Firm-Effects

—

Technischer Bericht

Boris Boesler Florian Liekweg

29. Juli 2005

Zusammenfassung

Um Komponenten zu beschreiben, die einem Übersetzer in einem Übersetzungslauf nicht in Quelltext vorliegen, weil sie entweder als Bibliotheken von Drittanbietern nur in binärer Form verfügbar sind, oder weil der Übersetzer aus Kapazitätsgründen ihren Quelltext nicht in einen Übersetzungslauf mit einbeziehen kann, wurde eine Beschreibungssprache entwickelt. Mit ihr lassen sich wichtige Eigenschaften der extern Komponenten beschreiben und dem Übersetzer zur Verfügung stellen, so daß Programmanalysen das Verhalten von Fragmenten der externen Komponenten mit berücksichtigen können.

Diese Arbeit wurde im Rahmen des Forschungsprojekts CATE (Go323/5-1) der Deutschen Forschungsgesellschaft (DFG) erstellt.

1 Einleitung

Bei der Programmanalyse eines Softwaresystems ist es nicht immer möglich, das System in seinem ganzen Umfang in die Analyse mit einzubeziehen.

Komponenten des Systemes können aus verschiedenen Gründen nicht in die Analyse mit einbezogen werden: Sie sollen nicht mit analysiert werden, um Zeit zu sparen; sie können nicht analysiert werden, weil die zur Verfügung stehenden Ressourcen nicht ausreichen; oder sie können deshalb nicht analysiert werden, weil — wie bei Komponenten, die von Drittanbietern bezogen wurden — ihr Quelltext nicht zur Verfügung steht.

Die Analyse muß die Effekte, die von Interaktionen mit solchen externen Komponenten verursacht werden, sehr konservativ abschätzen. Ohne weitere Informationen muß in einer Programmanalyse davon ausgegangen werden, daß auf sämtliche Programmfragmente, die *potentiell* für die unbekannte Komponente erreichbar sind, auch zugegriffen wird, so daß die Analyseinformationen für diese Fragmente konservativ abgeschätzt werden muß.

Mit der Beschreibungssprache für das Verhalten externer Komponenten, die in diesem Bericht bechrieben wird, kann das Verhalten solcher Komponenten in einer Form beschrieben werden, die von einer Programmanalyse statt des fehlenden Quellcodes analysiert werden kann.

2 Kriterien

Folgende Kriterien wurden bei dem Entwurf und der Implementierung der Spezifikation berücksichtigt:

Maschinenlesbares Format: Die Sprache, in der die Spezifikation verfasst wird, muß auf die Anwendung abgestimmt sein. Für Daten, die vornehmlich von Menschen erstellt und geändert werden (z. B. Programmtexte) bietet sich eine Sprache an, die in Form und Umfang dem Menschen entgegenkommt, wobei der Mehraufwand im Bereich der Zerteilung und semantischen Analyse gerechtfertigt ist. Für Daten, die maschinell erstellt werden können, und die nach ihrer Erstellung nicht in nennenswertem Umfang geändert werden, bietet sich eine maschinennahe Darstellung an.

Automatisch generierbar: Die Spezifikationen sollen automatisch aus existierendem Quelltext generiert werden können.

Quellsprachunabhängigkeit: Die Spezifikationen soll möglichst sprachunabhängig sein.

Geeigneter Detaillierungsgrad: Mit der Spezifikation soll das Verhalten externer Komponenten soweit beschrieben werden können, daß das Verhalten der Komponenten in Programmanalysen und Optimierungen ausreichend genau bekannt ist.

Integration in die Programmanalysen: Die einzelnen Informationen, die in einer Spezifikation geliefert werden, müssen von Programmanalysen berücksichtigt werden. Es ist wünschenswert, daß Programmanalysen diese Informationen mit einbeziehen können, ohne in nennenswertem Umfang angepasst oder erweitert werden zu müssen.

3 Umsetzung

Um die Kriterien aus Abschnitt 2 umzusetzen, wurden folgende Entscheidungen getroffen:

Internes Format: Die interne Repräsentation der Effekte bilded im Wesentlichen die Struktur des weiter unten beschriebenen `effect`-Elements ab. Auch wenn es möglich ist, die interne Repräsentation aus anderen Quellen als aus den im folgenden beschriebenen XML-Dateien aufzubauen, sind diese die Hauptquellen für Spezifikationen. Deshalb konzentriert sich dieser Bericht im folgenden auf die externe Darstellung.

Externes Format: Die Spezifikationen liegen als XML-Dateien vor. Für diese Format stehen zuverlässige und schnelle Zerteiler (zum Einlesen) und Formatierer (zur Generierung) in vielen gängigen Programmiersprachen (C, C++, Java[1], C#) zur Verfügung.

Für die Spezifikation existiert eine *Document Type Definition* (DTD), die das Format der Spezifikationen bestimmt. Spezifikationen lassen auf Konformität mit dieser DTD überprüfen.

Elemente der Spezifikation: Die Sprachelemente, die innerhalb der Spezifikation zur Verfügung stehen, sind sehr eng mit der Zwischensprache *Firm* [2] verwandt. Detaillierte Kentnisse über *Firm* sind jedoch nicht nötig, um eine Spezifikation zu erstellen oder die Generierung von Spezifikationen aus eigenem Quellcode zu implementieren.

In einer Spezifikation können atomare und zusammengesetzte Typen und Methoden beschrieben werden; für Methoden können Zugriffe auf Variablen und Objektfelder, Aufrufe von Methoden und die Verwendung von Argumenten und Zwischenergebnissen und die Rückgabe eines Wertes spezifiziert werden. Für eine Methode kann spezifiziert werden, ob und ggf. welche Ausnahme entstehen kann.

Ausdrucksmächtigkeit: Die Spezifikationen sind weitaus weniger mächtig als dies Quellcode typischerweise ist. Damit wird zum einen Komplexität bei der Erstellung der Spezifikationen vermieden. Zum anderen wird damit auch dem möglichen Wunsch von Drittanbietern Rechnung getragen, die genaue Funktionsweise ihrer Komponenten nicht offen zu legen.

Die Spezifikation enhält keine Anweisungen zur Ablaufsteuerung und keine Arithmetischen Operatoren, insbesondere keine Vergleiche und boolschen Operationen. Sie bietet die Möglichkeit, einen nicht näher spezifizierten Wert *„Unbekannt“* zu erstellen, der in andere spezifizierte Operationen mit eingeht.

Integration in den Übersetzer: Soll ein Übersetzer externe Komponenten nutzen, ohne daß deren Quellcode zur Verfügung steht, so muß zumindest eine Schnittstellenbeschreibung der externen Komponenten zugänglich sein (vgl. *Header*-Dateien in C und C++, *Bytecode*-Klassen in Java, Meta-Informationen in C#-*Assemblies*). Innerhalb dieser Schnittstellenbeschreibungen müssen auch sprachspezifische Merkmale enthalten sein, wie Alias-Namen für Typen (vgl. `typedef` in C) oder objektorientierte Beziehungen (Vererbung, Erweiterung, Zugriffsrechte, polymorphe Methoden). Solche Merkmale brauchen in der Spezifikation nicht vorhanden zu sein.

4 Details der Spezifikation

Die Details der Spezifikation wird anhand der *Document Type Definition* beschrieben, und durch kurze Beispiele erläutert.

4.1 Namesgebung

Die Elemente, die in der Spezifikation beschrieben sind, müssen vom Übersetzer mit den als extern deklarierten Komponenten in Bezug gebracht werden können. Dazu müssen den Elementen der Spezifikation geeignete Namen gegeben werden. Die Namen, die in der Spezifikation vergeben werden, müssen den kanonischen übersetzerinternen Namen entsprechen.

4.2 Inhaltsmodelle

Um Querverweise zwischen Elementen einer Spezifikation zu erlauben, werden Bezeichner deklariert:

```
<!-- content model for all nodes -->
<!-- allow nodes/entity accesses to reference other nodes/an entity -->
<!ENTITY    % firm.id       "id ID #REQUIRED">
```

Die Elemente, die mit einem Bezeichner versehen sind, können durch Bezug auf den Bezeichner referenziert werden.
Die einzelnen Elemente werden weiter unten beschrieben.
Nötig sind Referenzen auf Zwischenergebnisse durch Angabe einer `firm.id`, die in die weiteren Elemente offen eingebaut ist, auf Typen durch die Angabe eines `firm.type`-Bezeichners, und auf *Entities* durch `firm.entity`-Bezeichner.

```
<!-- allow a node to specify a type -->
<!ENTITY    % firm.type     "type CDATA #REQUIRED">

<!-- allow a node to specify a field -->
<!ENTITY    % firm.entity       "entity CDATA #REQUIRED">

<!-- allow an entity to specify an owner -->
<!ENTITY    % firm.owner        "owner CDATA #REQUIRED">
```

Eine Spezifikation besteht aus einer Folge von Typdeklarationen, einer Folge von *Entity*-Spezifikationen und einer Folge von Effekt-Spezifikationen:

```
<!-- the root element -->
<!ELEMENT   effects (type*, entity*, effect*)>
<!ATTLIST   effects
            module      CDATA       #IMPLIED>
```

Jede dieser drei Listen kann leer sein. Damit können Einträge, die häufig verwendet werden, in eine eigene Spezifikation ausgelagert werden.

Mit den Typ-Elementen umfassen atomare und zusammengesetzte Typen. Verbundtypen werden durch Angabe ihrer Elemente beschrieben; diese Elemente werden sowohl in *Firm* als auch innerhalb der Spezifikation als *Entities* angesprochen. Enthält ein Typ `T` eine *Entity* `e`, so wird `T` als Eigentümer (*owner*) von `e` bezeichnet.

```
<!-- a type reference -->
<!ELEMENT   type    EMPTY>
<!ATTLIST   type
            %firm.id;
            %firm.type;>
```

```
<!-- an entity reference -->
<!ELEMENT   entity  EMPTY>
<!ATTLIST   entity
            %firm.id;
            %firm.type;
            %firm.entity;
            %firm.owner;>
```

Eine Effekt-Spezifikation spezifiziert die Effekte einer einzelnen Methode. Für eine Methode können Null oder mehr Argumente spezifiziert werden, die als Zwischenergebnisse zur Verfügung stehen. Nach einer möglicherweise leeren Liste von Zwischenschritten (deren Elemente weiter unten beschrieben werden), kann optional das Entstehen einer Ausnahme beschrieben werden. Die Methodenbeschreibung endet mit der Rückkehr zum Aufrufer.

```
<!-- a single effect -->
<!ELEMENT   effect (arg*,
                    (load|store|alloc|call|join|unknown)*,
                    raise?, ret)>
<!ATTLIST   effect
            procname    CDATA       #REQUIRED
            %firm.owner;>
```

Die Argumente einer Methode werden zusammen mit ihren Typen und ihrer Position in der Argumentliste spezifiziert. Handelt es sich bei der Methode um eine Objekt-Methode, die im Kontext eines Objektes ausgeführt wird, so ist das erste Argument eine Referenz auf das Objekt.

Durch die Deklaration von Argumenten in der Spezifikation werden nicht nur Bezeichner für die weitere Verwendung der Werte der Argumente vereinbart; es wird auch bestimmt, daß die Argumente in der tatsächlichen Implementierung benötigt werden, und nicht durch eine Optimierung entfernt werden dürfen.

```
<!-- procedure argument -->
<!ELEMENT   arg         EMPTY>
<!ATTLIST   arg
            %firm.id;
            number      CDATA       #REQUIRED
            %firm.type;>
```

Um in anderen Elementen den Zugriff auf Zwischenergebnisse spezifizieren zu können, wird ein eigenes Verweis-Element eingeführt:

```
<!-- reference another value -->
```

```
<!ELEMENT   valref      EMPTY>
<!ATTLIST   valref
            refid       IDREF       #REQUIRED>
```

Die Berechnung einer Speicheradresse für Lade- und Speicheroperationen sowie für den statischen und polymorphen Aufruf von Methoden wird in dem Selektionselement zusammengefasst. Das Element ist immer in die Elemente integriert, die die Adresse benötigen, und hat deshalb keine eigene `firm.id`. Das Attribut `firm.entity` bezeichnet das Feld bzw. die Methode, deren Adresse berechnet wird.
Durch die Berechnung der Speicheradresse wird angezeigt, daß die referenzierten Programmelemente im tatsächlichen Programmcode benötigt werden. Optmierungen dürfen sie (insbesondere die *Entity*, über die gelesen bzw. geschrieben wird, und die Methode, die aufgerufen wird) nicht entfernen.

```
<!-- select element for use with store, load and call -->
<!-- for static entities, use a 'select' without a "valref" child
     element -->
<!ELEMENT   select      (valref?)>
<!ATTLIST   select      %firm.entity;>
```

Das Laden eines Wertes von einer Variable und das Speichern eines Wertes in einer Variable wird mit den Lade- bzw. Speicher-Elementen beschrieben. Der geladene Wert steht unter der `firm.id` des `load`-Elements zur Verfügung.
Durch die Spezifikation von Lade- oder Speicheroperationen wird angezeigt, daß der Inhalt des adressierten Speichers gelesen bzw. geändert wird. Programmanalysen müssen diese Änderung berücksichtigen.

```
<!-- load effect -->
<!ELEMENT   load        (select)>
<!ATTLIST   load
            %firm.id;>

<!-- store effect -->
<!ELEMENT   store       ((valref|select),valref)>
```

Die Allokation von Speicher auf der Halde wird durch das Allokations-Element beschrieben. Das Attribut `firm.type` beschreibt, von welchem Typ das allozierte Objekt ist. Eine Referenz auf das allozierte Objekt steht mit dem Bezeichner `firm.id` zur Verfügung.
Durch die Spezifikation der Allokation wird angezeigt, daß im tatsächlichen Programmcode Objekte der angegebenen Klasse alloziert und benutzt werden. Programmanalysen und Optimierungen müssen dies berücksichtigen (z. B. bei der Berechnung aufrufbarer Methoden im allgemeinen, und bei der Monomorphisierung von Aufrufen im speziellen).

```
<!-- alloc effect -->
<!ELEMENT   alloc       EMPTY>
<!ATTLIST   alloc       %firm.id;
```

Der Aufruf an eine Methode wird mit dem Aufruf-Element beschrieben. Um einen statischen oder dynamischen Aufruf über eine Referenz auf ein Objekt zu beschreiben, kann als erstes Unterelement ein Selektionselement angegeben werden. Um einen Aufruf über ein unbekanntes Objekt zu beschreiben, kann stattdessen ein Verweis-Element geschrieben werden. Mit den weiteren Verweisen werden die Argumente der Methode beschrieben.
Programmanalysen müssen hier die Effekte der aufgerufenen Methoden berücksichtigen.

```
<!-- call effect -->
<!ELEMENT   call        ((select|valref),(valref)*)>
<!ATTLIST   call        %firm.id;>
```

Um einen Wert zu beschreiben, der entweder unbekannt ist oder der nicht näher beschrieben werden soll, steht das Unbekannt-Element zur Verfügung:

```
<!-- unknown value -->
<!ELEMENT   unknown     EMPTY>
<!ATTLIST   unknown     %firm.id;>
```

Mit dem Verschmelzungs-Element kann man einen Wert spezifizieren, der zur Laufzeit einer von zwei oder mehrerern Zwischenergebnissen sein kann. Dies ist besonders dann interessant, wenn sich unter den Zwischenergebnisse, die verschmolzen werden, *kein* unbekannter Wert befindet.

```
<!-- join two or more values -->
<!ELEMENT   join        (valref,valref+)>
<!ATTLIST   join        %firm.id;>
```

Um zu spezifizieren, daß möglicherweise eine Ausnahme entstehen kann, dient das Ausnahme-Element. Mittels des Verweis-Unterelementes und des `firm.type`-Attributes lassen sich sprachspezifische Details über die Art der Ausnahme spezifizieren.

```
<!-- raise an exception -->
<!ELEMENT   raise       (valref)>
<!ATTLIST   raise       %firm.type;>
```

Der Rücksprung der Methode in ihren Aufrufer beendet die Spezifikation eines Methodeneffektes. Wenn die Methode einen Wert zurückgibt, so muß das Verweis-Unterelement vorhanden sein. Ist über den Rückgabewert nichts bekannt, oder soll er nicht näher spezifiziert werden, so kann ein Verweis auf einen als unbekannt spezifiziertes Zwischenergebnis übergeben werden.

```
<!-- ret effect -->
<!ELEMENT   ret     (valref?)>
```

4.3 Beispiele

Für die folgenden Beispiele setzen wir die folgende Klassendefinition(en) vorraus. Sie sind als *Java*-Quelltext [1] zu verstehen.

```
class Counter {
  public Counter (int initialCount) {
    _count = initialCount;
  }

  private int _count;

  public int getCount () {
    return (_count);
  }

  public void increaseCount () {
    _count = _count + 1;
  }
}
```

Nach der obligatorischen Einleitung für XML-Dokumente beginnen wir die Beschreibung dieser Klasse als ein externes Modul des Namens „`Counter-Effects`“:

```
<?xml version="1.0"?>
<!DOCTYPE effects
  SYSTEM "http://www.info.uni-karlsruhe.de/~firm/effect.dtd">

<effects module="Counter-Effects">
```

Zunächst werden die Typen spezifiziert, die in der Beschreibung benötigt werden. Die atomaren Typen sind `int` und `void`:

```
  <type id="Tint" type="int"/>
  <type id="Tvoid" type="void"/>
```

Als Verbundtyp ist hier die Klasse `Counter` nötig. Daraus ergibt sich weiterhin die Notwendigkeit, einen atomaren Typ *Zeiger auf* `Counter` zu spezifizieren. Als Platzhalter für den Verbund, der alle statischen Methoden enthält, legen wir den Typ `GlobalType` an.

```
<type id="Tglobal" type="GlobalType"/>
<type id="Tcounter" type="Counter"/>
<type id="Tcounter_ptr" type="Counter_ptr_t"/>
```

Das Feld `_count` der Klasse `Counter` spezifizieren wir als *Entity* mit der Klasse `Counter` als Eigentümer:

```
<entity id="E_counter" type="Tint"
        entity="_counter" owner="Tcounter"/>
```

Um die Spezifikation des Konstruktors `Counter(int)` zu erstellen, muß man die Semantik der zu Grunde liegenden Quellsprache beachten: Konstruktoren sind statische Methoden, werden also nicht im Kontext eines Objektes ausgeführt; sie allozieren den benötigten Speicher und führen dann die Anweisungen des Quelltextes aus. Damit wird der Konstruktor zu einer Methode, die den Platzhalter `GlobalType` als Eigentümer hat, die den nötigen Speicher alloziert und initialisiert. Implizit hat diese Methode den Rückgabewert *Zeiger auf* `Counter`.

Zunächst wird die Existenz der Methode spezifiziert:

```
<entity id="Ecounter_ctor"
        type="Tcounter_ptr"
        entity="Counter_Counter_int"
        owner="Tglobal"/>
```

Dann kann sich die Spezifikation des Verhalten der Methode darauf beziehen. Innerhalb des Effekt-Elements werden zuerst die Argumente spezifiziert:

```
<!-- effect -->
<effect procname="Counter_Counter_int" owner="Tglobal"/>
  <arg id="initialCount"  number="0001" type="Tint"/>
```

Dann wird der Speicherplatz alloziert und wird unter dem Bezeichner `count_obj` zur Verfügung gehalten:

```
<alloc id="count_obj" type="Tcounter"/>
```

Der Quelltext spezifiziert, daß das (einzige) Argument in das Objektfeld `_count` geschrieben wird. Dazu berechnen wir innerhalb des Speicher-Elements die Adresse und spezifizieren das Argumen als zu schreibenden Wert:

```
<store>
  <select entity="E_counter">
    <valref refid="count_obj"/>
  </select>
```

```
      <valref refid="initialCount"/>
    </store>
```

Der Rückgabewert des Konstruktors ist die Referenz auf das allozierte Objekt. Damit ist die Spezifikation des Konstruktors abgeschlossen:

```
    <ret>
      <valref refid="count_obj"/>
    </ret>
  </effect>
```

Die weiteren Methoden werden wie folgt spezifiziert:

```
  <effect procname="Counter_getCount_int" owner="Tcounter">
    <arg id="this" number="0001" type="Tcounter_ptr"/>

    <load id="count_val">
      <select entity="E_counter">
        <valref refid="this"/>
      </select>
    </load>

    <ret>
      <valref refid="count_val"/>
    </ret>
  </effect>
```

Für die Methode `Counter.increaseCount()` wird der neu berechnete Wert für das Feld `_count` durch einen Unbekannt-Element spezifiziert:

```
  <effect procname="Counter_increaseCount_void" owner="Tcounter">
    <arg id="this" number="0001" type="Tcounter_ptr"/>

    <load id="count_val">
      <select entity="E_counter">
        <valref refid="this"/>
      </select>
    </load>

    <unknown id="some_val"/>

    <join id="new_val">
      <valref refid="count_val"/>
      <valref refid="some_val"/>
```

```
    </join>

    <store>
      <select entity="E_counter">
        <valref refid="this"/>
      </select>
      <valref refid="new_val"/>
    </store>

    <ret/>
  </effect>
```

5 Technische Aspekte der Umsetzung

Die Struktur einer Spezifikation, insbesondere die des `effect`-Elements, ist sehr nahe an die Darstellung von Programmcode in *Firm* angelehnt. Daraus ergibt sich für die meisten Unterelemente des `effect`-Elements sowohl eine einfache, geradlinige Umsetzung innerhalb eines Übersetzers als auch eine klare Interpretation der Semantik.

5.1 Grundidee der Umsetzung

Für die Typ- und *Entity*-Elemente müssen die beschriebenen Typen, Klassen und Variablen mit ihren Gegenstücken aus der Schnittstellenbeschreibung in Übereinstimmung gebracht werden.

Während die Kenntnis der Zwischensprache *Firm* für die Erstellung von Spezifikationen nicht unbedingt nötig ist, ist für diesen Abschnitt ein Überblick über *Firm* wie zum Beispiel in [2] zu empfehlen.

Die Unterelemente eines `effect`-Elements lassen sich in einen einfachen *Firm*-Graphen umsetzen. In fast allen Fällen lassen sich die einzelnen Unterelemente direkt in *Firm*-Operationen umsetzen (`load`, `store`, `alloc`, `call`, `unknown`), die im Graph nacheinander in der Reihenfolge ausgeführt werden, in der sie in der Spezifikation auftreten.

Für die Elemente `join` und `raise` gestaltet sich der Aufbau etwas aufwendiger.
Um für ein `join`-Element die Verschmelzung der referenzierten Werte anzugeben, wird eine Verzweigung in der Ablaufsteuerung erstellt, an deren Ende die zu verschmelzenden Werte durch einen Φ-Knoten zusammengefasst werden. Die Verzweigung selber wird durch einen Wert gesteuert, der als unbekannt gekennzeichnet wird, so daß eine Optimierung keinen der angegebenen Fälle ausschließen kann.
Für ein `raise`-Element kann, ähnlich wie für das `join`-Element, eine Verzweigung in der Ablaufsteuerung erstellt werden, die durch einen unbekannten Wert gesteuert wird. In einem Zweig der Verzweigung wird dann die obligatorische Rückkehr zum Aufrufer durchgeführt, im anderen wird die beschriebene Aushahme erstellt.

6 Einleitung

Die Programmanalyse analysiert ein Programm auf Basis einer Zwischendarstellung. Für jeden Knoten ist die Semantik bekannt, und damit können die eingehenden abstrakten Beschreibungen der Werte der Operanden analysiert, und das Ergebnis der Operation in einer abstrakten Beschreibung bestimmt werden. Die Ergebnisse können nicht bestimmt werden, falls das Programm mit einem Aufruf einer Methode, die nicht zur Analyse zur Verfügung steht, verlassen wird. Dies ist im Allgemeinen bei Aufrufen an das Laufzeit- oder das Betriebssystems gegeben. Die aufgerufene Methode ist auch dann nicht bekannt, falls ein System partitioniert wird, und nur einzelne Partitionen analysiert werden.

Steht eine aufgerufene Methode der Analyse nicht zur Verfügung, so muß die Analyse vom schlimmsten Fall ausgehen und annehmen, daß alle Daten verändert werden. Das Ergebnis ist dann ein Verlust an Genauigkeit der Analyse.

Benötigt wird eine Beschreibung von unbekannten Methoden, sodaß diese Beschreibung, statt der Methode selbst, in die Analyse eingeht. Dadurch wird die Genauigkeit der Analyse erhöht.

7 Voraussetzungen und Anforderungen

Wir können eine Reihe von Voraussetzungen treffen, die erfüllt sein müssen, um abstrakte Beschreibungen für Methoden zu entwickeln:

- Die abstrakte Beschreibung einer Methode wird einer Methodenentität in Firm eindeutig zugewiesen. Zur Identifikation gehören die Klasse, in der die Methode enthalten ist, der Name der Methode, der Ergebnis- und die Operandentypen.
- Das abstrakte Verhalten ist exakt bekannt. Eine ungenaue Spezifikation des Verhaltens führt zu falschen Analyseergebnisse.
- Die relevanten Operation, die das Verhalten der Methode charakterisieren, können angegeben werden. Dazu gehören Speicherzugriffe, Objektallokationen, Methodenaufrufe, etc.

Aus den Voraussetzungen folgen auch Anforderungen an die Spezifikaion:

- Die Spezifikation muß das Verhalten modellieren können. Dazu muß sie Konzepte der Zwischensprache Firm wiederverwenden (z. B. den Feldzugriff über Sel-Knoten).
- Die Spezifikation enthält Konstrukte, die die Konzepte der Zwischensprache widerspiegeln (Methodenaufrufe, Ausnahmen, usw.).

8 Realisierung

Wir spezifizieren die Typen, die Entitäten und das Verhalten von Methoden in XML. Die Typen und Entitäten in der Spezifikation können wir anhand eines Namens eindeutig den

```
class External {
    native static void callMe(Foreign f);
}
```

Abbildung 1: Java-Klasse mit einer Methode und Feldern

Typen und Entitäten in Firm zuordnen. Die abstrakte Beschreibung des Verhaltens ist eine Liste von Anweisungen.

Die Operationen der Beschreibung haben Gegenstücke in Firm. sodaß aus der Beschreibung ein normaler Firm-Graph aufgebaut werden kann. Diese Firm-Graphen unterscheiden sich nicht von normalen Programmgraphen, sodaß Firm sie in die Analyse mit einbeziehen kann. Sie unterscheiden sich nur von anderen Graphen, weil sie nicht optimiert werden dürfen, da sich sonst das Verhalten ändern würde.

9 Beispiel

In Abbildung 2 ist ein Beispiel einer Beschreibung für die Java-Klassse in Abbildung 1 gegeben.

Zuerst werden die Klassen `GlobalType` und `Foreign` definiert, dann der primitive Typen `int` und der Zeiger `Foreign_ptr_t` auf Objekte der Klasse Foreign. Die Felder `used` und `modified` der Klasse `Foreign` werden anschließend definiert, danach die Effekte der Methode `callMe_void_Foreign`. Zuerst wird das erste Argument mit der Nummer 0 und Typ `Tforeign_ptr` definiert, auf das mit dem Identifikator `arg0` zugegriffen werden kann. Dananch wird ein Wert `unknown` definiert, der alle Werte repräsentiert. Eine Ladeoperation liest aus dem Objekte, das an die Methode übergeben wurde das Feld `used` aus. Anschließend wird in diesem Objekt das Feld `modified` mit dem Wert `unknown` beschrieben. Damit ist das Verhalten der Methode beschrieben.

Aus der Beschreibung kann direkt ein Firm-Graph konstruiert werden, der dem Verhalten entspricht.

Für das Programmfragment in Abbildung 3 kann nun die Heapanalyse das exakte Ergebnis berechnen: Das Feld `used` des Objektes `f` wird nicht beschrieben und enthält nach dem Aufruf den Wert 1.

Literatur

[1] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1997.

[2] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-44, Dez 1999.

```
<?xml version="1.0"?>
<!DOCTYPE effects SYSTEM "http://www.info.uni-karlsruhe.de/~firm/effect.dtd">
<effects module="EffectTest1 -Effects">
  <!-- Klassen -->
  <type id="Tglobal" type="GlobalType"/>
  <type id="Tforeign" type="Foreign"/>

  <!-- Primitive Typen -->
  <type id="Tint" type="int"/>

  <!-- Referenzen auf Objekte -->
  <type id="Tforeign_ptr" type="Foreign_ptr_t"/>

  <!-- Felder in Objekten der Klasse Foreign -->
  <entity id="Eused"     type="Tint" entity="used"     owner="Tforeign"/>
  <entity id="Emodified" type="Tint" entity="modified" owner="Tforeign"/>

  <!-- Effekte fuer Methode callMe_void_Foreign -->
  <effect procname="callMe_void_Foreign" owner="Tglobal">
    <arg id="arg0" number="00000" type="Tforeign_ptr"/>
    <unknown id="unknown"/>
    <!-- Lese Feld used -->
    <load id="N00001">
      <select entity="Eused">
        <valref refid="arg0"/>
      </select>
    </load>
    <!-- Schreibe Feld modified -->
    <store>
      <select entity="Emodified">
        <valref refid="arg0"/>
      </select>
      <valref refid="unknown"/>
    </store>
    <ret/>
  </effect>

</effects>
```

Abbildung 2: Beschreibung einer Methode

```
class EffectTest1 {
  public static void main (String[] args) {
    Foreign f = new Foreign();
    f.used = 1;
    f.modified = 2;
    External.callMe(f);
    print(f.used);
    print(f.modified);
  }
}
```

Abbildung 3: Aufruf der unbekannten Methode