according to their membership in different data structures as well as the specification of some essential data structure heap invariants. The invariant specification sublanguage allows the communication of additional context-specific reachability and aliasing properties. Finally, the procedure effect sublanguage is designed to capture precise effects of short procedures and to summarize complex modifications performed in regions of the heap reachable from procedure parameters.

We have constructed this language to serve as a foundation for a compositional flow-sensitive interprocedural program analysis. Such an analysis can increase a programmer's confidence in program correctness. Moreover, it can enable a variety of program transformations.

References

1. Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for linked data structures. In *Proc. 8th European Symposium on Programming*, 1999.
2. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.
3. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th European Conference on Object-Oriented Programming*, LNCS 2072, pages 130–149. Springer, 2001.
4. Pascal Fradet and Daniel Le Metayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
5. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2001.
6. Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, 1999.
7. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Mass., 2000.
8. Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3), September 1993.
9. Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proc. 8th International Parallel Processing Symposium*, Cancun, Mexico, April 26–29 1994.
10. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.
11. Naoki Kobayashi. Quasi-linear types. In *Proc. 26th ACM POPL*, 1999.
12. Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.
13. Trygve Reenskaug. *Working With Objects*. Prentice Hall, 1996.
14. Radu Rugina and Martin Rinard. Design-driven compilation. In *Proc. 10th International Conference on Compiler Construction*, 2001.
15. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. 23rd ACM POPL*, 1996.
16. F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th European Symposium on Programming*, Berlin, Germany, March 2000.
17. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.