0.1 Semantische Verarbeitung des AST

Für die korrekte Verwendung des ASTs benötigt das DMF auch eine Softwarekomponente, welche den AST durchläuft und ein semantisches Modell erstellt. Anhand dieses semantischen Modells werden semantische Regeln überprüft.

Diese Softwarekomponente ist die erste Komponente welche mithilfe von Golang implementiert wird.

0.1.1 Das semantische Modell

Das semantische Modell bildet alle Informationen, die aus dem AST entnommen werden können, ab. Dazu gehören Referenzen zum AST für Positionen in der Modelldatei, die verschiedenen PackageElemente und die NamendElemente.

ErrorElemente

Für eine annehme und effiziente Entwicklung ist die verständliche Kommunikation von Fehlern essenziell. Deshalb enthält das semantische Modell das ErrorElement.

```
1  package err_element
2  type ErrorElement struct {
3      Fehler FehlerStelle
4      Error  error
5      // Falls es einen Grund gibt, weshalb ein Fehler aufgetreten ist und dieser
6      Cause    *FehlerStelle
7      rendered *string
8  }
9
10 type FehlerStelle struct {
11     // Die Node von der, der ModelCode stammt, und die ein Parent von der eigent
12     // Ist nil, wenn der ModelCode von der eigentlichen Node kommt
13     ContextNode *tree_sitter.Node
14     ModelCode   string
15     Node        *tree_sitter.Node
16 }
```

Mithilfe dieser Strukturen lassen sich Fehler festhalten ohne die spätere Darstellung mit einzubeziehen zu müssen. Der Error dokumentiert um welchen Fehler es sich handelt.
Die FehlerStelle “Fehler” dokumentiert den Modellcode, der falsch ist.
Die FehlerStelle “Cause” dokumentiert den Modellcode, wodurch der Modellcode, welcher in der “Fehler”-FehlerStelle enthalten ist, falsch wurde.
Jede FehlerStelle beinhaltet zwei Nodes. Diese beinhalten die Positionen in der Modelldatei. Die ContextNode ist optional und wird gesetzt wenn für einen Fehler der umliegende Code wichtig. Es handelt sich dabei um eine der Parent-Nodes.

Der ModelCode wird erst beim Rendern der FehlerStelle genutzt. Die Variable dient als Zwischenspeicher für den Code den die Nodes referenzieren.

ErrorElemente werden in der gesamten Semantik-Komponente genutzt und erzeugt. Spätere Komponenten nutzen die ErrorElemente, um den Entwickler*innen die Fehler zu erläutern.

0.1.2 Übersetzen des ASTs

Die Übersetzung des ASTs beginnt mit dem Erstellen eines SemanticContext. Dieser Kontext beinhaltet die erkannten Fehler, das bisherige Modell, den Text der Modelldatei und die TreeCursor. Mithilfe des TreeCursors kann der AST durchlaufen werden. Der Kontext durchläuft den AST in der PreOrder-Reihenfolge. Für jedes Element des ASTs enthält der Kontext eine Methode zum Parsen.
Sollte der AST Import Statements beinhalten so werden zunächst die referenzierten Modelle verarbeitet. Die importierten Packages werden nun in das Modell übernommen. Das Laden der verschiedenen Modelle wird mithilfe einer Callbackstruktur außerhalb der Semantik-Komponente definiert. So können verschiedene Logiken genutzt werden.
Am Ende des Parsen ist Modell vollständig mit allen gültigen Elementen. Elemente die durch den Parser im AST als Fehlerhaft gekennzeichnet wurden, werden ignoriert.

Der Lookup

Um anhand des vollständigen Namens (Package Pfad + Name) ein Element schnell zu finden, wird nach dem semantischen Parsen ein Lookup erstellt. Dieser Lookup nutzt eine Map für den schnellen Zugriff. Zum Befüllen des Lookups wird das semantische Modell durchlaufen. Jedes erreichte Package Element wird dem Lookup hinzugefügt. Beim Packages werden auch alle enthaltenen Elemente durchlaufen.

0.1.3 Die semantischen Regeln

Die semantischen Regeln basieren alle auf dem Typen “walkRule”. Dieser Typ verallgemeinert das Iterieren über den Lookup und nutzt eine Instanz der “iWalkRule” um die Elemente zu verarbeiten.

```
1 package semantic_rules
2 // Supertyp alle Regeln, welche alle Typen durchlaufen.
3 type walkRule struct {
4     lookup    *smodel.TypeLookUp
5     elements  []errElement.ErrorElement
6     iWalkRule iWalkRule
7 }
```

Semantische Regeln implementieren das Interface “iWalkRule” und erweitern die “walkRule”. Durch das Nutzen der eignen Instanz werden die Methoden der Regel-Implementierung aufgerufen, um Elemente zu verarbeiten.

```
1  package semantic_rules
2  func newComputeSuperTypes(lookup *smodel.TypeLookUp) *computeSuperTypes {
3      types := computeSuperTypes{
4          walkRule: &walkRule{
5              lookup:   lookup,
6              elements: make([]errElement.ErrorElement, 0),
7          },
8      }
9      types.iWalkRule = &types
10     return &types
11 }
```

Die Reihenfolge der semantischen Regeln ist sehr relevant, denn sie überprüfen nicht nur das semantische Modell, sondern setzen auch Referenzen und befüllen Lookups innerhalb der Elemente.
Im Folgenden werden die Regeln in ihrer Ausführungsreihenfolge erläutert.

Compute Supertypes Regel

Die “Compute Supertypes Regel” ermittelt und überprüft die Supertypen der PackageElementen. Dazu gehören alle Referenzen in den Extends- und Implements-Blöcken. Dabei werden folgende Bedingungen überprüft:

1. Der referenzierte Typ muss im Lookup vorhanden sein.
2. Die Vererbung darf nicht rekursiv sein.
3. Structs dürfen nur von Structs erben.
4. Entities dürfen nur von Structs und Entities erben.
5. Es können nur Interfaces implementiert werden.
6. Ein Interface kann sich nicht selber implementieren.

Nachdem die referenzierten Type ermittelt wurden, werden Referenzen zu diesen Typen in die jeweiligen PackageElemente eingetragen. So kann später noch schneller auf diese Elemente zugegriffen werden.

Compute Elements Regel

Die “Compute Elements Regel” ermittelt alle Elemente innerhalb der PackageElemente. Diese werden als NamedElements in dem jeweiligen PackageElement eingetragen, sodass mithilfe des Namens schnell auf das Element zugegriffen werden kann. Zu den NamedElements gehören:

1. Argumente
2. Referenzen

3. Multireferenzen
4. Funktionen
5. Konstanten

Beim Hinzufügen eines Elementes wird überprüft, ob der Name schon von einem Element in der Map genutzt wird.

Die “Compute Elements Regel” einzigartig, denn sie überschreibt das Verhalten des Iterierens durch den Lookup mit einem zweifachen Durchlauf. Denn um alle geerbten Elemente zu Bestimmen müssen die NamedElements-Maps der anderen PackageElemente schon die eigenen Elemente enthalten.

```
1  package semantic_rules
2  func (c *computeElements) walk() []errElement.ErrorElement {
3      c.walkRule.walk()
4
5      for _, element := range *c.lookup {
6          c.handleAbstraction(element)
7      }
8
9      return c.elements
10 }
```

Das Bestimmen der gerbten Elemente (handleAbstraction) ist rekursiv implementiert. So werden auch alle Elemente, die das erweiterte Element erbt mit eingeschlossen.

Check Entity Identifier Regel

Die “Check Entity Identifier Regel” überprüft, ob alle im Entity-Identifier referenzierten Variablen enthalten sind.

Check Enum Constants Regel

Die Konstanten eines Enums müssen semantische Regeln folgen. Deshalb überprüft diese Regel folgende Bedingungen für jede Konstante:

1. Der Name jeder Konstante muss innerhalb eines Enums einzigartig sein.
2. Der erste Wert muss ein Integer sein (Index).
3. Die Anzahl der Werte muss mit der Anzahl der Argumente im Enum übereinstimmen.
4. Der Typ der Werte muss mit den entsprechenden Argumenten übereinstimmen.
   Für die Reihenfolge der Argumente wird die Reihenfolge in der Modelldatei genutzt.

Check Referenzen Regel

Die “Check Referenzen Regel” überprüft, dass alle Referenzen existieren. In folgenden Elementen werden Referenzen überprüft:

1. Referenzen in Structs und Entities.
2. Return Typ und Parameter von Funktionen in Structs, Entities und Interfaces.
3. Generische Typen der Multireferenzen in Structs und Entities.