Entwicklung eines Editors für Datenflussdiagramme

Simon Schwarz

Institut für Programmstrukturen und Datenorganisation (IPD)
Betreuender Mitarbeiter: M.Sc. Stephan Seifermann

1 Einleitung

Bei Datenflussdiagrammen handelt es sich um eine datenorientierte Modellierung von Systemen [1]. Sie werden in Bereichen der Systemanalyse und Softwareentwicklung verwendet, insb. der Erfassung von Anforderungen (*Requirements Engineering*) und der Sicherheitsanalyse. Ihre grundlegende Syntax und Semantik sind überschaubar, eine Charakteristik führt allerdings zu Komplexität: Die Unterstützung für sogenannte *Hierarchisierung* (*Leveling*), dass heißt die Darstellung von Abläufen in verschiedenen Abstraktionsebenen. Dies geschieht durch die sukzessive Verfeinerung eines Ausgangsdiagrammes. Solche Verfeinerungen müssen konsistent erfolgen. Der im Rahmen dieses Praktikums erstellten Editors für Datenflussdiagramme muss also sowohl die programmatische Hierarchisierung als auch die Konsistenzprüfung von manuell hierarchisierten Diagrammen unterstützen. Konzeptionell ist dafür eine präzise Definition dieses Vorgangs, der Entwurf eines Meta-Modells für hierarchisierbare Datenflussdiagramme und eines Validierungsalgorithmus erforderlich.

2 Aufgabenstellung und Hintergrund

Diese Sektion gibt einen Überblick über Datenflussdiagramme, die verwendeten Werkzeuge und die Aufgabenstellung.

2.1 Datenflussdiagramme

Komponenten Datenflussdiagramme bestehen aus zwei Basiskomponenten: Den eigentlichen Elementen, dargestellt in Abb. 3 und dem sogenannten Data Dictionary. Letzteres enthält Informationen über die für das Diagramm verfügbaren Typen. Im Rahmen des Editors enthält dieses drei verschiedene Entitäten: *Primitive* (nur ein Name), *Collection*

(Name und Typ) sowie *Composite* (Name und Einträge, bestehend jeweils aus einem Namen und Typ). Die explizit dargestellten Elemente eines Datenflussdiagrammes sind die folgenden:

- **Prozess**: Prozesse transformieren ihre eingehenden Datenflüsse in die ausgehenden, modellieren also aktive Daten-Verarbeitung.
- **Externer Aktor**: Externe Aktoren können nur ausgehende Datenflüsse haben, modellieren also die initiale Einführung von Informationen in das System.
- **Store**: Speicher führen in der Regel keine Transformation von Daten durch sondern modellieren die klassische Lese/Schreibe - Semantik mittels ihrer ausgehenden/eingehenden Datenflüsse.
- **Datenfluss**: Ein Datenfluss transportiert eine Menge von Daten-Objekten von der Quelle zum Ziel; diese bleiben dabei unverändert. Es existiert dabei kein Konzept von Quantität.

Außerdem existieren implizit, d.h. nicht grafisch dargestellt, Daten-Objekte. Diese haben einen Namen und einen Typ (aus dem Data Dictionary). Für diesen Editor wurden außerdem neue Varianten der oben genannten Elemente entworfen. Verfeinerte Prozesse, d.h. Prozesse für die ein verfeinerndes Diagramm existiert haben eine neue Darstellung. Ebenso wird in solchen verfeinernden Diagrammen die Nachbarschaft des Prozesses als externe Elemente in grau dargestellt. Diese Elemente sind nicht modifizierbar und dem verfeinerten Datenflussdiagramm zuzuordnen. Es werden also die selben Knoten (nicht Kopien davon) gegebenenfalls in mehreren Diagrammen angezeigt.

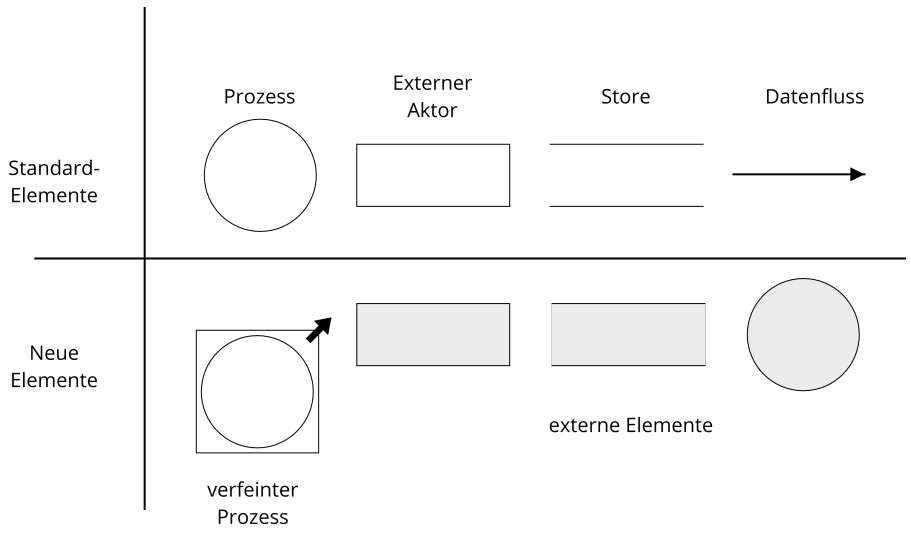

Abbildung 1: Die grafischen Elemente von Standard- und verfeinerten Datenflussdiagrammen.

Hierarchisierung Die Prozesse (nicht die Stores und Externen Aktoren) eines Datenflussdiagrammes können verfeinert werden, was das Erstellen eines neuen Datenflussdiagrammes bedeutet. Einem Datenflussdiagramm können auf diese Weise mehrere verfeinernde

Diagramme zugeordnet werden. Diese sind im Allgemeinen als Beschreibung ihres jeweiligen Prozesses in einer geringeren Abstraktionsebene anzusehen, formell gibt es jedoch für ihren Inhalt keine Einschränkungen. Es müssen jedoch alle eingehenden und ausgehenden Datenflüsse jeweils konsistent zu den oben ein- und ausgehenden sein. Das bedeutet, dass diese entweder identisch oder gemäß der in Abb. 2 dargestellten Semantik verfeinert worden sein. Es gibt zwei Arten der Hierarchisierung: Enthält ein Datenfluss mehrere Datenobjekte, werden aus diesen jeweils neue Datenflüsse, die nur noch aus einem Daten-Objekt bestehen. (*Leveling A*). Falls ein Datenfluss nur ein Daten-Objekt enthält, dessen Typ aber *Composite* ist, so ist auch eine Verfeinerung möglich. Aus jedem Eintrag dieses Typen wird ein neuer Datenfluss mit dem Eintrag als neues Datenobjekt (*Leveling B*). An dieser Stelle ist die Namensgebung nicht mehr eindeutig. Deswegen kommen sogenannte *naming schemes* zum Einsatz. Das aktuell implementierte Schema fügt einfach (jeweils inkrementierte) Zahlen an den Namen des Daten-Objekts hinzu.

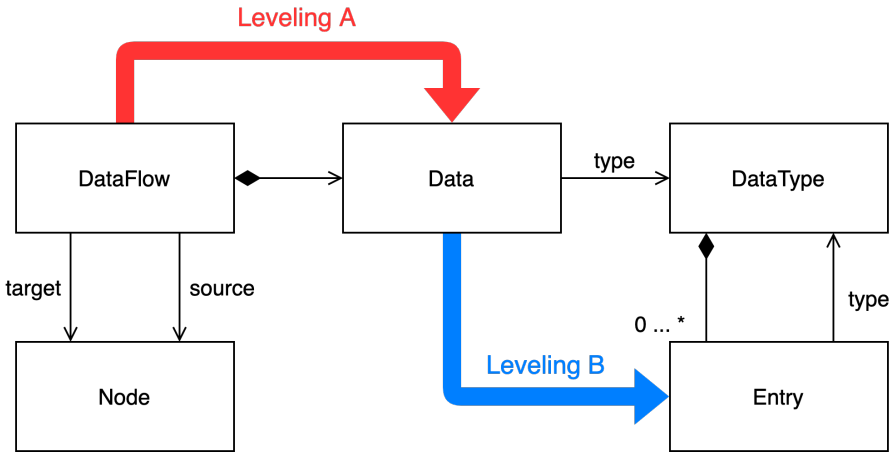

Abbildung 2: Die zwei Arten von Leveling: Daten-Objekt- oder Datentyp-basiert.

2.2 Eclipse-Sirius

Sirius [2] ist Teil von Eclipse und ein Framework für das Erstellen von Editoren. Es kann als Weiterentwicklung des *Eclipse Modelling Frameworks* (EMF) angesehen werden. Ausgangspunkt für die Entwicklung ist ein mittels EMF erstelltes Meta-Modells für eine Domain. Sirius stellt dann eine Reihe von Werkzeugen bereit, die die Implementierung von Editoren ermöglichen. Diese Editoren können dann verwendet werden um Instanzen des Meta-Modells zu modifizieren. Die wesentlichen Bestandteile von Sirius sind:

- **Sirius-Elemente**: Wie in Abb. 3 dargestellt können Objekten des Meta-Modells grafische Elemente zugeordnet werden. Dies geschieht durch die explizite Angabe der zugehörigen Klasse.
- **AQL**: Die *Acceleo Query Language* (AQL) kann als Weiterentwicklung der bekannten *Object Constraint Language* (OCL) gesehen werden. Die Editor-Umgebung unterstützt beide Sprachen: OCL im Rahmen des EMF und AQL als Sprache für Sirius.

Mittels AQL-Ausdrücken wie in Abb. 4 können Elemente des Meta-Modells ausgewählt (aber nicht modifiziert) werden.

- **Sirius-Operationen**: Modifikationen des Modells sowie die Interaktion des Editor-Interfaces mit dem Modell werden mittels Operationen definiert. Beispiele für solche Operationen sind das Erstellen von Instanzen einer gegebenen Klasse, das Zuweisen von Attributwerten oder das Öffnen eines Editor-Fensters.
- **Services**: Für Aufgaben, die nicht (oder nicht einfach) mittels Operationen gelöst werden können gibt es die Möglichkeit Java-Services aufzurufen. Darunter versteht man vom Entwickler deklarierte Methoden, die mindestens einen Parameter (ihren Aufrufkontext) erwarten. Hier stehen außerdem eine Reihe von Sirius-APIs zur Verfügung.

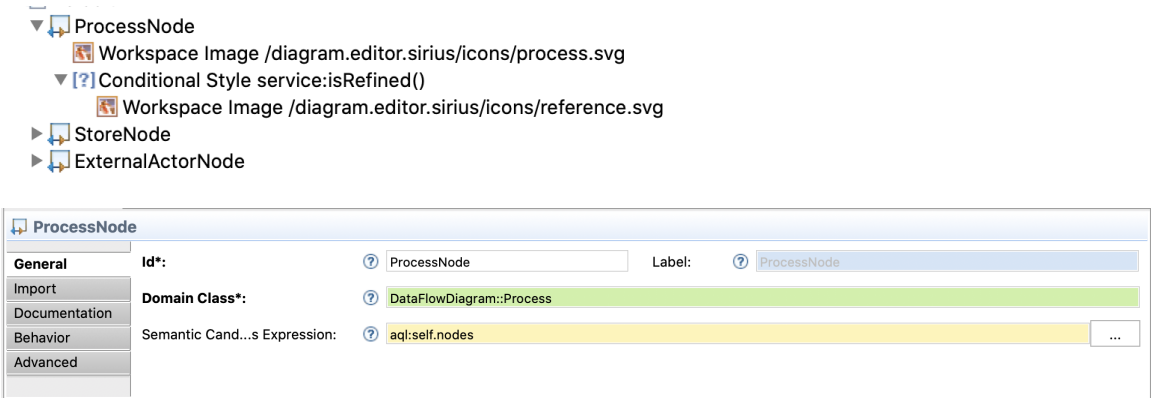

Abbildung 3: Das Erstellen von im Editor sichtbaren Elementen erfolgt in zwei Schritten: Durch das Zuweisen der eigentlichen Grafik und der semantischen Bedeutung.

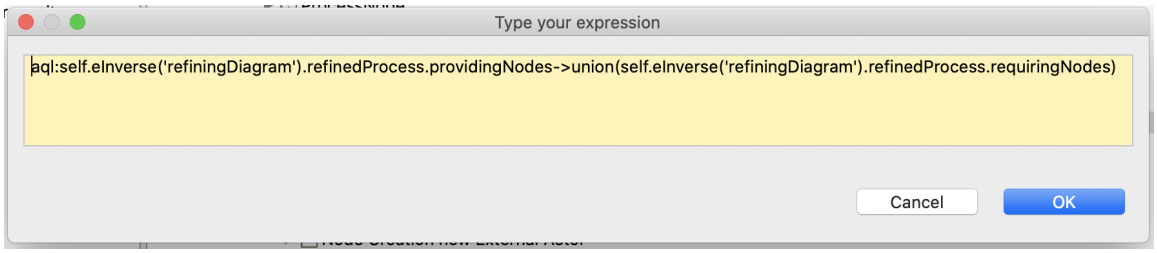

Abbildung 4: Ein AQL-Statement im Sirius-Editor.

2.3 Aufgabenstellung

Ziel dieses Praktikums ist die Erstellung eines grafischen Editors für Datenflussdiagramme im Rahmen des Eclipse-Sirius Frameworks. Konkreter soll dieser EMF-Meta-Modelle auf die Standard-Syntax abbilden und dem Nutzer somit das Erstellen und Editieren solcher Diagramme erlauben. Zusätzlich soll die programmatische Hierarchisierung von Datenflussdiagrammen möglich sein. Inhärent zu diesen beiden Anforderungen ist die Konzeption und Implementierung eines Validierungs-Algorithmus, der dem Nutzer bei der

Konsistenzhaltung der Diagramme unterstützt. Ferner erfordert dies eine präzise Definition der Semantik des (bisher vagen) Konzepts der Hierachisierung und eine Abbildung derselben auf die Elemente eines selbst erstellten Meta-Modells solcher Datenflussdiagramme. Auf technischer Ebene gilt es Prinzipien des Software-Engineerings einzuhalten, da der Editor als Grundlage für weitere Arbeiten bzw. Anwendungsbereiche dienen soll.

3 Ergebnisse

Die folgenden Sektionen geben einen kurzen Überblick über die erzielten Ergebnisse. Dazu zählen die erarbeiteten Meta-Modelle, der implementierte Editor und das Validierungs-Konzept in Form eines *Generate and Test*-Algorithmus.

3.1 Meta-Modelle

Allen Sirius-Projekten liegt stets ein Meta-Modell zu Grunde. Dieses kann instanziiert werden. Diese konkreten Modelle können dann mit einem (wiederum mittels des Meta-Modell erstellten) Editors bearbeitet werden. Wäre nur eine grafische Darstellung von Datenflussdiagrammen gewünscht, wären große Teile des in Abb. 5 dargestellten Meta-Modells nicht erforderlich. Aber aufgrund der Umsetzung der oben beschriebenen Hierarchisierung müssen vom Modell bzw. dem Editor viele für den Nutzer nicht sichtbare Elemente verwaltet werden. Diese werden im Nachfolgenden kurz beschrieben.

- **berechnete Relationen**: Die mit Schrägstrich dargestellten Relationen sind auf Basis der anderen Elemente des Objekts berechnet und müssen nicht manuell gesetzt oder aktualisiert werden. So führt z.B. die Relation */refiningDiagram* nicht zu einer zirkulären Abhängigkeit, sondern ist eine Invertierung der *refinedBy*-Relation.
- **Typreferenzen**: Die gestrichelt dargestellten Teile des Modells stellen eine externe Referenz dar, d.h. das Objekt *Data Type* ist Teil des Data Dictionary-Meta-Modells.
- **Verfeinerungsreferenzen**: Die Objekte *Data Flow Diagram Refinement* und *Edge Refinement* ermöglichen die Hierarchisierung von Datenflussdiagrammen. Erstere ermöglichen unter anderem die Navigation zwischen Diagrammen verschiedener Hierarchieebenen und Zuordnung zu dem jeweils verfeinerten Prozess. *Edge Refinement*-Objekte ermöglichen die Zuordnung von verfeinerten und verfeinernden Datenflüssen im Rahmen des weiter unten beschriebenen Validierungsalgorithmus. Bei programmatischem Leveling werden diese automatisch zugeordnet; bei der manuellen Erstellung von Datenflüssen hingegen muss der Nutzer gegebenenfalls mittels eines Dialogs die Zuordnung angeben.

Für den ebenfalls implementierten DD-Editor existiert (notwendigerweise) auch ein Meta-Modell, dieses ist aber weniger komplex, weitgehend selbsterklärend und wird nicht weiter beschrieben. Eines der in Abb. 5 nicht dargestellten Details ist die Verwendung einer (d.h. Referenz auf eine) Identifier-Klasse des am SDQ entwickelten Palladio-Simulators. Diese ist auch Teil der eigentlichen Vererbungshierarchie sorgt für das automatische Generieren einer eindeutigen ID beim Erstellen eines Objektes.

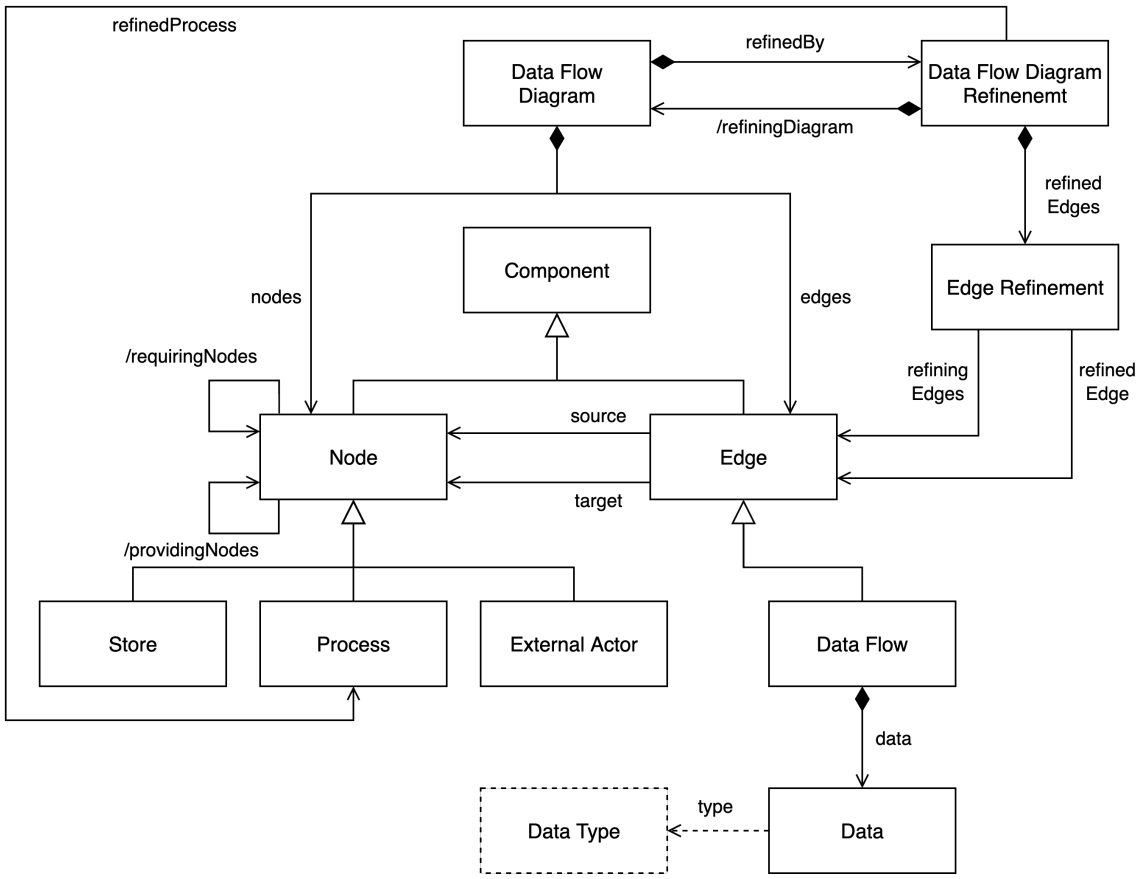

Abbildung 5: Das leicht vereinfachte Meta-Modell eines Datenfluss-Diagramms.

3.2 Editor

Wie oben und in Abb. 6 beschrieben besteht ein Sirius-Editor im Wesentlichen aus drei Elementen: Der zentralen grafischen Darstellung des zu editierenden Diagramms, einer Auswahl von Werkzeugen und der kontextsensitiven Property-Ansicht, die das Verändern des aktuell ausgewählten Elements ermöglicht. Die eigentliche Funktionalität des Editors besteht aus dem Modifizieren einer Instanz des in Abb. 5 dargestellten Meta-Modells. Die Implementierung dieser Modifikationen teilt sich auf zwei Komponenten auf deren Rollen im folgenden kurz diskutiert werden soll.

Sirius, AQL und EMF Auf Sirius-Ebene können *model queries* mittels AQL formuliert werden, d.h. das Auswählen von bestimmten Elementen des Modells. Auf diese Weise kann das Modell allerdings nicht modifiziert werden. Das Hinzufügen und Löschen von Elementen sowie das Setzen von Attributwerten geschieht mittels Sirius-Aktionen oder Java-Services, die weiter unten beschrieben werden. Nicht alle Elemente des Modells sind grafisch dargestellt. Man kann grob zwischen drei Arten von Elementen unterscheiden:

- **direkt sichtbar**: normale Prozesse, Stores, Externe Aktoren und Datenflüsse.
- **indirekt sichtbar**: verfeinerte Prozesse, also Prozesse, die zwar der Prozess-Klasse zugeordnet sind aber ein *custom design* haben oder Elemente eines anderen DFDs, die mittels eines *semantic candidate*-Statement sichtbar gemacht werden.
- **nicht sichtbar**: Ist einem Element (wie der Edge-Refinement-Klasse) kein Sirius-Element zugeordnet wird es entsprechend auch nicht angezeigt. Die zugehörigen Elemente im Modell können aber unverändert modifiziert werden.

Java Sirius-Aktionen können stets einen sogenannten *Service* aufrufen. Dabei handelt es sich um eine Java-Methode, die in einer bestimmten Datei deklariert sein und mindestens einen Parameter haben muss: ein Ecore-Objekt. Durch diese *hook* steht nun die ganze Ausdrucksmächtigkeit von Java zur Verfügung. Dies beinhaltet auch eine Reihe von EMF- bzw. Sirius-spezifischen Modulen mit Hilfsfunktionen für das Erstellen und Löschen von Meta-Modell-Objekten, das Laden von Ressourcen, Zugriff auf offene Editor-Instanzen etc.

Für data dictionaries existiert auch ein Editor; dieser ist allerdings konzeptionell deutlich einfacher und wird deswegen hier nicht gesondert beschrieben. Einzige technische Herausforderung ist hierbei das Verbinden der DFD- und DD-Meta-Modelle. Dies erfolgt über einen File-Selection-Dialog im DD-Editor und muss explizit in Java behandelt werden.

3.3 Validierung

Die zentrale Frage bei der Validierung des Zustandes eines Datenflussdiagrammes betrifft die Konsistenz des *Levelings*. Ein Leveling ist dann konsistent, wenn die ein- und ausgehenden Datenflüsse des verfeinernden DFDs aus den des verfeinerten DFDs abgeleitet werden können. Dieser (etwas abstrakte) Zusammenhang ist in Abb. 7 dargestellt.

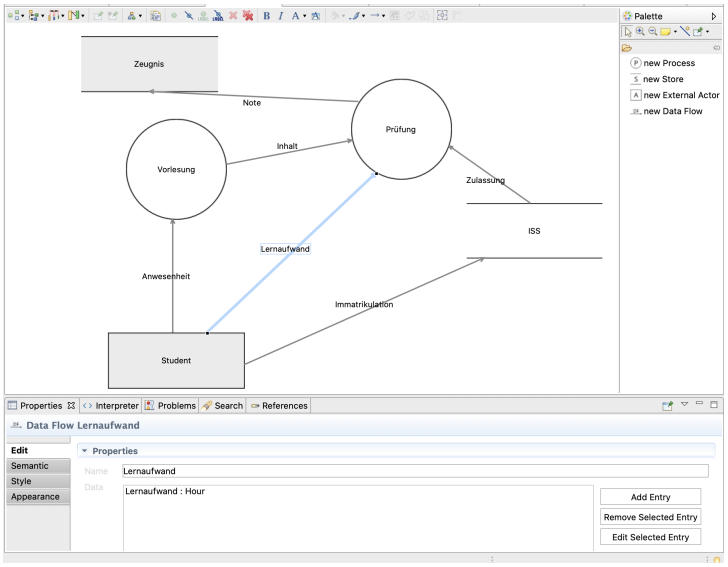

Abbildung 6: Das Interface des Editors, bestehend aus drei Teilen: Properties am unteren Rand, ein zentrales Klassendiagramm und ein Werkzeugbereich rechts.

Formelle Korrektheit Für die folgende Validierung werden Datenflussdiagramme graphentheoretisch modelliert. Ein Datenflussdiagramm D besteht aus einer Menge von Knoten V (Prozesse, externe Aktoren sowie Stores) und einer Mengen von Kanten E (Datenflüsse), es gilt also: $D = (V, E)$. Um die Verfeinerungsbeziehung zwischen Datenflussdiagrammen darzustellen wird im folgenden davon ausgegangen, dass die Menge der Knoten und Kanten eines (Basis-)Datenflussdiagramm D_0 eine Menge von verfeinernden Datenflussdiagrammen $D_1, \ldots, D_m$ induziert (bzw. induzieren kann). D_0 ist dann konsistent, wenn alle diese induzierten Datenflussdiagramme D_i konsistent bezüglich allen verfeinerten Knoten sind. (Hierbei ist zu beachten, dass ein induziertes Datenflussdiagramm selbst wieder DFDs induzieren kann). Neben dieser Rekurrenzbeziehung gibt es im Wesentlichen zwei weitere Konsistenzbedingungen: Sei p ein verfeinerter Prozess in einem Datenflussdiagramm $D_j = (V_j, E_j)$ mit $0 \leq j < m$. Seien $\Delta_+ \subseteq E_j$ sowie $\Delta_- \subseteq E_j$ die Mengen der ein- bzw. ausgehenden Datenflüsse. Entsprechend bezeichnen $\Theta_+ \subseteq V_j$ sowie $\Theta_- \subseteq V_j$ die benachbarten Elemente, die die Quelle bzw. Ziel der Datenflüsse in $\Delta = \Delta_+ \cup \Delta_-$ sind. Die Vereinigung dieser Elemente $\Theta = \Theta_+ \cup \Theta_-$ wird nachfolgend auch als Nachbarschaft von p bezeichnet. D_j ist bezüglich p konsistent verfeinert wenn das p-verfeinernde Diagramm $D_k = (V_k, E_k)$ mit $k = j + 1$ die folgenden beiden Bedingungen erfüllt:

1) Alle benachbarten Elemente von p müssen auch im verfeinernden Diagramm enthalten sein: $\Theta \subseteq E_k$.

2) Sei $\lambda : E \longrightarrow 2^E$ die Leveling-Funktion. Diese ist durch das (hier nicht modellierte) Data Dictionary sowie die oben dargestellte Semantik definiert. Weiter bezeichnen $\lambda(\Delta_+), \lambda(\Delta_-) \subseteq 2^E$ die Mengen aller (möglichen) verfeinerten Datenflüsse, die p als Ziel oder Quelle haben und mittels λ ableitbar sind. Um Wiederholungen zu vermeiden wird im folgenden Δ_k als Platzhalter für die eigentlich zu betrachtenden

$\Delta_{k,+}$ und $\Delta_{k,-}$ verwendet. Für die tatsächlich vorhandenen Datenflüsse $\Delta_k \subseteq 2^{E_k}$, die an p in D_k angrenzen, muss dann gelten:

$$\forall \delta \in \Delta_k.\ \exists \delta' \in \Delta.\ \delta \in \lambda(\delta')$$

Das heisst alle angrenzenden Datenflüsse im verfeinernden Diagramm D_k müssen aus einem Datenfluss in D_j ableitbar sein. Ein δ' kann mehreren δ zugeordnet werden. Weiter muss gelten:

$$\forall \delta \in \Delta.\ \delta' \in \lambda(\delta)\ \rightarrow \delta' \in \Delta_k$$

Das heisst alle verfeinernden Datenflüsse eines in D_j existierenden Datenflusses müssen auch in D_k existieren.

Zusammenfassend muss die Leveling-Funktion λ also *rechtstotal*, *linkstotal* und *rechtseindeutig*, aber i.A. nicht *linkseindeutig* sein. Jeder verfeinernde Datenfluss muss zu genau einem verfeinerten Datenfluss gehören und jeder Datenfluss muss konsistent verfeinert werden.

In den folgenden Überlegungen kann die erste Bedingung ignoriert werden, da der Editor es nicht erlaubt, dass benachbarte Elemente in einem verfeinernden Diagramm nicht vorhanden sind. Auf technischer Ebene sind diese nämlich nicht Teil des Diagramms sondern werden lediglich mittels eines *semantic candidates*-Ausdrucks dargestellt.

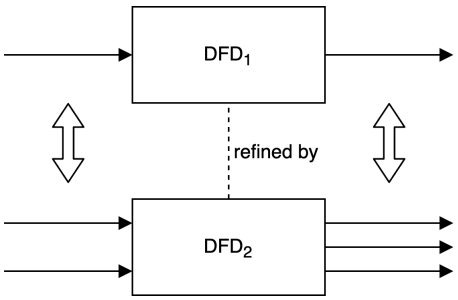

Abbildung 7: Das Ziel der Validierung von hierarchisierten Datenflussdiagrammen ist es festzustellen ob ein- und ausgehende Datenflüsse der beiden Datenflussdiagramme äquivalent, d.h. konsistent ableitbar sind.

Der Validierungs-Algorithmus Der im Rahmen des Editors entworfene und implementierte Algorithmus folgt dem *Generate and Test*-Paradigma. Er basiert auf dem rekursiven Verfeinern der zu verifizierenden Datenflüsse. Für jeden eingehenden/ausgehenden Datenfluss eines verfeinerten Prozess wird ein Edge-Refinement-Objekt erzeugt. Dieses enthält eine Zuordnung eines Datenflusses im verfeinerten (*refined*) DFD und einer Liste von Datenflüssen im verfeinernden (*refining*) DFD, die zusammen äquivalent zu diesem sein müssen. Dies ist bei Nutzung der Editor-gestützten Hierarchisierung immer der Fall, die Möglichkeit des Nutzers Verfeinerungen entweder komplett manuell zu erstellen oder die automatischen Ergebnisse zu modifizieren erfordert aber eine Konsistenzprüfung. Diese

funktioniert wie folgt: Initial wird der zu prüfende Datenfluss einmalig verfeinert und für die entstandenen Kandidaten jeweils geprüft ob sie äquivalent zu den tatsächlich existierenden Datenflüssen sind. Ist dies nicht der Fall beginnt die Rekursion. In jedem Schritt werden aus einem Kandidaten jeweils alle Kombination erzeugt, die entstehen wenn jeweils ein beliebiges Element dieses Kandidaten einmalig verfeinert wird. Wie in Abb. 8 dargestellt wird also ein hierarchischer Baum aufgebaut. Bevor nun für die neu entstandene Ebene geprüft wird ob es einen korrekten Kandidaten (d.h. einen, der äquivalent zu den tatsächlich existierenden ist) gibt, werden alle Duplikate entfernt. Ein Pfad in diesem Baum kann somit als eine Sequenz von Verfeinerungen (d.h. Klicks des Users) gesehen werden. Jeder erzeugte Kandidat ist also konsistent und alle konsistenten Kandidaten werden erzeugt.

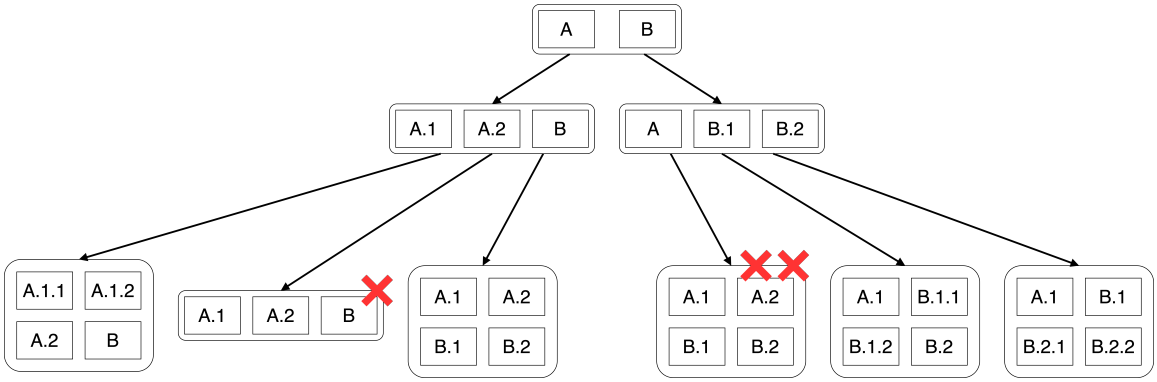

Abbildung 8: Der Validierungs-Algorithmus generiert schrittweise Kandidaten und baut somit implizit einen Baum auf. An diesem kann *pruning* durchgeführt werden falls Duplikate entstehen.

4 Zusammenfassung und zukünftige Arbeiten

Im Rahmen dieses Praktikums wurde ein Editor für Datenflussdiagramme entworfen, der die programmatische Hierarchisierung und Validierung manuell hierarchisierter Diagramme unterstützt. Implementiert wurde der Editor mittels des Eclipse Sirius-Framework verwendet, ein Meta-Modell-basierter Ansatz für die Umsetzung von Editoren.

Auf konzeptioneller Ebene wurde zunächst die Semantik von Datenflussdiagrammen sowie deren Hierarchisierung präzise definiert und entsprechende Konsistenzbedingungen sowie Meta-Modelle ausgearbeitet. Für die Validierung wurde ein *generate-and-test*-Algorithmus entworfen. Dieser kann in Zukunft erweitert und modifiziert werden, beispielsweise durch Vorberechnungen und Heuristiken, die die Zahl der explizit konstruierten Kandidaten einschränken. Eventuell kann dies auch durch speziell entworfene Datenstrukturen oder weitere theoretische Überlegungen geschehen.

Der Editor kann für speziellere Anwendungsfälle wie Sicherheitsanalysen oder Requirements Engineering sowohl bezüglich des Interfaces als auch auf Meta-Modell-Ebene erweitert werden. Ebenso denkbar sind Features wie das automatische Wiederherstel-

len von Konsistenz im Fehlerfall oder das Definieren von vorgefertigten Diagrammen als wiederverwendbare *Templates*.

Literatur

[1] Tom DeMarco. *Structured Analysis and System Specification*. USA: Prentice Hall PTR, 1979. ISBN: 0138543801.

[2] Eclipse Foundation. *Sirius*. `https://www.eclipse.org/sirius/overview.html`. [Online; letzter Zugriff 17.03.2020]. 2020.