Freie Universität Berlin
Fachbereich für Mathematik und Informatik
Institut für Informatik

Diplomarbeit

Software Factories for Embedded Systems

Design and Implementation of a Visual Domain Specific Language for Wireless Sensor Networks

Bearbeitet von:
Benjamin Schröter

Betreut von:
Prof. Dr.-Ing. Jochen Schiller
Dipl.-Inform. Tomasz Naumowicz

Bearbeitungszeitraum:
27. März – 29. September 2008

b.schroeter@taskon.biz http://flow.irgendwie.net

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 29. September 2008

Benjamin Schröter

panta rhei

Alles fließt.

Zusammenfassung

Wireless Sensor Networks *(WSN)* werden immer öfter neben der Forschung in der Informatik auch in anderen Gebieten verwendet. Dies führt dazu, dass auch fachfremde Anwender Sensornetzwerke als Werkzeuge einsetzen. Um Software für Sensorknoten programmieren zu können, ist aber meist Spezialwissen notwendig, da diese eingebetteten Systeme sehr hardwarenah in C programmiert werden. Der Austausch mit diesen Anwendern hat die grundlegenden Anforderungen bestimmt, um eine modellgetriebene Entwicklungsumgebung *(eine Software Factory)* zu entwerfen, die es auch nicht-Informatikern ermöglicht Software für Sensornetzwerke zu entwickeln.

Im Rahmen dieser Arbeit wurden drei domainspezifische Sprachen *(DSLs)* entworfen und implementiert, um alle Aspekte einer Sensorknotenanwendung zu spezifizieren. Diese Sprachen bilden mit weiteren Werkzeugen die modellgetriebene Software Factory ***Flow*** und erlauben es einem Anwender Programme für Sensorknoten graphisch zu erzeugen und daraus Quellcode zu generieren. Dieser Code ist – ohne weitere manuelle Schritte – auf den Sensorknoten einer gegebenen Plattform lauffähig. Die Sprachen und Werkzeuge sind unabhängig von der eingesetzten Sensorknotenplattform, so dass es für Hardwarehersteller möglich ist, *Flow*-kompatible Plattformen für unterschiedliche Hardware bereitzustellen.

Des Weiteren kann mit diesen Sprachen eine Schnittstelle zu einem *(eingebetteten)* PC definiert werden. Aus dieser Schnittstellenbeschreibung wird neben entsprechendem Code für die Sensorknoten eine Bibliothek generiert, die einen einfachen Zugriff auf das Sensornetzwerk erlaubt, ohne dass Kommunikations- und Protokolldetails bekannt sein müssen.

Als Referenzimplementierung wurde eine *Flow*-kompatible Plattform für die ScatterWeb Sensorknoten vom Typ MSB-430H erstellt. Auf dieser Basis sind Beispielprogramme angegeben und eine umfangreichere Fallstudie, die ein reales Einsatzszenario von Sensornetzwerken nachbildet, implementiert worden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Wireless Sensor Networks *(WSN)* sind schon seit einigen Jahren Gegenstand der Forschung in der Informatik und haben sich zu einer eigenen Forschungsdisziplin entwickelt. War anfangs die Hauptaufgabe, eine geeignete Hardwareplattform zu schaffen, wurde später immer mehr Energie in die Entwicklung von Software investiert, so dass beispielsweise eigene Betriebssysteme für Sensorknoten entwickelt wurden. Nachdem auf diese Art und Weise eine Basis geschaffen wurde, begann man, die Sensorknoten und Sensornetzwerke so zu optimieren, dass sie auch in praktischen Szenarien eingesetzt werden können. Besonders an der Optimierung des Stromverbrauchs und an einer möglichst guten Funkanbindung der einzelnen Sensorknoten sowie der Selbstoptimierung des Netzwerkes wurde viel gearbeitet. Innerhalb kurzer Zeit sind beispielsweise eine große Anzahl von verschiedenen Routingalgorithmen speziell für Sensornetzwerke entwickelt worden [IGE00; RM99; HKB99].

Um eine der wichtigsten Eigenschaften der Sensornetze, eine geringe Energieaufnahme und somit eine lange Lebenszeit, sicherzustellen, werden meist Mikrocontroller mit geringer Leistungsfähigkeit eingesetzt. Diese Mikrocontroller liefern zwar genug Leistung, um die Aufgaben eines einzelnen Sensorknotens erfüllen zu können, haben dabei aber kaum Reserven, um komplexe Laufzeitumgebungen auszuführen. Die Einbußen an Leistungsfähigkeit, die man in Kauf nehmen müsste, wenn man z.B. eine virtuelle Maschine wie Java verwenden würde, sind für die meisten Anwendungsfälle zu groß [SCC+06; LC02]. Die Kosten *(gemessen z.B. in der Energieaufnahme)*, die entstehen würden, wenn man einen leistungsfähigeren Prozessor einsetzt, sind für Sensorknoten ebenfalls in vielen Fällen zu hoch. Die Konsequenz daraus ist, dass sie meist ohne *(umfangreiches)* Betriebssystem auskommen und in C programmiert werden müssen.

Aufgrund der Möglichkeiten, die Sensornetzwerke bieten, setzen Forscher sie auch vermehrt in anderen Umfelder neben der Informatik ein. Beispielsweise nutzen Biologen Sensornetzwerke zur Beobachtung von Tieren *(siehe Kapitel 1.1 sowie [ZSLM04])*. Diese fachfremden Forscher betrachten ein solches Sensornetzwerk als Werkzeug, als Mittel zum Zweck für die Erfüllung ihrer Aufgaben, und nicht mehr als eigenständigen Forschungsgegenstand. Dies führt dazu, dass sie andere Anforderungen, wie die einfache Bedienung und Anpassbarkeit an das konkrete Forschungsumfeld, an dieses Werkzeug stellen. Doch diese Anforderungen können heute nur sehr eingeschränkt erfüllt werden. Meist ist es nötig, dass den Forschern ein Programmierer zur Seite steht, der die Implementierung der Software für das WSN übernehmen kann. Ein solcher Ansatz ist heute

unumgänglich, da ein großes technisches Verständnis erforderlich ist, um erfolgreich Software für Mikrocontroller zu schreiben, das bei Anwendern nicht vorausgesetzt werden kann. Dies führt zu Schwierigkeiten, denn ein Anwender ist immer auf die Hilfe eines Programmierers angewiesen und kann auch kleinste Anpassungen nicht selbst vornehmen. Die Benutzerfreundlichkeit der Entwicklungswerkzeuge und die Anwendbarkeit für Endbenutzer werden einen entscheidenden Einfluss auf die Verbreitung von WSNs in anderen Gebieten als der Informatik haben.

Mit dieser Arbeit soll ein Weg aufgezeigt und die dazugehörige Entwicklungsumgebung implementiert werden, um einen Anwender in die Lage zu versetzen, Anwendungen für Sensornetzwerke selbst zu entwickeln. Natürlich ist es unumgänglich, dass der Anwender sich zuvor mit einigen grundlegenden Konzepten der Sensornetze vertraut macht, aber dies soll auf einer möglichst abstrakten und weniger technischen Ebene passieren. Er soll z.B. nicht mit Interrupts sondern mit Ereignissen der Sensoren arbeiten. Der Anwender soll die grundlegende „Geschäftslogik“ umsetzen bzw. anpassen können und nur in speziellen Situationen auf die Hilfe eines Programmierers angewiesen sein.

Es soll eine *Software Factory (siehe Abschnitt 2.2.2)* entwickelt werden, die es einem Anwender ermöglicht das Verhalten seiner Sensorknoten auf einem sehr hohen Abstraktionsniveau zu spezifizieren, statt es in C oder einer anderen Programmiersprache implementieren zu müssen. Dazu soll er einen graphischen Editor verwenden und einem ereignis- und datengetriebenen Paradigma folgen. Aus diesen Modellen wird dann C-Code generiert, der – ohne weitere manuelle Eingriffe – auf den Sensorknoten lauffähig ist. Die im Rahmen dieser Diplomarbeit entwickelte Software Factory wird, aufgrund des zentralen Gedankens des Datenflusses, den Namen ***Flow*** tragen.

1.1. Beispielanwendung Skomer Island

Die Notwendigkeit, eine Entwicklungsumgebung für Endanwender von Sensornetzwerken bereitzustellen, ist aus einem konkreten Anwendungsfall[1] geboren. Dabei handelt es sich um ein Kooperationsprojekt von Microsoft Research Cambridge *(MSRC)*[2], dem Zoologischen Institut der University of Oxford[3] sowie dem Institut für Informatik der Freien Universität Berlin[4]. Es werden die auf Skomer Island, einer naturgeschützten Insel an der Westküste von Wales, brütenden Schwarzschnabel-Sturmtaucher *(Manx Shearwater, Puffinus puffinus)* beobachtet. Die einzelnen Erdhöhlen, die von den Vögeln bewohnt werden, wurden mit Temperatur- und Luftfeuchtigkeitssensoren sowie Bewegungsmeldern ausgestattet. Durch zwei Bewegungsmelder *(einer im Inneren der Höhle und einer davor)* ist es möglich, zwischen dem Betreten und Verlassen der Höhle zu unterscheiden.

[1] http://research.microsoft.com/habitats/
[2] http://research.microsoft.com/aboutmsr/labs/cambridge/
[3] http://www.ox.ac.uk/
[4] http://www.inf.fu-berlin.de/

Des Weiteren sind Radio Frequency Identification *(RFID)* Reader am Höhleneingang positioniert, so dass ein RFID-Transponder am Ring der Vögel zur Identifikation der Tiere verwendet werden kann. Eine Waage, über die die Vögel die Höhle betreten müssen, misst ihr Gewicht. Die Daten, die ein einzelner Sensorknoten erfasst, werden zu einem zentralen Sensorknoten mit angeschlossenem PC und Mobilfunkverbindung gesendet. Der genaue Aufbau ist detailliert in [NFH+08] beschrieben.

Die Anforderungen der Zoologen erscheinen einfach: Es muss in festgelegten Intervallen Temperatur und Luftfeuchtigkeit gemessen, protokolliert und per Funk verschickt werden. Sobald eine Bewegung festgestellt wird, muss der RFID-Reader und die Waage für einige Sekunden mit Strom versorgt und die Daten dieser Sensoren aufgezeichnet werden. Auch wenn diese Beschreibung so einfach klingt, können intelligenten Sensorknoten Messdaten in einem Umfang erheben, der kaum denkbar war, als man die Nester der Vögel noch durch Menschen beobachtet hat. Selbst diese einfachen Aufgaben können nur mit einem Programmierer und Experten für Sensornetzwerke gemeinsam umgesetzt werden. Für einen Anwender ohne Programmierhintergrund stellt die hardwarenahe Programmierung in C eine zu große Hürde dar, obwohl er durchaus in der Lage wäre, ähnliche Aufgaben zur Datenauswertung mit Programmen wie Excel oder MatLab durchzuführen.

Der Austausch mit den Forschern dieses Projektes hat einen Großteil der Anforderungen, die an die Entwicklungsumgebung gestellt werden, bestimmt. Auch soll das Skomer Island Szenario als Beispielanwendung und Fallstudie herangezogen werden.

Im ersten Schritt soll es möglich sein, die oben beschriebene Sensorknotenanwendung graphisch zu modellieren. Dies kann anfangs durchaus durch Programmierer geschehen, die Erfahrung auf dem Gebiet der Sensorknotenprogrammierung haben. Ein Anwender ohne Erfahrung in der Programmierung soll die Diagramme zumindest verstehen und verändern können, wenn sich z.B. in der Praxis gewisse Parameter oder Algorithmen als nicht zielführend erweisen. Ein solcher Anwender kann dann mit der Zeit in die Lage versetzt werden, auch selbst entsprechende Anwendungen graphisch zu erzeugen oder die vorhandenen zu erweitern.

Ganz ohne die Programmierung von C-Code wird man allerdings auch in diesem Umfeld nicht auskommen: Für die Sensorknoten auf Skomer Island wurde eine spezielle Zusatzhardware entwickelt, die die oben beschriebenen Sensoren enthält. Es erscheint wenig sinnvoll, die Ansteuerung dieser Sensoren graphisch spezifizieren zu wollen. Dies wird weiterhin durch den Hardwarehersteller in einem Treiber geschehen, der Funktionen anbietet, die zur Modellierung der Sensorknotenanwendung verwendet werden. Durch die Kapselung der Hardwareansteuerung in einen Treiber kann die Hardwareerweiterung auch in zukünftigen Projekten ohne großen weiteren Aufwand zum Einsatz kommen, da der Rest der Anwendung graphisch modelliert werden kann, sobald ein Treiber verfügbar ist.

1.2. Verwandte Arbeiten

In mehreren anderen Arbeiten wurden z.T. ähnliche Wege aufgezeigt, um die Komplexität bei der Entwicklung von Software für Sensornetzwerke zu reduzieren.

In [TWS06a] und [TWS06b] wird die regelbasierte Middelware *FACTS* für Sensornetzwerke vorgestellt, die Nebenläufigkeit und manuelles Stackmanagement vor dem Programmierer verbirgt, indem Regeln das Verhalten des Sensorknotens beschreiben. Dazu werden die Begriffe der Fakten und Regeln eingeführt. Daten *(beispielsweise von den lokalen Sensoren oder per Funk empfangen)* werden als Fakten repräsentiert. Regeln geben, basierend auf diesen Fakten, Bedingungen an, unter welchen Umständen Aktionen ausgelöst werden. Das Faktenrepository sowie die Regeln einer Anwendung werden von einer virtuellen Maschine auf den Sensorknoten verwaltet, ausgewertet und ausgeführt.

Weitere Ansätze basieren meist für allgemeingültige Systeme auf visueller Programmierung oder für Spezialanwendungen auf domainspezifischer Modellierung. *Viptos* [CLZ06] ist eine graphische Entwicklungs- und Simulationsumgebung für Sensornetzwerke, die auf *TinyOS*[5] basiert. Dabei ist der Abstraktionslevel, auf dem die Programme gezeichnet werden, nur minimal höher, als wenn man direkt *nesC*-Code[6] für TinyOS schreiben würde.

In einer Studienarbeit [Jes05] wurde es ermöglicht, Windows-Anwendungen auf .NET-Basis, die mit Sensorknoten der *ScatterWeb*-Plattform kommunizieren, mittels *SoftWIRE*[7] in *Visual Studio* graphisch zu programmieren. Mit dieser Arbeit wird zwar das Abstraktionsniveau gegenüber der hardwarenahen Programmierung erhöht, aber es werden keine Programme erzeugt, die auf den Sensorknoten laufen. Stattdessen empfängt die erzeugte PC-Software Daten aus dem Sensornetzwerk und steuert die Sensorknoten fern.

Bereits zwei Diplomarbeiten am Institut für Informatik der Freien Universität Berlin haben sich mit der visuellen Programmierung von Sensorknoten auseinandergesetzt. Der Abstraktionsgrad beider Arbeiten ist in etwa gleich und entspricht dem einer gewöhnlichen textuellen General Purpose Language *(GPL)*, wobei auch hier der Ablauf der Programme imperativ, aber graphisch, angegeben wird. Während in [Bly06] sowohl die Entwicklungsumgebung als auch eine speziell dazu passende Firmware für die MSB-430 Sensorknoten entwickelt wurde, setzt [Pie05] auf eine virtuelle Maschine, die auf den Sensorknoten Bytecode interpretiert und ist daher unabhängiger von der zugrundeliegenden Plattform. Da erst seit kurzem Tools zur einfachen Erzeugung von modellgetriebenen Entwicklungsumgebungen und Editoren zur Verfügung stehen *(siehe Abschnitt 3.3.2)* wurde in beiden Arbeiten ein graphischer Editor sowie der Codegenerator von Hand implementiert.

Während bisher meist visuelle, aber doch universelle, Ansätze vorgestellt wurden, wird

[5] http://www.tinyos.net/
[6] nesC ist ein erweiterter C-Dialekt, der für die Bedürfnisse von TinyOS geschaffen wurde.
[7] http://www.softwire.com/

in [KSLB03] das modellgetriebene Vorgehen auch für embedded Software empfohlen und Grundlagen sowie existierende Frameworks vorgestellt. Ein solches Vorgehen wurde in einer Diplomarbeit [Hen06] umgesetzt. Auch wenn mit der dort erstellten *ScatterFactory* keine fertige Sensorknotensoftware erzeugt werden kann, so ist es möglich, durch die graphische Spezifikation der benötigten Komponenten eine angepasste Firmware für Sensorknoten zu erzeugen. Der eigentliche Anwendungscode muss allerdings weiterhin per Hand und in C programmiert werden.

In [Sad07] wird eine domainspezifische Sprache *(DSL)* zur Beschreibung von WSN-Programmen vorgestellt, die einen viel engeren Anwendungsbereich *(Domäne)* anspricht. Es wurde eine prototypische DSL entwickelt, die streamorientiert Algorithmen zur Erdbebenerkennung beschreiben kann. Durch diese sehr spezielle Sprache werden Seismologen in die Lage versetzt, die benötigten Algorithmen direkt – ohne auf die Hilfe von Programmierern angewiesen zu sein – zu spezifizieren und zu testen.

In einer weiteren Diplomarbeit [Kam08] wurden ebenfalls modellgetriebene Methoden verwendet allerdings nicht, um Software für Sensorknoten zu entwickeln, sondern um diese zu testen. Die domainspezifische Sprache von *ScatterUnit* erlaubt es, verteilte Testfälle zu spezifizieren, vorhandenen Anwendungscode entsprechend zu instrumentalisieren, die Testfälle in einem realen Sensornetzwerk auszuführen und die Ergebnisse zu visualisieren.

1.3. Überblick

Im folgenden Kapitel werden die Grundlagen der modellgetriebenen Softwareentwicklung erläutert und Fachbegriffe aus diesem Bereich eingeführt. Diese Techniken bilden die theoretische Basis für die weiteren Teile dieser Arbeit. Die technische Basis *(in Form von verwendeter Hard- und Software)* ist in Kapitel 3 dargestellt.

In Kapitel 4 werden die Anforderungen, die an eine Software Factory für Embedded Systems, gestellt werden, beschrieben. Dazu werden an dieser Stelle bereits die verschiedenen Anwender von *Flow* unterschieden und die benötigten Werkzeuge aufgeführt. Aus der Anforderungsanalyse geht hervor, dass drei domainspezifische Sprachen *(DSLs)* zu entwickeln sind, deren Design in Kapitel 5 erläutert wird. In diesem Kapitel wird auch die Semantik und das Zusammenspiel der Sprachen detailliert aufgezeigt. Kapitel 6 beschreibt das Design der weiteren Komponenten von *Flow*, die im Rahmen dieser Arbeit erstellt wurden und geht auf einige Implementierungsdetails ein.

Die Anwendung von *Flow* wird anhand von drei einfachen Beispielen und einer umfangreicheren Fallstudie *(dem Skomer Island Szenario)* in Kapitel 7 dargestellt. Hier werden Modelle vorgestellt, wie sie mit *Flow* erzeugt werden, um Sensorknotenanwendungen zu spezifizieren. Der durch *Flow* erzeugte Quellcode ist an einem Beispiel ebenfalls in diesem Kapitel beschrieben.

Die Diplomarbeit endet in Kapitel 8 mit einer Zusammenfassung sowie einem Ausblick darauf, wie sich *Flow* in Zukunft entwickeln könnte.

Die Anhänge A bis C enthalten weitere Design- und Implementierungsdetails und ergänzen die Kapitel 5 und 6. Ein Anwenderhandbuch, das die grundlegende Bedienung von *Flow* beschreibt ist in Anhang D enthalten. Anhang E gibt den Inhalt der dieser Diplomarbeit beigelegten CD an, auf der sich vor allem der unter BSD-Lizenz stehende Quellecode von *Flow* befindet sowie ein ausführbares Installationsprogramm. Auch die beschriebenen Hardwareplattformen, die Fallstudie und diese Arbeit sind auf der CD enthalten.

2. Modellgetriebene Softwareentwicklung

Nach Stahl et al. ist die modellgetriebene Softwareentwicklung wie folgt definiert:

> *„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“* [SVEH07, Kapitel 2.1]

Es werden formale Modelle gefordert, „die irgendeinen Aspekt der Software vollständig beschreiben“, automatisiert ausgewertet und weiterverarbeitet werden können. Im Gegensatz zu vielen heute gebräuchlichen Modellierungsarten und -werkzeugen soll durch die MDSD-Techniken lauffähige Software automatisch erzeugt werden. Dies kann durch die Generierung von Quellcode aus Modellen oder durch Interpretation der Modelle geschehen. In dieser Arbeit wird vor allem von der Codegenerierung ausgegangen, da dieses Verfahren hier zum Einsatz kommen wird. Diese Erzeugung zu automatisieren betrifft dabei zwei Aspekte: Zum einen soll das Modell nicht durch Programmierer in lauffähige Software umgesetzt werden – was heute viel zu oft der Fall ist –, zum anderen soll das Erzeugen wiederholt durchgeführt werden können. Es gibt zwar heute viele Werkzeuge, die initialen Code zur manuellen Weiterverarbeitung aus Modellen generieren können, aber das MDSD-Vorgehen sieht vor, am Modell statt dem Quellcode zu arbeiten und regelmäßig Code zu generieren *(in einer ähnlichen Häufigkeit, wie man die Anwendung kompiliert)*.

Durch diese Betrachtungsweisen soll ein weiteres Ziel der modellgetriebenen Softwareentwicklung erreicht werden: Modellen einen festen Platz auch bei der Implementierung von Software einzuräumen, dadurch unter anderem die Softwarequalität zu erhöhen und die Entwicklungszeit zu verkürzen *(vgl. [SVEH07, Kapitel 2.2])*.

Zwar haben Modelle seit langem, vor allem durch den Erfolg von UML, einen festen Platz insbesondere bei der Softwarearchitektur gefunden, doch werden sie meist lediglich während einer Entwurfsphase oder im Anschluss an die Implementierung zur Dokumentation erstellt. Oft ändern sich während eines Projektes die Anforderungen oder Architekturentscheidungen werden angepasst, so dass sich die konkrete Software langsam aber stetig von den Entwurfsmodellen entfernt. Dies kann meist nur verhindert werden, wenn die Modelle zeitaufwändig und manuell synchron gehalten werden. Natürlich bezweifelt niemand den Nutzen von Modellen in der Softwareentwicklung, aber sie werden auch oft als „Overhead“ angesehen. Mit der MDSD soll nun versucht werden, Modellen einen höheren Stellenwert einzuräumen und sie auf eine Ebene mit dem Quellcode des Endproduktes zu heben.

Wenn ein Architekturmodell nicht nur eine Architektur aufzeigt, sondern aus diesem Modell auch ein Architekturrahmen erzeugt werden könnte, dann wäre der Nutzen dieses Modells ungleich größer, als wenn es nur zur Dokumentation dient. Es wäre nicht mehr nötig, die entsprechenden Programmteile durch Programmierer umsetzen zu lassen, sondern dieser Schritt wäre bereits mit der Modellierung abgeschlossen. Nicht nur, dass man durch ein solches Vorgehen den reinen Programmieraufwand reduzieren würde, es ergeben sich auch weitere positive Folgen.

Es ist unwahrscheinlicher, dass Defekte im automatisch erzeugten Teil der Software entstehen. Menschen neigen dazu, besonders dann Fehler beim Programmieren zu begehen, wenn sie monotone und wenig kreative Arbeiten ausführen. Doch genau diese Aufgaben können sehr gut automatisiert werden. Diese Annahme setzt voraus, dass der Generator, der diese Teile der Software erzeugt, fehlerfrei arbeitet. Jedoch zum einen wird davon ausgegangen, dass ein solcher Generator ausgiebiger getestet werden kann, als die konkrete Implementierung, wenn sie manuell erzeugt wird. Zum anderen wäre ein Fehler im Generator einfacher zu beheben, als ein Fehler in der konkreten Implementierung. Im endgültigen Softwareprodukt wäre ein Fehler, der durch den Generator entstanden ist, an vielen Stellen zu finden und in etwa mit einem Fehler in einer Vorlage für die manuelle Implementierung zu vergleichen *(von der in vielen Fällen kopiert wurde)*. Bei der manuellen Implementierung müsste nun jede betroffene Stelle gefunden, untersucht und entschieden werden, in wie fern sie betroffen ist, bevor der Code manuell korrigiert werden kann. Dahingegen könnte dieser Code ohne Aufwand erneut automatisch generiert werden, nachdem der Codegenerator korrigiert wurde. Dadurch, dass jederzeit Code aus den Modellen erzeugt werden kann, ergeben sich weitere Vorteile: Änderungen an zugrundeliegenden Frameworks oder Bibliotheken könnten durch Änderungen am Generator genauso einfach umgesetzt werden, wie Erweiterungen und neue Features oder gar die Verwendung einer vollkommen anderen Plattform.

Darüber hinaus steigt die Qualität allein schon dadurch, dass ein Generator erstellt wird, der viele verschiedene Fälle unterstützen soll, da er alle Möglichkeiten, die modelliert werden können, abbilden muss. Dadurch wird der Entwickler des Generators gezwungen, diese Fälle zu bedenken und es geschieht wesentlich seltener, dass Sonderfälle im Produktcode nicht beachtet werden, weil z.B. durch *(falsche)* Annahmen an einer Stelle davon ausgegangen wurde, dass diese Sonderfälle dort nicht auftreten können. Wenn der Generator zuverlässig getestet wurde und in mehreren Projekten eingesetzt wird, ist von einer weitaus geringeren Defektrate auszugehen, als wenn die Software per Hand programmiert wird. Auch die Performance der erzeugten Software liegt meist auf zufriedenstellendem Niveau. Zwar ist manuell optimierte Software durchaus in vielen Fällen schneller, aber es kommt auch immer wieder vor, dass manuell geschriebene Software ineffizient arbeitet.

Durch die Modellierung entsteht weiterer Nutzen, der bereits durch die Verwendung von UML als Architektursprache versprochen wurde. Dadurch, dass Modelle auf einer

höheren Abstraktionsebene angesiedelt sind, sind sie auch von Domänenexperten *(z.B. dem Kunden)*, wenn die richtige Abstraktion gefunden wurde, besser zu verstehen und können auf dieser Ebene diskutiert werden. Da nun weniger Code implementiert werden muss und mehr auf dieser hohen Ebene modelliert werden kann, begibt sich auch der Softwareentwickler näher zur Problemstellung und kann diese effizienter lösen. Im Gegensatz dazu herrscht heute eine „rätselhafte Situation [die] der Grund dafür [ist], warum wir derart grobkörnige Konzepte wie Kreditautorisierung mit derart feinkörnigen Konzepten wie Strings und Ganzzahlen ausdrücken“ [GS06, S. 117].

Außerdem können nun die Softwareentwickler ihre begrenzte und wertvolle Zeit vermehrt auf den kreativen Prozess lenken, statt Arbeiten zu erledigen, die automatisiert werden können.

Es stellt sich die Frage, in wie fern ein solches Vorgehen realistisch ist. Die Modellierungssprache und der Codegenerator müssen schließlich alle aktuellen und zukünftigen Eventualitäten eines Softwareprojektes erfüllen können. Dies scheint vor allem ein Risiko darzustellen, da der Softwareentwickler scheinbar Kompetenz an die Modellierungssprache und den Codegenerator abgibt und sich so zu einem gewissen Teil von ihnen abhängig macht. Dies gilt insbesondere, wenn der Codegenerator nicht selbst entwickelt, sondern als Produkt zugekauft wurde. Eine manuell implementierte Architektur scheint zumindest flexibler zu sein, da die Illusion herrscht, man könne jede Anforderung *irgendwie* einbauen. In der Praxis lassen sich aber durchaus Projekte auf Basis von MDSD entsprechend flexibel oder gar flexibler als klassische Projekte durchführen. Ein bewährtes Mittel ist es, den Codegenerator Code erzeugen zu lassen, der sich an vielen Stellen durch Erweiterungspunkte *(Extension Points)* um zusätzlichen *(dann manuell geschriebenen)* Code erweitern lässt. Wie solche Erweiterungspunkte in der Praxis aussehen, hängt von der verwendeten Plattform ab und könnte z.B. durch Vererbung von Basisklassen, partiellen Klassen und Methoden oder einer Plugin-Architektur umgesetzt werden.

Eine weitere Möglichkeit besteht darin, neue benötigte Funktionen durch Anpassung und Erweiterung der Modellierungssprache und des Codegenerators hinzuzufügen. Dies mag zwar gegenüber der manuellen Entwicklung ein Mehraufwand sein, bringt aber wiederum enorme Vorteile mit sich. Wenn eine Modellierungssprache samt ihrem Codegenerator in mehreren Projekten eingesetzt wird, dann profitieren diese Projekte alle von Anforderungen, die lediglich für ein Projekt umgesetzt wurden. In Zukunft können auch die übrigen Projekte mit minimalem Mehraufwand diese neuen Funktionen durch entsprechende Modellierung nutzen.

2.1. Definitionen

Die Verwendung eines neuen Paradigmas in der Softwareentwicklung bringt es zwangsläufig mit sich, dass einige Begriffe, die unter Umständen bereits intuitiv klar sind, definiert werden sollten. Die Definitionen in diesem Abschnitt stellen die Sichtweise aus [SVEH07, Kaptiel 3.1] dar.

2.1.1. Domäne / Domain

Die meisten Projekte in der Softwareentwicklung und insbesondere alle Projekte, die mit den Techniken von MDSD umgesetzt werden, finden in einem Kontext bzw. im Rahmen einer Domäne oder einem Fachbereich statt. Die Domäne ist insbesondere eine wichtige Basis für den Kontext und die Darstellungskraft der Modelle.

Oft wird zwischen einer *horizontalen* und *vertikalen* Domäne unterschieden. Horizontale oder auch *architekturzentrierte* Domänen sind auf einem technischen Niveau angesiedelt. Beispiele sind Benutzeroberflächen *(GUI)*, Datenbanken, Deployment usw. Dahingegen beschreiben vertikale Domänen eher Bereiche, in denen sich der klassische Endkunde einer Software bewegt, wie z.B. Versicherungswesen, Bankenprodukte usw.

Derzeit finden die meisten Arbeiten im Bereich von MDSD in horizontalen Domänen statt. Der Grund dafür scheint zu sein, dass die ersten Anwender dieser Technologie Softwareentwickler mit diesem architekturzentrischen Fokus sind. Doch es wird erwartet, dass mit dem Reifen von MDSD diese Techniken auch vermehrt in vertikale Domänen vordringen werden.

In der deutschen Fachliteratur hat sich der Begriff „Domäne“ eingebürgert. Bei zusammengesetzten Wörtern wie „domainspezifisch“ wird aber auch oft die englische Schreibweise verwendet.

2.1.2. Domain Specific Language (DSL)

Eine domainspezifische Sprache *(Domain Specific Language, DSL)* bezeichnet eine Sprache, die speziell für die Anforderungen einer Domäne entwickelt oder angepasst wurde. Diese Sprachen haben den Vorteil, dass sie von Domänenexperten meist relativ leicht erlernt werden können und keinen überflüssigen Ballast mit sich tragen müssen. DSLs können sowohl textuell als auch graphisch sein. Im Umfeld der MDSD verwendet man häufig graphische DSLs als Modelle, aber es besteht explizit die Möglichkeit, auch textuelle Sprachen zu nutzen.

Als Beispiele für *(textuelle)* DSLs können Reguläre Ausdrücke und SQL angeführt werden, die jeweils nur einen sehr begrenzten Fokus haben *(Stringmatching bzw. Datenbankanfragen)* und außerhalb dieser Aufgaben nicht einzusetzen sind. Aber gerade diese Spezialisierung hat sie zu Standards in der Informatik werden lassen.

2.1.3. Metamodell

Ein Metamodell spezifiziert die Struktur eines Modells *(beispielsweise das Modell einer DSL, die im Rahmen von MDSD verwendet wird)* formal. Es besteht aus der *abstrakten Syntax* sowie der *statischen Semantik* der Sprache und bestimmt, ähnlich wie die Grammatik einer textuellen Sprache, welche Konstrukte innerhalb der Sprache gültig sind. Es dient als Definition, kann aber auch als Dokumentation der Modellierungssprache verwendet werden.

Es existieren seit kurzem Tools und Frameworks, die die Methoden der MDSD selbst auf Metamodelle anwenden. Mit diesen Tools ist es möglich, ein Metamodell nicht nur zur Dokumentation zu erstellen, sondern aus dem Metamodell automatisiert beispielsweise einen graphischen Editor für die modellierte Sprache zu erzeugen. Als Beispiele sollen das Graphical Modeling Framework[1] *(GMF)* und die Microsoft DSL Tools *(siehe Abschnitt 3.3.2)* genannt werden. Beide Tools bieten in etwa den gleichen Umfang an, um aus Metamodellen graphische Editoren für DSLs zu erzeugen. Während GMF auf Java und die Eclipse Plattform aufsetzt *(die erzeugten Editoren und Tools sind Erweiterungen für die Eclipse IDE)*, verwenden die Microsoft DSL Tools als Plattform und Laufzeitumgebung Microsoft Visual Studio *(siehe Kapitel 3.3)*.

2.1.4. Abstrakte und konkrete Syntax

Die abstrakte Syntax einer Sprache beschreibt, wie die einzelnen Elemente des Metamodells einer Sprache zueinander in Beziehung gestellt werden können. Hier wird beispielsweise beschrieben, dass ein Element vom Typ A Elemente vom Typ B enthalten kann, nicht aber vom Typ C. Außerdem können Typ A Elemente Beziehungen *(einer gewissen Art)* mit Elementen vom Typ D eingehen.

Die konkrete Syntax beschreibt hingegen, wie diese Elemente und Beziehungen dargestellt werden. Sie gibt konkret an, dass Elemente eines Typs als Kreise und andere als Rechtecke mit gewissen Eigenschaften dargestellt werden und dass Beziehungen als gestrichelte, nicht als durchgezogene Linie repräsentiert werden.

Für eine textuelle Sprache wie C# gibt die abstrakte Syntax an, dass es Klassen gibt, die Methoden und Properties enthalten. Wohingegen die konkrete Syntax beschreibt, dass Klassen mit dem Schlüsselwort `class` eingeleitet und ihre Methoden in einem Block aus geschweiften Klammern eingeschlossen werden.

2.1.5. Statische Semantik

Mittels der statischen Semantik ist definiert, wie ein inhaltlich wohlgeformtes Modell einer gewissen Sprache aussieht. Sie kann Bedingungen und Einschränkungen *(Constraints)* festlegen und dadurch den Anwender bei der Modellierung unterstützen.

[1] http://www.eclipse.org/modeling/gmf/

Die statische Semantik zu der Sprache aus dem vorhergehenden Abschnitt könnte beispielsweise festlegen, dass jedes Element vom Typ A mindestens ein, aber höchstens fünf Typ B Elemente enthalten darf. Bei statisch getypten textuellen Programmiersprachen wird die statische Semantik vom Compiler überprüft und Verstöße gegen sie als Fehlermeldungen angezeigt.

2.1.6. Metametamodell

Das Metametamodell stellt das Metamodell des Metamodells einer DSL dar und kommt immer dann zum Einsatz, wenn das Metamodell einer DSL ebenfalls mit Techniken der MDSD erstellt werden soll. Theoretisch kann es weitere Meta-Ebenen geben, aber oft wird das Metametamodell durch sich selbst beschrieben. Dieses sogenannte Bootstrapping kommt sowohl beim Graphical Modeling Framework *(GMF)* für Eclipse als auch bei den DSL Tools von Microsoft zum Einsatz. Bemerkenswert ist, dass diese Tools selbst eindrucksvoll belegen, welche Möglichkeiten durch sie bestehen, da sie auf denselben Grundlagen basieren, die auch einem Anwender dieser Tools zur Verfügung stehen.

Interessanterweise ähneln sich die Metametamodelle der beiden hier betrachteten Modellierungsplattformen sehr. In [BHJ+05] wurden die jeweiligen Metametamodelle miteinander verglichen und die Autoren kamen zu dem Schluss, dass die Modelle ähnlich genug sind, um einen Adapter von GMF- zu Microsoft DSL-Modellen erstellen zu können. In Zukunft könnte dies eine Interoperabilität zwischen den Eclipse und den Microsoft Tools ermöglichen.

2.1.7. Plattform

Beim modellgetriebenen Vorgehen wird gefordert, dass die Modelle mächtig genug sind, um daraus den Anwendungscode zu generieren. Dieser Code setzt wiederum auf Bibliotheken und Frameworks auf, da es sinnvoll ist, die bewährte Funktionalität der Bibliotheken und Frameworks auch beim modellgetriebenen Vorgehen zu nutzen. Die Menge dieser Voraussetzungen, auf die der generierte Code aufsetzt, wird als Plattform bezeichnet.

Es ist durchaus denkbar und auch erwünscht, aus einem Modell Code für verschiedene Plattformen zu erzeugen, indem verschiedene Codegeneratoren eingesetzt werden. So können kleine Unterschiede in der Plattform adressiert werden, indem z.B. Code für zwei verschiedene J2EE-Application-Server generiert wird. Es wäre auch denkbar, Code sowohl für einen in C zu programmierenden Mikrocontroller, als auch für ein Embedded Device, auf dem das .NET Micro Framework läuft, zu erzeugen. Plant man eine solche Austauschbarkeit der Plattform, muss allerdings bereits bei der Erstellung des Metamodells beachtet werden, in ihm keine impliziten oder expliziten Annahmen zu machen, die nur eine Plattform wiederspiegeln.

2.2. Verwandte Ansätze

2.2.1. Model Driven Architecture

Model Driven Architecture[2] *(MDA)* wird oft synonym für MDSD verwendet, was aber nicht korrekt ist. Bei dem Begriff „Model Driven Architecture“ handelt es sich um einen markenrechtlich geschützten Begriff der OMG[3] *(Object Management Group)*, der Organisation, die unter anderem auch den UML-Standard entworfen hat. Die Beziehung von MDSD zu MDA lässt sich mit einem Zitat aus [Eva03] beschreiben, in dem MDSD als „MDA für Praktiker“ dargestellt wird.

Auch wenn sich MDA den gleichen oder ähnlichen Zielen verschrieben hat, wie das oben beschriebene MDSD, gibt es gewisse Unterschiede *(vgl. [SVEH07, Kapitel 3.2.1])*. Die Kernunterschiede sind, dass die OMG versucht, viele Aspekte viel stärker zu kontrollieren, als die Ideen der MDSD es vorsehen. Dies ist vor allem aus der Notwendigkeit entstanden, einen Standard „MDA“ zu erarbeiten. Als Modellierungssprache soll UML 2.0 zum Einsatz kommen. Auch wenn man UML mit Profilen und Stereotypen an seine Domäne anpassen kann, geben Verfechter einer freieren Auslegung wie MDSD zu bedenken, dass eine Sprache mit einer beliebigen Darstellung für viele Domänen besser geeignet sei. Dave Thomas geht so weit und fragt in [Tho04] ob „MDA die Rache der Modellierer“ dafür sei, dass sich die bisherigen CASE-Ansätze nicht durchsetzen konnten.

Die OMG definiert durch MDA nicht nur UML als Modellierungssprache, sondern auch OCL *(Object Constraint Language)* zur Spezifikation der statischen Semantik sowie weitere Standards zur Definition von plattformspezifischen und -unspezifischen Modellen *(PSM und PIM)*.

In dieser Arbeit werden zwar viele Ideen und Konzepte verwendet, die sowohl in MDSD als auch MDA beheimatet sind, aber es werden nicht die konkreten Spezifikationen der MDA verwendet, da unter anderem die verwendeten Werkzeuge MDA nicht direkt unterstützen.

2.2.2. Software Factories

Im Zusammenhang mit MDSD und MDA trifft man auch auf den Begriff der Software Factory *(vgl. [GS06, Kapitel 17.1.2])*. Software Factories verwenden die gleichen Ansätze, gehen aber in einigen Bereichen ein wenig weiter. So versteht man unter einer Software Factory im allgemeinen nicht nur eine Modellierungssprache sowie einen Codegenerator, sondern eine komplette Entwicklungsumgebung zur Entwicklung ähnlicher Softwareprodukte.

Bei Software Factories steht nicht ein Produkt, sondern eine *Produktlinie* von ähnlichen Produkten im Mittelpunkt. Das Konzept der Produktlinie baut auf der Beobachtung auf, dass häufig viele ähnliche Softwareprodukte in einer Domäne entwickelt werden, die sich

[2] http://www.omg.org/mda/
[3] http://www.omg.org/

nur leicht oder nur in gewissen Aspekten unterscheiden. Produktlinien kommen auch bei MDSD zum Einsatz, sind aber innerhalb von MDA unbekannt. Man kann sich z.B. eine gewisse Art von Software für Sensornetzwerke vorstellen, die zwar viel Funktionalität – nach klassischen Entwicklungsmethoden auch Code – teilen, aber unterschiedliche Aufgaben erfüllen sollen. Eine Software Factory ist für diese Art von Anwendungen gebaut und nimmt die Spezifikation auf, die die einzelnen Anwendungen unterscheiden. Daraus kann dann entweder die gesamte Anwendung oder ein großer Teil von ihr erzeugt werden.

Als Analogie zu Software Factories wird gerne die Industrialisierung zu Anfang des 19. Jahrhunderts herangezogen. Zu dieser Zeit wurde begonnen, Produkte nicht mehr einzeln von Handwerkern herstellen zu lassen, sondern Fabriken zu bauen, in denen das gleiche oder ähnliche Produkte mehr oder weniger automatisiert hergestellt wurden. Bis heute hat sich dies so weit entwickelt, dass beispielsweise Autos hoch automatisiert in vielen Konfigurationsvarianten auf Kundenwunsch produziert werden können.

Vergleicht man dies mit dem Stand der Softwareindustrie, kann man behaupten, dass Software immer noch manuell von „Handwerkern“ hergestellt wird. Zwar werden zunehmend vorgefertigte Komponenten eingesetzt, aber ein Großteil der Arbeit muss weiterhin und wiederholt per Hand ausgeführt werden. Software Factories sollen einen Weg aufweisen, diese Handarbeit zumindest für Produktlinien zu reduzieren.

Um dieses Ziel zu erreichen, setzen Software Factories auf die Konzepte des oben beschriebenen MDSD auf und bieten eine Entwicklungsumgebung an, mit der möglichst effizient Softwareprodukte der entsprechenden Produktlinie erstellt werden können.

Greenfield und Short weisen im Zusammenhang mit dieser Analogie darauf hin, dass sie keineswegs der Meinung sind, Softwareentwicklung könnte ohne weiteres so automatisiert werden, wie es heute in der produzierenden Industrie der Fall ist. Sie gehen auch nicht davon aus Softwareentwickler und Programmierer arbeitslos zu machen aber sie möchten ihnen andere, nämlich kreativere Aufgaben zuteilen, um die Produktivität zu steigern:

> *„[Der Schlüssel] zur Befriedigung der Nachfrage im industriellen Maßstab besteht darin, aufzuhören, die Talente von Fachentwicklern mit mechanischen und niedrigen Aufgaben zu vergeuden, so dass sie mehr Zeit mit Denken und weniger Zeit mit lästigen Kleinarbeiten verbringen können. Außerdem müssen wir diese wenigen, aber wertvollen Ressourcen besser nutzen, als sie für die Konstruktion von Endprodukten zu verschwenden, die bereits wieder angepasst oder gar ersetzt werden müssen, wenn das nächste Hauptrelease der jeweiligen Plattform veröffentlicht wird oder wenn sich die Geschäftsanforderungen aufgrund der Marktlage ändern.“* [GS06, S. 26]

2.2.3. Visuelle Programmierung

Während die oben beschriebenen Modellierungsmethoden die Modelle in den Mittelpunkt der Betrachtung rücken und erst in einem weiteren – durchaus nicht trivialen – Schritt Quellcode und dadurch lauffähige Software erzeugen, ist der Fokus bei visuellen Programmiersprachen meist ein anderer. Es wird versucht die Konstrukte einer textuellen Programmiersprache graphisch darzustellen. Schiffer schlägt in seinem Buch „Visuelle Programmierung“ [Sch97] die folgenden Definitionen für die Begriff „Visuelle Sprache“ und „Visuelle Programmiersprache“ vor:

> *„Eine visuelle Sprache ist eine formale Sprache mit visueller Syntax oder visueller Semantik und dynamischer oder statischer Zeichengebung.“*
>
> *„Eine visuelle Programmiersprache ist eine visuelle Sprache zur vollständigen Beschreibung der Eigenschaften von Software. Sie ist entweder eine Universalprogrammiersprache oder eine Spezialprogrammiersprache.“*

Auch wenn Schiffer zwischen Universalprogrammiersprachen *(d.h. Sprachen die Turingvollständig sind)* und Spezialprogrammiersprachen *(die einer DSL gleichen)* unterscheidet, wird in den meisten Fällen bei der visuellen Programmierung doch mit Universalprogrammiersprachen gearbeitet. Wenn Spezialprogrammiersprachen verwendet werden, erreichen diese meist nur ein geringfügig höheres Abstraktionsniveau als gewöhnliche General Purpose Languages. In den folgenden Betrachtungen in diesem Abschnitt wird dieser, aus der Praxis stammende, Fokus angewandt, wenn der Begriff „visuelle Programmiersprache“ verwendet wird. Auch die meisten Beispiele, die Schiffer anbringt sowie die später erwähnten Studien und Erfahrungsberichte, stützen sich auf solche visuellen Universalprogrammiersprachen.

Die meisten visuellen Universalprogrammiersprachen basieren auf einem von nur wenigen Konzepten zur visuellen Darstellung von Programmen. Diese Konzepte sollen hier nicht detailliert beschrieben, aber zumindest genannt werden:

- Programmablaufplan
- Nassi-Shneiderman-Diagramme
- Datenflussdiagramme
- Zustandsautomaten

Viele dieser Diagrammtypen arbeiten auf einem ähnlichen Abstraktionsniveau wie andere Programmiersprachen *(z.B. C oder Java)*, obwohl es durchaus denkbar ist, auch solche Diagramme auf einem höheren Niveau anzusiedeln, so wie beispielsweise *Flow* Datenflussdiagramme nutzt. Aber in vielen visuellen Programmiersystemen ist dies nur selten der Fall.

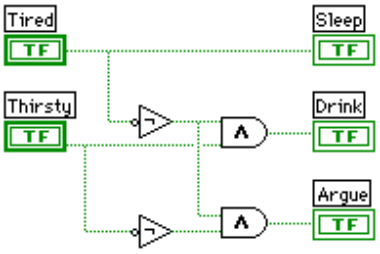


Abbildung 2.1.: Zwei äquivalente logische Ausdrücke als LabVIEW-Programm und Pseudocode [Sch97, Bild 3-9]

In den 80er und 90er Jahren des vergangenen Jahrhunderts wurde besonders intensiv an visuellen Programmiersprachen gearbeitet und geforscht. Es wurde sehr viel Hoffnung mit dieser Art des Programmierens verbunden und man hatte große Erwartungen. Schiffer zitiert eine Studie [Sne95] aus dem Jahr 1995 mit einer Prognose für das Jahr 1999, nach welcher der Marktanteil der visuellen Programmierung 3,8 Milliarden US-Dollar betragen sollte. Allerdings wird der Begriff der visuellen Programmierung hier sehr frei aufgefasst und unter anderem Visual Basic und ähnliche Werkzeuge mit betrachtet. Dies ist allerdings weder nach heutiger Ansicht, noch nach den oben angegebenen Definitionen der Fall, vielmehr war Visual Basic eine der ersten Programmiersprachen bzw. eher Entwicklungsumgebungen, bei denen Benutzeroberflächen mit einem graphischen Editor erzeugt werden konnten. Die Programmierung der Anwendungslogik erfolgte aber weiterhin textbasiert in dem Basic Dialekt „Visual Basic“. Nach dieser Auffassung wäre beinahe jede Entwicklungsumgebung, die heute verwendet wird, visuell.

Allerdings haben sich die visuellen Programmiersprachen der angedachten Form *(als graphische Universalprogrammiersprache)* bis heute nicht etablieren können. Viele Skeptiker geben damals wie heute zu bedenken, dass komplexe Probleme nicht durch Visualisierung einfacher gelöst werden können. Besonders für große Programme wurde kein Weg gefunden, die entsprechenden Graphiken übersichtlich zu gestallten, um das Skalierungsproblem zu lösen. Die Behauptung von Hirakawa und Ichikawa in [HI94], dass bei der Benutzung visueller Ausdrücke keine Notwendigkeit mehr besteht, computerspezifische Konzepte zu erlernen, würde heute kaum noch jemand unterstützen.

Des Weiteren scheint es, dass professionelle Programmierer in den letzten Jahrzehnten sehr erfolgreich gelernt haben, ihre Intentionen in textueller Form zu formulieren. Studien legen nahe, dass auch beim Lesen von Programmen die textuelle Form oft einfacher verständlich ist als Graphiken. Als Beispiel soll ein sehr bekannter Versuch [GP92] genannt werden, bei dem Programmierer mit und ohne LabVIEW[4]-Erfahrung einfache graphische LabVIEW-Programme und äquivalente textuelle Programme in Pseudocode evaluieren sollten. Als Ergebnis geben Green und Petre an, dass die textuelle Form zweifelsfrei besser lesbar ist. Abbildung 2.1 stützt diese Aussage auch intuitiv.

[4] http://www.ni.com/labview/

Ein weiteres Beispiel beschreibt Schiffer aus eigener Erfahrung. Im Software-Engineering-Praktikum des Institutes für Wirtschaftsinformatik an der Universität Linz sollten Studenten eine einfache Büroapplikation programmieren. Eine Gruppe verwendete dazu Smalltalk *(eine textuelle objektorientierte Programmiersprache)* und eine andere Gruppe die visuelle Sprache Serius:

> *„Die Serius-Gruppe konnte in kurzer Zeit und ohne viel Mühe einen Prototypen bauen, während die Smalltalk-Gruppe noch mit den Sprachkonzepten und der Programmierumgebung kämpfte. Das Blatt wendete sich jedoch zu Gunsten der zweitgenannten Gruppe, als die Studenten gelernt hatten, die Möglichkeiten von Smalltalk zu nutzen. Ab diesem Zeitpunkt war die Smalltalk-Gruppe der Serius-Gruppe weitaus überlegen. Sie konnten alle funktionalen Anforderungen erfüllen und schnell und flexibel auf Änderungswünsche reagieren. Die Serius-Gruppe hingegen kam über die erste prototypische Implementierung kaum hinaus und mußte frustriert feststellen, daß die Konzepte von Serius, die anfänglich so überzeugend und motivierend wirkten, bei weitem nicht ausreichten, um ein taugliches Programm zu schreiben.“* [Sch97, Kapitel 3.4.3]

Dieser Erfahrungsbericht, wie auch die Abbildung 2.2, zeigen anschaulich, dass der bisherige visuelle Programmierungsansatz mit Universalprogrammiersprachen nur partiell zum Ziel führen kann. Im Vergleich dazu versuchen die oben beschriebenen modellbasierten Ansätze unter anderem die Komplexität der Modelle zu verringern, indem ein höheres Abstraktionsniveau zur Modellierung verwendet wird. Auch in dieser Arbeit soll daher ein solcher Ansatz unter Verwendung von domainspezifischen Modellen umgesetzt werden.

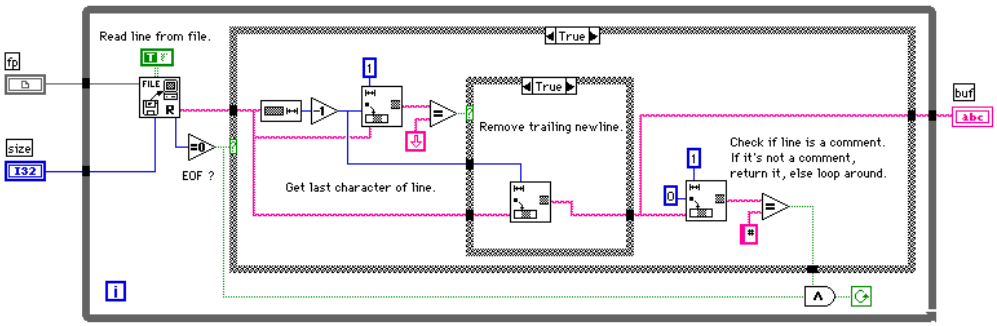


Abbildung 2.2.: Visuelles LabVIEW-Programm der Funktion `getsline` [Sch97, Bild 2-8]

3. Werkzeuge und Plattformen

Diese Arbeit baut auf einer gegebenen Hardwareplattform für Sensorknoten sowie auf einigen weiteren Programmen und Softwareumgebungen auf, die in diesem Kapitel kurz vorgestellt werden.

3.1. Modular Sensor Board MSB-430H

Bei dem MSB-430H[1] *(siehe Abbildung 3.1)* handelt es sich um einen Sensorknoten, der an der Freien Universität Berlin, auf Basis des MSP430 Mikrocontrollers, entwickelt wurde. Der MSB-430H ist eine leicht modifizierte Version des MSB-430 *(vgl. [BKLS07])*, wobei das „H“ für „highspeed“ steht, da hier ein u.a. schnellerer Transceiver als beim Vorgängermodell verwendet wird. Wie der Begriff „modular“ im Namen andeutet, ist die Hardwarearchitektur für eine besonders einfache Erweiterbarkeit ausgelegt. Das Hauptmodul inklusive des Mikrocontrollers kann durch Aufsetzen von Erweiterungsmodulen um Sensoren ergänzt werden. Auch die Stromversorgung ist modular aufgebaut, so dass das Hauptmodul auf verschiedene Trägerboards mit unterschiedlichen Stromquellen und Schnittstellen zu z.B. einem PC aufgesetzt werden kann.

Abbildung 3.1.: MSB-430 mit optionalem Sensorboard MSB-430S und Trägerboard [Quelle: ScatterWeb-Homepage[1]]

[1] ScatterWeb-Homepage: http://cst.mi.fu-berlin.de/projects/ScatterWeb/hardware/msb/

Auch wenn diese Arbeit im Weiteren prinzipiell unabhängig von dem verwendeten Sensorknoten ist, so nimmt das MSB doch eine Sonderrolle ein. Durch seine modulare Architektur und den vorhandenen Sensoren und Schnittstellen eignet es sich zur allgemeinen Betrachtung von Sensorknoten. Als Referenzimplementierung *(siehe Kapitel 7.1)* werden eine Firmware sowie Treiber für diese Hardware zur Verfügung gestellt. Es wird zwar möglich sein, *Flow* auch mit anderer Hardware zu verwenden, aber nur für das MSB kann in der Bearbeitungszeit einer Diplomarbeit eine kompatible Firmware angepasst werden.

Das Herzstück des MSB-430H ist ein Mikrocontroller des Typs MSP430F1612 *(vgl. [Tex06])* von Texas Instrumentes. Er enthält 5 kByte RAM sowie 55 kByte Flashspeicher, um Programmcode nicht flüchtig abzulegen. Der Controller ist so konzipiert, dass er trotz einer Taktrate von bis zu 8 MHz mit einer minimalen Stromaufnahme auskommen kann, die sich in verschiedenen Schlafmodi bis auf wenige μA drosseln lässt. Es stehen zwei UARTS zur seriellen Kommunikation sowie zwei Hardwaretimer zur Verfügung. Des Weiteren enthält er analoge und digitale I/O-Ports, um weitere Hardware ansteuern zu können.

Das Hauptmodul MSB-430H hat an seinen Anschlüssen 16 digitale I/O Ports sowie je zwei analoge Ein- und Ausgänge *(mit 12 Bit Genauigkeit)* herausgeführt. Auch die Leitungen eines der beiden UARTS zur seriellen – mittels Adapterkabel auch USB – Kommunikation stehen über diese Pins bereit. Diese Anschlüsse können für Erweiterungen genutzt werden. Auf der Rückseite enthält das Modul einen Slot für SD-Speicherkarten und die Firmware kann ein FAT16-Dateisystem auf diesen Karten lesen und schreiben.

Neben dem Mikrocontroller ist der Chipcon Transceiver CC1100 *(vgl. [Tex08])* von Texas Instruments ein wichtiger Bestandteil des Hauptmoduls. Er stellt Funktionalität zur paketorientierten Datenübertragung mit bis zu 500 kBit/s im ISM-Band *(868 MHz)* bereit. Durch die verwendete hohe Übertragungsrate *(und dadurch kurze Sendedauer)* sowie dem Wake-on-Radio-Schlafmodus des Transceivers ist eine besonders energieeffiziente und verlässliche Kommunikation im Sensornetzwerk möglich *(vgl. [Hil07])*.

3.1.1. ScatterWeb .NET SDK

ScatterWeb-Sensorknoten und ein PC kommunizieren für gewöhnlich durch das Austauschen von Textnachrichten über die serielle Schnittstelle. Damit ein solches Protokoll nicht für jede Anwendung erneut implementiert werden muss, bietet das ScatterWeb .NET SDK[2] umfangreiche Funktionen an, um von einer PC-Anwendung *(basierend auf dem .NET-Framework)* mit Sensorknoten zu kommunizieren. Das SDK versteckt die Low-Level-Kommunikation über die Schnittstelle vor dem Anwender. Es kann selbständig den entsprechenden seriellen Port, an dem ein Gatewayknoten angeschlossen ist, finden und einen Kommunikationskanal aufbauen. Für jeden im Netzwerk gefundenen

[2] http://cst.mi.fu-berlin.de/projects/ScatterWeb/tools/scatterwebnet.htm

Sensorknoten wird eine Instanz einer Sensorknoten-Klasse erzeugt, mit der der Anwender arbeiten kann. Je nach Software auf dem Sensorknoten kann dies eine andere Klasse sein und auch der Anwender hat die Möglichkeit eigene Klassen zu ergänzen.

Diese Klassen können unabhängig davon, ob ein Knoten direkt mit dem PC verbunden oder nur über Funk vom Gatewayknoten erreichbar ist, Textnachrichten an den jeweiligen Knoten verschicken und werden über eingehende Nachrichten mittels .NET-Events informiert. Das SDK enthält des weiteren Funktionalität, um auf Antworten einer gewissen Form des Sensorknotens zu warten und solange die Programmausführung zu blockieren. Dabei werden Nachrichten an andere Knoten weiterhin verarbeitet und das Warten über einen Timeout-Mechanismus nach einiger Zeit abgebrochen.

Durch die Verwendung des ScatterWeb .NET SDKs kann ein Anwender sehr einfach die Logik einer PC-Anwendung, die mit Sensorknoten kommuniziert, implementieren ohne viel Zeit für die unteren Kommunikationsschichten aufwenden zu müssen.

3.2. Doxygen

Doxygen[3] ist ein Quellcode-Dokumentationswerkzeug, das aus entsprechend kommentiertem Quellcode Dokumentationen in verschiedenen Ausgabeformaten erzeugen kann. Doxygen ist eins der bekanntesten Produkte seiner Art und durch seinen Umfang und seine Erweiterbarkeit universell einsetzbar. Die Software ist unter der GNU General Public License *(GPL)* veröffentlicht.

Doxygen kann Quellcode vieler Sprachen, darunter C, C++, Objective-C, Java, Python, Fortran, VHDL, PHP und C#, einlesen und Dokumentationen in unterschiedlichen Formaten wie HTML, RTF, PDF und LATEX erzeugen.

Einige typische Doxygen-Kommentare zu Dokumentationszwecken sind in Abbildung 3.2 zu sehen.

Im Rahmen dieser Arbeit wird Doxygen allerdings nicht zur Dokumentation, sondern zur Annotation der Sensorknoten-Firmware verwendet. Der Quellcode einer in C geschriebenen Firmware mit speziellen Annotationen soll eingelesen und automatisiert in ein Modell überführt werden *(siehe Kapitel 6.4)*. Dazu wird die Möglichkeit genutzt, Doxygen mittels Parametern neue Annotationsformate hinzuzufügen und die Ausgabe als XML zu erstellen, die dann weiter verarbeitet werden kann.

3.3. Microsoft Visual Studio

Visual Studio *(VS)* ist die integrierte Entwicklungsumgebung *(IDE)* aus dem Hause Microsoft, aktuell in Version 2008, die für alle Programmiersprachen von Microsoft verwendet werden kann. Dies beinhaltet zum einen die Sprachen der .NET-Familie,

[3] http://www.doxygen.org/

```
/**
* A test class. A more elaborate class description.
*/
class Test
{
  public:
  /**
  * A constructor.
  * A more elaborate description of the constructor.
  */
  Test();

  /**
  * a normal member taking two arguments and returning an
      integer value
  * @param a an integer argument.
  * @param s a constant character pointer.
  * @return The test results
  */
  int testMe(int a,const char *s)
  {
    // ...
  }
}
```

Abbildung 3.2.: Kommentierter C++ Quellcode aus der Doxygen Dokumentation *(verkürzt)*

wie C# und Visual Basic.NET, aber es ist auch möglich, nativen *(x86)* C++ Code zu erstellen. Des Weiteren besteht die Möglichkeit, C bzw. C++ Code in VS zu schreiben, aber statt des Microsoft Compilers ein Makefile sowie einen externen Compiler einzusetzen. Die IDE enthält weitere Tools zur Softwareentwicklung, angefangen von graphischen GUI-Designern für Windows Forms und WPF[4] über die Anbindung an Datenbankserver, einem Editor für XML und eine Anbindung an den Team Foundation Server zur Quellcodeverwaltung und als Ticketingsystem sowie vieles mehr.

Im Rahmen dieser Arbeit ist Visual Studio besonders wegen seiner Offenheit und Erweiterungsmöglichkeiten von Interesse.

Da es verschiedene kostenlose und kommerzielle Editionen von Visual Studio gibt, die Einfluss auf die Erweiterungsmöglichkeiten haben, sollen diese Editionen kurz beschrieben werden:

Microsoft bietet Visual Studio in drei sogenannten „Express“ Versionen kostenlos an. Diese Express Versionen beherrschen jeweils nur eine Sprache und sind leicht eingeschränkt. Leider erlaubt es Microsoft nicht, diese Express Versionen zu erweitern, so dass sie für diese Arbeit keine weitere Rolle spielen.

Die drei kommerziellen Editionen mit den Namen „Standard“, „Professional“ und

[4] Windows Presentation Foundation: ein neues GUI-Framework von Microsoft als Teil des .NET-Frameworks

„Team System“ sprechen verschiedene Benutzergruppen an, was am Funktionsumfang[5] deutlich wird. Sie alle lassen sich durch eigenen Code erweitern und eignen sich so für die in dieser Arbeit beschriebenen Vorhaben.

Seit der Version 2008 existiert eine weitere kostenlose Edition mit dem Namen „Visual Studio Shell“[6]. Diese Version enthält keine eingebaute Programmiersprache, aber erlaubt es Erweiterungen zu verwenden und grenzt sich dadurch von den ebenfalls kostenlosen Express Versionen ab. Die VS Shell richtet sich an Entwickler von Visual Studio Erweiterungen, die ihre Erweiterungen auch an Endkunden ohne eine Visual Studio Lizenz verteilen möchten. Sie können dabei auf den gesamten Rahmen von Visual Studio aufsetzen *(inklusive Menü- und Symbolleisten, Undo- und Redo-Funktionalität und vielem mehr)*, aber es fehlen beispielsweise die Quellcodeeditoren. Ein Texteditor ohne besondere Sprachfeatures ist vorhanden. Eine weitere große Einschränkung ist, dass der Visual Studio Shell keine Projekttypen bekannt sind. Projekte werden in Visual Studio verwendet, um mehrere Dateien *(z.B. Quellcode und Ressourcen)* als Einheit zu betrachten. Verschiedene Projekttypen *(z.B. C#, Visual Basic aber auch Bibliothek oder Webservice)* enthalten unterschiedliche Vorlagen für neue Dateien und verhalten sich z.B. beim kompilieren unterschiedlich. Da die Visual Studio Shell ohne solche Projekttypen geliefert wird, muss in den meisten Fällen ein eigenes Projektsystem bereitgestellt werden, um diese Variante von Visual Studio sinnvoll einsetzen zu können.

3.3.1. Visual Studio Extensibility (VSX)

Es ist möglich, Visual Studio auf zwei grundlegenden Wegen zu erweitern: Durch *Add-Ins* und sogenannte *Visual Studio Integration Packages (kurz VSPackages oder Packages)*. Während Add-Ins nur auf einen beschränkten Teil von VS zugreifen und diesen manipulieren können, können mit Packages nahezu alle Teile von Visual Studio verändert und erweitert werden. Die übrigen Möglichkeiten, Visual Studio um eigene Funktionalität zu ergänzen, sollen hier nicht weiter betrachtet, aber zumindest kurz erwähnt werden:

- *Macros* können durch den Benutzer aufgezeichnet, editiert und ausgeführt werden.
- *Code Snippets* und *Project Templates* erlauben es, wiederkehrende Vorlagen als solche zu definieren und zu verwenden.
- Der Debugger kann durch *Visualizers* so erweitert werden, dass er weitere Datentypen in einem benutzerdefinierten Format anzeigen kann.

Im Gegensatz zu Add-Ins benötigt man zur Entwicklung von Integration Packages zwingend das kostenlose Visual Studio SDK[7] *(VSSDK)*, aber auch für die anderen Erweiterungsoptionen empfiehlt es sich, das SDK zu installieren, da mit diesem auch

[5] http://msdn.microsoft.com/en-us/vs2008/products/cc149003.aspx
[6] http://msdn.microsoft.com/en-us/vsx2008/products/bb933751.aspx
[7] http://msdn.microsoft.com/en-us/vsx/bb980956.aspx

Dokumentation und Beispiele geliefert werden.

Für die Art, wie *Flow* sich in Visual Studio integrieren soll, ist es unumgänglich, Integration Packages zu erstellen. Denn nur mittels Packages ist es möglich, neue Editoren für neue Dateitypen der IDE hinzuzufügen oder beispielsweise bereits geöffnete Dateien zu manipulieren und sogenannte globale Services innerhalb der IDE anzubieten.

Die Schnittstellen, die Add-Ins zur Verfügung stehen, sind relativ geradlinig und gut dokumentiert. Wohingegen Microsoft für die Entwicklung von Packages interne Schnittstellen offengelegt hat, die früher nicht als öffentliche Schnittstellen gedacht waren und nur Microsoft-intern bzw. ausgesuchten Partnern zur Verfügung standen. Diese Schnittstellen basieren auf COM und gliedern sich nur schwer in aktuelle Programmierparadigmen von .NET ein. Auch wenn dank COM-Interopt die Nutzung technisch kein Problem darstellt, stößt man immer wieder auf Gegebenheiten, die für .NET gewöhnte Augen umständlich erscheinen, wie z.B.:

- Um den Erfolg eines Funktionsaufrufes zu ermitteln, muss der Rückgabewert der Funktion untersucht werden. Exceptions sind in der COM-Welt unbekannt.
- Es wird an vielen Stellen mit skalaren Variablen *(`string`, `integer`)* anstatt mit Objekten oder `struct`s gearbeitet. Dies lässt Parameterlisten von Funktionen sehr lang werden.
- Für die .NET-Welt werden oft eher untypische Datentypen wie `uint` verwendet.
- .NET-Objekte müssen teilweise manuell in COM-Interfaces *(z.B. `IUnknown`)* oder umgekehrt gewandelt werden.
- Statt Enumeratoren – die einfach in der `foreach`-Schleife verwendet werden können – werden Methoden in der Form `GetFirstElement()` und `GetNextElement()` angeboten, die man entsprechend korrekt verwenden muss.
- Statt Events zu abonnieren, muss der Abonnent ein Interface *(oft mit vielen Methoden, auch wenn für ihn nur eine einzige von Bedeutung ist)* implementieren, um sich für Callbacks registrieren zu können.
- Statt Objektreferenzen wird an vielen Stellen mit Integer-Cookies gearbeitet, die man sich merken muss um, zu einem späteren Zeitpunkt auf eine gewisse Objektreferenz verweisen zu können.

Leider sind diese Schnittstellen zum Teil nur wenig oder überhaupt nicht dokumentiert, so dass man sich vielfach auf Methodennamen verlassen und anhand der Signatur raten muss, wie eine Funktion verwendet werden soll. Bei der Arbeit mit dem VSSDK werden deshalb Online-Ressourcen wie das *Microsoft MSDN Visual Studio Extensibility Forum*[8],

[8] http://forums.msdn.microsoft.com/en-US/vsx/threads/

die *VSX Team Website*[9] und das *Team Blog* sowie die Blogs der einzelnen Teammitglieder *(auf der Team Website verlinkt)* zu wichtigen Anlaufstellen. Auch Open Source Projekte sowie Webseiten und Blogs von engagierten Programmierern sind sehr hilfreich. Besonders hervorheben möchte ich das Blog von DiveDeeper[10], der einen ausführlichen Lehrgang in mehr als 30 Folgen zusammengestellt hat, der besonders Einsteigern verschiedene Aspekte der Programmierung von VSPackages näher bringt.

3.3.2. DSL Tools

Einen besonderen Platz bei der Visual Studio Erweiterbarkeit nehmen die DSL Tools[11] ein. Während die DSL Tools bei der Vorgängerversion *(Visual Studio 2005)* nur als separater Download erhältlich waren, sind sie nun Teil des normalen Visual Studio SDKs. Die Laufzeitbibliotheken stehen mit jeder Visual Studio Installation zur Verfügung.

Die DSL Tools stellen Microsofts Antwort auf die Frage nach Tools dar, die durch MDSD und das Thema Software Factories gefordert werden. Im Prinzip handelt es sich bei den DSL Tools um eine Software Factory, um domainspezifische Software Factories zu erstellen. Das Vorgehen ist dem ähnlicher Tools gleich: Der Anwender erhält einen graphischen Editor in dem er das Metamodell für seine DSL erzeugen kann. Dieses Metamodell trennt strikt zwischen der abstrakten und konkreten Syntax und erlaubt es, den zu erzeugenden Editor an vielen Stellen durch die Modellierung zu beeinflussen.

Als Resultat werden zwei Assemblies *(`.dll`)* mit C# Quellcode erzeugt, von der die eine das Metamodell enthält und Funktionen anbietet, um Modelle aus Dateien zu laden, im Speicher zu manipulieren und zu speichern. Das andere Assembly implementiert ein Visual Studio Integration Package, welches einen Editor und die übrige Infrastruktur *(z.B. zur Codegenerierung)* innerhalb von Visual Studio bereitstellt. Der so erzeugte Code ist an vielen Stellen durch Extension Points mittels eigenem Code erweiterbar, so dass der Editor bzw. die Software Factory den eigenen Bedürfnissen angepasst werden kann. Des Weiteren enthält der erzeugte Code ein Grundgerüst, um aus den Modellen Quellcode oder andere Artefakte mittels des Text Templating Transformation Toolkit *(siehe Abschnitt 3.3.3)* zu erzeugen. Auch Validierungsregeln als Teil der statischen Semantik lassen sich der erzeugten Software Factory einfach hinzufügen.

In Abbildung 3.3 ist die Visual Studio IDE zu sehen, in der ein Metamodell geöffnet ist. Im linken Teil *(beschriftet mit „Classes and Relationships“)* ist die abstrakte Syntax, bestehend aus *Domain Classes (blaue Rechtecke)* und *Relationships (gelbe Rechtecke)* modelliert. Die Beziehungen zwischen Domain Classes können in zwei Ausprägungen vorkommen: *embedding* und *reference*. Jede Domain Class *(außer der Wurzel)* muss in genau eine andere eingebettet sein, um einen Baum der Klassen des Modells aufbauen

[9] http://msdn.com/vsx
[10] http://dotneteers.net/blogs/divedeeper/
[11] http://msdn.microsoft.com/en-us/library/bb126235.aspx

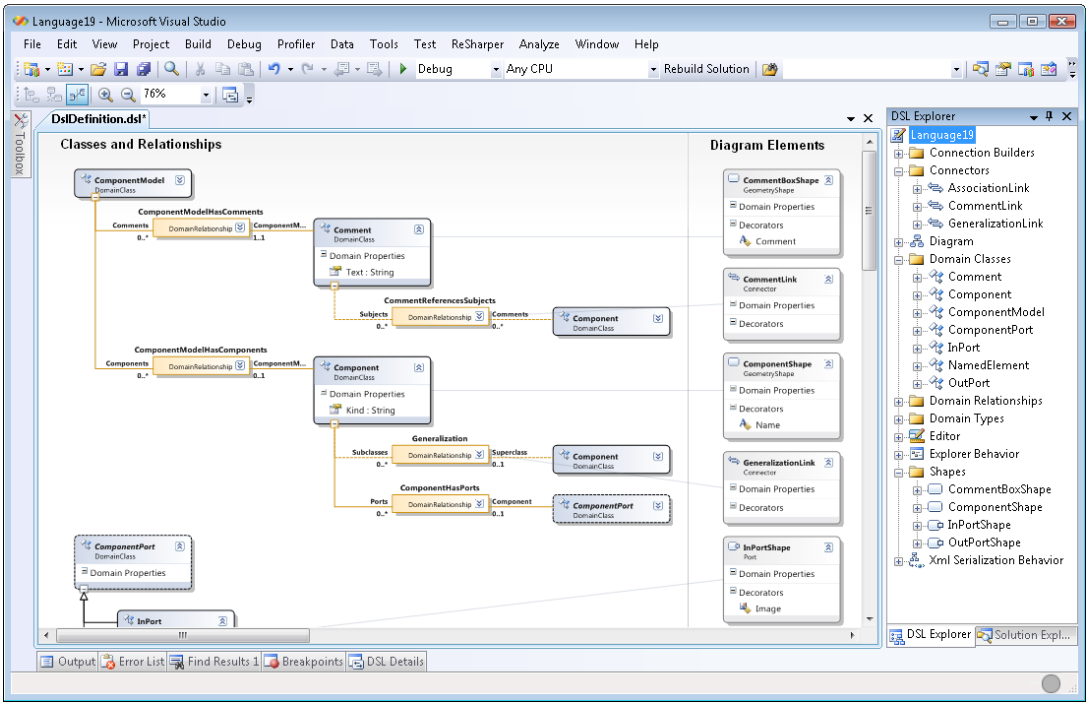


Abbildung 3.3.: Visual Studio: Metamodell im DSL-Editor

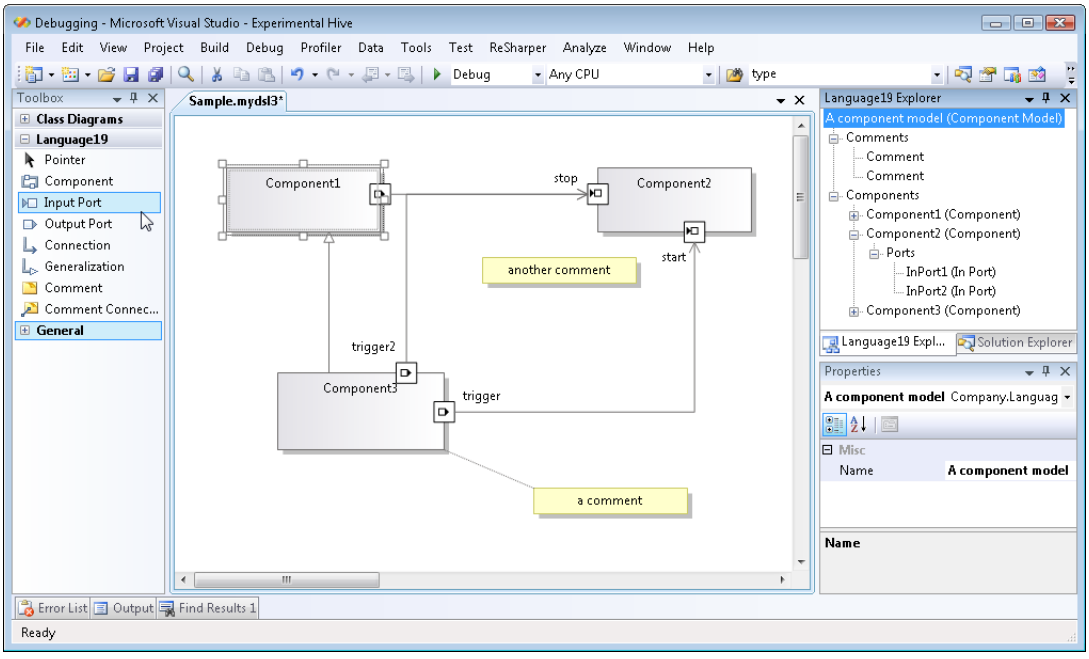


Abbildung 3.4.: Visual Studio: Editor einer benutzerdefinierten DSL

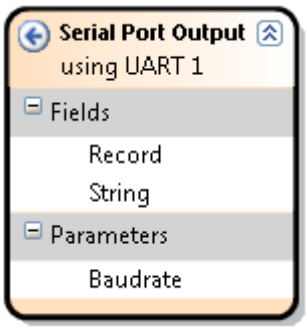


Abbildung 3.5.: Compartment Shape

zu können. Zusätzlich kann jede Klasse beliebige weitere Klassen referenzieren. Sowohl Domain Classes als auch Relationships können Daten in Form von *Domain Properties* enthalten.

Im rechten Teil des Metamodells *(beschriftet mit „Diagram Elements“)* ist die konkrete Syntax definiert, indem den Elementen der abstrakten Syntax aus dem linken Teil *Shapes* und *Connectoren* zugeordnet werden. Diese graphischen Elemente definieren den Grundtyp, wie ein Element im Diagramm dargestellt wird, können aber durch Konfiguration weiter verändert werden. An dieser Stelle wird auch mittels sogenannter *Decorators* definiert, wie die Daten *(Domain Properties)* der Domain Classes im Modell als Teil der Shapes angezeigt werden sollen. Als Shapes zur Repräsentation der Domain Classes stehen in den DSL Tools verschiedene Arten zur Verfügung:

- *Geometry Shape*: Kann als Rechteck, Rechteck mit abgerundeten Ecken, Kreis oder Ellipse eine Domain Class repräsentieren.
- *Image Shape*: Erlaubt es, ein beliebiges Bitmap zu verwenden.
- *Port Shapes* werden auf dem Rand eines anderen Shapes positioniert und können dort als besondere Endpunkte von Connectoren verwendet werden.
- *Compartment Shapes* werden als Rechtecke dargestellt, können darüber hinaus aber zeilenweise weitere Daten ihrer untergeordneten Domain Class anzeigen. Abbildung 3.5 zeigt ein Beispiel eines Compartment Shapes mit zwei sogenannten Compartments *(„Fields“ und „Parameters“)*.
- *Swimmlanes* sind eine besondere Art Shape, denn auf dem Diagramm erkennt man sie kaum als Shape, das eine Domain Class repräsentiert, sondern nimmt Swimmlanes viel mehr als Teil des Hintergrunds wahr. Ein Beispiel sind die beiden Bereiche „Classes and Relationships“ und „Diagram Elements“ in Abbildung 3.3. Swimmlanes können zur Strukturierung von Diagrammen verwendet werden.
- *Connectoren* repräsentieren Beziehungen zwischen Domain Classes bzw. Shapes und werden als Linie oder Pfeil dargestellt.

Neben den Beziehungen, die bereits beschrieben wurden, können im Metamodell auch Vererbungshierarchien von Domain Classes und Shapes definiert werden. Dies ist nützlich, da aus den Elementen des Modells .NET-Klassen erzeugt werden, mit denen im weiteren Verlauf gearbeitet wird.

Aus dem in Abbildung 3.3 dargestellten Metamodell kann ein Editor inklusive Integration in Visual Studio erzeugt werden. Das Ergebnis mit einem Beispielmodell ist in Abbildung 3.4 gezeigt. In der Mitte ist die Zeichenfläche mit einigen Shapes und Connectoren des Beispielmodells zu sehen. Aus der Toolbox auf der rechten Seite kann der Benutzer Elemente *(Domain Classes)* per drag-and-drop auf die Zeichenfläche ziehen und so neue Instanzen erzeugen. Im Model Explorer *(rechts oben)* wird die hierarchische Baumstruktur *(gegeben durch die Embedded Relationships)* der Elemente angezeigt. Die Domain Properties des ausgewählten Elementes *(hier „Component1“)* werden im Properties Window *(rechts unten)* angezeigt und können dort bearbeitet werden.

Während bei der Erweiterung von Visual Studio mittels Integration Packages die größten Kritikpunkte die wenige Dokumentation sowie die COM-Wurzeln sind, basieren die DSL Tools auf .NET und erzeugen ihre Ergebnisse als C#-Klassen. Dieser Code ist gut strukturiert, so dass man sich in kurzer Zeit einarbeiten kann. Neben der im Vergleich zu VSPackages um einiges besseren online Dokumentation[12] ist das Buch „Domain-Specific Development with Visual Studio DSL Tools“ [CJKW07], geschrieben von einigen der Entwicklern der DSL Tools, sehr zu empfehlen und bereits zum Standardwerk für dieses Toolset geworden.

3.3.3. Text Templating Transformation Toolkit (T4)

Das Text Templating Transformation Toolkit ist ebenfalls ein Bestandteil von Visual Studio und wird unter anderem verwendet, um aus Modellen Quellcode zu erzeugen. Sowohl die DSL Tools Software Factory selbst, als auch die erstellten Software Factories verwenden dieses Toolkit.

Die Templatesfiles *(siehe Abbildung 3.6)* mit der Dateiendung `.tt` enthalten den zu erzeugenden Quellcode, aber auch Kontrollstrukturen, die in C# oder Visual Basic programmiert werden können. Diese Templates erinnern in gewisser Weise an Webseiten, die mit Techniken wie JSP, ASP oder einer anderen eingebetteten Skriptsprache dynamisch erzeugt werden. Die Templates werden bei der Ausführung kompiliert und im Rahmen des .NET-Frameworks ausgeführt. Dabei haben sie Zugriff auf alle Bibliotheken und Funktionalitäten, die das .NET-Framework zur Verfügung stellt.

Zwar werden solche Templates leicht unübersichtlich, da sich oft der zu erzeugende Quellcode und die Kontrollstrukturen sehr ähnlich sind, aber es ist dennoch ein sehr effizienter Weg, um aus Modellen Quellcode zu erzeugen. Auch wenn meist vom Ziel

[12] http://msdn.microsoft.com/en-us/library/bb126235.aspx

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".cs" #>
<#@ Language20 processor="Language20DirectiveProcessor" requires="fileName='Sample.myds14'" #>

<#
  foreach (ModelClass class in this.ModelRoot.Types)  {
#>
public class  <#= class.Name #>
{
<#
    foreach (Method method in class.Methods) {
#>
        public <#= method.ReturnType #> <#= method.Name #>()
        {
            // TODO: put your code here
        }
<#  } // foreach method #>
}
<# } // foreach class #>
```

Abbildung 3.6.: Beispiel eines T4-Templates

der Quellcode-Erzeugung ausgegangen wird, können mit T4 beliebige Artefakte erzeugt werden, solange es sich dabei um Textdateien handelt. Es wäre z.B. ohne weiteres möglich XML-Konfigurationsdateien oder HTML-Dateien aus den Modellen zu erstellen.

3.4. Nullsoft Scriptable Install System

Das Nullsoft Scriptable Install System[13] *(NSIS)* ist ein Generator, um Installationsprogramme für Windows zu erzeugen. Dabei enthält er die wichtigsten Funktionen von Hause aus, die man sich von einem Programm dieser Gattung wünscht: Er führt den Anwender mittels eines Assistenten durch die verschiedenen Schritte der Installation und erzeugt ein Uninstall-Programm. Es ist möglich Dateien auf dem Zielsystem zu installieren und die Registry zu bearbeiten sowie Einträge im Startmenü zu erstellen. Viele der am häufigsten benötigten Funktionen sind sehr einfach zu verwenden und für exotische Anwendungsfälle steht eine große Anzahl von Plugins zur Verfügung *(beispielsweise um während der Installation Teile aus dem Internet nachzuladen)*.

Zwar können auch mit Visual Studio Installationsprogramme erzeugt werden, aber diese werden graphisch innerhalb der IDE konfiguriert und sind relativ unflexibel und es ist nur schwer möglich, die Schritte zum Erzeugen eines Setups zu automatisieren. Für die Installation der DSLs wird daher ein anderes Tool *(WiX[14])* empfohlen, welches allerdings auch manuelle Arbeit erfordert, um so erweitert zu werden, dass mit einem Setup gleichzeitig mehrere DSLs installiert werden können *(die mitgelieferte XML-Setupbeschreibung für eine DSL ist bereits 800 Zeilen lang)*. Diese Gründe sprechen für den Einsatz eines universellen und scriptbasierten Tools wie NSIS, das über eine einfache Skriptsprache gesteuert wird. Diese NSI-Skripte stellen im Prinzip eine textuelle DSL dar, die für die Anforderungen eines Installationsprogramms entworfen wurde. Das Installieren

[13] http://nsis.sourceforge.net/
[14] Windows Installer XML: http://wix.sourceforge.net/

von Dateien und Bearbeiten der Registry ist relativ einfach, wohingegen benutzerdefinierte Methoden und andere selten genutzte Funktionen eher etwas umständlich erscheinen. Mittels des NSI-Compilers kann ein solches NSI-Skript kompiliert werden, was eine einzelne komprimierte ausführbare Datei *(`.exe`)* erzeugt.

NSIS steht unter der BSD-ähnlichen freien zlib/libpng-Lizenz und kann somit kostenlos in Open Source, aber auch kommerziellen Projekten eingesetzt werden. Die mit NSIS erzeugten Installationsprogramme – und natürlich die damit installierte Software – sind von diesen Lizenzbedingungen in keiner Weise betroffen und können insbesondere nach eigenem Belieben verteilt werden, ohne dass Quellcode offengelegt werden müsste.

Neben der einfachen Bedienung und dem großen Umfang zeigt sich im Rahmen dieser Arbeit der skriptbasierte Ansatz als großer Vorteil. Nicht nur, dass *Flow* mittels eines NSIS-Setups installiert werden kann, es ist auch möglich, aus *Flow* heraus *(mittels des Codegenerators)* NSI-Skripte zu erzeugen und direkt Installationsprogramme für vom Anwender erstellte Hardwareplattformen zu kompilieren.

4. Anforderungsanalyse

Die Anforderungen, die an *Flow* gestellt werden, sind maßgeblich durch zwei Parteien geprägt. Diese beiden Hauptanwender sind im Folgenden kurz Charakterisiert, bevor in den folgenden beiden Abschnitten ihre Anforderungen beschrieben werden. Darauf aufbauend werden weitere nicht-funktionale Anforderungen ergänzt und am Ende des Kapitels Werkzeuge definiert, die benötigt werden um den Anforderungen gerecht zu werden.

- Der *Anwender* verwendet *Flow* als Entwicklungsumgebung, um das Verhalten der Sensorknoten zu spezifizieren. In dieser Hinsicht ist er der primäre Anforderungssteller und Hauptnutzer der Software. Alle weiteren Anforderungen müssen sich so in das Gesamtkonzept eingliedern, dass der Anwender seine Aufgaben möglichst effizient erfüllen kann und die in Kapitel 1 beschriebenen Ziele erreicht werden.
- Der *Hardwarehersteller* ist in dieser Betrachtung auch der Hersteller der zugrundeliegenden Firmware und aus diesem Grund dafür verantwortlich, diese Firmware so zu gestalten, dass sie zu *Flow* kompatibel ist. Daher ist es notwendig, ihn bei seiner Arbeit zu unterstützen. Der Hersteller von Hardwareerweiterungen ist dem Hardwarehersteller gleichgestellt, da er auf dem gleichen Weg wie dieser eine Firmware *(Treiber)* für die Hardwareerweiterung bereitstellen muss.

4.1. Flow aus Anwendersicht

Wie beschrieben soll der Anwender datenorientiert mit *Flow* arbeiten und die Datenflüsse innerhalb seiner Anwendung graphisch beschreiben. Dazu muss es möglich sein, graphische Repräsentationen von Ein- und Ausgängen anlegen zu können. Die Eingänge werden mittels Kanten – unter Umständen über Transformationen – mit den Ausgängen verbunden. Ein solcher beispielhafter und nicht an die Notation der konkret verwendeten DSL angepasster *Dataflow* ist in Abbildung 4.1 zu sehen.

Neben den Dataflows *(die das „Was soll geschehen?“ beschreiben, aber nicht „wie“ dies geschehen soll)* ist es nötig Ereignisse definieren zu können, um das „Wann?“ zu spezifizieren. Dazu sollen Ereignisse, die die Hardware bereitstellt, verwendet werden. Ereignisse sollen, optional an Bedingungen geknüpft, die Ausführung eines Dataflows starten können.

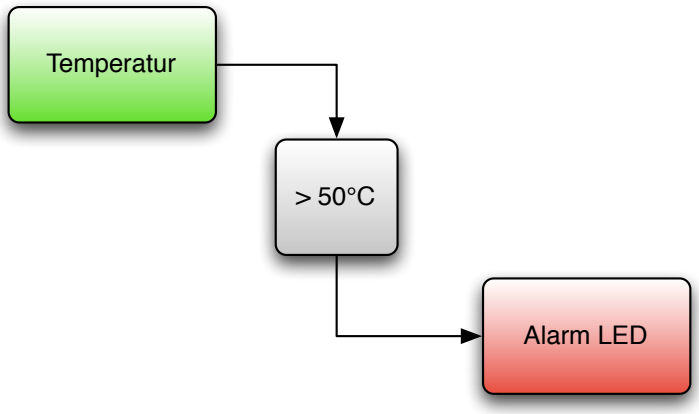


Abbildung 4.1.: Beispielhafter Dataflow: Bei einer Temperatur über 50° schalte eine LED ein

Solche Ereignisse können z.B. sein:

- ein Taster wurde gedrückt
- ein periodischer Timer wird regelmäßig aktiviert
- eine Nachricht wurde über die Funkschnittstelle empfangen

Aus diesen Dataflows mit den Ereignissen und Bedingungen muss dann C-Code generiert werden, der ohne weiteres Zutun durch den Anwender kompilierbar und auf der Zielplattform lauffähig ist.

Die hohe Abstraktionsebene, die das Arbeiten mit Dataflows erlaubt, erleichtert dem Anwender die Arbeit, doch kann auch leicht geschehen, dass man ihn so sehr einschränkt, dass ihm gewisse Möglichkeiten genommen werden. Beim Entwurf von *Flow* soll dies immer beachtet und verhindert werden. Ziel ist, dass ein Großteil der vom Anwender benötigten Programme durch Modellieren der Dataflows zu erzeugen ist, aber dass der Anwender, wenn dies nötig ist, die Dataflows um C-Code anreichern kann, um an keiner Stelle eingeschränkt zu werden. Diese Anforderung scheint dem entgegen zu stehen, was bereits in Kapitel 1 gefordert wurde: Der Anwender soll kein C beherrschen müssen. Allerdings zeigt die Erfahrung mit visuellen Programmiersprachen, dass es oft nicht möglich ist, alles visuell abzubilden, bzw. wenn dies möglich ist, dass es oft umständlich ist. In einem solchen Fall ist es besser, vom Paradigma des Modellierens ein wenig Abstand zu nehmen, um dem Anwender keine Möglichkeiten zu verwehren und damit das gesamte System für ihn nutzlos zu machen.

Zusätzlich zu den Dataflows, die das dynamische Verhalten der Sensorknotenanwendung beschreiben, muss der Anwender auch die Möglichkeit haben, die statische Sicht auf die Anwendung zu modellieren. Die statische Sicht definiert vor allem globale Datenstruktu-

ren, die in verschiedenen Dataflows und anderen Teilen, wie z.B. dem Proxy, verwendet werden.

4.1.1. Proxy

Es ist nur selten der Fall, dass ein Sensornetzwerk vollkommen autark eingesetzt wird. Es besteht, sei es zu Verwaltungszwecken oder zur Datenauswertung, in den meisten Fällen eine Verbindung zu einem *(eingebetteten)* PC. Auch um diesen Teil eines Sensornetzwerkes zu programmieren, ist es für den PC-Programmierer notwendig, ein gewisses konzeptionelles Wissen über Sensornetzwerke aufzubauen. Unter Umständen muss auf dem PC sehr hardwarenah programmiert werden, um mit dem Sensornetzwerk zu kommunizieren.

Auch hier soll es durch *Flow* möglich sein, Aufwand zu reduzieren und vom PC-Programmierer nicht verlangen zu müssen Spezialwissen aufzubauen. Dazu soll neben der Software für die Sensorknoten eine Bibliothek für den PC generiert werden, die direkte Zugriffe auf die konkrete Sensorknotenanwendung ermöglicht. Damit dies durchgeführt werden kann, muss der Anwender die entsprechenden Schnittstellen zwischen den Sensorknoten und dem PC modellieren können.

4.2. Flow aus Hardwareherstellersicht

Da *Flow* nicht nur mit einer einzigen Hardware zusammenarbeiten soll, ist es notwendig, eine Abstraktionsschicht zwischen der Hardware und *Flow* einzuführen. Diese Schicht lässt verschiedene Hardware gleich erscheinen und bietet dem Anwender verschiedene Elemente an *(wie auch jede Hardware z.B. verschiedene Sensoren anbietet)*. Ein Hardwarehersteller muss nicht nur diese Abstraktionsschicht erstellen, sondern sie auch in eine Form überführen, die *Flow* als Basis zur Codegenerierung verwenden kann. Um diese Aufgabe zu vereinfachen, soll mit Annotationen angereicherter C-Quellcode der Firmware eingelesen werden, so dass *Flow* daraus alle benötigen Informationen extrahieren kann.

Nachdem der Hardwarehersteller die Firmware so entworfen hat, dass sie mit *Flow* kompatibel ist, soll es ihm möglich sein, aus der oben erstellten Beschreibung und der Firmware ein Paket zu erzeugen, das der Anwender als Basis für seine Entwicklung nutzen kann.

4.2.1. Treiber

Treiber stellen eine Spezialform der Hardwarebeschreibung dar. Ähnlich wie die Basis-Hardware, sollen Treiber Elemente *(z.B. Eingänge mit Sensordaten und Ereignisse)*, die zur Modellierung verwendet werden können, anbieten. Die Anforderungen, die *Flow* an die Treiberfirmware stellt, sind somit denen, die an die Hardwarefirmware gestellt

werden, sehr ähnlich: Der C-Quellcode wird mit Annotationen angereichert und diese Daten werden von *Flow* zur Codegenerierung verwendet.

Auch die weiteren Funktionen, die dem Hardwarehersteller an dieser Stelle durch *Flow* angeboten werden, sind die gleichen: Aus der Treiberfirmware soll ein Paket erzeugt werden, das der Anwender im Rahmen seiner Entwicklung laden und nutzen kann.

4.3. Nicht-funktionale Anforderungen

Neben den oben beschriebenen Kernanforderungen gibt es eine Reihe von weiteren nicht-funktionalen Anforderungen, die hier kurz zusammengefasst werden sollen:

Da eine graphische Software Factory erstellt werden soll, ist es wichtig, diese graphischen Editoren sowie alle weiteren Bestandteile möglichst nahtlos in einem einzigen Tool, mit dem der Anwender arbeiten wird, zu integrieren. Dazu soll Visual Studio mit Hilfe der DSL Tools um zusätzliche Editoren für die verschiedenen Modelle erweitert werden. Das gesamte System soll so weit integriert werden, dass, vom Neuanlegen eines Projektes bis zum Aufbringen der Anwendung auf einen Sensorknoten, alle Aufgaben innerhalb der Entwicklungsumgebung ausgeführt werden können.

Des Weiteren ist es für den Anwender nicht vollkommen trivial, semantisch korrekte Modelle zu erzeugen, so dass er dabei Unterstützung benötigt. Im Idealfall würde der Editor es nicht erlauben, semantisch falsche Modelle zu erstellen, und dies soll er auch überall verhindern, wo es möglich ist. Faktisch kann dies nicht immer verhindert werden. In solchen Fällen sollen Validierungsregeln den Anwender mit möglichst aussagekräftigen Meldungen auf Fehler hinweisen und bei der Korrektur unterstützen. Beispielsweise kann der Editor verhindern, dass der Anwender einem Objekt mehr ausgehende Kanten zu anderen Objekten hinzufügt als erlaubt sind. Doch wenn für die Anzahl der Kanten auch eine untere Grenze existiert, so ist diese Grenze bei einem leeren Modell mit nur einem Objekt zunächst unterschritten. Der Editor kann den Anwender nur mit Meldungen der Validierungsregeln auf einen solchen Fehler hinweise, aber den Fehler nicht verhindern.

Natürlich werden die allgemeinen Regeln guter Softwareentwicklung angewendet. Das heißt insbesondere, den Benutzer z.B. bei Fehleingaben zu unterstützen, robuste Software zu entwickeln, die nicht abstürzt, sondern qualifizierte Fehlermeldungen ausgibt und nach Möglichkeit aus einem ungültigen Zustand möglichst einfach wieder in einen gültigen überführt werden kann oder sich selbst überführt.

4.4. Abgrenzung

Der Fokus dieser Arbeit liegt in erster Linie auf dem Design der domainspezifischen Sprachen und der Implementierung der Software Factory. Es ist nicht vorgesehen, den durch *Flow* für die Sensorknoten erzeugten Code in irgendeiner Weise zu optimieren. Mit der ersten Version von *Flow* ist zunächst beabsichtigt, lauffähigen Code zu generieren,

der den graphischen Spezifikationen genügt. Insbesondere wird dieser Code weder in Hinblick auf den Speicherplatz – sowohl Programm- als auch Arbeitsspeicher – noch die Performance optimiert.

4.5. Werkzeuglandschaft

Aus diesen Anforderungen lassen sich verschiedene Werkzeuge ableiten, die benötigt werden, um so wie beschrieben arbeiten zu können.

Um eine Anwendung für einen Sensorknoten vollständig beschreiben zu können, sind verschiedene Blickwinkel nötig, die durch verschiedene Modelle abgebildet werden: Die *Hardwaredescription* beschreibt die verwendete Sensorknotenhardware und die von dieser Hardware angebotenen Funktionen *(z.B. die Sensoren)*. Diese Modelle werden vom Hardwarehersteller erzeugt *(bzw. von ihm aus der Firmware automatisch erstellt)*, können aber auch vom Anwender betrachtet und leicht modifiziert werden. Die statische Sicht auf die Sensorknotenanwendung wird in den *Datastructures* beschrieben. Hier definiert der Anwender Datenstrukturen und globale Variablen, die er bei der weiteren Programmierung verwenden kann. Außerdem wird hier die Schnittstelle zum PC-Proxy definiert. Das Verhalten der Anwendung wird in mehreren *Dataflow*-Modellen spezifiziert, die die Aktionen beschreiben, die bei bestimmten Ereignissen ausgeführt werden sollen.

Für jeden dieser Modelltypen wird ein Editor innerhalb von Visual Studio benötigt. Diese Editoren werden mit den DSL Tools erzeugt und müssen dann nur noch geringfügig angepasst werden. Insbesondere ist es notwendig, in den Dataflow-Modellen jene Elemente verwenden zu können, die in der Hardwaredescription und den Datastructures definiert sind. Darüber hinaus muss der Hardwaredescription-Editor in verschiedenen kontextabhängigen Modi betrieben werden können: Der Anwender darf, im Gegensatz zum Hardwarehersteller, nur einen geringen Teil des Modells bearbeiten. Außerdem sollen für ihn irrelevante Details versteckt werden. Da sich die Treiber und die Hardwareplattform sehr ähnlich sind, kann der gleiche Editor mit kleinen Anpassungen auch zur Bearbeitung von Treiberdefinitionen verwendet werden. Neben den Editoren werden weitere Funktionen, deren Zusammenspiel in Abbildung 4.2 dargestellt ist, benötigt: Für den Hardwarehersteller muss es möglich sein, aus der Firmware *(für die Basishardware oder für Hardwareerweiterungen)* ein Hardwaredescription-Modell zu erzeugen. Dazu sollen die Firmware eingelesen und gewisse Annotationen ausgewertet werden. Im Anschluss daran soll der Hardwarehersteller die Basishardware als Plattform bzw. die Treiber in einer Form exportieren können, die vom Anwender wiederum importiert und als Basis für seine Programme verwendet werden kann.

Ein Sensorknotenprojekt des Anwenders besteht immer aus einer Hardwaredescription, optional einem oder mehreren eingebetteten Treibern, den Datastructures und mehreren Dataflows *(siehe Abbildung 4.3)*. Da diese Modelle voneinander abhängen, müssen sie nicht nur jeweils einzeln validiert werden können, sondern es ist zusätzlich eine

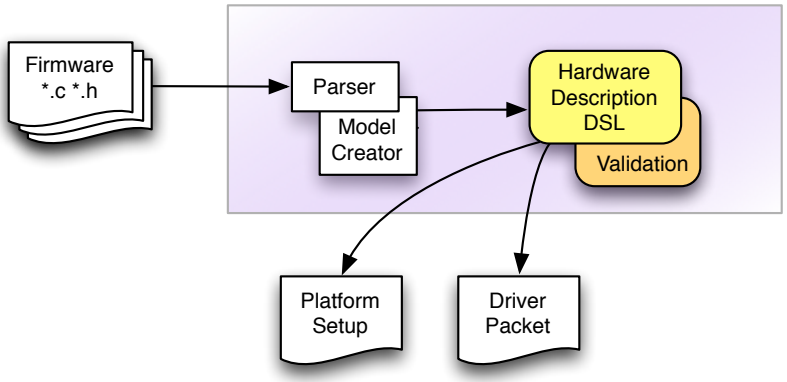


Abbildung 4.2.: Zusammenspiel der Werkzeuge für den Hardwarehersteller

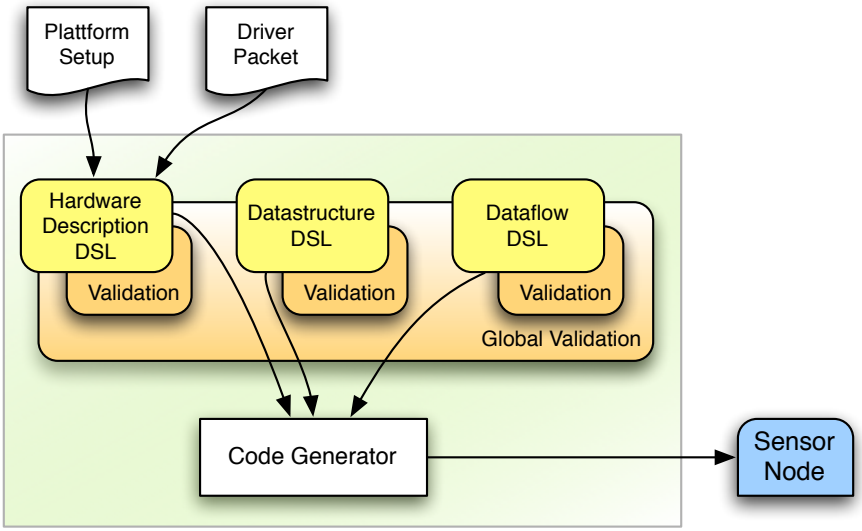


Abbildung 4.3.: Zusammenspiel der Werkzeuge für den Anwender

Validierungsfunktion notwendig, die alle Modelle im Kontext des Projektes auf ihre Gültigkeit prüfen kann *(Global Validation)*. Zu guter Letzt wird ein *Codegenerator* benötigt, der aus allen Modellen eines Projektes C-Code erzeugen kann, der dann kompiliert auf den Sensorknoten ausführbar ist.

5. Design der domainspezifischen Sprachen

Das Design der domainspezifischen Sprachen zur Beschreibung von Sensorknotenhardware und -anwendungen ist der Mittelpunkt dieser Arbeit. Nicht nur, dass alle weiteren Komponenten darauf aufbauen und um diese Sprachen herum positioniert sind, sondern sie stellen auch die Schnittstelle zwischen *Flow* und seinen Benutzern dar. Daher ist auch aus Sicht der Benutzerfreundlichkeit das Design der Sprachen von zentraler Bedeutung.

Aufgrund einer Einschränkung der DSL Tools, dass jeder DSL genau ein Editor zugeordnet sein muss und es keine einfachen Wege gibt, Editoren für verschiedene Sichten auf ein Modell zu erzeugen, müssen die gewünschten Sichten als separate DSLs modelliert werden. Da diese DSLs, wenn auch nur lose gekoppelt, in Beziehung zueinander stehen, muss zusätzliche Funktionalität implementiert werden, um die einzelnen Modelle konsistent zu halten *(siehe Abschnitt 6.2.2)*.

In den folgenden Abschnitten wird die abstrakte und konkrete Syntax der drei in *Flow* verwendeten Sprachen *Hardwaredescription-DSL*, *Datastructure-DSL* und *Dataflow-DSL* beschrieben. Die Validierungsregeln *(Constraints)* der statischen Semantik sind in Anhang C angegeben.

Sowohl die verwendeten Klassennamen, als auch die Begriffe, die in der Benutzeroberfläche angezeigt werden, sind Englisch. Da diese Namen innerhalb von *Flow* wie Eigennamen betrachtet werden, werden sie auch in der Beschreibung nicht ins Deutsche übersetzt.

5.1. Hardwaredescription-DSL

Da mit *Flow* verschiedene Hardwareplattformen für Sensorknoten angesprochen werden sollen, muss bekannt sein, welche konkreten Funktionen die verwendeten Sensorknoten anbieten. Diese Definition soll in der *Hardwaredescription* hinterlegt werden. Darüber hinaus muss angegeben werden, welche Methoden die Firmware anbietet und wie der Codegenerator diese ansprechen muss.

Um verschiedene Sensorknoten beschreiben zu können, muss eine gemeinsame Beschreibungsebene gefunden werden. Bereits in Kapitel 4.1 wurden Ein- und Ausgänge sowie Ereignisse als zentrale Entitäten festgehalten. Zusätzlich erscheint es sinnvoll, eine weitere Entität einzuführen, die von einem Sensorknoten bzw. der entsprechenden Firmware angeboten werden kann: Variablen. Variablen können vom Anwender gelesen und optional geschrieben werden. Sie speichern Werte, die von der Firmware für spezielle Aufgaben benötigt werden *(z.B. die Adresse des Sensorknotens im Funknetzwerk)*.

Abbildung 5.2 zeigt das Metamodell der Hardwaredescription-DSL in einer vereinfachten Darstellung *(einige Domain Classes und die Domain Properties fehlen)*. Für eine vollständige Abbildung des Metamodells sei auf Anhang A verwiesen.

5.1.1. Input- und Output-Ports

Da sich Input- und Output-Ports sehr ähnlich sind, sollen sie gemeinsam beschrieben und vom Anwender auf eine gleiche Art und Weise benutzt werden.

Sensorknoten haben oft viele Ein- und Ausgänge, die zu Gruppen zusammengefasst sind. Daher sollen auch die Input- und Output-Ports der Hardwaredescription nicht atomar sein, sondern aus einer Menge von atomaren Werten *(Feldern)* bestehen. Des Weiteren muss es möglich sein, sowohl Input- als auch Output-Ports mit Parametern zu versehen. Diese Parameter können vom Anwender verwendet werden, um die Sensorknotenhardware zu initialisieren. Abbildung 5.1 zeigt die Shapes, wie sie in *Flow* für Input- und Output-Ports angezeigt werden. Dabei müssen nicht nur Komponenten des Sensorknotens repräsentiert werden, die tatsächlich mit der Außenwelt kommunizieren, sondern es ist durchaus möglich, virtuelle Ein- und Ausgänge zu definieren, die lediglich durch die Firmware in Software implementiert sind. Die Realtime Clock ist ein Beispiel dafür. Beim „Serial Output“ in diesem Beispiel ist ein Parameter definiert, mit dem der Anwender die Geschwindigkeit der seriellen Kommunikation steuern kann.

Aus den Graphiken ist bereits ersichtlich, dass diese Elemente jeweils einen fettgedruckten Namen und eine Beschreibung besitzen. Auch die Felder und Parameter besitzen Namen und Beschreibungen, allerdings wird die Beschreibung nicht im Shape angezeigt. Da *Flow* stets typisiert arbeitet, muss für jedes Feld und auch für jeden Parameter ein Datentyp *(siehe Abschnitt 5.1.5)* festgelegt werden. Des Weiteren enthalten alle Elemente, also Input- und Output-Ports, Felder und Parameter, intern einen Globally Unique Identifier *(Guid)*, der an vielen Stellen von *Flow* zur Referenzierung von Elementen verwendet wird.

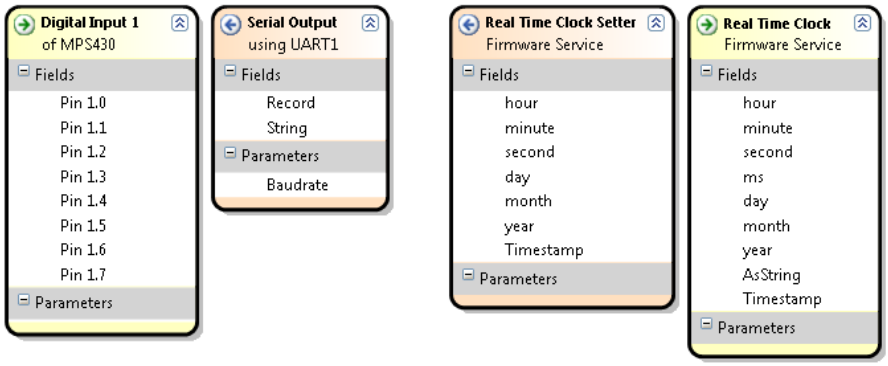


Abbildung 5.1.: Hardwaredescription: Input- und Output-Ports

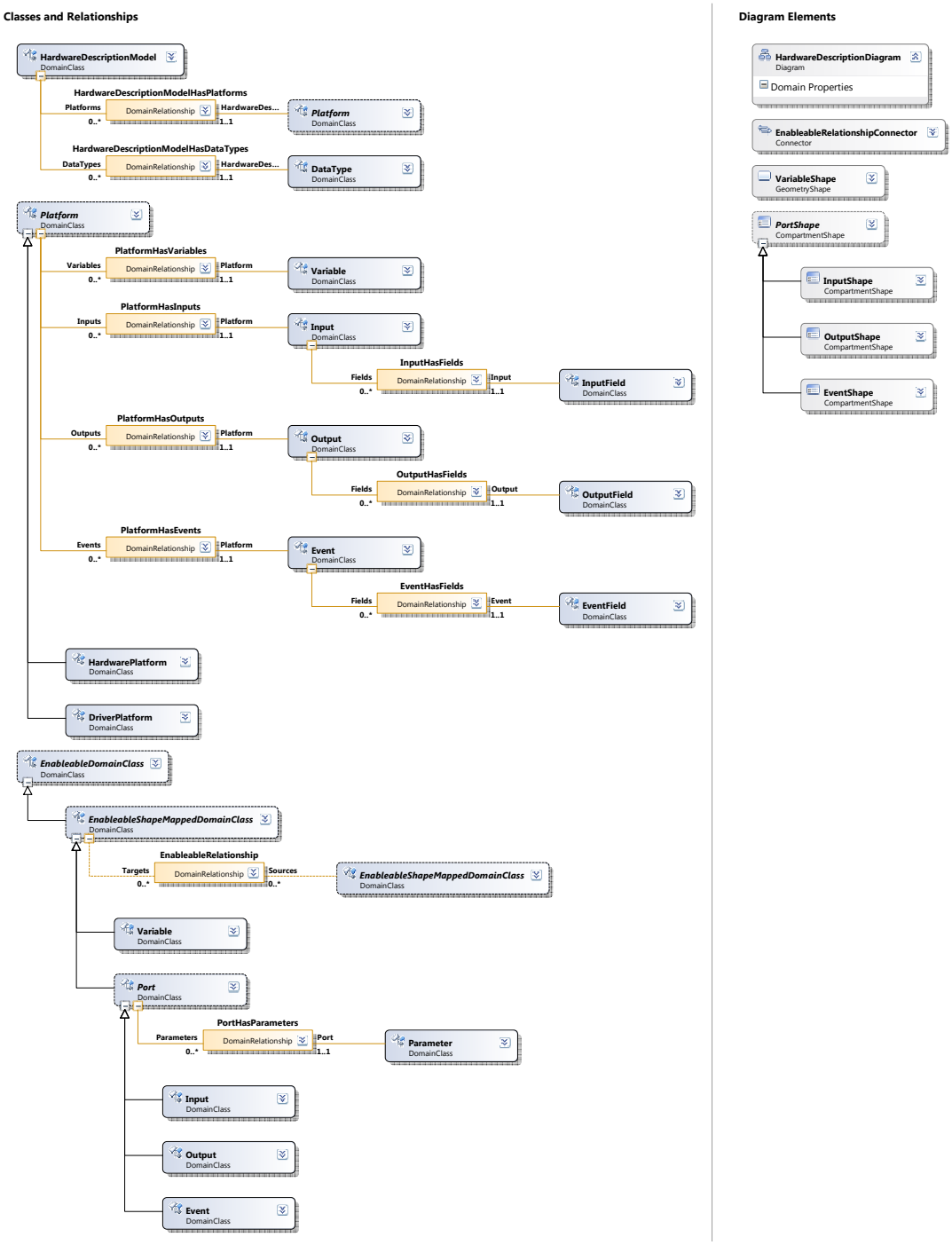


Abbildung 5.2.: Hardwaredescription-DSL Metamodell *(vereinfacht)*

Parametern, im Gegensatz zu Feldern, kann bereits in der Hardwaredescription ein Wert zugewiesen werden. Der Hardwarehersteller gibt an dieser Stelle einen Standardwert vor, den der Anwender verändern kann. Alle Daten, die hier beschrieben sind, aber nicht in den Shapes dargestellt werden, können in Visual Studio über das Properties Window bearbeitet werden, wenn das entsprechende Shape, Feld oder der Parameter ausgewählt ist.

Während der Hardwarehersteller in dieser DSL ohne Einschränkungen arbeiten kann, ist der Editor für den Anwender eingeschränkt. Er kann keine Shapes, Felder oder Parameter hinzufügen oder löschen und auch viele Properties *(z.B. den Datentyp)* nicht verändern. Er kann aber sehr wohl die Namen der Elemente – und auch die Beschreibung – an seine eigenen Bedürfnisse anpassen. Dies ist immer dann sinnvoll, wenn der Anwender an die universellen Ein- und Ausgänge konkrete Zusatzhardware angeschlossen hat. Beispielsweise könnte er dann das Feld „Pin 1.0“ in „rote LED“ umbenennen. Diese veränderten Namen erscheinen in den Dataflows und erlauben dem Anwender intuitiver mit seiner Sensorknotenhardware zu arbeiten.

Neben dem Ändern der Namen soll dem Anwender ein weiteres Instrument geboten werden, um die vom Hardwarehersteller gelieferte Hardwaredescription an seine Bedürfnisse anzupassen. Oft wird ein Teil der Sensorknotenhardware nicht benötigt. Beispielsweise verwendet der Anwender meist nur einige der 16 angebotenen digitalen Eingänge, so dass er einige Teile – ganze Shapes oder einzelne Felder – deaktivieren kann. Deaktivierte Shapes werden grau, deaktivierte Felder mit einem entsprechenden Symbol dargestellt. Das Deaktivieren von Elementen ist an zwei Stellen von großem Nutzen. Zum einen werden diese Shapes bzw. die Felder beim Bearbeiten der Dataflows nicht angezeigt, so dass die Komplexität verringert wird. Zum anderen wird der Firmware in der Initialisierungsphase bekanntgegeben, welche Teile deaktiviert sind, so dass diese Teile auch hardwareseitig ausgeschaltet werden können. Wenn z.B. ein GPS-Empfänger nicht benötigt wird, dann kann dieser ausgeschaltet bleiben um Strom zu sparen.

Bis zu dieser Stelle wurden Input- und Output-Ports gleich behandelt, doch die Felder der Output-Ports benötigen eine weitere Eigenschaft, die der Hardwarehersteller festlegen kann. Einzelne Felder können als optional gekennzeichnet werden, um zu steuern, ob in einem Dataflow immer alle Felder eines Output-Ports gemeinsam gesetzt werden müssen oder nicht. Ersteres ist z.B. bei einem Output-Port zum Versenden von Funknachrichten der Fall *(die Empfängeradresse und die zu übertragenden Daten müssen immer gemeinsam gesetzt werden)*, wohingegen die acht einzelnen Pins eines digitalen Ausgangs auch separat voneinander gesetzt werden können.

5.1.2. Events

Events haben große Ähnlichkeit zu den im letzten Abschnitt beschriebenen Input- und Output-Ports, aber auch wichtige semantische Unterschiede. Visuell werden sie auf eine ähnliche Art und Weise repräsentiert *(siehe Abbildung 5.3)*. Auch sie bestehen aus Feldern und Parametern sowie einem Namen, einer Beschreibung und den anderen beschriebenen Daten.

Events sorgen dafür, dass Dataflows ausgeführt werden und können sie mit Daten versorgen. Das „Radio Received"-Event *(Abbildung 5.3)* liefert einem Dataflow zwei Felder mit Informationen, die in einem Dataflow verwendet werden können. Da ein Event, sobald es ausgelöst wird, die vom Hardwarehersteller definierten Daten liefert, ist es für den Anwender nicht möglich einzelne Felder zu deaktivieren. Es steht ihm aber frei, gewisse Daten im Dataflow nicht zu verwenden. Das Event als Ganzens kann dahingegen deaktiviert werden. Wenn der Anwender beispielsweise das „Radio Received"-Event deaktiviert, könnte die Firmware den Transceiver vollständig ausschalten, da offensichtlich kein Interesse daran besteht, Funkpakete zu empfangen.

Es ist möglich dasselbe Event in mehreren Dataflows zu verwenden. Allerdings werden zwei Arten von Events unterschieden. Einige Events existieren nur ein einziges Mal beispielsweise das „Power On Event" und „Radio Received" *(Multiplicity = 1)*. Wenn diese Events auftreten, werden alle Dataflows, in denen sie verwendet werden, ausgewertet. Andere Events *(z.B. „Periodic Timer Activated Event")* können mehrfach und unabhängig voneinander existieren. Wird ein solches Event in einem Dataflow verwendet, entspricht dies einer neuen Ausprägung des Events. Diese Ausprägungen, deren Anzahl der Hardwarehersteller mittels der Eigenschaft „Multiplicity" begrenzen kann, können durch die Parameter in den Dataflows ein unterschiedliches Verhalten haben. Beispielsweise kann der Timer aus Abbildung 5.3 in bis zu vier Dataflows verwendet werden und jeden dieser Dataflows in einem anderen durch den Parameter einstellbaren Rhythmus aktivieren.

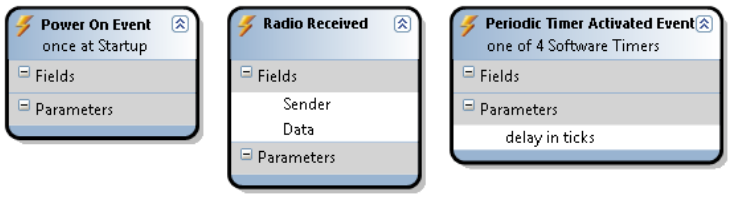


Abbildung 5.3.: Hardwaredescription: Events

5.1.3. Variablen

Variablen stellen die einfachsten Elemente der Hardwaredescription dar. Sie repräsentieren atomare Daten und enthalten daher weder Felder noch Parameter. Der Hardwarehersteller definiert die Variablen und kann sie wahlweise für den Anwender nur lesbar oder auch schreibbar machen. Abbildung 5.4 zeigt die graphische Repräsentation der Variable „Node Id“.

5.1.4. Kommentare

In der Hardwaredescription – und ebenso in den weiteren DSLs – können an beliebigen Stellen Kommentare abgelegt und mit Elementen verbunden werden. Diese Kommentare haben keinen Einfluss auf das Verhalten von *Flow* und dienen nur dem Benutzer. Abbildung 5.4 zeigt einen solchen, mit einer Variable verbundenen, Kommentar.

5.1.5. Datentypen

Auch die Datentypen, die für Felder, Parameter und Variablen verwendet werden, sind nicht durch *Flow* vorgegeben, sondern müssen ebenfalls in der Hardwaredescription definiert werden. Dabei verwendet *Flow* intern die folgenden Metadatentypen:

- Boolean
- Number
- String
- Record – für die in den Datastructures definierten Records *(siehe Abschnitt 5.2.2)*
- Other – um es dem Hardwarehersteller zu erlauben, spezielle Datentypen zu verwenden, die für *Flow* keine Bedeutung haben, aber in Dataflows genutzt werden können.

Der Hardwarehersteller kann beliebige Datentypen mit Namen, die innerhalb von *Flow* verwendet werden, und zugrundeliegendem C-Datentyp definieren und diesen Metadatentypen zuordnen. In der Praxis wird er für jeden Metadatentyp maximal einen eigenen Typ definieren. Für eine weitere Diskussion dieses Typsystems sei auf Abschnitt 8.1 verwiesen.

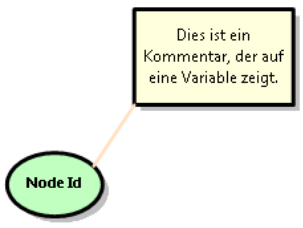


Abbildung 5.4.: Hardwaredescription: Variable mit Kommentar

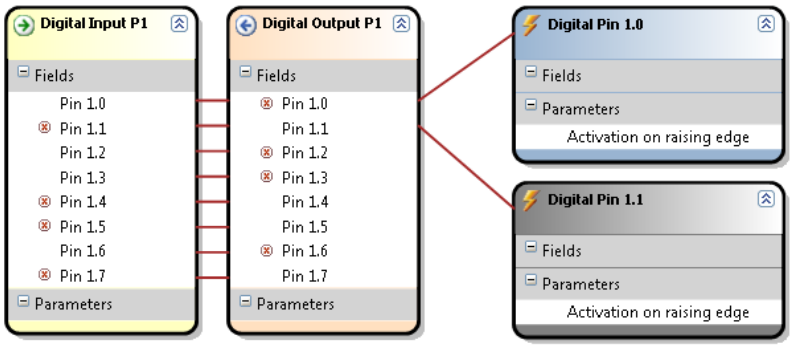


Abbildung 5.5.: Hardwaredescription: Kanten zwischen Ports und Events

5.1.6. Kanten in der Hardwaredescription

Die bisher vorgestellten Entitäten der Hardwaredescription beschreiben die Fähigkeiten eines Sensorknotens ausreichend, doch ein Detail kann noch nicht modelliert werden: Bei Mikrocontrollern, aber auch bei anderer Hardware, schließen sich gewisse Komponenten u.U. gegenseitig aus. Das wohl prominenteste Beispiel ist ein digitaler I/O-Port, der je nach Konfiguration als Eingang oder Ausgang verwendet werden kann. Nach obigen Definitionen müsste hierfür ein Input- und ein Output-Port in der DSL definiert werden, aber es können niemals beide gleichzeitig Verwendung finden. Dadurch, dass der Anwender einzelne Felder aktivieren und deaktivieren kann, kann zwar gesteuert werden, wie dieser I/O-Port verwendet werden soll, aber es muss sichergestellt werden, dass entweder der Input- oder der Output-Port, niemals aber beide, durch den Anwender aktiviert werden. Es ist auch nicht vorgesehen aus *Flow* dynamisch zwischen Eingang und Ausgang eines einzelnen Ports hin und her schalten zu können. So etwas ist zwar für gewisse Kommunikationsprotokolle notwendig, aber diese sollen nicht graphisch im Dataflow, sondern als Treiber in C programmiert werden.

Um diese Abhängigkeiten zu definieren, kann der Hardwarehersteller Kanten zwischen Feldern und Shapes erstellen. Wenn eine solche Kante vorhanden ist, dürfen die Elemente an ihren Enden niemals gleichzeitig aktiviert werden.

Das Beispiel aus Abbildung 5.5 beschreibt einen typischen I/O-Port, den der Anwender bitweise als Eingang oder Ausgang verwenden kann. Die Kanten verhindern aber, dass ein Feld gleichzeitig im Input- und im Output-Port aktiviert ist. Die Events für zwei der digitalen Eingänge sind ebenfalls dargestellt und dürfen nur dann aktiviert werden, wenn der entsprechende Output-Port deaktiviert ist.

5.1.7. Plattformen und Treiber

Die Beschreibung einer *Flow*-Hardwareplattform und die eines Treibers für *Flow* unterscheiden sich nur minimal, so dass der Hardwarehersteller diese mittels derselben DSL erzeugen kann. Es sind die gleichen Elemente *(Input-Port, Output-Port, Event und*

Variable) erlaubt, die ebenso über Kanten miteinander verbunden werden können.

Zusätzlich muss für Treiber angegeben werden, zu welchen Hardwareplattformen sie kompatibel sind. Wenn ein Anwender einen Treiber verwendet, kann somit die Kompatibilität überprüft werden. Treiber bauen zumeist auf anderen Komponenten *(z.B. Ein- und Ausgängen)* der Sensorknotenhardware auf, so dass diese Komponenten nicht mehr vom Anwender verwendet werden können. Um auch dies abzubilden, werden die oben beschriebenen Kanten eingesetzt. Der Hardwarehersteller kann dazu in der Treiberbeschreibung angeben, welche Entitäten der Hardwaredescription mit welchen der Treiberbeschreibung durch Kanten verbunden werden und sich somit gegenseitig ausschließen.

Der Hardwarehersteller modelliert in der Hardwaredescription-DSL immer genau eine Hardware- oder Treiberplattform pro Modell. Der Anwender hingegen hat die Möglichkeit, zu einer Hardwareplattform mehrere Treiber zu importieren, die dann im gleichen Modell dargestellt werden, aber in Bereichen voneinander getrennt sind.

5.2. Datastructures-DSL

Im Gegensatz zur Hardwaredescription, die vom Hardwarehersteller erzeugt und vom Anwender nur leicht verändert werden kann, sind die *Datastructure*-Modelle vollständig in der Hand des Anwenders. Diese Modelle sind relativ einfach aufgebaut, insbesondere da die einzelnen Entitäten nicht in Beziehung zueinander gesetzt werden. Die im Folgenden beschriebenen Entitäten werden lediglich auf der Zeichenfläche abgelegt und entsprechend durch den Anwender konfiguriert.

Jedes Sensorknotenprojekt kann mehrere Datastructure-Modelle enthalten, so dass der Anwender Elemente nach Belieben gruppieren oder auf mehrere Modelle verteilen kann. Für *Flow* macht es keinen Unterschied, ob die Entitäten in demselben oder in verschiedenen Modellen definiert sind, solange sie innerhalb des Sensorknotenprojektes, in dem sie verwendet werden sollen, definiert sind.

Diese Entitäten werden dann später aus den Dataflows heraus verwendet. Kommentare können auf die gleiche Art und Weise wie in der Hardwaredescription genutzt werden. In Abbildung 5.6 ist das vereinfachte Metamodell der Datastructures-DSL dargestellt, das vollständige Metamodell ist in Anhang A zu finden.

5.2.1. Variablen

Ähnlich wie der Hardwarehersteller Variablen in der Hardwaredescription definieren kann, kann auch der Anwender Variablen zur eigenen Verwendung in den Datastructure-Modellen anlegen. Diese Variablen werden graphisch so wie in Abbildung 5.4 repräsentiert. Der Anwender muss einen Datentyp aus der Liste der in der Hardwaredescription definierten Datentypen auswählen und kann optional einen Wert angeben, mit dem die Variable beim Starten des Sensorknotens initialisiert wird.

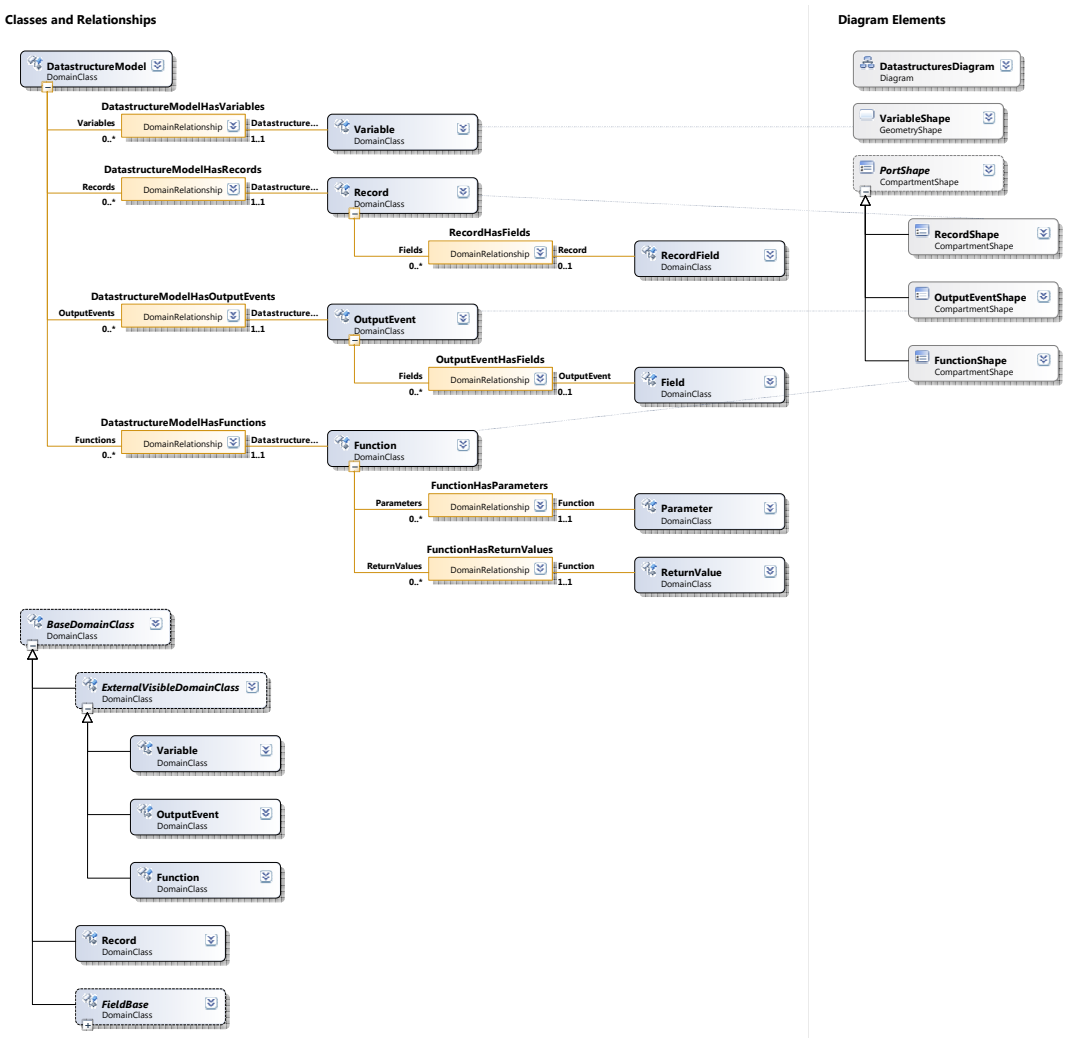


Abbildung 5.6.: Datastructures-DSL Metamodell *(vereinfacht)*

Diese Variablen können von allen Dataflows innerhalb des Projektes gelesen und geschrieben werden und erlauben dem Anwender dadurch, Informationen zwischen den Ausführungen von einzelnen Dataflows zu speichern.

Variablen können als öffentlich markiert werden, um es dem Proxy als Teil einer PC-Anwendung *(siehe Abschnitt 6.7.1)* zu erlauben, den Wert der Variable auslesen und optional verändern zu können.

5.2.2. Records

Nicht an allen Stellen soll mit atomaren Datentypen gearbeitet werden. Immer dann, wenn mehrere Daten zusammengefasst und als Einheit betrachtet werden, kommen Records zum Einsatz. Einer der wichtigen Einsatzorte von Records ist die Funkkommunikation zwischen Sensorknoten. Die Firmware bietet mittels Output-Ports und Events Funktionen an, um Records zu versenden und zu empfangen. Der Anwender kann Records in der Form definieren, wie es für seinen Anwendungsfall notwendig ist.

Records *(siehe Abbildung 5.7)* sind wie alle anderen Daten innerhalb von *Flow* typisiert, das heißt, dass jedes Feld einen Datentyp haben muss, aber auch, dass jedes Record eine eindeutige Id benötigt. Beim Anlegen neuer Recordausprägungen innerhalb eines Dataflows soll der Anwender nicht gezwungen werden, alle Felder des Records mit Werten zu füllen, deshalb können einige Felder als optional markiert werden. Weil aber für alle Felder Daten benötigt werden, ist für optionale Felder zusätzlich ein Standardwert anzugeben.

Da Records insbesondere auch über Funk übertragen werden, der Funkverkehr aber besonderen Beschränkungen unterliegt *(u.a. durch eine geringe Paketgröße)*, muss bei Feldern vom Typ String zusätzlich die maximale Länge des Strings angegeben werden, um die Effizienz an dieser Stelle optimieren zu können. In gewisser Weise stellt diese zusätzliche Information einen Bruch der Abstraktionsebenen dar: An dieser Stelle der Modellierung sollten Aspekte der Codegenerierung und der verwendeten Plattform *(noch)* keine Rolle spielen. Doch für Records wurde diese Ausnahme gemacht, um effizienten Code erzeugen zu können. Sollte ein anderer Codegenerator zu einem späteren Zeitpunkt keine besonderen Einschränkungen für die Verarbeitung von Strings haben, so kann dieser Codegenerator die zusätzliche Information der Stringlänge ignorieren.

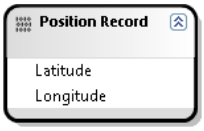


Abbildung 5.7.: Datastructures: Record

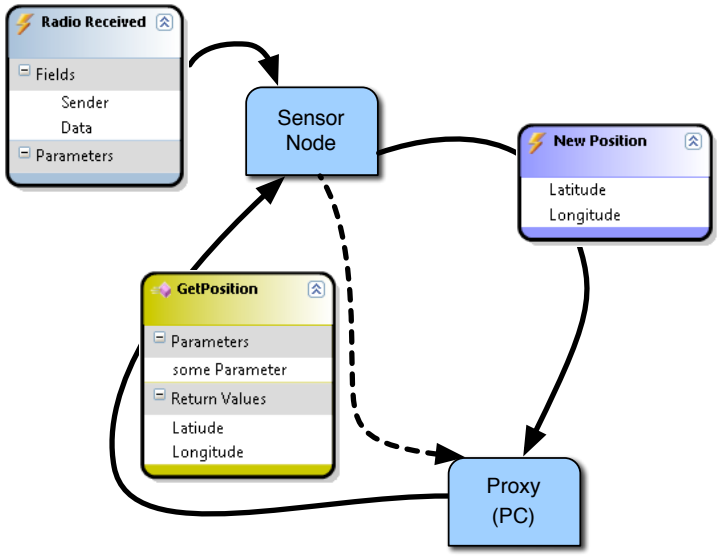


Abbildung 5.8.: **Zusammenspiel von Events und Funktionen mit dem Sensorknoten und Proxy**:
Das „Radio Received“ Event ist in der Hardwaredescription definiert und kann einen Dataflow auf dem Sensorknoten auslösen.
Das „New Position“ Output Event ist vom Anwender in den Datastructures definiert. Es kann vom Sensorknoten ausgelöst und vom Proxy empfangen werden.
Die Funktion „GetPosition“ ist ebenfalls in den Datastructures definiert und kann vom Proxy aufgerufen werden, was dazu führt, dass der Sensorknoten einen gewissen Dataflow abarbeitet. Wenn in der Funktion Rückgabewerte definiert sind kann der Sensorknoten diese an den Proxy zurückschicken.

5.2.3. Output Events

Die in der Hardwaredescription definierten Events stellen Ereignisse dar, die „in der Welt“ geschehen und durch die Firmware an den Sensorknoten gemeldet werden. Die Output Events, die der Anwender in den Datastructure-Modellen definiert, sind dahingegen Ereignisse, die der Sensorknoten auslösen kann und auf die der Proxy in einer PC-Anwendung *(siehe Abschnitt 6.7.1)* reagieren kann. Dieser Zusammenhang ist in Abbildung 5.8 dargestellt.

Diese Output Events enthalten, ähnlich wie Records, Felder, die innerhalb von Dataflows vom Anwender gefüllt werden müssen. Im Gegensatz zu einem Record, das nichts über die weitere Verarbeitung aussagt, werden die Daten der Output Events automatisch versendet und vom Proxy auf einem PC empfangen. Dieser Proxy ist ebenfalls von *Flow* generiert und ihm sind auch diese Datenstrukturen bekannt. Der Programmierer der PC-Anwendung kann die Events abonnieren und wird benachrichtigt, sobald ein Sensorknoten das entsprechende Event auslöst.

5.2.4. Funktionen

Während die Output Events die Kommunikation vom Sensorknoten zum Proxy ermöglichen, werden Funktionen für die entgegengesetzte Richtung eingesetzt *(siehe Abbildung 5.8)*. Funktionen definieren Parameter, die der Proxy an einen Sensorknoten schicken kann. Ein Sensorknoten kann mittels eines Dataflows auf diesen Funktionsaufruf reagieren *(Beispiel siehe Abschnitt 5.3.1 sowie Abbildung 5.11)* und, falls in der Funktion definiert, Rückgabewerte an den Proxy zurück senden.

5.3. Dataflow-DSL

Die wichtigste DSL innerhalb von *Flow* ist die *Dataflow*-DSL, denn in ihr wird das Verhalten der Sensorknotenanwendung spezifiziert. Jedes Sensorknotenprojekt enthält mehrere Dataflow-Modelle. In diesen Modellen wird auf unterschiedliche Ereignisse reagiert. Darüber hinaus ist dies auch die umfangreichste Sprache, wie das vereinfachte Metamodell in Abbildung 5.9 und das vollständige Metamodell in Anhang A belegen.

Jedes Dataflow-Modell besteht aus zwei Teilen *(siehe Abbildung 5.10)*: dem grau hinterlegtem Trigger-Bereich oben und dem weiß hinterlegtem eigentlichen Dataflow-Bereich darunter. Im oberen Bereich werden Events und Funktionen abgelegt, die den Dataflow – optional an Bedingungen geknüpft – auslösen. Der Dataflow-Bereich darunter wird nur ausgeführt, wenn dies durch den Trigger-Bereich freigegeben wird. Dann wird der gesamte Dataflow ausgewertet und die dort modellierten Aktionen ausgeführt.

Viele Elemente, die der Anwender in diesem Modell verwenden kann, stammen aus den bereits beschriebenen Hardwaredescription- und Datastructure-Modellen. Diese und weitere, die nur in den Dataflows verwendet werden können, sollen zunächst beschrieben werden, bevor ab Abschnitt 5.3.3 die Semantik dieser Elemente im Dataflow-Modell definiert wird. Alle Elemente in diesem Modell werden mit Kanten so miteinander verbunden, dass Daten an diesen Kanten aus Datenquellen *(z.B. Input Ports)* über Transformationen *(z.B. logische Verknüpfungen)* zu Datensenken *(z.B. Output Events)* „fließen“.

5.3.1. Elemente aus Hardwaredescription- und Datastructure-Modellen

Alle Elemente, die in der Hardwaredescription definiert sind, können im Dataflow verwendet werden. Dazu werden die Elemente in der Toolbox von Visual Studio angezeigt und können von dort per drag-and-drop auf dem Modell platziert werden. Der Anwender kann keine eigenen neuen Elemente dieser Art anlegen. Abbildung 5.10 zeigt ein Dataflow-Modell, in dem die verschiedenen Elemente der Hardwaredescription verwendet werden. Die Semantik der Kanten in Abschnitt 5.3.3 beschrieben. Zu beachten ist, dass die Input- und Output-Ports mit weniger Feldern, als in der Hardwaredescription angegeben, dargestellt werden, da die übrigen Felder vom Anwender deaktiviert wurden. Des Weiteren

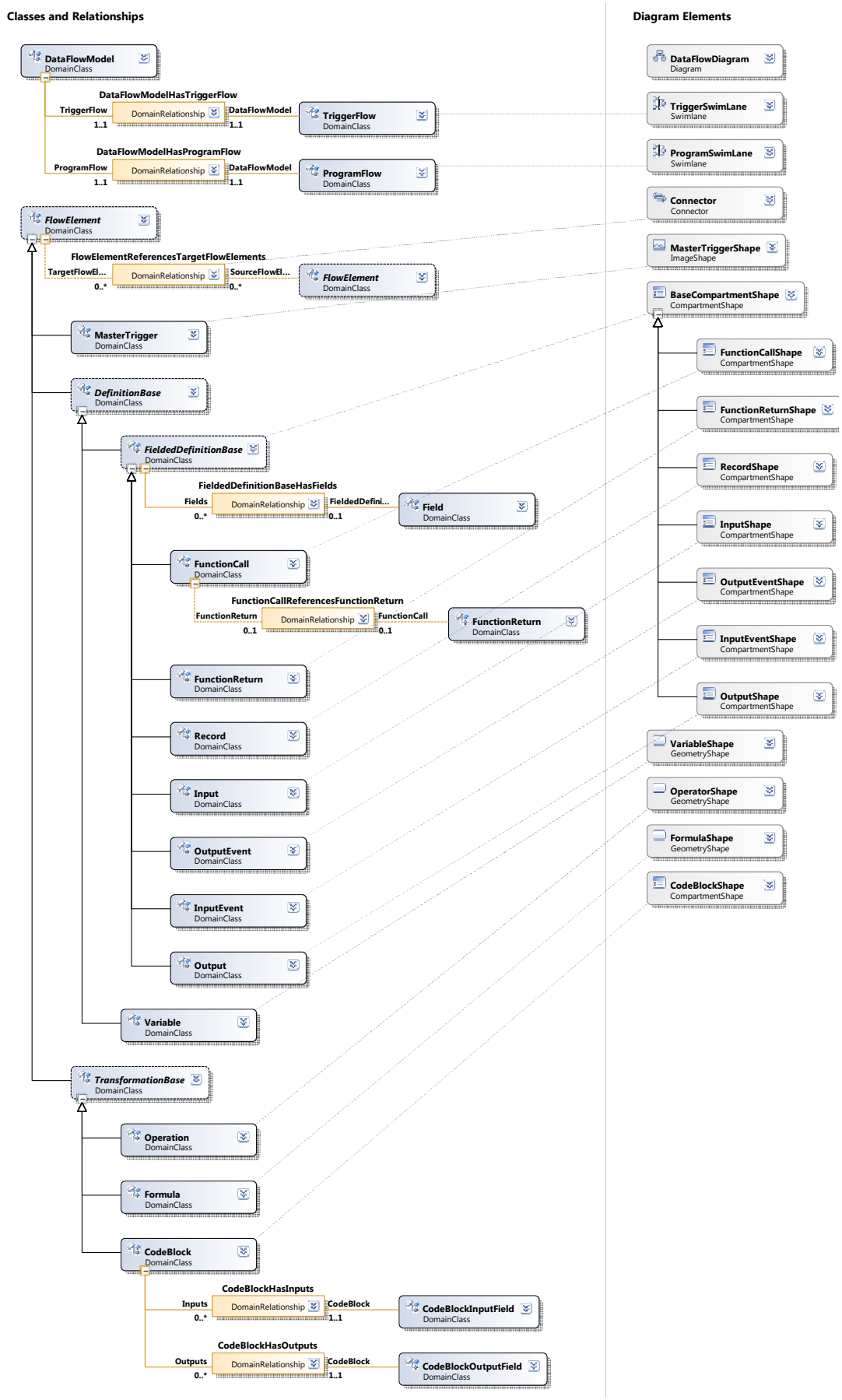


Abbildung 5.9.: Dataflow-DSL Metamodell *(vereinfacht)*

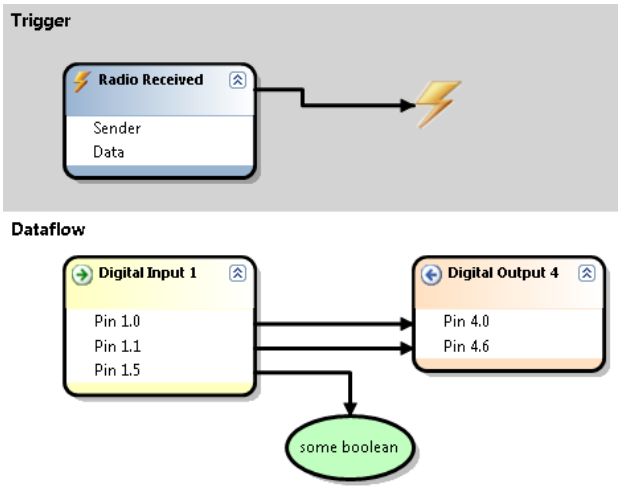


Abbildung 5.10.: Dataflow: Elemente aus der Hardwaredescription

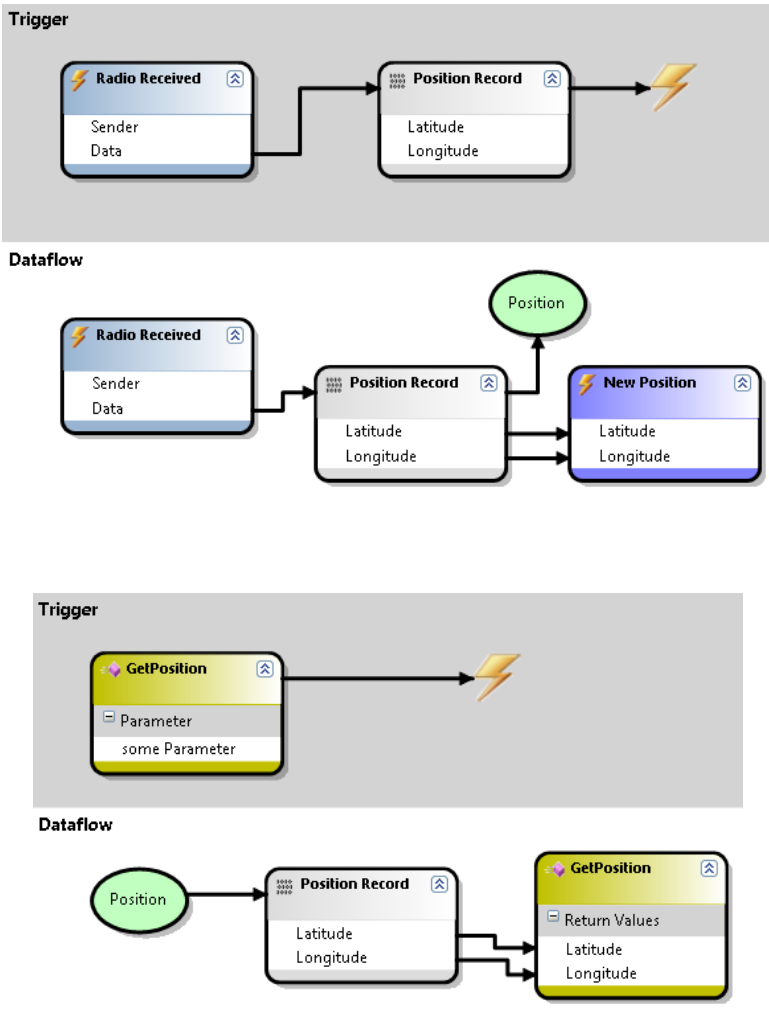


Abbildung 5.11.: Dataflow: Zwei Modelle mit Elementen aus dem Datastructure-Modell

wird im Dataflow auf die Anzeige von Parametern verzichtet, da diese für den Dataflow nicht von Bedeutung sind.

Ähnlich wie die Elemente der Hardwaredescription können auch die vom Anwender in den Datastructures definierten Elemente im Dataflow verwendet werden. Variablen, Records und Output Events werden dabei genauso wie in den Datastructure-Modellen dargestellt, doch Funktionen werden in zwei Teile aufgeteilt, so dass die Parameter und Rückgabewerte getrennt positioniert werden können. Das Beispiel aus Abbildung 5.11 empfängt „Position Records“ per Funk und löst immer wenn dies geschieht ein Output Event für den Proxy aus, außerdem wird das Record in der Variable „Position“ gespeichert. Mit dem zweiten Dataflow erhält der Proxy zusätzlich die Möglichkeit mittels der Funktion „GetPosition“ aktiv nach der letzten Position zu fragen, welche dann über die Rückgabewerte der Funktion an ihn geliefert wird.

5.3.2. Transformationen

Mit den bisher beschriebenen Elementen ist es zwar möglich, Daten zu empfangen, auf Ereignisse zu reagieren und Daten weiter zu geben, aber diese Daten können noch in keiner Weise verändert werden. Es ist auch noch nicht möglich, Bedingungen zu definieren. Um diese Aufgaben zu erfüllen existieren verschiedene Transformationen, die der Anwender im Dataflow verwenden kann. Diese Transformationen zeichnen sich gemeinsam dadurch aus, dass sie Daten über Kanten entgegennehmen, sie verändern und neue Daten zur weiteren Verarbeitung liefern.

Die einfachsten Transformationen sind die logischen Operationen: `NOT`, `AND`, `OR` und `XOR`. Auch die booleschen Konstanten `TRUE` und `FALSE` werden in einer ähnlichen Form angeboten. In Abbildung 5.12 werden einige dieser Operationen mit booleschen Input- und Output-Ports verwendet.

Da es wenig sinnvoll erscheint, komplexe *(z.B. arithmetische)* Ausdrücke mit so feingranularen Operationen zu modellieren, existiert ein Formel-Shape *(siehe Abbildung 5.13)*, in das der Anwender eine Formel eingeben kann. Alle eingehenden Kanten zu diesem Shape erhalten automatisch einen Variablennamen *(vom Anwender veränderbar)*, mittels dem innerhalb der Formel auf diese Kante verwiesen werden kann. Da alle Kanten

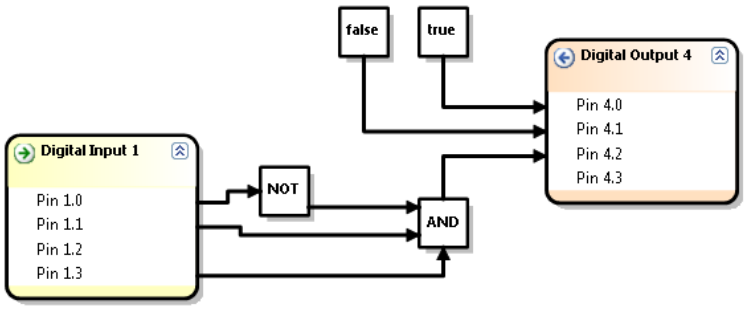


Abbildung 5.12.: Dataflow: boolesche Operatoren

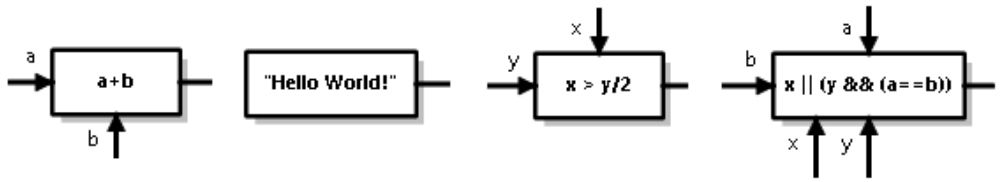
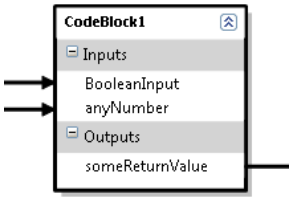


Abbildung 5.13.: Dataflow: Formel-Shapes




```
void Dataflow1_CodeBlock1(
  bool      BooleanInput,
  uint32_t  anyNumber,
  bool*     someReturnValue) {
}
```

Abbildung 5.14.: Dataflow: Codeblock mit dazugehöriger Signatur

typisiert sind, kann innerhalb des Shapes zum einen die Wohlgeformtheit der Formel überprüft werden, andererseits ist es möglich, den Typ des Ergebnisses der Formel zu bestimmen und so die ausgehende Kante ebenfalls korrekt mit einem Typ zu versehen. Verzichtet man auf eingehende Kanten, dann kann das Formel-Shape auch zur Definition von konstanten Ausdrücken *(z.B. Zahlen oder Strings)* verwendet werden. Der Anwender soll mit dem Formel-Shape ähnlich arbeiten können wie er es z.B. aus einer Tabellenkalkulation gewohnt ist.

Wenn eine Aufgabe derart komplex wird, dass sie auch durch die Verwendung von Formel-Shapes nicht zu lösen ist, so bleibt dem Anwender der Weg, C-Code für diese Aufgabe zu schreiben. Auch wenn dies ein wenig vom Paradigma des Modellierens der Anwendung abweicht, stellt es eine durchaus wichtige Anforderung dar. Sie soll zwar nur möglichst selten Einsatz finden müssen, aber würde sie fehlen, so könnte es vorkommen, dass *Flow* für den Anwender nicht weiter nützlich ist und demselben nur die Möglichkeit bleibt, die gesamte Anwendung mit klassischen Tools – und dann meist komplett in C – zu entwickeln. Einem Codeblock muss der Anwender typisierte Ein- und Ausgänge hinzufügen *(siehe Abbildung 5.14)*. Durch Doppelklick auf das Shape wird ein C-Quellcodeeditor geöffnet, in dem bereits eine Methodensignatur vorgegeben ist. Der Anwender muss lediglich den Funktionsrumpf implementieren.

5.3.3. Dataflowmodellierung

In den vorhergehenden Abschnitten wurden die einzelnen Entitäten des Dataflow-Modells beschrieben. Einige dieser Entitäten sind dabei sowohl im Trigger-Bereich als auch im Dataflow-Bereich zulässig. Andere nur in einem dieser Bereiche und einige Entitäten verhalten sich je nach Bereich, in dem sie abgelegt werden, unterschiedlich.

Im Trigger-Bereich sollen nur Daten des Sensorknotens ausgewertet, nicht aber verän-

dert werden, daher sind hier nur Entitäten zugelassen, die lesend arbeiten. Transformationen *(logische Operationen, Formel-Shapes und Codeblöcke)* können an beliebiger Stelle des Modells verwendet werden, da sie zwar Daten verändern, dies aber nur innerhalb des Datenflusses tun und so frei von Seiteneffekten sind.

Der Trigger-Bereich enthält ein weiteres Symbol, das bisher noch nicht beschrieben wurde: den *Master-Trigger (siehe Abbildung 5.15)*. Jedes Dataflow-Modell enthält im Trigger-Bereich immer genau einen Master-Trigger, der vom Anwender weder gelöscht noch neu erzeugt werden kann. Bei der Auswertung des Trigger-Bereichs wird geprüft, ob dieser Master-Trigger einen „Impuls“ erhält, um den Dataflow-Bereich auszuführen. Ein solcher „Impuls“ ist im einfachsten Fall direkt ein Event oder für komplexere Entscheidung ein boolescher Wert, der `TRUE` repräsentiert. Dadurch, dass die Events ebenfalls als boolesche Werte aufgefasst werden können *(die zu `TRUE` ausgewertet werden)* können direkt verschiedene Arten von Entscheidungen im Trigger-Bereich modelliert werden. Abbildung 5.16 zeigt einige Varianten, wie Events im Trigger-Bereich verwendet werden können: Die einfachste Variante (a) löst den Dataflow immer dann aus, wenn das „Button 1 pressed“ Event geschieht. Variante (b) ist auch von diesem Event abhängig, aber der Dataflow wird nur dann ausgelöst, wenn der boolesche Parameter „pressed a long time“ `TRUE` ist. In Variante (c) sind zwei Events vorhanden und der Dataflow wird dann ausgelöst, wenn eine der beiden Kanten am Master-Trigger einen „Impuls“ dazu

Abbildung 5.15.: Dataflow: Master-Trigger

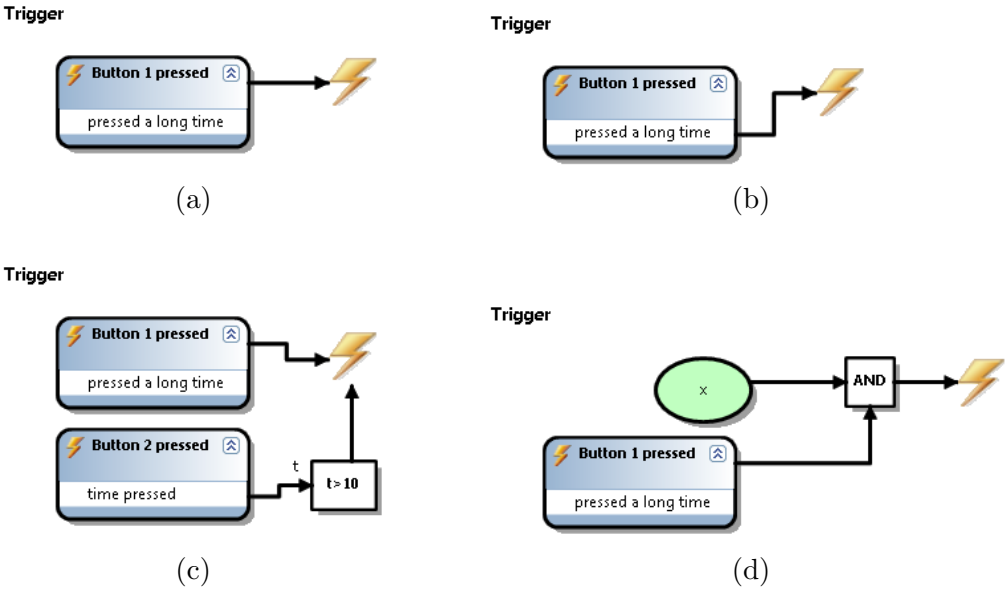


Abbildung 5.16.: Dataflow: Verschiedene Events im Trigger-Bereich

gibt. Dies geschieht entweder beim „Button 1 pressed“ Event oder wenn die angegebene Bedingung unter Verwendung des numerischen Parameters von Event „Button 2 pressed“ wahr ist.

Neben Events können auch andere Elemente Einfluss auf die Ausführung eines Dataflows haben. Abbildung 5.16 (d) zeigt beispielsweise einen Dataflow, der nur ausgeführt wird, wenn „Button 1“ gedrückt wird und die Variable x `TRUE` ist. Außer Variablen hätte man hier u.a. auch Input-Ports verwenden können. Es ist aber nicht möglich, eine Variable im Trigger-Bereich zu verändern oder einem Output-Port einen Wert zuzuweisen.

Ein Event kann durchaus in verschiedenen Dataflow-Modellen verwendet werden, was allerdings die Frage aufwirft, in welcher Reihenfolge diese Dataflows abgearbeitet werden. Ohne weiteres Zutun durch den Anwender ist die Reihenfolge nicht festgelegt. Er kann allerdings jedem Dataflow eine numerische Priorität zuweisen, so dass die Dataflows in aufsteigender Reihenfolge ausgeführt werden. Dadurch kann sichergestellt werden, dass bei gewissen Events beispielsweise zuerst Daten des Sensorknotens verändert und danach – aus einem anderen Dataflow heraus – per Funk verschickt werden.

Auch Records können über Events empfangen und im Trigger-Bereich verwendet werden *(siehe Abbildung 5.17)*. Dabei können entweder die Daten aus einem Record als „Impuls“ für den Master-Trigger herangezogen werden (a) oder nur das Vorhandensein eines Records des gegebenen Typs. Events *(z.B. wenn Funkpakete empfangen wurden)* liefern einen allgemeinen Record, der erst in einen Record eines konkreten Typs gewandelt werden muss. Oftmals soll ein Dataflow nur dann ausgeführt werden, wenn ein Record eines bestimmten Typs empfangen wird. Dazu kann eine Kante aus dem Kopf des Record-Shapes zum Master-Trigger gezogen werden (b). Dieser „Impuls“ wird nur dann gegeben, wenn das Record entsprechend umgewandelt werden kann.

Außer Events können auch Funktionsaufrufe vom Proxy einen Dataflow auslösen.

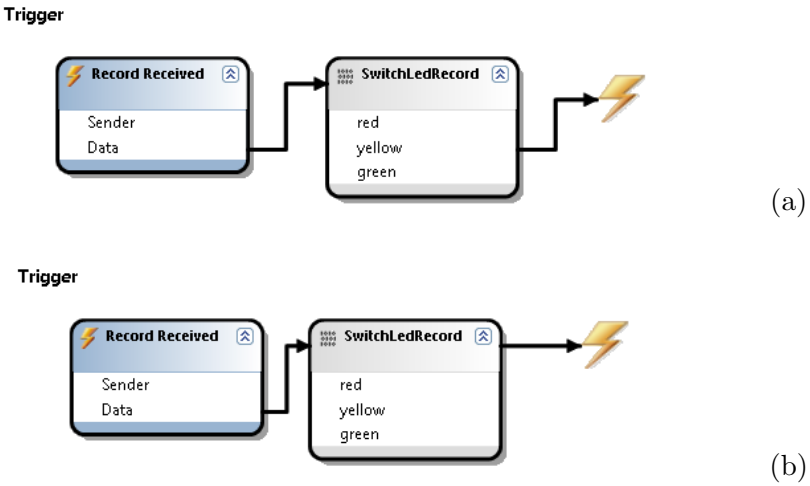


Abbildung 5.17.: Dataflow: Records im Trigger-Bereich

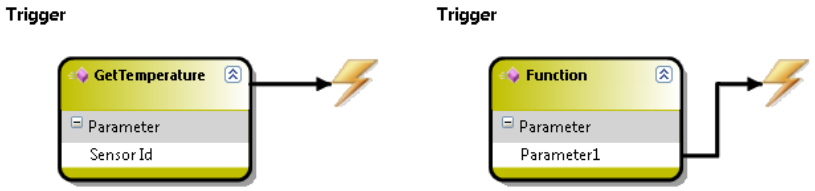


Abbildung 5.18.: Dataflow: Funktion im Trigger-Bereich

Die Funktionsaufrufe verhalten sich genauso wie Events, so dass sie direkt oder über ihre Daten einen „Impuls“ an den Master-Trigger auslösen können *(siehe Abbildung 5.18)*.

Sowohl im Trigger- als auch im Dataflow-Bereich können Kanten immer dann zwischen beliebigen Elementen erzeugt werden, wenn die Datentypen gleich sind. Mittels dieser Kanten werden die Daten von einem Element zum nächsten transportiert, welches diese Daten elementabhängig verarbeitet. Abbildung 5.19 zeigt alle Elemente, die im Dataflow vorkommen können und wie diese mit Kanten verbunden werden dürfen.

Anzumerken ist eine Besonderheit bei Records: Wenn Records geschrieben werden, darf entweder eine Kante auf den Kopf gerichtet sein, um das gesamte Record zu schreiben, oder die einzelnen Felder können geschrieben werden, nicht aber beides. Falls eine Kante das gesamte Record schreiben soll, aber zur Laufzeit festgestellt wird, dass die Typen nicht kompatibel sind, wird der gesamte Dataflow abgebrochen[1].

Die logischen Operationen können je nach Art nur eine gewisse Anzahl an eingehenden Kanten besitzen: `NOT` nur eine, `XOR` zwei und `AND` sowie `OR` jeweils beliebig viele. Auch Formeln und der Master-Trigger können beliebig viele eingehende Kanten besitzen, alle anderen Entitäten jeweils nur eine.

Es wird strikt zwischen dem Trigger- und dem Dataflow-Bereich eines Dataflow-Modells getrennt, so dass es nicht möglich ist, Kanten zwischen Elementen dieser beiden Bereiche zu erstellen. Es ist aber dennoch in gewissen Situationen notwendig, Daten vom Trigger- in den Dataflow-Bereich zu übermitteln, und zwar immer dann, wenn Events oder Funktionen mit Parametern im Trigger-Bereich verwendet werden.

Legt ein Anwender ein Event mit Feldern im Trigger-Bereich ab, so wird das gleiche Event automatisch auch im Dataflow-Bereich erzeugt, auf dessen Daten dort zugegriffen werden kann. Sollten mehrere Events im Trigger-Bereich verwendet werden, so ist es nicht möglich, diese Daten in den Dataflow-Bereich zu übergeben. In diesem Fall müssen alle Events aus dem Dataflow-Bereich entfernt werden.

Wenn der Anwender eine Funktion des Proxies im Trigger-Bereich ablegt, so wird sie, wenn sie Parameter enthält, wie ein Event auch im Dataflow-Bereich erzeugt. Sollte sie darüber hinaus Rückgabewerte definiert haben, so wird ein weiteres Shape für die Rückgabewerte im Dataflow-Bereich angelegt, das der Anwender mit Daten füllen

[1] Da keine Möglichkeit existiert, diesen Fehler weiter zu geben, passiert dies während der Ausführung unbemerkt.

muss *(siehe Abbildung 5.11)*. Es können zwar mehrere Events in einem Trigger-Bereich verwendet werden, aber nur maximal eine Funktion. Das gleichzeitige Verwenden von Events und Funktionen ist ausgeschlossen.

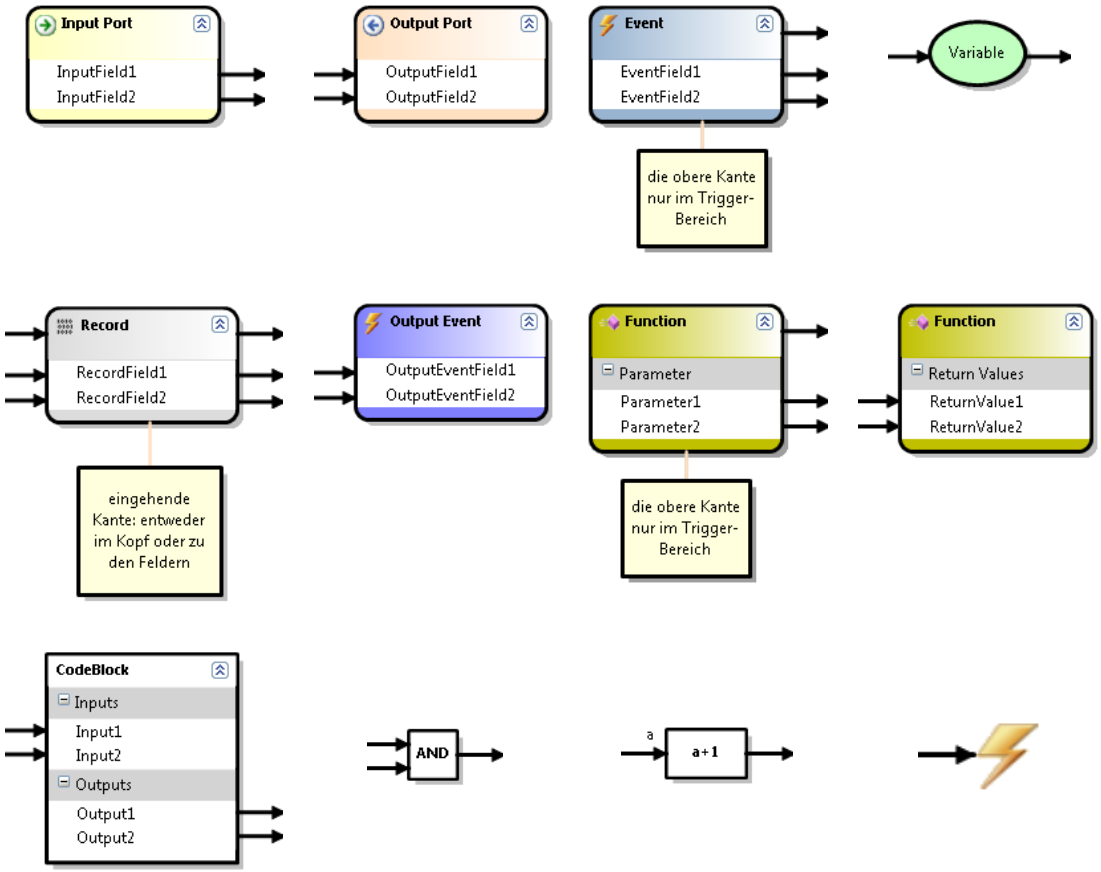


Abbildung 5.19.: Alle Shapes des Dataflows und die Positionen, an denen Kanten angelegt werden können

6. Softwarearchitektur

Neben den im letzten Kapitel beschriebenen domainspezifischen Sprachen besteht *Flow* aus einigen weiteren Komponenten, die zusammen mit den DSLs eine vollständige Software Factory bilden. Viele dieser Komponenten erweitern die durch die DSL Tools erzeugten Sprachen und Editoren oder Visual Studio. So ist der Spielraum für Architekturentscheidungen bereits grundlegend eingeschränkt, da sich die Komponenten an die Vorgaben der vorhandenen Schnittstellen und Frameworks halten müssen.

Im folgenden Abschnitt werden diese Komponenten zunächst kurz vorgestellt und ihr Zusammenspiel dargestellt. Ab Kapitel 6.2 werden dann die einzelnen Komponenten detailliert beschrieben und ihr internes Design sowie z.T. Implementierungsdetails erklärt.

6.1. Komponenten von Flow

Zunächst sollen die logischen Komponenten von *Flow* beschrieben werden, bevor das technische Design dieser Komponenten in den folgenden Abschnitten erläutert wird. Abbildung 6.1 zeigt, wie die einzelnen Teile von *Flow* zusammenspielen, um in Visual Studio eine vollständige Entwicklungsumgebung zur modellgetriebenen Programmierung von Sensorknoten anzubieten.

In Abbildung 6.1 sind die Bereiche, die vom Hardwarehersteller bzw. vom Anwender genutzt werden, farbig hinterlegt. Die Hardwaredescription-DSL kann in beiden Bereichen gefunden werden, da sie von beiden Anwendertypen verwendet wird. Zur Übersichtlichkeit ist sie im Diagramm zweimal dargestellt.

Der Hardwarehersteller benutzt den Firmwareparser *(Kapitel 6.4)* um ein Hardwaredescription-Modell aus der in C geschriebenen Firmware zu erzeugen. Dazu muss die Firmware von ihm entsprechend annotiert worden sein *(Abschnitt 6.4.1)*. Er kann dieses Modell im Modelleditor weiter verfeinern und als *Plattformsetup* exportieren. Falls er statt einer Hardwareplattform einen Treiber für Zusatzhardware entwickelt, exportiert er stattdessen ein *Driver Packet*.

Das Plattformsetup und die Driver Packets werden an den Anwender gegeben, der diese auf seinem PC installiert. Dort bilden sie für ihn die Basis von Sensorknotenprojekten. Wenn der Anwender ein neues Sensorknotenprojekt in Visual Studio anlegt, ist die Hardwaredescription bereits vorgegeben *(es wird das Modell verwendet, welches der Hardwarehersteller mit dem Plattformsetup geliefert hat)*. Er selbst kann Datastructure- und Dataflow-Modelle erzeugen und seine Anwendung wie gewünscht modellieren. Dazu

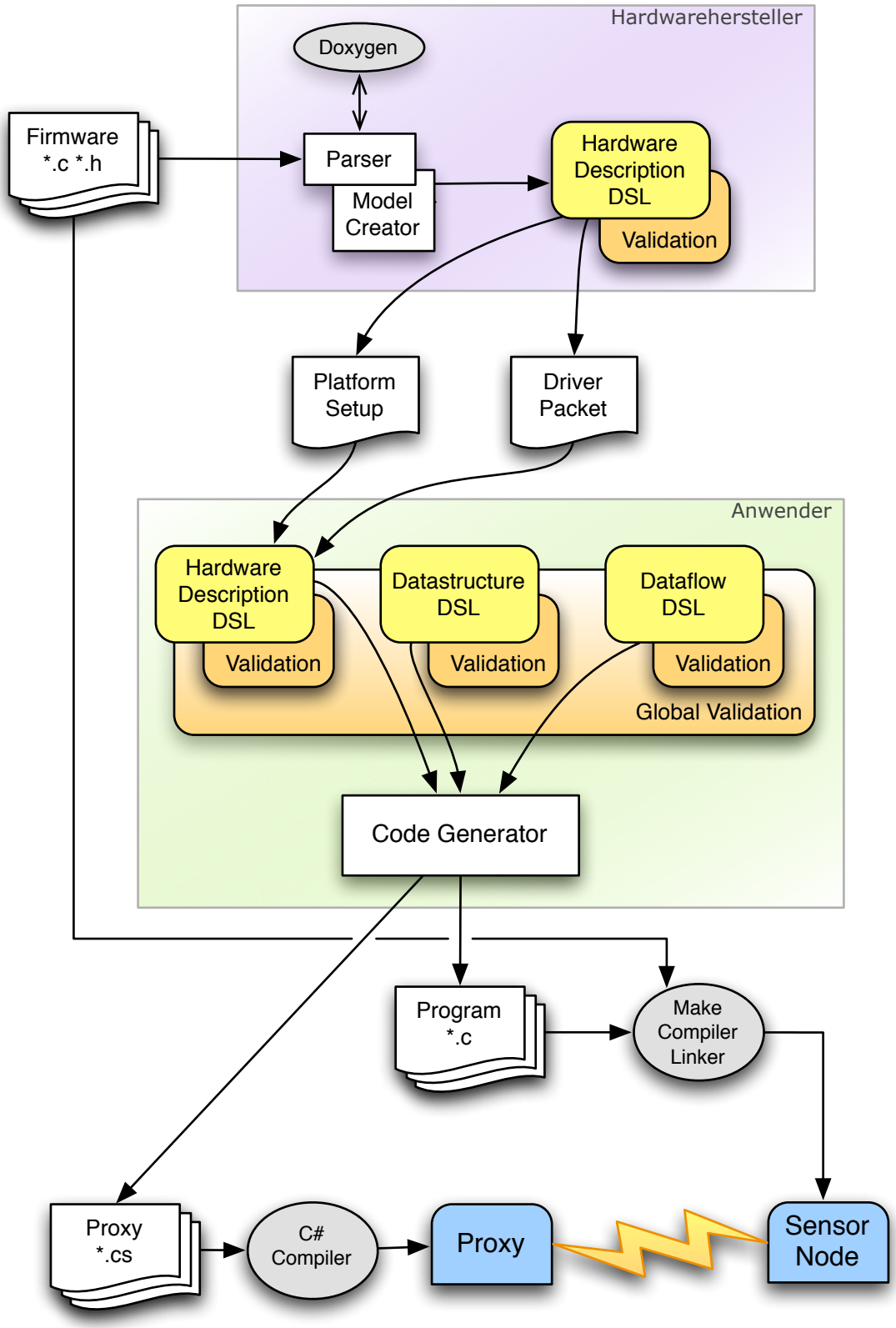


Abbildung 6.1.: Zusammenspiel der logischen Komponenten

werden die entsprechenden DSLs sowie die darin eingebetteten Validierungsregeln verwendet *(Kapitel 6.2)*. Neben der Validierung der einzelnen Modelle wird ein zusätzlicher Validierungsschritt benötigt, der alle Modelle im Kontext des Projektes überprüft. Dieser Validierungsschritt wird von der *Global Validation (Kapitel 6.5)* durchgeführt.

Der *Codegenerator (Kapitel 6.7)* erzeugt aus den Modellen eines Projektes Quellcode für die Sensorknoten. Dieser Quellcode wird zusammen mit der Firmware *(die ebenfalls als Teil des Plattformsetups geliefert wurde)* kompiliert und kann auf den Sensorknoten ausgeführt werden. Der Codegenerator erzeugt auch den Code für den Proxy *(Abschnitt 6.7.1)*, der mittels des C#-Compilers zu einer Bibliothek für PC-Anwendungen kompiliert wird.

Eine weitere Komponente von *Flow* ist in der Abbildung 6.1 nicht dargestellt, da sie Querschnittsfunktionalität enthält. Das `UiSupportPackage` *(Kapitel 6.6)* steuert die Integration aller gezeigten Komponenten in Visual Studio. Dort sind die Menübefehle sowie die Logik implementiert, um Plattformsetups und Driver Packets sowohl zu exportieren *(durch den Hardwarehersteller)*, als auch zu importieren *(durch den Anwender)*. Des Weiteren sind im `UiSupportPackage` die Elemente der Benutzerschnittstelle definiert, die es dem Anwender erlauben, dem Projekt neue Modelle hinzuzufügen und die Global Validation sowie den Codegenerator auszuführen.

Die hier erwähnten Komponenten sind innerhalb von *Flow* in 14 Assemblies implementiert, die in den folgenden Abschnitten detailliert beschrieben werden. Des Weiteren sind im Rahmen dieser Arbeit drei Bibliotheken entstanden, die allgemeine Funktionen anbieten, um mit den DSL Tools und Visual Studio zu arbeiten. Auch wenn diese Bibliotheken nicht direkt zu *Flow* zu zählen sind, sollen sie der Vollständigkeit halber in Kapitel 6.9 kurz vorgestellt werden.

6.2. Domainspezifische Sprachen

Wie bereits in Abschnitt 3.3.2 beschrieben, erzeugt der Codegenerator der DSL Tools für jede der in Kapitel 5 beschriebenen Sprachen je zwei Assemblies:

- `HardwaredescriptionDsl` und `HardwaredescriptionDslPackage`
- `DatastructuresDsl` und `DatastructuresDslPackage`
- `DataFlowDsl` und `DataFlowDslPackage`

Die jeweiligen `Dsl`-Assemblies enthalten die Datenstrukturen der domainspezifischen Sprachen sowie die Shapes zur graphischen Repräsentation. In diesen Assemblies ist ebenfalls der Code enthalten, um die Modelle aus Dateien lesen und wieder speichern zu können. Auch große Teile der Modelleditoren sind hier zu finden.

Die `DslPackage`-Assemblies enthalten Code, um diese Editoren in Visual Studio zu integrieren. Dabei handelt es sich vor allem um das Bekanntgeben der Editoren für die

Dateitypen[1] der Modelle und das Erzeugen von neuen Menübefehlen sowie Symbolleistenbuttons, um mit den Editoren zu arbeiten.

Ein Großteil des Codes, der für *Flow* per Hand geschrieben wurde, ist ebenfalls in diesen Assemblies enthalten, da so die generierten Klassen um weitere Funktionalität ergänzt werden *(z.B. die Darstellung der Shapes und das Verhalten der Editoren)*. Bevor auf die Struktur der Assemblies eingegangen wird, sollen allerdings die Extension Points, die durch die DSL Tools vorgesehen sind, genauer beschrieben werden.

Es wird bei nahezu allen generierten Klassen Gebrauch von dem C#-Feature der partiellen Klassen gemacht: Eine Klasse kann sich in C# über mehrere Quellcodedateien erstrecken und in jeder dieser Dateien kann Code enthalten sein, der durch den Compiler zu einer einzigen Klasse zusammengesetzt wird. Dazu muss die Klassendeklaration mit den Schlüsselwörtern `partial class` eingeleitet werden. So kann eigener Code, der z.B. weitere Methoden einer generierten Klasse hinzufügt, in eine separate Datei ausgelagert werden. Dies erhöht nicht nur die Übersichtlichkeit, es erlaubt auch problemlos jederzeit den DSL-Code neu zu generieren *(z.B. nach Änderungen am Metamodell)*, da hierdurch niemals der Code in der zusätzlichen Datei überschrieben wird.

Nicht immer reicht es aus, den generierten Klassen Code hinzuzufügen, da man gelegentlich auch generiertes Verhalten austauschen möchte. Dazu bieten die DSL Tools das sogenannte „Double-derived"-Pattern *(vgl. [CJKW07, S. 398f.])* an, um es dem Programmierer zu erlauben, jede beliebige Methode einer generierten Klasse zu überschreiben und so zu ergänzen oder vollständig durch eigenen Code zu ersetzen. Wendet man dieses Pattern an, so wird beispielsweise statt der Klasse `AbcShape` eine Klasse `AbcShapeBase` mit sämtlichem Code generiert. Zusätzlich existiert eine leere Klasse `AbcShape` die von `AbcShapeBase` erbt und im übrigen Code verwendet wird. Der Programmierer kann nun in der partiellen Klasse `AbcShape` alle Methodenaufrufe überschreiben.

Normalerweise strukturiert man Software in einer Klassenhierarchie *(und meist auch Dateihierarchie)*, die der Problemstellung nahe kommt, allerdings ist die Klassenhierarchie bei generiertem Code bereits vorgegeben. Bei den DSL Tools werden Domain Classes und Domain Properties zur Verwaltung der Daten sowie Shapes zur graphischen Repräsentation erzeugt. Alle weiteren Funktionen müssen sich in diese Hierarchie einordnen. Betrachtet man die Menge von gewünschten Erweiterungen, so bemerkt man, dass verschiedene Features zum Teil in den gleichen Klassen implementiert werden müssen. Um weiterhin eine gewisse Struktur zu erhalten, wurde der Quellcode in diesen Assemblies auf eine eher ungewöhnliche Art und Weise strukturiert: Für alle Features, die den generierten Code berühren, wurden Verzeichnisse *(unterhalb von `UserCode`)* angelegt. In diesen Verzeichnissen sind alle Quellcodedateien abgelegt, die zusammen dieses Feature implementieren, auch wenn dieselben Klassen bereits an anderen Stellen

[1] Visual Studio entscheidet aufgrund der Dateiendungen, in welchem Editor Dateien geöffnet werden.

Datei	Implementierte Funktion
`GeneratedCode\ DomainClasses.cs`	Der generierte Code als Basis für die Klasse. Die Klasse `Variable` erbt von `EnableableShapeMappedDomainClass` und definiert hier u.a. die Domain Properties aus dem Metamodell.
`UserCode\CallParameter\ Variable.cs`	Dieser Code fügt der Klasse zwei weitere Properties *(`CallParameterGetter` und `CallParameterSetter`)* hinzu, die intern an einigen Stellen benötigt werden.
`UserCode\DataTypes\ Variable.cs`	Code, um die Domain Property `Type` zu initialisieren.
`UserCode\Restrictions\ CanDelete.cs`	Hier wird ein zusätzliches Interface `IDynamicCanDelete` implementiert, über das das Löschen von Variablen aus dem Modell verhindert werden kann. *(Dies ist notwendig, wenn der Anwender statt des Hardwareherstellers mit dem Modell arbeitet.)*
`UserCode\Restrictions\ Properties.cs`	In dieser Datei werden der Klasse lediglich Attribute hinzugefügt, die steuern, welche Domain Properties vom Anwender bzw. vom Hardwarehersteller gesehen und bearbeitet werden können.
`ValidationRules\ Variable.cs`	Mehrere Regeln, um die statische Semantik von Variablen im Modell zu prüfen.

Tabelle 6.1.: Dateien, die Code der Klasse `Variable` enthalten

um anderen Code erweitert wurden *(was durch partielle Klassen möglich ist)*.

In dem Assembly `HardwareDescriptionDsl` ist der Code der Klasse `Variable` *(dies ist die Domain Class aus dem DSL-Metamodell)* über sechs Dateien verteilt. Dies soll als Beispiel in Tabelle 6.1 näher betrachtet werden. In den einzelnen Verzeichnissen befinden sich neben dem Code der Klasse `Variable` auch weitere Dateien, so dass beispielsweise aller Code für alle Klassen, der Einschränkungen des Editors behandelt, in dem Verzeichnis `UserCode\Restrictions` gefunden werden kann.

Eine vollständige Liste aller auf diese Weise implementierten Funktionen kann in Anhang B gefunden werden. Viele der so implementierten Funktionalitäten steuern das Verhalten der DSL-Editoren bzw. die Darstellung der Shapes, da in Kapitel 5 zum Teil Anforderungen gestellt wurden, die nicht allein durch die Modellierung des Metamodells erfüllt werden können. Des Weiteren sind in diesen Assemblies die Validierungsregeln der statischen Semantik implementiert, die in einer vollständigen Liste im Anhang C angegeben sind.

6.2.1. Visual Studio Integration

Neben den Editoren für die DSLs integrieren sich diese Packages an weiteren Stellen in Visual Studio.

Das `HardwareDescriptionPackage` registriert eine *OptionsPage*, die im Options-

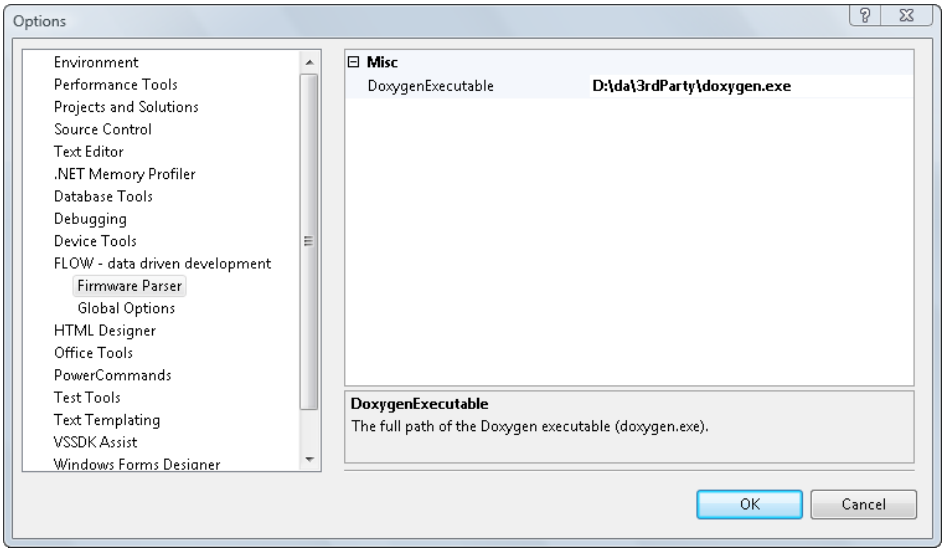


Abbildung 6.2.: Visual Studio Options-Dialog mit der OptionsPage der Hardwarebeschreibung

Dialog von Visual Studio angezeigt wird und es dem Anwender ermöglicht, den Pfad zu der Datei `Doxygen.exe` anzugeben *(siehe Abbildung 6.2)*. Eine weitere OptionsPage wird vom `UiSupportPackage` bereitgestellt.

Einträge in Menüs *(auch Kontextmenüs)* und Symbolleisten werden in Visual Studio durch *Commands* repräsentiert. Diese Commands werden in einer XML-Struktur *(`.vsct`-Dateien)* definiert. In einem beliebigen Package kann ein *CommandHandler* registriert werden, um auf diese Commands zu reagieren. Dadurch wird eine Trennung der Oberfläche von der Implementierung erreicht. Außerdem ist es Visual Studio so möglich, das Package erst dann in den Speicher zu laden, wenn ein Command tatsächlich benötigt wird *(lazy loading)*. Solange der Anwender beispielsweise einen Menübefehl nicht verwendet, muss das entsprechende Package nicht geladen werden; sobald es allerdings einmal geladen wurde, verbleibt es aus Performancegründen im Speicher. Durch die Packages der DSLs sind fünf Commands, die in Tabelle 6.2 aufgelistet sind, definiert.

6.2.2. Services

Neben den Commands, die vom Benutzer verwendet werden, sieht die Komponentenarchitektur von Visual Studio es vor, sogenannte *Services* anzubieten, um über die Grenzen von Komponenten *(meist Packages)* hinweg Dienste anzubieten. Diese Services spezifizieren ihre öffentlichen Schnittstellen und werden bei Visual Studio registriert. Beliebige andere Komponenten können diese Services konsumieren, ohne den Anbieter eines Services zu kennen, lediglich die Schnittstelle muss bekannt sein. Um diese Trennung von Schnittstellen und Serviceanbietern auch entsprechend umzusetzen, sind die meisten Schnittstellen der Services in dem `ServciceInterfaces`-Assembly beheimatet. Die Services, die *Flow* definiert, sind in Tabelle 6.3 aufgelistet.

im `HardwareDescriptionDslPackage`:	
„Show Hardware Description Explorer“	Im Kontextmenü des Modells, um das Hardware Description Explorer Toolwindow zu öffnen.
„Show Connectors“	Im Kontextmenü des Modells, um die Kanten ein- bzw. auszublenden.
„Import Driver...“	Im Kontextmenü des Modells, um ein Driver Packet zu importieren *(nur für Anwender, nicht für Hardwarehersteller).*
im `DatastructuresDslPackage`:	
„Show Datastructure Explorer“	Im Kontextmenü des Modells, um das Datastructure Explorer Toolwindow zu öffnen.
im `DataFlowDslPackage`:	
„Open CodeBlock Editor...“	Im Kontextmenü der CodeBlock-Shapes, um den Quellcode des Shapes in einem C-Editor zu öffnen.

Tabelle 6.2.: Durch die *Flow*-Packages definierte Commands

Interaktion der DSLs

Die drei *Flow*-DSLs bauen aufeinander auf, so dass beim Arbeiten mit den Datastructures die Daten aus der Hardwaredescription und beim Arbeiten mit den Dataflows die Daten aus der Hardwaredescription und den Datastructures benötigt werden. Leider stellen die DSL Tools keine Möglichkeiten bereit, um innerhalb eines Projektes von einer DSL auf eine andere zugreifen zu können. Außerdem muss definiert werden, wie mit mehreren Modellen eines Typs in einem Projekt umgegangen werden soll: In *Flow* darf jedes Sensorknotenprojekt nur eine Hardwaredescription, aber mehrere Datastructure-Modelle enthalten. Zwei Services liefern für ein gegebenes Sensorknotenprojekt die Hardwaredescription bzw. alle in den Datastructures definierten Elemente.

Diese beiden Data-Services sollen keine Instanzen der durch die DSL Tools generierten Domain Classes liefern, da diese Domain Classes zu viel Logik enthalten, um zum Datenaustausch geeignet zu sein. Würde man extern mit diesen Klassen arbeiten, so wären Seiteneffekte auf die Ursprungsmodelle nicht auszuschließen, da die Daten der Domain Classes nicht direkt in Variablen innerhalb der Klassen, sondern in einem transaktionalen Container *(implementiert durch die Klasse `Store`)*, gespeichert sind. Insbesondere führt jede Veränderung der Properties außerhalb einer Transaktion zu einer Exception; Änderungen innerhalb der Transaktion würden aber u.U. Auswirkungen auf das Modell haben.

Daher wurden Datenklassen erzeugt, die was die Domain Properties betrifft das gleiche Interface wie die generierten Domain Classes besitzen, ihre Daten aber lediglich speichern, keine weitere Logik enthalten und so zur Datenweitergabe geeignet sind. Diese Daten-

Service	**definiert in**	**implementiert in**	**konsumiert von**
IHardwareDescriptionDataService	HardwareDescriptionData	HardwareDescriptionDslPackage	DatastructuresDslPackage DataFlowDslPackage
IFirmwareParser	ServiceInterfaces	GlobalValidation *(registriert im UiSupportPackage)*	UiSupportPackage
IDatastructureDataService	DatastructuresData	DatastructuresDslPackage	DataFlowDslPackage
IGlobalValidation	ServiceInterfaces	UiSupportPackage	HardwareDescriptionDslPackage DatastructuresDslPackage DataFlowDslPackage UiSupportPackage
ICodeBlockEditor	DataFlowDsl	UiSupportPackage	DataFlowDslPackage
IImportDriverService	ServiceInterfaces	UiSupportPackage	HardwareDescriptionDslPackage

Tabelle 6.3.: Durch *Flow* definierte Services

klassen sind in zwei Assemblies *(`HardwareDescriptionData` und `DatastructuresData`)* ausgelagert, um weniger Abhängigkeiten zwischen den einzelnen Packages zu erzeugen. In diesen Assemblies sind auch die Service-Interfaces `IHardwareDescriptionDataService` und `IDatastructureDataService` definiert.

Auch diese Klassen wurden nicht per Hand implementiert, sondern mittels eines eigenen Codegenerators[2] aus den Metamodellen generiert. Auch die Logik, die diese Klassen aus den Domain Classes füllt, wurde generiert, so dass bei Änderungen am Metamodell auch diese Datenklassen synchron dazu bleiben.

Immer wenn ein Dataflow-Modell geöffnet wird, werden über die beschriebenen Data-Services die zugrundeliegenden Modelle aus dem Projekt geladen[3]. Im Anschluss werden alle Elemente des Dataflow-Modells daraufhin untersucht, ob sie den Definitionen in den zugrundeliegenden Modellen entsprechen. Zu diesem Zweck kann die Guid eines jeden Elements der DSLs zur Zuordnung verwendet werden. Der Name wäre zur Zuordnung nicht geeignet, da es dem Anwender frei steht, die Namen in der Hardwaredescription zu verändern. Sollten Unterschiede gefunden werden, so wird das Dataflow-Modell entsprechend verändert. Da dies – z.B. wenn Elemente gelöscht wurden – zu einer Veränderung der Logik des Modells führen kann, muss der Anwender das veränderte Modell kontrollieren und es manuell speichern, um die Änderungen zu akzeptieren. Dieser Mechanismus ist zum einen dazu geeignet, Dataflows nach Veränderungen am Hardwaredescription-Modell *(durch den Anwender)* zu aktualisieren, kann zum anderen aber auch verwendet werden, wenn die Hardwareplattform – beispielsweise durch eine aktualisierte und erweiterte Version – ersetzt wird.

Die Datastructure-Modelle sind ebenfalls von der Hardwaredescription abhängig, so dass hier ein ähnliches Verfahren eingesetzt wird. Allerdings sind diese beiden Modelltypen nicht so stark miteinander verbunden. Es müssen lediglich die Datentypen, die im Hardwaredescription-Modell angegeben sind, in das Datastructure-Modell übernommen werden.

Ebenfalls beim Öffnen von Dataflow-Modellen muss die Toolbox des Modelleditors gefüllt werden. In der Toolbox sollen alle Elemente, die in der Hardwaredescription oder den Datastructures definiert sind, angezeigt werden, um es dem Anwender zu ermöglichen, diese Elemente in das Dataflow-Modell zu übernehmen. Auch hierfür werden die Daten verwendet, die die Data-Services liefern. Da das Manipulieren der Visual Studio Toolbox relativ zeitaufwändig ist, werden nur dann neue Elemente in der Toolbox zu erzeugen, wenn dies nötig ist. Um dies zu erkennen, wird die Modellversion herangezogen, die in den Hardwaredescription- und Datastructure-Modellen geführt und bei jedem Speichern erhöht wird.

[2] Dieser Codegenerator basiert auf dem der DSL Tools, verwendet aber ein angepasstes Template.
[3] Sollte das zwingend notwendige Hardwaredescription-Modell nicht existieren, wird eine Fehlermeldung ausgegeben.

6.3. Gemeinsame Datenstrukturen

Einige wenige Datentypen werden von allen DSLs und weiteren Programmteilen verwendet. Dies ist z.B. ein spezieller .NET-Datentyp, um die Typ-Eigenschaft von Variablen und Feldern in den DSLs zu verwalten. Er enthält des weiteren die Logik, um im Properties Window einen Wert aus einer Vorgabenliste auswählbar zu machen. Diese Typen sind im `FlowCommonTypes`-Assembly definiert.

6.4. Firmware Parser

Der Firmware-Parser wird von den Hardwaredescription-Assemblies verwendet, um den C-Quellcode der Sensorknotenfirmware einzulesen und daraus ein Hardwaredescription-Modell zu erzeugen.

Als Parser wird Doxygen *(siehe Kapitel 3.2)* verwendet, der für gewöhnlich HTML-Dokumentation erzeugt. Allerdings kann Doxygen so gesteuert werden, dass er seine Ausgabe in *(mehreren)* XML-Dateien erzeugt und weitere Annotationen, als die standardmäßig definierten, aus dem Quellcode ausliest. Die Ausgabe von Doxygen wird mittels einer XSL-Transformation in ein Format überführt, das für die Weiterverarbeitung mit *Flow* geeignet ist und als Objektstruktur *(in der Klasse `CodeDescription`)* in den Speicher geladen werden kann. Diese Daten werden an den `ModelCreator` *(ein Teil der `HardwareDescriptionDsl`)* übergeben, der aus den Methodendefinitionen und den Annotationen das Hardwaredescription-Modell erzeugt. Der beschriebene Ablauf ist in Abbildung 6.3 dargestellt.

Sollten beim Parsen durch Doxygen oder in der nachgeschalteten Validierungsphase Fehler oder Verstöße gegen die Konventionen auftreten, so werden entsprechende Mel-

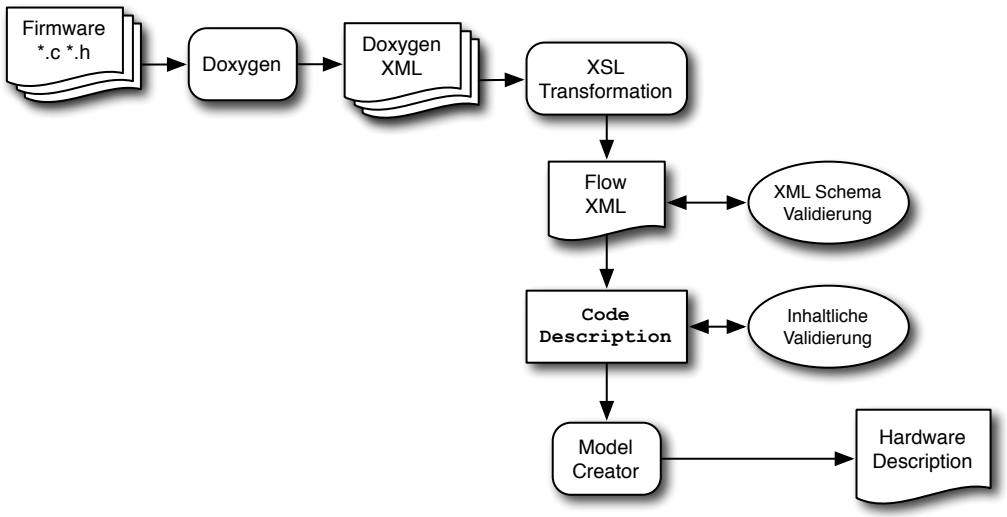


Abbildung 6.3.: Ablauf von der Firmware bis zum Hardwaredescription-Modell

Quellcodeelement	Annotationen
Datei	`@firmwarename`, `@drivername`, `@description`, `@productname`, `@producturl`, `@vendorname`, `@vendorurl`, `@version`
Methode	`@getVariable`, `@setVariable`, `@input`, `@inputInit`, `@output`, `@outputInit`, `@event`, `@eventCallback`, `@multiplicity`, `@description`
Parameter	`@default`, `@enabled`, `@field`, `@fieldenabled`, `@parameter`, `@description`

Tabelle 6.4.: Annotationen, die verwendet werden können, um den Firmware-Quellcode zu beschreiben

dungen ausgegeben anstatt ein inkonsistentes Modell zu erzeugen. Die Regeln, die hier überprüft werden, sind in Anhang C.5 aufgelistet.

6.4.1. Quellcodeannotationen

Für die Modellerzeugung werden die in Tabelle 6.4 aufgelisteten Annotationen aus dem Quellcode ausgelesen. Ein Beispiel-Quellcode ist in Abbildung 6.4 angegeben.

Auf Dateiebene können einige Annotationen verwendet werden, um Metainformationen zu der Firmware zu hinterlegen. Je nachdem, ob `@firmwarename` oder `@drivername` verwendet wird, wird entweder eine Hardware- oder Treiberplattform in der Hardwaredescription erzeugt.

Aus Methoden werden je nach Annotation Input-Ports, Output-Ports, Events oder Variablen erstellt. Die Methodenparameter können in den Shapes als Felder *(`@field`)* oder Parameter *(`@parameter`)* wiedergefunden werden. Ist ein Methodenparameter mit `@enabled` oder `@fieldenabled` versehen, so wird dieser Methodenparameter nicht im Shape der Hardwaredescription angezeigt. Stattdessen wird bei der Initialisierung des Sensorknotens über diese Methodenparameter der Firmware mitgeteilt, welche Shapes bzw. Felder durch den Anwender aktiviert wurden. Die aus dem Quellcode erzeugten Elemente sind ebenfalls in Abbildung 6.4 zu sehen.

6.5. Globale Validierung

Die DSL Tools haben bereits Mechanismen vorgesehen, um Validierungsregeln innerhalb der Domain Classes zu implementieren und ausführen zu lassen. So werden die Modelle vor dem Speichern mit diesen Regeln geprüft und Fehler in der Error-List von Visual Studio angezeigt. Die Regeln *(die in Anhang C aufgelistet sind)* können jeweils nur ein Modell untersuchen. Doch in *Flow* müssen zusätzlich modellübergreifende Bedingungen überprüft werden. Beispielsweise muss sichergestellt sein, dass in einem Sensorknotenprojekt nur eine Hardwaredescription enthalten ist und dass keine Elemente mit gleichen Namen in mehreren Datastructure-Modellen definiert sind.

```
/** \file
    @firmwarename{MSB430H}
    @description{Flow-Firmware for the MSB430H sensor node}
    @version{1.0} */

/// @getVariable{Node Id}
bool GetNodeId(uint32_t* id);
/// @setVariable{Node Id}
bool SetNodeId(uint32_t id);

/// @input{Digital Input 1}
/// @param P10 @field{Pin 1.0}
/// @param P11 @field{Pin 1.1}
bool DigitalIO_P1_Get(bool* P10, bool* P11);
/// @inputInit{Digital Input 1}
/// @param P10 @fieldenabled{Pin 1.0}
/// @param P11 @fieldenabled{Pin 1.1}
bool DigitalIO_P1_Get_Init(bool P10, bool P11);

/// @output{Serial Port Output}
/// @field{Text}
void PrintLine(char* line);
/// @outputInit{Serial Port Output}
/// @param enabled @enabled
/// @param baudrate @parameter{Baudrate} @default{115200}
bool UART_Output_Init(bool enabled, uint32_t baudrate);

/// @event{Periodic Timer}
/// @multiplicity{4}
/// @param ticks @parameter{delay in ticks} @default{8192}
    @description{8192 ticks are 1 second)}
/// @param timerHandler
bool PeriodicTimers_Init(uint32_t ticks, void* timerHandler);
/// @eventCallback{Periodic Timer}
bool PeriodicTimers_Callback();
```

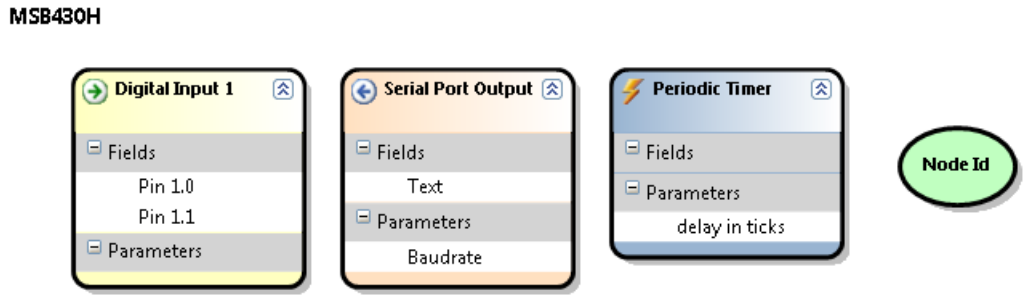


Abbildung 6.4.: Firmware-Quellcode mit beispielhaften Annotationen und dem daraus erstellten Hardwaredescription-Modell

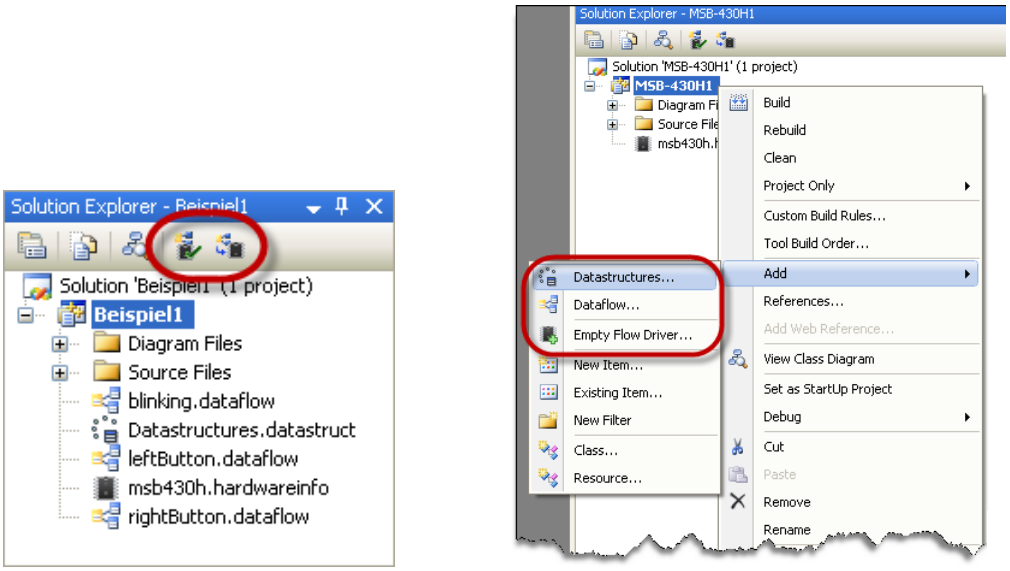
Im `GlobalValidation`-Assembly wird der `GlobalValidationService` angeboten, der das gesamte Projekt, welches in Visual Studio ausgewählt ist, überprüfen kann. Dazu werden sämtliche Modelle der drei Typen ermittelt und ein Regelsystem verwendet. Die Klassen, die diese Regeln implementieren, erhalten jeweils alle Modelle des Projektes, prüfen eine Bedingung und melden bei Bedarf Fehler. Eine Liste aller implementierten Regeln ist in Anhang C.4 zu finden. Nachdem diese globalen Regeln geprüft wurden, werden zusätzlich die übrigen Regeln durch den internen Validierungsmechanismus der DSL Tools ausgewertet.

6.6. Integration in Visual Studio

Alle Funktionalität von *Flow*, die mit der Benutzerschnittstelle von Visual Studio interagiert, ist im `UiSupportPackage` implementiert. Für einen Teil der Funktionen stellt das Package lediglich die Commands bereit und leitet die Aufrufe an die bereits zuvor beschriebenen Services weiter. Andere Funktionalitäten, insbesondere zur Projektverwaltung, sind direkt in diesem Package enthalten.

Dem Kopf des Solution Explorers wurden zwei Icons hinzugefügt *(siehe Abbildung 6.5 (a))*. Mit dem linken wird die Validierung des gesamten Projektes *(Kapitel 6.5)* ausgelöst. Der rechte Button startet den Codegenerator *(Kapitel 6.7)*.

Auch dem Kontextmenü des Solution Explorers wurden mehrere *(kontextabhängige)* Befehle hinzugefügt. Der Anwender kann über das Add-Menü *(Abbildung 6.5 (b) im Kontextmenü; aber auch in der Hauptmenüleiste von Visual Studio)* ein neues Datastructure-



(a) (b)

Abbildung 6.5.: (a) Buttons zum Validieren des Projektes und zur Codegenerierung
(b) Kontextmenü, um dem Projekt Elemente hinzuzufügen

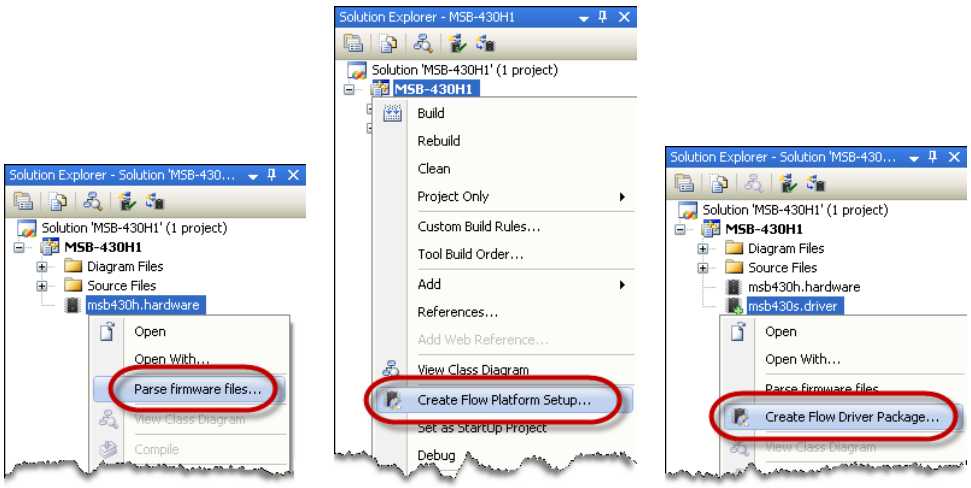


Abbildung 6.6.: Weitere Kontextmenübefehle für den Hardwarehersteller

bzw. Dataflow-Modell dem Sensorknotenprojekt hinzufügen. Über diese Befehle wird sowohl ein leeres Modell, als auch ein Template zur Codegenerierung dem Projekt hinzugefügt. Der Hardwarehersteller kann zusätzlich einen neuen leeren Treiber anlegen; für Anwender ist dieser Befehl nicht sichtbar.

Die weiteren Befehle im Kontextmenü des Solution Explorers sind nur für den Hardwarehersteller relevant *(siehe Abbildung 6.6)*. Das Command „Parse firmware files“ ermöglicht es den Firmware-Parser *(Kapitel 6.4)* zu verwenden, um ein vorhandenes Modell mit den Daten aus der Firmware zu ergänzen bzw. zu überschreiben. Das Command „Create Flow Platform Setup“ erzeugt aus einem Projekt ein Installationspaket, um die Hardwareplattform an den Anwender zu geben. Das Paket enthält alle Dateien des Projektes und installiert die Hardwareplattform im New-Project-Dialog von Visual Studio auf dem System des Anwenders. Zur Erzeugung dieses Setups wird das Nullsoft Installer System *(siehe Kapitel 3.4)* verwendet. Auch Treiber können *(mit dem „Create Flow Driver Package“-Command)* exportiert werden, um sie an den Anwender zu geben *(siehe Abschnitt 4.2.1)*.

Wird ein Modell innerhalb von Visual Studio umbenannt, muss auch das Text-Template umbenannt und innerhalb des Templates der Dateiname des Modells angepasst werden. Wird ein Modell aus dem Projekt entfernt, muss auch das Template gelöscht werden. Auch diese Aufgaben werden vom `UiSupportPackage` übernommen, indem die entsprechenden Events von Visual Studio beobachtet werden.

Des Weiteren sorgt das `UiSupportPackage` dafür, dass nicht gleichzeitig Modelle verschiedener Modelltypen geöffnet werden können. Da die Dataflows auf Daten der anderen beiden Modelltypen aufbauen, wäre es nur sehr schwer möglich, Änderungen in der Hardwaredescription oder den Datastructures in ein gleichzeitig geöffnetes Dataflow-Modell zu übertragen. Wenn aber verschiedene Modelltypen niemals gleichzeitig geöffnet sind, redu-

ziert sich dieser Aufwand auf das Synchronisieren des Dataflow-Modells mit den anderen Modellen auf den Zeitpunkt, wenn das Modell geöffnet wird *(siehe Abschnitt 6.2.2)*.

6.6.1. Services

Neben den oben beschriebenen Commands sind im `UiSupportPackage` zwei Services implementiert *(vgl. Tabelle 6.3)*: `IImportDriverService` und `ICodeBlockEditor`. Die Funktionalität, die diese Services anbieten, werden vom `HardwareDescriptionDslPackage` bzw. `DataFlowDslPackage` benötigt. Man hätte sie auch dort implementieren können, allerdings sollte die Logik zur Interaktion mit dem Projektsystem von diesen beiden Packages losgelöst werden, so dass diese Services besser in das `UiSupportPackage` passen.

Der `IImportDriverService` wird aufgerufen, wenn der Anwender den entsprechenden Befehl *(„Import Driver...“)* aus dem Kontextmenü des Hardwaredescription-Modells auswählt. Hier wird zunächst nach der Treiberdatei gefragt und diese geöffnet. Nachdem die Definitionen aus dem Treiber in das Hardwaredescription-Modell als eine eigene Plattform eingefügt wurden, werden die übrigen *(Quellcode-)* Dateien aus dem Treiber dem Projekt hinzugefügt.

Der `ICodeBlockEditor`-Service erzeugt, falls nötig, eine neue C-Datei für ein in einem Dataflow-Modell definiertes CodeBlock-Shape und öffnet sie, so dass der Anwender den Code für den Codeblock bearbeiten kann. Falls das CodeBlock-Shape im Modell verändert wurde, wird die Signatur in dieser Datei an die Definition aus dem Modell angeglichen *(siehe auch Abschnitt 5.3.2 sowie Abbildung 5.14 auf Seite 52)*.

6.7. Codegenerator

Für die Codeerzeugung aus den Modellen eines Sensorknotenprojektes wird das Text Templating Transformation Toolkit *(siehe Abschnitt 3.3.3)* verwendet. Die verwendeten Templates geben detailliert die Struktur des Codes vor und nutzen zu dessen Erzeugung Funktionalität aus dem `CodeGenerator`-Assembly.

In einem Sensorknotenprojekt wird aus jedem Modell *(egal von welchem Typ)* eine C-Datei generiert, die das in dem Modell beschriebene Verhalten bzw. die Definitionen aus dem Modell als ausführbaren Code enthält. Zusätzlich wird eine weitere Datei *(`Main.c`)* erzeugt, die den Code aller Dateien zusammenführt und beispielsweise Events an Methoden in den anderen Dateien weiterleitet.

Auch der Code für den Proxy wird mit diesen Techniken erzeugt, dabei entsteht eine einzige C#-Datei, die die Definitionen aus allen Datastructure-Modellen eines Sensorknotenprojektes enthält.

Aus der Hardwaredescription wird lediglich eine Methode zur Initialisierung der Sensorknotenhardware erzeugt. Für jedes Input- und Output-Shape wird eine entsprechende

Methode aufgerufen und die Parameter sowie die Enabled-Flags aus dem Modell übergeben. Events werden an anderer Stelle registriert und initialisiert. Für Variablen der Hardwaredescription ist keine Initialisierung durch *Flow* vorgesehen, da dies im Aufgabenbereich der Firmware liegt.

Aus den Datastructure-Modellen werden für die dort definierten Variablen globale Variablen im Quellcode sowie `structs` für die definierten Recordtypen angelegt. Da für die Kommunikation mit dem Proxy interne Records verwendet werden *(siehe Abschnitt 6.7.1)*, werden des Weiteren `structs` für die definierten Funktionen, Output Events und die öffentlichen Variablen erzeugt. Zusätzlich entsteht eine Methode `<Dataflow>_ProcessInternalRecord()`, die solche internen Records entgegennehmen und verarbeiten kann, um Zugriffe des Proxies auf die Variablen zu ermöglichen.

Der Code, der für die Dataflows erzeugt werden muss, ist um einiges komplexer als der Code der beiden vorherigen Modelltypen. In jedem Dataflow werden fünf verschiedene Arten von Methoden definiert, die den Namen des Dataflows als Präfix tragen, um Namenskonflikte zu vermeiden:

- `<Dataflow>_Init()`: Diese Methode wird während der Startphase des Sensorknotens aufgerufen und sorgt dafür, dass sich der Dataflow für Events registrieren und je eine Callback-Methode *(die ebendfalls in dieser Datei generiert wird)* angeben kann. Ausnahmen bilden Events mit einer Multiplicity von 1, die in Abschnitt 5.1.2 beschrieben sind und stattdessen in `Main.c` registriert werden.

- `<Dataflow>_Callback_SomeEvent()`: Für jedes Event, das im Trigger-Bereich des Dataflows verwendet wird, wird eine Callback-Methode benötigt. Diese Methode wird immer dann aufgerufen, wenn das entsprechende Event eintritt. Optional werden dieser Methode die vom Event gelieferten Daten übergeben. Innerhalb dieser Methode wird dann der Teil des Trigger-Bereichs ausgewertet, der vom Event zum Master-Trigger führt. Wird der Master-Trigger mit einem „Impuls“ erreicht, wird der Dataflow mittels der Methode `<Dataflow>_Flow()` ausgeführt.

- `<Dataflow>_ProcessInternalRecord()`: Neben Events können auch Funktionsaufrufe einen Dataflow auslösen. Funktionsaufrufe des Proxies werden über interne Records empfangen, die an diese Methode weitergeleitet werden. Hier wird geprüft, ob das interne Record einem Funktionsaufrufe entspricht, der im Trigger-Bereich vorkommt. Wenn dies der Fall ist, wird der Trigger-Bereich von diesem Funktionsaufruf bis zum Master-Trigger wie bei Events ausgewertet und wenn nötig der Dataflow mittels `<Dataflow>_Flow()` aufgerufen.

- `<Dataflow>_Flow()`: Diese Methode repräsentiert den Dataflow-Bereich eines Dataflow-Modells und wird immer dann aufgerufen, wenn der Trigger-Bereich über

den Master-Trigger einen „Impuls“ dazu bekommt. Hier werden alle Shapes mit dem im Modell spezifizierten Verhalten ausgeführt.

- Methoden für Formel-Shapes: Falls im Dataflow-Modell Formel-Shapes verwendet wurden, wird für jedes Shape eine eigene Methode erzeugt, die dann aufgerufen wird, wenn das Formel-Shape verwendet wird. Durch diese einzelnen Methoden für jede Formel wird die Codegenerierung ein wenig vereinfacht.

Innerhalb der `Callback()`-, `ProcessInternalRecord()`- und `Flow()`-Methoden muss der Dataflow aus Shapes und Kanten ausgewertet werden. Dabei werden die Shapes in einer bestimmten Reihenfolge abgearbeitet. Für jede ausgehende Kante eines Shapes werden lokale Variablen eingeführt, die mit den entsprechenden Daten aus dem Shape gefüllt werden, so dass diese lokalen Variablen als Eingabe für die weiteren Shapes verwendet werden können. Wie genau das Verhalten der verschiedenen Shapes aussieht, ist in das `CodeGenerator`-Assembly ausgelagert und wird nicht in den Template-Dateien angegeben. Für jedes Shape existiert eine Klasse *(abgeleitet von `BlockGeneratorBase`)*, die den Code für das Verhalten des Shapes erzeugt. Das Template kann mittels einer Fabrikmethode den Generator für ein Shape ermitteln und diesen zum Generieren des entsprechende Codes verwenden.

Die Reihenfolge, in denen die Shapes abgearbeitet werden, kann bestimmt werden, indem man den Dataflow als gerichteten Graphen auffasst und auf diesem Graphen eine Tiefensuche *(vgl. [CLRS01, Kapitel 22.3])* ausführt. Die umgekehrte Reihenfolge, in der bei der Tiefensuche die Knoten *(Shapes)* schwarz markiert wurden, garantiert eine Reihenfolge, in der kein Shape bearbeitet wird, bevor alle Shapes, von denen es abhängt, bearbeitet wurden.

Neben dem Code, der aus den drei Modelltypen erzeugt wird, wird eine weitere Datei *(`Main.c`)* generiert, die das Zusammenspiel aller Dateien steuert. In dieser Datei befindet sich die Methode `EntryPoint()`, die von der Firmware zum Starten der Sensorknotenanwendung aufgerufen wird. Aus dieser Methode werden sämtliche oben generierten Initialisierungsmethoden *(Hardwaredescription und Dataflows)* aufgerufen.

Außerdem werden hier Eventhandler für Events, die nur ein einziges Mal existieren *(Multiplicity = 1)*, erzeugt und registriert. Aus den Eventhandlern werden der Reihe nach alle Dataflows aufgerufen, die dieses Event verwenden. Auch für das Empfangen von internen Records, die sowohl von den Datastructures, als auch von den Dataflows benötigt werden, wird ein Eventhandler angelegt, um die Aufrufe an die Datastructures und Dataflows weiterzuleiten.

Beispiele für das Zusammenspiel der generierten Dateien sind in den Abbildungen 6.7 bis 6.9 zu sehen. Der generierte Code eines Beispieldataflows ist in Abschnitt 7.2.3 gezeigt und detailliert beschrieben.

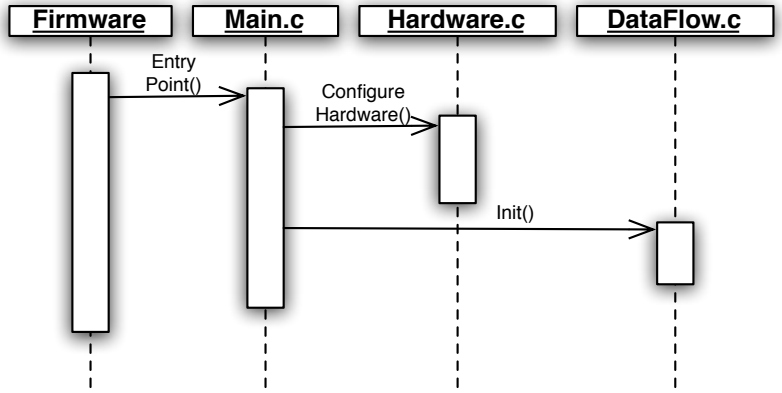


Abbildung 6.7.: Sequenzdiagramm: Start der Sensorknotenanwendung und Initialisierung

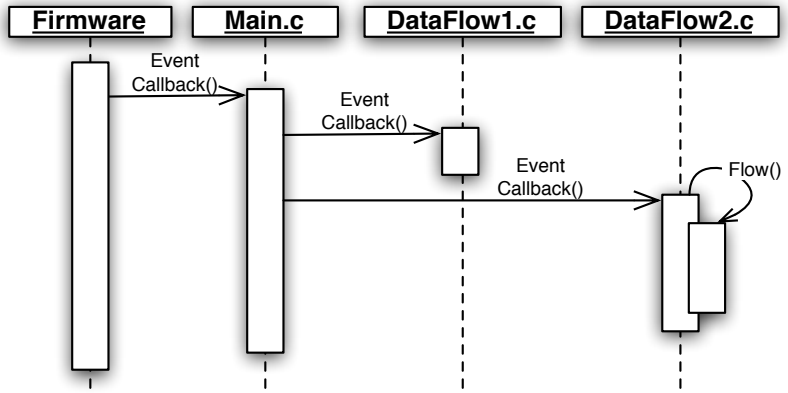


Abbildung 6.8.: Sequenzdiagramm: Auftreten eines Events, auf das ein Dataflow reagiert

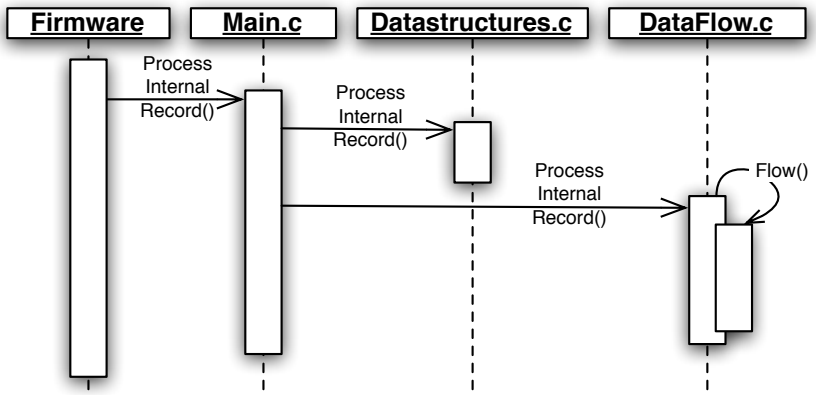


Abbildung 6.9.: Sequenzdiagramm: Empfang eines Records vom Proxy, das einen Funktionsaufruf bewirkt

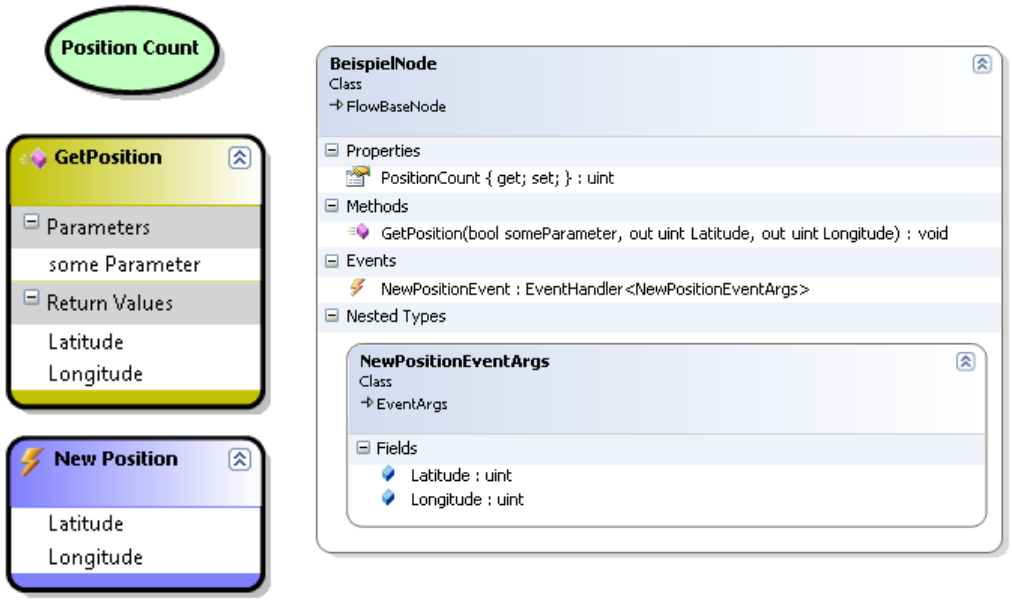
6.7.1. Proxy

Für die Kommunikation zwischen den Sensorknoten und einem PC kommt das ScatterWeb .NET SDK *(siehe Abschnitt 3.1.1)* zum Einsatz. Es bietet bereits die grundlegende Funktionalität an, um Funkpakete vom PC zu einem Sensorknoten und in umgekehrter Richtung zu übertragen. Für *Flow* wurden weitere Klassen innerhalb des SDKs angelegt, die es ermöglichen auf Basis von Records mit den Sensorknoten zu kommunizieren.

Für jedes *Flow*-Sensorknotenprojekt wird eine Proxy-Klasse generiert, die von `FlowBaseNode` erbt. Das öffentliche *(vom PC-Programmierer zu verwendende)* Interface dieser Klasse enthält die Elemente, die durch den Anwender in den Datastructure-Modellen definiert wurden. Variablen, die in den Datastructures definiert und als öffentlich markiert sind, sind über die Proxy-Klasse als .NET-Properties verfügbar. Funktionen und Output Events werden ebenfalls als .NET-Funktionen bzw. .NET-Events angeboten. Abbildung 6.10 zeigt ein Datastructure-Modell sowie das öffentliche Interface der generierten Proxy-Klasse *(vgl. Abbildung 5.11 und Abschnitt 5.3.1)*.

Die Kommunikation zwischen dem PC und den Sensorknoten erfolgt ausschließlich durch die Übertragung von Records, die sowohl auf dem Sensorknoten, als auch in der Proxy-Klasse definiert sind. Diese Records werden intern verwendet und sind für den Anwender nicht sichtbar.

Um eine Variable zu lesen oder zu schreiben, wird ein Record vom PC an den Sensorknoten übertragen. Über ein Flag des Records wird dem Sensorknoten mitgeteilt, ob er mit dem aktuellen Wert antworten oder den Wert aus dem Record speichern soll.



(a) (b)

Abbildung 6.10.: Datastructure-Modell (a) und die daraus generierte Proxy-Klasse (b)

In der Proxy-Klasse ist dazu eine .NET-Property definiert.

Tritt auf dem Sensorknoten ein Output Event auf, wird ein entsprechendes Record mit den Daten des Output Events an den Proxy verschickt. Dieser empfängt das Record, entpackt die Daten und löst ein .NET-Event aus, welches ein PC-Programmierer abonnieren kann.

Auch für Funktionsaufrufe wird ein Record, das die Daten der Funktionsparameter enthält, an den Sensorknoten geschickt. Der Sensorknoten antwortet mit einem anderen Record, das die Daten der Rückgabewerte enthält. Der PC-Proxy stellt eine entsprechende .NET-Funktion bereit, die, falls Rückgabewerte definiert sind, so lange die Ausführung blockiert, bis das Antwortrecord eingegangen ist oder ein Timeout ausgelöst wird.

Der Anwender kann mit diesen Proxy-Klassen so arbeiten, wie er es von anderen .NET-Klassen gewohnt ist, lediglich die Zeit, die gewisse Funktionsaufrufe benötigen, ist immer dann länger, wenn über Funk mit einem Sensorknoten kommuniziert werden muss.

6.8. Erweiterbarkeit von Flow

Flow sollte so offen entworfen werden, dass es möglich ist, Sensorknoten verschiedener Plattformen zu adressieren. Eine zentrale Rolle spielt dafür die Hardwaredescription, die nicht nötig wäre, wenn man lediglich Sensorknoten eines Typs ansprechen wollte.

Die drei DSLs sind so allgemein gehalten, dass es durchaus möglich ist auch Software für andere Plattformarten[4] zu spezifizieren, solange diese Arten auch auf der Ebene von Input- und Output-Ports, Events, Variablen, Records, Output Events und Funktionen beschrieben werden können. Allerdings wurde bereits bei der Datastructure-DSL in Kapitel 5.2.2 eine erste Ausnahme gemacht. Für Records wurde in der DSL eine Annahme getroffen, wie der Codegenerator mit Strings umgehen wird. Abgesehen von diesem Bruch der Abstraktionsebenen können besonders die DSLs, die einen Großteil des *Flow*-Systems darstellen, problemlos für andere Plattformarten eingesetzt werden.

Einige der Komponenten, die für *Flow* entwickelt wurden, sind eher plattformspezifisch und sprechen eine Sensorknotenplattform auf Basis der Programmiersprache C an. Der Firmwareparser sowie der Codegenerator zählen zu dieser Gruppe. Der Codegenerator ist des Weiteren an die Bedürfnisse der ScatterWeb-Firmware angepasst. Es ist zwar denkbar, auch andere Systeme anzusprechen, aber diese müssten die gleichen Makros und Methoden bereitstellen. Beim Codegenerator stellt dies allerdings keinen Designfehler dar, da es vorgesehen ist, für grundsätzlich andere Plattformen auch verschiedene Codegeneratoren zu verwenden. Um dies zu erleichtern, ist der Codegenerator nur lose in das Gesamtsystem *Flow* eingebunden. Durch Austauschen der entsprechenden Templates

[4] In der aktuellen Form spricht *Flow* vor allem Plattformen auf C-Basis an. Durch die Modellierung der Hardwaredescription werden aber kaum weiteren Annahmen über die Plattform getroffen *(siehe Text)*. Es wäre denkbar, in Zukunft auch vollkommen andere Plattformarten, z.B. auf Java- oder .NET-Basis, zu unterstützen.

kann leicht ein anderer Generator verwendet werden.

Das `UiSupportPackage` spielt eine Sonderrolle. Indem es die Integration in Visual Studio steuert, muss es an vielen Stellen Annahmen über die verwendete Plattformart treffen. Dies beginnt damit, dass es auf dem Visual Studio Projekttyp „Makefile-Project“ aufsetzt und Funktionen nur für diesen Projekttyp anbietet. Es definiert die Struktur der Plattformsetups und Driver Packets und gibt vor, wie dem Projekt Modelle *(u.a. mit welchen Templates zur Codegenerierung)* hinzugefügt werden. Auch hier müssten für eine grundsätzlich andere Plattform *(z.B. eine, die nicht auf C-Makefile-Projekten basiert)* Änderungen vorgenommen werden. Es wäre denkbar, mehrere `UiSupportPackage`s parallel zu verwenden, die jeweils eine andere Plattformart ansteuern.

6.9. Bibliotheken

Die oben beschriebenen Assemblies bilden die Kernbestandteile von *Flow* und sind ausschließlich für die Anforderungen von *Flow* entwickelt worden. Während der Entwicklungsarbeit sind auch etliche Erweiterungen für die DSL Tools und Visual Studio entstanden, die so allgemein umgesetzt wurden, dass sie auch in anderen Projekten in Verbindung mit den DSL Tools und Visual Studio eingesetzt werden können. Da diese Bibliotheken während der Arbeit an *Flow* entstanden sind, sollen sie in den folgenden Abschnitten kurz vorgestellt werden.

6.9.1. JaDAL

Um *Flow* zu realisieren mussten die DSL Tools erweitert bzw. Funktionen nachrüstet werden, die ansonsten fehlen würden. Diese Funktionalität ist in der Bibliothek *JaDAL* zusammengefasst. *JaDAL* ist ein Akronym und steht für „Just another DSL Tools Addon Library“ und ist im Rahmen der Diplomarbeit entstanden. Sie wurde durch den Autor unter der neuen BSD-Lizenz auf CodePlex[5] veröffentlicht. Diese Bibliothek kann so auch von anderen Nutzern der Microsoft DSL Tools verwendet werden. Im Folgenden soll kurz beschrieben werden, welche Funktionalität durch *JaDAL* geboten und an welchen Stellen sie von *Flow* verwendet wird.

Compartment Mapping

In den DSL Tools sind zwar Compartment-Shapes *(siehe Abbildung 3.5 auf Seite 27)* bekannt, um eine Domain Class mit untergeordneten Domain Classes zeilenweise in einem Shape darzustellen, aber die DSL Tools ermöglichen es nicht, Kanten zwischen den Zeilen von Compartment-Shapes zu erstellen. Kanten können nur zwischen Shapes erzeugt werden. Es ist zwar möglich, Kanten zwischen zwei Compartment-Shapes zu erstellen, aber es können keine Zeilen adressiert werden.

[5] http://www.codeplex.com/JaDAL

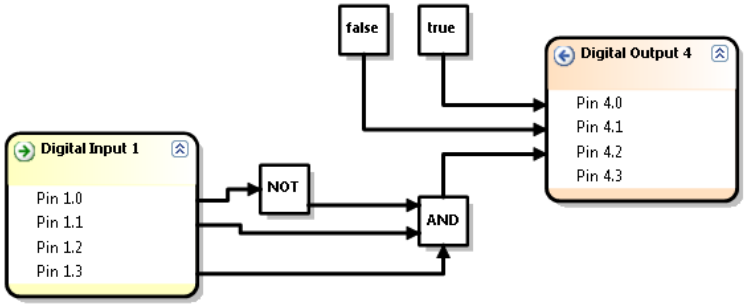


Abbildung 6.11.: Compartment- und reguläre Shapes, verbunden durch Kanten unter Verwendung des *JaDAL Compartment Mappings*

Für die Dataflows ist dies aber eine wichtige Anforderung. Würde man keine Kanten von Zeilen eines Input-Shapes erstellen können, so müsste jede Zeile *(jedes Feld des Input-Ports)* durch ein einzelnes Shape repräsentiert werden. Dies wäre extrem unübersichtlich und würde unter Umständen das gesamte Konzept der Dataflows, wie es nun umgesetzt wurde, in Frage stellen, da ein Anwender es nicht mehr so einfach bedienen könnte wie gewünscht.

Abbildung 6.11 zeigt solche Kanten zwischen Zeilen von Compartment-Shapes und auch regulären Shapes, die erst durch die *Compartment Mapping* Funktionalität von *JaDAL* möglich werden.

User Restrictions

Im Metamodell der DSLs werden die Domain Classes und Domain Properties definiert. An dieser Stelle kann auch definiert werden, wie ein Anwender den Editor zur Laufzeit verwenden darf. Domain Properties können beispielsweise nur-lesbar oder sogar als versteckt definiert werden. Das Hinzufügen von gewissen Domain Classes zum Modell kann dem Anwender gestattet oder verboten werden.

Diese Konfiguration im Metamodell führt zur Generierung von Code, der dafür sorgt, dass sich der Editor entsprechend verhält. Doch das Verhalten ist statisch und kann nicht auf einfache Weise zur Laufzeit umgeschaltet werden. Wenn man einen Editor in verschiedenen Modi betreiben möchte *(z.B. in einem Modus mit Vollzugriff und in einem anderen, in dem nur gewisse Properties bearbeitet werden dürfen)*, bieten die DSL Tools keine Unterstützung dafür an.

Durch die *JaDAL User Restrictions* können für jeden DSL-Editor verschiedene Modi definiert werden. Die einzelnen Domain Classes und Domain Properties können dann über Attribute so konfiguriert werden, dass sie sich in den unterschiedlichen Modi wie gewünscht verhalten. Es können Properties als nur-lesend oder versteckt definiert werden. Des Weiteren kann es verboten werden, gewisse Domain Classes dem Modell hinzuzufügen.

In *Flow* wird diese Funktionalität verwendet, um die Hardwaredescription in vier verschiedenen Modi zu betreiben. Welcher Modus verwendet wird, hängt von der Dateiendung des Modells ab:

- `.hardware`: voller Zugriff auf alle Properties durch den Hardwarehersteller.
- `.hardwareinfo`: eingeschränkter Zugriff auf die Properties durch den Anwender. Es können keine Klassen hinzugefügt werden *(und auch nicht gelöscht werden; siehe folgenden Abschnitt).*
- `.driver`: voller Zugriff auf alle Properties, allerdings minimal anders dargestellt als bei `.hardware`, da es sich um eine Treiberplattform handelt.
- `.driverinfo`: wie `.hardwareinfo`, allerdings für Treiberplattformen.

Dynamic Can Delete

Auch das Löschen von Domain Classes bzw. Shapes aus dem Modell kann im Metamodell gestattet oder verboten werden. Wenn aber diese Erlaubnis nicht statisch, sondern erst zur Laufzeit entschieden werden kann oder soll, muss individueller Code geschrieben werden. Der Code, der dazu nötig ist, kann durch die Verwendung von *JaDAL* und der *Dynamic Can Delete* Funktionalität um einen Großteil reduziert werden.

Die entsprechende Domain Class muss lediglich um das Interface `IDynamicCanDelete` mit einer einzigen Methode `bool CanDelete()` erweitert werden. *JaDAL* sorgt dafür, dass diese Methode jedes Mal, wenn ein Exemplar der Domain Class gelöscht werden soll, abgefragt wird und verbietet bei negativem Ergebnis das Löschen.

Dynamic Properties

Im Metamodell werden für jede Domain Class die Domain Properties angegeben, die diese Klasse besitzt. Diese Domain Properties werden zur Laufzeit beispielsweise im Properties Window von Visual Studio angezeigt. Es kann aber vorkommen, dass zur Laufzeit weitere Properties hinzugefügt werden sollen. *Flow* benötigt weitere Properties z.B. bei Events im Dataflow *(das typische Timer-Event benötigt eine zusätzliche Property, um den Intervall einzustellen).*

Die *JaDAL Dynamic Properties* Funktionalität stellt eine Containerklasse bereit, die mit den DSL Tools verwendet werden kann. Über das Interface der Containerklasse können jederzeit weitere *(untergeordnete)* Properties hinzugefügt und wieder entfernt werden. Die Containerklasse ist entsprechend ausgestattet, so dass ihre Daten im Properties Window angezeigt werden. Abbildung 6.12 zeigt das Properties Window einer Domain Class, die im Metamodell nur zwei Properties *(`Name` und `Props`)* definiert hat. Alle weiteren angezeigten Daten stammen aus den *Dynamic Properties*.

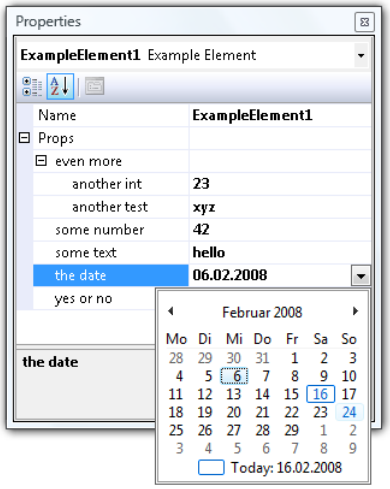


Abbildung 6.12.: Properties Window mit *JaDAL Dynamic Properties*

Directive Processors

Directive Processors werden vom Text Templating Transformation Toolkit verwendet, um aus den Templates auf bestimmte Daten zuzugreifen. Die DSL Tools generieren für jede DSL einen Directive Processor, der auf Modelle dieser Sprache zugreifen kann. Innerhalb von *JaDAL* sind zwei weitere Directive Processors definiert: Einer, um auf beliebige XML-Dateien zugreifen und diese Daten innerhalb der Templates verwenden zu können, und ein anderer, um auf Visual Studio Project-Files zugreifen zu können. Beide Processors werden von *Flow* zur Codegenerierung benötigt.

6.9.2. VSXTools

Im Gegensatz zu *JaDAL* enthält die Bibliothek *VSXTools*, die ebenfalls im Rahmen der Diplomarbeit entstanden ist, eher kleine Erweiterungen, die meist nur aus einer Methode bestehen, um einfacher mit den Klassen aus dem Visual Studio SDK zu arbeiten. An vielen Stellen wurden hier Best Practices zum Zweck der Wiederverwendung zusammengefasst.

Der größere Teil der *VSXTools* erweitert die Visual Studio Extensibility Klassen und nicht die Klassen der DSL Tools. Dennoch kann beispielsweise Code in den *VSXTools* gefunden werden, um Texte in Shapes mehrzeilig anzuzeigen. Auch der Tiefensuchalgorithmus, der z.B. auf Modellen der DSL Tools angewandt werden kann, ist hier definiert.

6.9.3. System.Linq.Dynamic

Der Code aus diesem Assembly ist nicht im Rahmen der Diplomarbeit entstanden, sondern stammt aus den *C# Samples for Visual Studio 2008*[6] und ist von Microsoft unter der

[6] http://code.msdn.microsoft.com/csharpsamples

Microsoft Public License *(Ms-PL)* veröffentlicht. Da diese Beispiele ausschließlich im Quellcode und nicht als kompiliertes Assembly veröffentlicht wurden, ist es notwendig, den Quellcode in das *Flow*-Projekt zu übernehmen. Um diesen Code vom restlichen *Flow*-Code zu trennen, wurde diese Funktionalität in ein separates Assembly aufgenommen.

Bei dem Code handelt es sich um einen Parser, der C#-Ausdrücke verarbeiten und daraus zur Laufzeit ausführbare Ausdrücke erzeugen kann. Diese Funktionalität wird für das Formel-Shape verwendet, um zu prüfen, ob die eingegebene Formel einen syntaktisch korrekten Ausdruck darstellt. Dies ist möglich, da die Syntax von C und C#, was einfache Formeln betrifft, gleich ist. Der Parser gibt, unter Beachtung der Eingabedatentypen, aussagekräfigte Fehlermeldungen aus, die dem Anwender bei der Modellvalidierung präsentiert werden. Außerdem kann so der Datentyp des Ergebnisses einer Formel ermittelt werden, wenn die Datentypen der Parameter bekannt sind.

7. Anwendung und Fallstudie

In diesem Kapitel wird gezeigt, wie *Flow* verwendet werden kann, um Sensorknotenanwendungen zu erstellen. Dazu wird zunächst eine Referenzplattform für den Sensorknoten MSB-430H und die Erweiterung MSB-430S vorgestellt. Erst eine solche Plattform ermöglicht es, Anwendungen für Sensorknoten mit *Flow* anzufertigen. Diese Plattform wird im weiteren Verlauf verwendet, um einige einfache Beispielanwendungen zu erstellen. Die Anwendungen sind durch die entsprechenden *Flow*-Modelle gezeigt und textuell erklärt. Für eines dieser Modelle ist der generierte Quellcode angegeben und im Detail beschrieben. Neben diesen einfachen Beispielanwendungen, die nur die Grundlangen von *Flow* aufzeigen sollen, ist in Kapitel 7.3 eine umfangreichere Fallstudie am Skomer Island Beispiel *(siehe Kapitel 1.1)* dargestellt.

7.1. Referenzplattform MSB-430H

Bereits in Kapitel 3.1 wurde die Sensorknotenhardware MSB-430H vorgestellt. Da diese Hardware und die dazugehörige Firmware an der Freien Universität Berlin entwickelt wurden, bietet es sich an, auf dieser vorhandenen Firmware aufzusetzen und eine Abstraktionsschicht *(siehe Kapitel 4.2)* für *Flow* anzubieten. Diese Abstraktionsschicht wurde von Tomasz Naumowicz für *Flow* entwickelt. Die ursprüngliche Firmware musste dazu kaum erweitert werden, so dass die Abstraktionsschicht relativ dünn ist. Sie enthält Methoden in der von *Flow* erwarteten Form und leitet die meisten Anfragen direkt an Firmwaremethoden weiter. Lediglich zur Funkkommunikation mit Records musste ein wenig Code geschrieben werden, da Records, in der Form, in der *Flow* sie verwendet, der Firmware nicht bekannt sind. Die Abstraktionsschicht ist mit den entsprechenden Annotationen versehen, so dass die Hardwaredescription *(siehe Abbildung 7.1)* automatisch erzeugt werden kann.

Im oberen Teil der Abbildung sind die I/O-Ports des MSB-430H zu sehen. Die 16 digitalen Ports sind zu zwei Gruppen zusammengefasst und jeweils durch ein Input- und ein Output-Shape repräsentiert. Da jeder einzelne Pin entweder als Eingang oder als Ausgang verwendet werden kann, sind Kanten *(siehe Abschnitt 5.1.6)* zwischen diesen Shapes modelliert, die den gegenseitigen Ausschluss erzwingen. Für den Input-Ports „Digital Input P1“ sind des Weiteren acht Events verfügbar[1]. Da die Events unabhängig voneinander in den Dataflows verwendet werden sollen, sind sie durch acht einzelne Shapes dargestellt. Auch an dieser Stelle sorgen Kanten dafür, dass die Events nur dann

[1] Der MSP-Mikrocontroller bietet nur für diese Ports Interrupt-Unterstützung an.

aktiviert werden können, wenn der entsprechende Port als Eingang konfiguriert ist. Neben den digitalen Ports existiert ein analoger Eingang *(„ADC“)* sowie ein Output-Shape, um die Status-LED des MSB-430H-Boards anzusteuern.

Zur Kommunikation stehen die serielle Schnittstelle sowie Funk zur Verfügung. Das „Text Line Received“-Event liefert eine Textzeile, die über die serielle Schnittstelle empfangen wurde. Der „Serial Port Output“ gibt Daten in Textform über diese Schnittstelle aus. Dieser Output-Port kann entweder einen String oder ein Record, dessen Stringrepräsentation ausgegeben wird, verarbeiten. Auch für die Funkkommunikation werden zwei Shapes *(der Output-Port „Transmit Record“ und das Event „Record Received“)*, die als Daten ausschließlich Records verwendet, bereitgestellt. Das Event teilt dem Sensorknoten über das Flag „Broadcast“ mit, ob das empfangene Record als Broadcast versendet wurde. Über den Parameter „Receive Broadcasts“ kann der Anwender im Vorfeld steuern, ob Broadcasts überhaupt an die Dataflows weitergeleitet werden. Die Variable „Node ID“ ermöglicht es dem Anwender, die Adresse des Knotens auszulesen, nicht aber sie zu verändern.

Der Input-Port „Real Time Clock“ liefert die Zeit in ihren Komponenten, als Timestamp oder als Text. Die beiden dazugehörigen Output-Ports ermöglichen es die Uhrzeit aus einem Dataflow heraus zu stellen, indem entweder alle Komponenten oder ein Timestamp angegeben werden.

Mittels der Shapes „Write File with Timestamp“ und „Write File without Timestamp“ können Records in eine Datei auf der SD-Karte des Sensorknotens geschrieben werden. Die beiden Shapes haben gemeinsam, dass sie die Datei zeilenweise ergänzen und Records in einer menschenlesbaren Textrepräsentation ablegen. Sie unterscheiden sich darin, dass der Zeile optional die aktuelle Uhrzeit mit Datum vorangestellt wird.

Zwei weitere Events ermöglichen es Dataflows zu starten. Das „Power On Event“ wird beim Starten der Sensorknotenanwendung *(beispielsweise nach einem Hardwarereset)* ein einziges Mal ausgelöst und ermöglicht es so Initialisierungsaufgaben durchzuführen. Das „Periodic Timer Activated“-Event kann in mehreren[2] Dataflows verwendet werden und aktiviert diese Dataflows regelmäßig. Die Dauer zwischen zwei Aktivierungen kann dabei für jede Ausprägung des Events über einen Parameter im jeweiligen Dataflow eingestellt werden.

7.1.1. Treiber für MSB-430S

In Abbildung 3.1 auf Seite 19 ist das Sensorboard MSB-430S als Erweiterung des MSB-430H zu sehen. Dieses Erweiterungsboard enthält zwei Taster, drei LEDs sowie einen Bewegungsmelder und einen Beeper. Auch für diese Erweiterung wurde ein *Flow*-Treiber mit entsprechender Hardwaredescription *(Abbildung 7.2)* erstellt.

[2] In der aktuellen Implementierung stehen vier Timer zur Verfügung, aber diese Anzahl kann durch Konfiguration der Firmware erhöht werden. Die derzeitige Begrenzung ist vor allem so gewählt worden, um Ressourcen zu sparen.

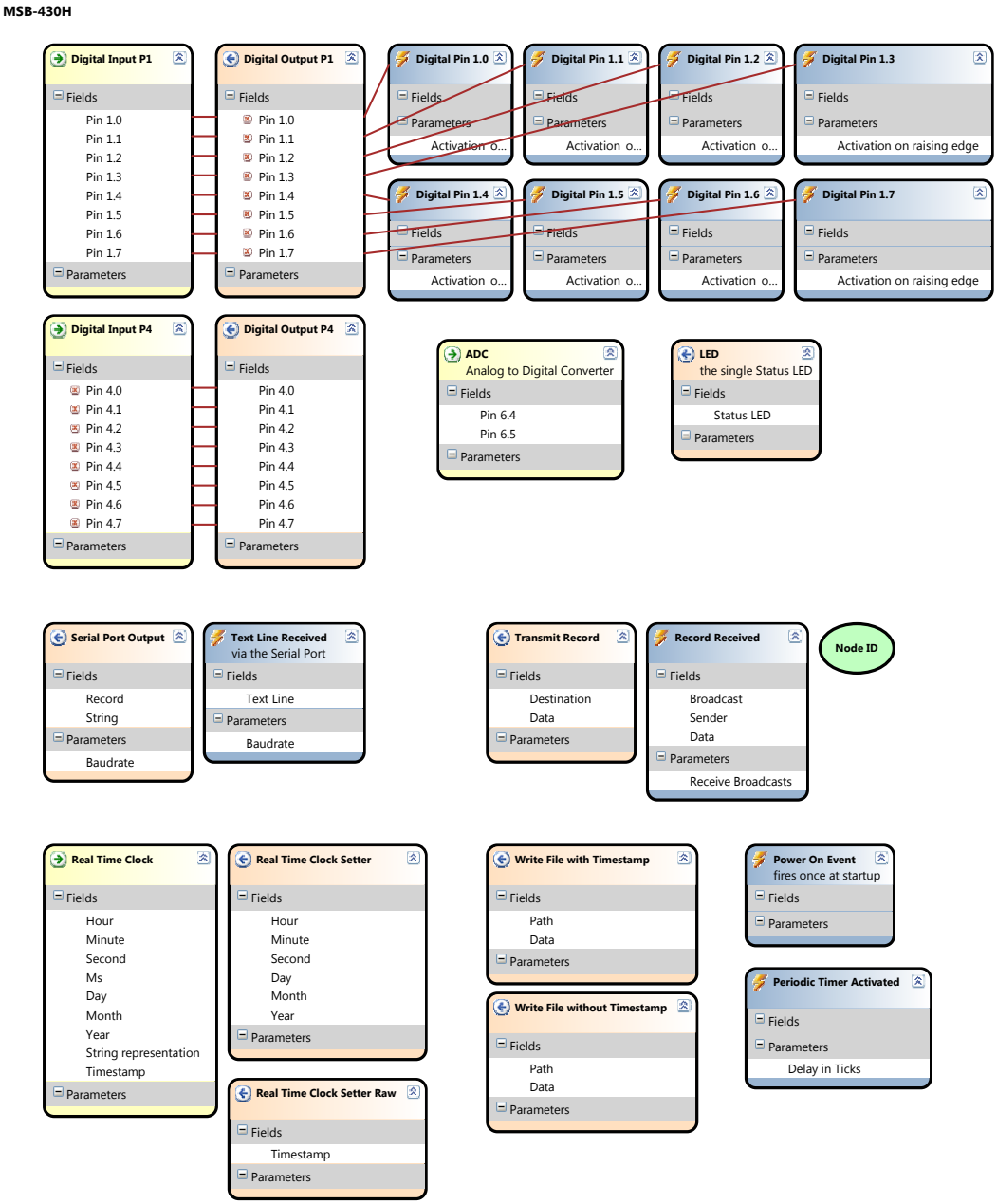


Abbildung 7.1.: Hardwaredescription des MSB-430H

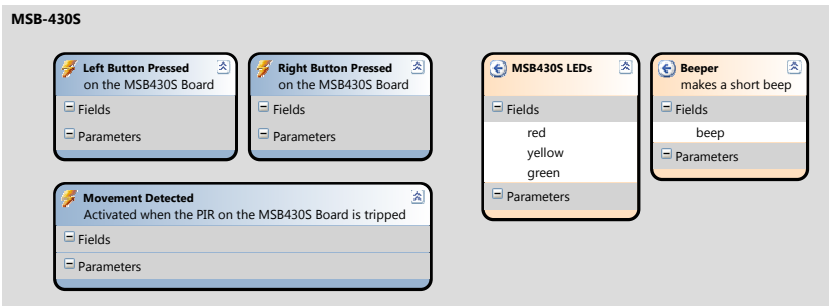


Abbildung 7.2.: Hardwaredescription des MSB-430S-Treibers

Die Buttons können als Events verwendet werden, um Dataflows auszulösen. Auch der Bewegungsmelder löst, immer wenn er aktiviert wird, ein Event aus. Die LEDs sind als Output-Port modelliert, über die jede LED einzeln geschaltet werden kann. Der Beeper wird ebenfalls als Output-Port zur Verfügung gestellt. Er gibt immer dann einen kurzen Pfeifton von sich, wenn das Feld „beep“ in einem Dataflow mit einem booleschen `TRUE` gesetzt wird.

Da diese Erweiterung auf den gewöhnlichen Ports des MSB-430H aufsetzt *(sowohl physisch als auch logisch)*, stehen bei Verwendung des MSB-430S einige der Elemente des MSB-430H nicht mehr zur Verfügung. Diese werden – durch das Eintragen von Kanten in die Hardwaredescription – automatisch deaktiviert, sobald ein Anwender den Treiber lädt.

7.2. Beispielanwendungen

In den folgenden Abschnitten sind drei einfache Beispielprogramme, die die Referenzplattform für den MSB-430H verwenden, gezeigt und detailliert beschrieben. Zusätzlich ist im Abschnitt 7.2.3 der Quellcode angegeben und beschrieben, der aus einem dieser Modelle generiert wurde.

7.2.1. Blinklicht

Eins der einfachsten Beispielprogramme um die Funktionalität eines Sensorknotens zu demonstrieren, ist ein Programm, das eine LED blinken lässt. Um dies mit *Flow* zu realisieren muss in einem Datastructure-Modell eine boolesche Variable angelegt werden. Der Dataflow, der das LED ein- und ausschaltet, ist in Abbildung 7.3 dargestellt. Dieser Dataflow wird durch einen periodischen Timer aktiviert *(das Intervall kann, wenn das Event im Dataflow ausgewählt ist, über das Visual Studio Properties Window angegeben werden)*. Im Dataflow wird die Variable x gelesen, negiert und das Ergebnis wiederum in x gespeichert. Zusätzlich wird das Ergebnis durch die Status LED angezeigt, die somit blinkt.

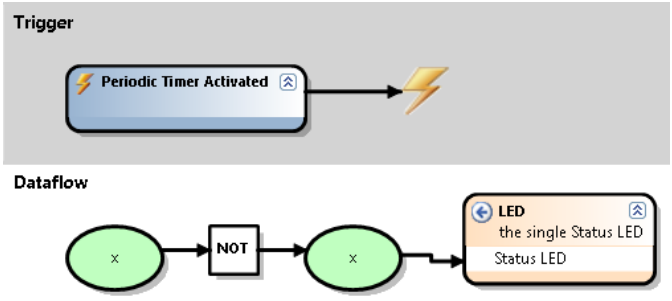


Abbildung 7.3.: Dataflow der Beispielanwendung „Blinklicht“

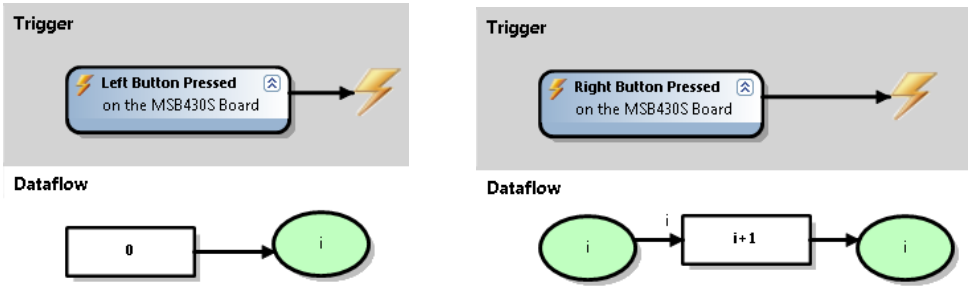


Abbildung 7.4.: Zwei Dataflows der Beispielanwendung „Binärzähler“ zum Ändern der Variable i

7.2.2. 3-Bit Binärzähler

Die zweite Beispielanwendung zeigt, wie mehrere Events und Dataflows in einer Sensorknotenanwendung zusammenarbeiten können, um eine Aufgabe zu erfüllen. In dieser Anwendung wird der MSB-430H-Sensorknoten sowie die Erweiterung MSB-430S mit ihrem Treiber verwendet, um einen 3-Bit Binärzähler umzusetzen. Da das MSB-430S-Erweiterungsboard nur drei LEDs besitzt, wird es dem Zähler nur möglich sein von null bis sieben zu zählen.

In einem Datastructure-Modell ist auch für diese Anwendung nur eine einzige Variable definiert *(i vom Typ* `Number`*)*. In Abbildung 7.4 sind zwei Dataflows gezeigt, die jeweils beim betätigen der linken oder rechten Taste ausgeführt werden. Mit dem einen wird die Variable i auf 0 gesetzt und mit dem anderen wird ihr Wert um eins erhöht.

Ein dritter Dataflow *(Abbildung 7.5)* reagiert ebenfalls auf diese beiden Events und sorgt dafür, dass der Wert von i mittels der LEDs als Binärzahl angezeigt wird. Dazu dient i als Eingabe für drei Formel-Shapes, die mittels des Modulo-Operators bestimmen, ob die einzelnen LEDs eingeschaltet werden müssen. Da nun mehrere Dataflows auf dieselben Events reagieren, muss sichergestellt werden, dass zuerst i verändert und dann die LEDs umgeschaltet werden. Dazu werden die Prioritäten der Dataflows so konfiguriert, dass der Dataflow aus Abbildung 7.5 eine höhere Priorität aufweist als die beiden anderen. Der Dataflow aus Abbildung 7.5 reagiert zusätzlich auch auf das „Power On Event“. Dadurch werden die LEDs nach dem Einschalten des Sensorknotens ein erstes Mal initialisiert, so dass ein Startwert, der im Datastructure-Modell angegeben ist, ebenfalls sichtbar wird.

7.2.3. Kommunikation

Nur selten werden Sensorknoten alleine betrieben, so dass ein weiteres Beispiel die Kommunikation zwischen zwei Sensorknoten sowie einem PC über die serielle Schnittelle demonstrieren soll. Es soll ein Record *(definiert in einem Datastructure-Modell auf beiden Sensorknoten, siehe Abbildung 7.6)* auf Knopfdruck vom Sender-Knoten zu einem Empfänger geschickt werden. Der Empfänger zeigt ein boolesches Flag aus dem Record

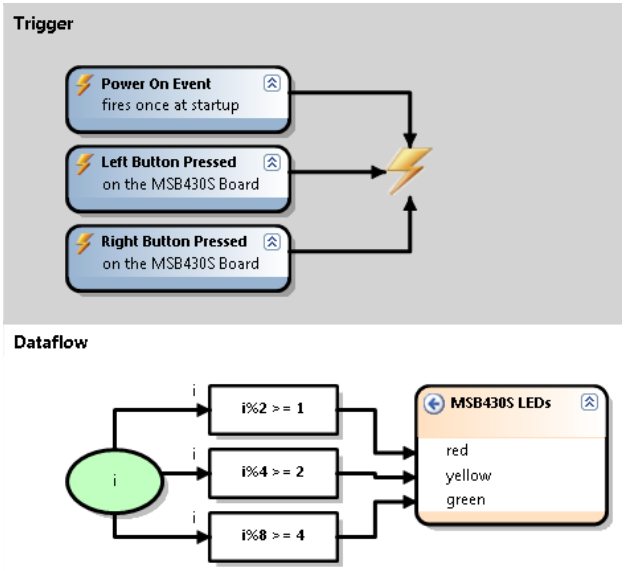


Abbildung 7.5.: Dataflow der Beispielanwendung „Binärzähler“ zur Anzeige des Binärwerts

mittels der LED an und gibt das gesamte Record über die serielle Schnittstelle aus.

Für beide Sensorknotenanwendungen muss das Record auf dieselbe Art und Weise definiert werden *(insbesondere die Record-Id muss gleich sein)*. Auf Senderseite kommen zwei Variablen i und x zum Einsatz, die wie in den vorhergehenden Beispielen beim Tastendruck verändert werden *(siehe Abbildung 7.7)*. Anschließend wird in diesem Dataflow ein „DataRecord“ erzeugt und per Funk an den Sensorknoten mit der Adresse 2 verschickt. Zusätzlich wird das Flag auch beim Sender mittels der LED angezeigt.

Im Dataflow des Empfängers *(Abbildung 7.8)* ist eine Bedingung im Trigger-Bereich angegeben. Dieser Dataflow wird nur dann ausgeführt, wenn das empfangene Record vom Sensorknoten mit der Adresse 1 geschickt wurde. Dann wird das Record über die serielle Schnittstelle ausgegeben und das LED dem Flag entsprechend geschaltet. Einige Zeilen der seriellen Ausgabe sind ebenfalls in der Abbildung zu sehen.

In diesem Beispiel wurden zur Übersichtlichkeit die Adressen der beiden Sensorknoten fest in den Dataflows hinterlegt. Es ist natürlich möglich diese Daten über Variablen – auch zur Laufzeit – konfigurierbar zu halten.

Quellcode

Der Quellcode, der aus dem Dataflow in Abbildung 7.7 generiert wird, soll exemplarisch vorgestellt werden und ist in Abbildung 7.10 abgedruckt. Lediglich auf die beiden leeren Methoden `button_Init()` und `button_ProcessInternalRecord()` wurde verzichtet. Diese Methoden kommen hier nicht zum Einsatz, da es sich bei den Button-Events um globale Events *(Multiplicity = 1)* handelt, deren Callback-Methoden in der `Main.c`-Datei implementiert sind und auch dort registriert werden. Immer wenn das Event ausgelöst

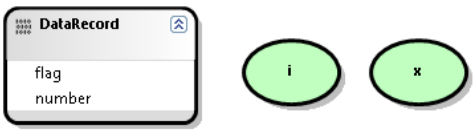


Abbildung 7.6.: Datastructure-Modell der Beispielanwendung „Kommunikation“ *(Sender)*. Der Empfänger verwendet ein ähnliches Modell, in dem allerdings nur das „DataRecord“ definiert ist.

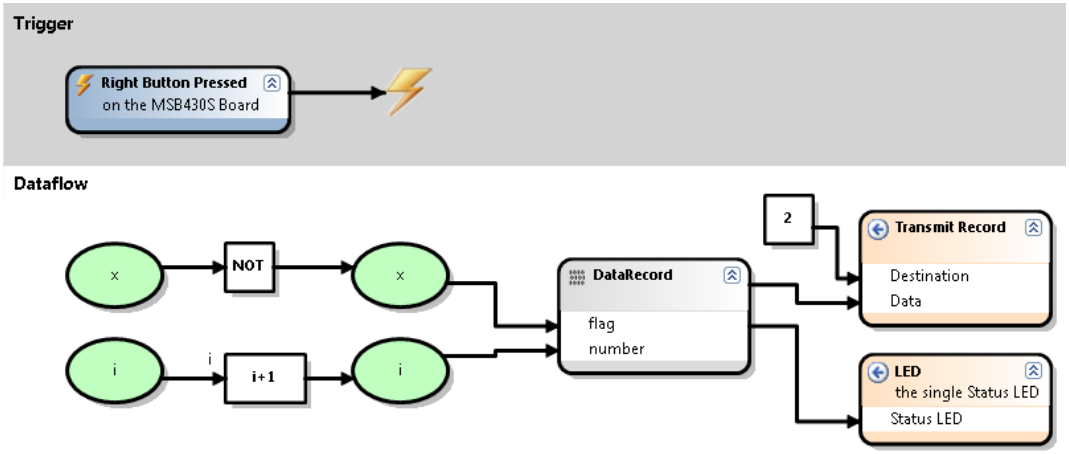
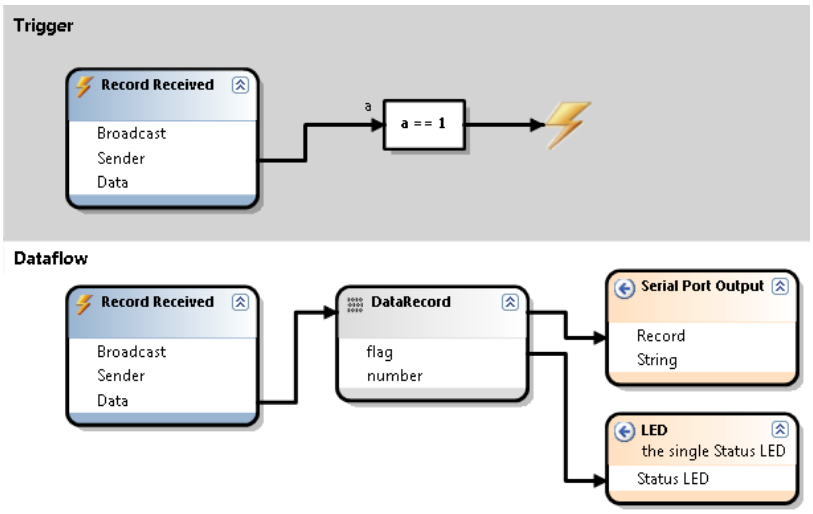


Abbildung 7.7.: Dataflow der Beispielanwendung „Kommunikation“ *(Sender)*




```
flag:1 number:59
flag:0 number:60
flag:1 number:61
flag:0 number:62
flag:1 number:63
flag:0 number:64
```

Abbildung 7.8.: Dataflow der Beispielanwendung „Kommunikation“ *(Empfänger)*. Zusätzlich sind einige Zeilen der seriellen Ausgabe dargestellt.

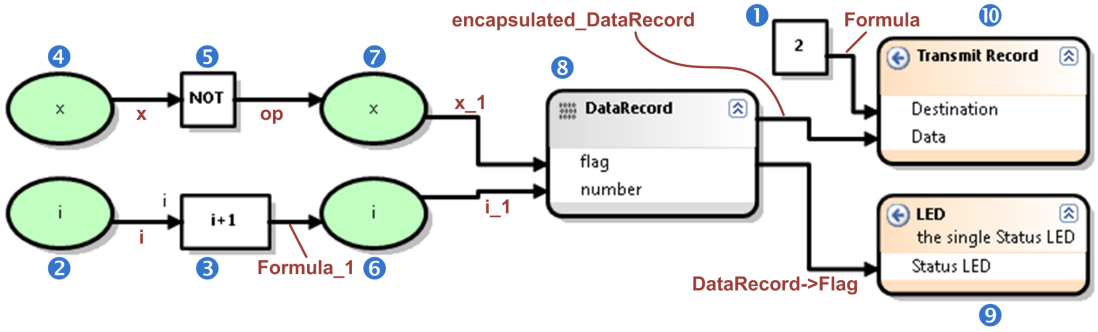


Abbildung 7.9.: Dataflow mit Kommentaren zu Codegenerierung *(siehe Text)*

wird, wird die Methode `button_Callback_RightButtonPressed()` *(Zeile 54)* von dort aus aufgerufen. Da im Trigger-Bereich keine Bedingungen definiert sind, wird in dieser Methode direkt der Dataflow-Bereich *(`button_Flow()`)* aufgerufen.

Interne Records werden in Dataflows nur verwendet, wenn Funktionen des Proxies zum Einsatz kommen. Da dies in diesem Beispiel nicht der Fall ist, bleibt auch die Methode `button_ProcessInternalRecord()` leer.

Im eigentlichen Dataflow *(`button_Flow()`, ab Zeile 8)* erkennt man die zehn Blöcke der zehn Shapes, die im Dataflow verwendet werden. Für jede von einem Shape ausgehende Kante werden lokale Variablen angelegt, die weiter unten als Eingänge zu den übrigen Shapes verwendet werden. Durch dieses Vorgehen werden derzeit zum Teil unnötig viele lokale Variablen angelegt und so ein wenig Arbeitsspeicher des Sensorknotens verschwendet. Doch wie bereits in Abschnitt 4.4 bemerkt, ist es nicht vorgesehen den Code im Rahmen dieser Diplomarbeit dahingehend zu optimieren. Der Codegenerator ist aber durchaus darauf vorbereitet, in Zukunft solche Optimierungsschritte durchführen zu können. Zuvor sollte allerdings geprüft werden, welche Optimierungen der Compiler bereits selbständig durchführen kann. Durch das gleiche Schema von Codeabschnitten für jedes Shape und neuen Variablen für jede Kante gestaltet sich die Codegenerierung in der aktuellen Version ein wenig einfacher.

Abbildung 7.9 zeigt den Dataflow erneut und gibt mit Zahlen in blauen Kreisen an, in welcher Reihenfolge die Shapes abgearbeitet werden. An den Kanten sind in Rot die lokalen Variablennamen angegeben.

Die Auswertung des Dataflows beginnt in Zeile 10 mit dem Formel-Shape, das die Konstante 2 enthält. Es muss lediglich die Variable `Formula` mit dieser Konstanten gefüllt werden. Ebenso wird die globale Variable i *(`var_i`)* in der lokalen Variable `i` abgelegt *(Zeile 14)*, um dann die Berechnung $(i + 1)$ auf Basis dieses Wertes in Zeile 20 durchführen zu können. Mit der globalen Variable x und dem `NOT`-Operator wird ähnlich verfahren *(Zeile 22-28)*. Dann werden die veränderten Werte zurück in die globalen Variablen gespeichert *(Zeile 30-38)*, zusätzlich werden hier lokale Kopien *(`i_1` und `x_1`)*

```
// ****************** Formula Shapes (if any) **********************
uint32_t button_Formula(uint32_t i)
{
  return i+1;
}

// ****** DataFlow (the actual behavior of the code / model) ********
void button_Flow()
{
  // 2 (Formula)
  uint32_t Formula;
  Formula = 2;

  // i (Variable)
  uint32_t i;
  i = var_i;

  // i+1 (Formula)
  uint32_t Formula_1;
  Formula_1 = button_Formula(i);

  // x (Variable)
  bool x;
  x = var_x;

  // LogicalNot (Operation)
  bool op;
  op = !x;

  // i (Variable)
  var_i = Formula_1;
  uint32_t i_1;
  i_1 = var_i;

  // x (Variable)
  var_x = op;
  bool x_1;
  x_1 = var_x;

  // DataRecord (Record)
  CREATE_USER_STRUCT_INSTANCE(DataRecord_t, DataRecord);
  DataRecord->flag = x_1;
  DataRecord->number = i_1;
  ENCAPSULATE_USER_STRUCT(encapsulated_DataRecord, 1, DataRecord);

  // LED (Output)
  LEDs_statusSet(DataRecord->flag);

  // Transmit Record (Output)
  SendRecord(Formula, &encapsulated_DataRecord);
}

// ******* EventCallbacks (for all events from trigger flow) ********
void button_Callback_RightButtonPressed()
{
  // Event points directly to the MasterTrigger
  button_Flow();
}
```

Abbildung 7.10.: Aus dem Dataflow in Abbildung 7.7 generierter Quellcode

für die ausgehenden Kanten angelegt.

Das „DataRecord“ wird in Zeile 41 angelegt und in den beiden darauf folgenden Zeilen gefüllt, bevor es in Zeile 44 in den allgemeinen Recordtyp verpackt wird. Zuletzt wird in Zeile 47 das LED geschaltet und in Zeile 50 das Record mittels der Methode `SendRecord()` und ihren beiden Parametern verschickt.

7.3. Skomer Island Fallstudie

In Kapitel 1.1 wurde ein konkretes Einsatzszenario von Sensorknoten vorgestellt. Diese Sensorknotenanwendung wurde als Fallstudie mit *Flow* umgesetzt, um zu demonstrieren, dass *Flow* unter realen Bedingungen eingesetzt werden kann. Dazu wurde in Zusammenarbeit mit Microsoft Research Cambridge *(MSRC)* und den Betreuern des Projektes die Anwendung, die derzeit auf den Sensorknoten auf Skomer Island verwendet wird, mit *Flow* nachgebildet und im Labor getestet. Nach einer umfangreicheren Evaluation soll eine Anwendung auf Basis dieser Fallstudie zu weiteren Tests auf Skomer Island eingesetzt werden.

Zur Beobachtung der Schwarzschnabel-Sturmtaucher wird eine spezielle Hardwareerweiterung verwendet, die den Sensorknoten MSB-430H aufnimmt und die benötigten Sensoren sowie die Stromversorgung bereitstellt [NFH+08]. Um diese Hardware mit *Flow* nutzen zu können, wurde ein entsprechender Treiber entwickelt. Dieser Treiber verwendet die Funktionen der vorhandenen Firmware und bietet sie in einer *Flow*-kompatiblen Form an. Auch dieser *Flow*-Treiber ist mit Annotationen versehen, so dass die Hardwaredescription aus Abbildung 7.11 automatisch erzeugt werden konnte. Der Treiber stellt die folgenden Komponenten zur Verwendung mit *Flow* bereit:

- Zwei Bewegungsmelder *(PIR1 und PIR2)*, die vor und in der Höhle der Vögel positioniert werden.
- Einen Input-Port um die Spannung der beiden Batterien zu messen.
- Eine externe Real Time Clock, die ausgelesen werden kann.
- Einen kombinierten Temperatur- und Luftfeuchtigkeitssensor.
- Einen RFID-Reader, dessen Spannungsversorgung für eine gegebene Zeit eingeschaltet werden kann *(RFID Power Activator)*, und der Events auslöst, sobald er einen RFID-Transponder ausgelesen hat.
- Eine Waage, deren Spannungsversorgung ebenfalls für eine gewisse Zeit aktiviert werden kann *(Scale Activator)*, und die, wenn sie eingeschaltet ist, in regelmäßigen Intervallen mittels eines Events Daten liefert.

Für diese Fallstudie arbeiten zwei verschiedene Sensorknotenanwendungen sowie ein Proxy auf einem PC zusammen. Beide Anwendungen sind mit *Flow* programmiert. Die

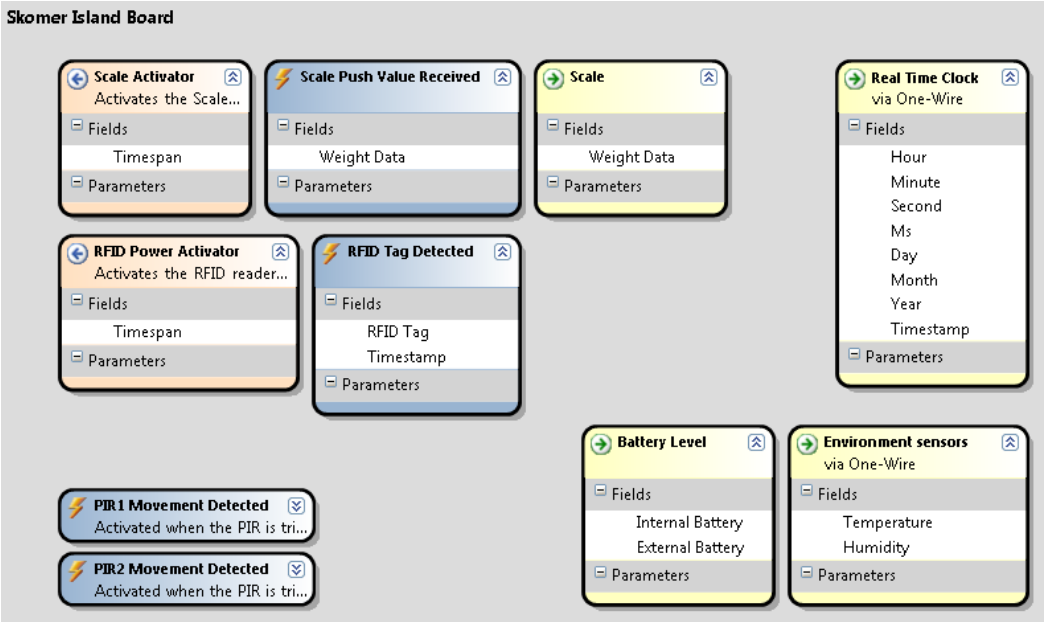


Abbildung 7.11.: Hardwaredescription des Treibers für die Hardwareerweiterung

Hauptknoten sind mit der oben beschriebenen Hardwareerweiterung versehen. Diese Knoten sind in der Nähe der Höhlen positioniert und sammeln dort Messdaten. Ein Logger-Knoten empfängt die Daten aller Hauptknoten und speichert sie an zentraler Stelle auf seiner SD-Speicherkarten. Der PC kann die verschiedenen Knoten abfragen und empfängt ebenfalls einige der Events der Hauptknoten. Für die Zukunft ist geplant, den PC durch ein Embedded Device zu ersetzen, so dass beispielsweise ein PDA zur Diagnose des Sensornetzwerkes verwendet werden kann.

7.3.1. Hauptknoten

Die Knoten, die an den Höhlen der Vögel positioniert sind, müssen einige Aufgaben erfüllen, um sämtliche Messdaten zu ermitteln, diese speichern und weiterleiten zu können:

- Alle 15 Minuten wird die Temperatur und Luftfeuchtigkeit gemessen und in Dateien auf der lokalen SD-Karte gespeichert. Zusätzlich werden die Werte per Funk an den Logger geschickt.

- Wenn einer der Bewegungsmelder eine Bewegung feststellt wird dies zunächst auf der SD-Karte protokoliert. Zusätzlich werden der RFID-Reader und die Waage für 30 Sekunden eingeschaltet. Während dieser Zeitspanne können diese Geräte Events auslösen. Die Daten jedes dieser Events werden auf der lokalen SD-Karte gespeichert und an den Logger gesendet. Zusätzlich werden die Daten als Output Event an den Proxy geschickt.

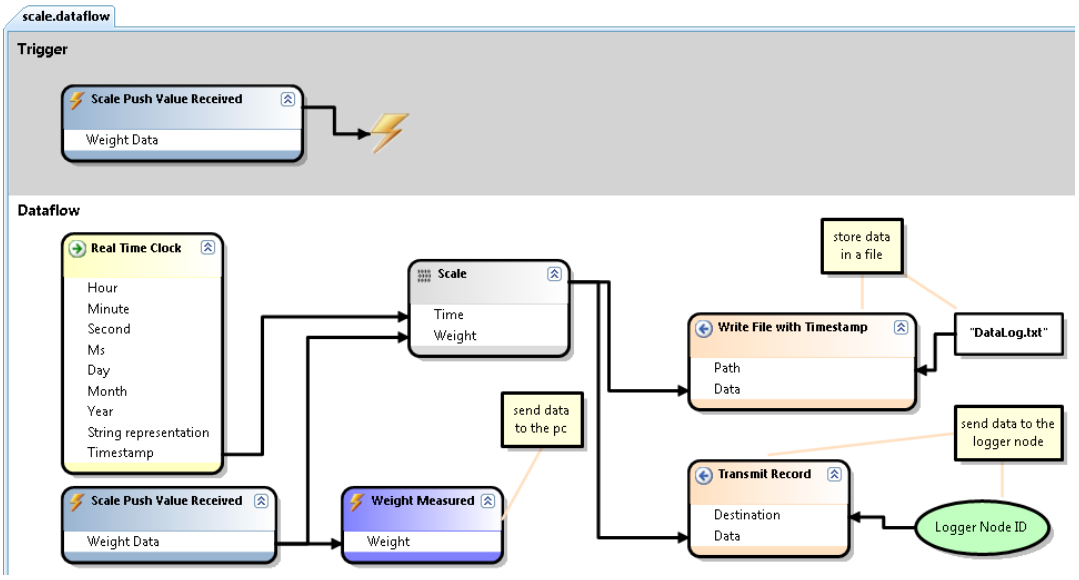


Abbildung 7.12.: Dataflow: Die Messergebnisse der Waage werden gespeichert, an den Logger-Knoten verschickt und ein Output Event ausgelöst.

- Alle 30 Minuten wird die Batteriespannung überprüft. Falls sie unter einen Schwellwert sinkt, wird der Proxy über ein Output Event sowie der Logger informiert.
- Der Proxy kann jederzeit über zwei Funktionen die aktuelle Batteriespannung und die Daten des Temperatur- und Luftfeuchtigkeitssensors abfragen.
- Beim Starten des Sensorknotens wird die interne Real Time Clock mit der Uhrzeit der externen Real Time Clock initialisiert. Außerdem wird auch das Einschalten des Knotens sowie die aktuelle Batteriespannung auf der lokalen SD-Karte protokolliert.
- Um Ereignisse an den Logger schicken zu können, muss dem Knoten die Id des Loggers bekannt sein. Dazu verschickt er nach dem Einschalten eine Anfrage per Broadcast, auf die der Logger mit seiner Id antwortet. Diese wird in einer Variable gespeichert und in den übrigen Dataflows verwendet.

Um diese Funktionen mit *Flow* zu implementieren sind elf Dataflow-Modelle spezifiziert. Eins dieser Modelle ist exemplarisch in Abbildung 7.12 dargestellt.

7.3.2. Logger

Der Logger-Knoten hat die Aufgabe, die von den Hautpknoten produzierten Messdaten entgegenzunehmen und zu speichern. Durch ihn wird eine Sicherheitskopie der Daten erzeugt, die verwendet werden kann, falls die Daten eines Knotens verloren gehen sollten.

Seine Aufgaben sind in folgender Liste aufgeführt:

- Sämtliche empfangenen Records werden in einer Datei gespeichert. Um zwischen den Senderknoten differenzieren zu können, wird für jeden Sender eine Datei mit dem Namen seiner Id angelegt.

- Wenn ein Knoten per Broadcast den Logger sucht, muss dieser mit seiner Id antworten, um in Zukunft Records des Knotens entgegennehmen zu können.

- Ein interner Zähler – der auch über den Proxy ausgelesen werden kann – zählt die Anzahl der empfangenen Records.

Diese Anforderungen sind mittels vier Dataflows modelliert.

7.3.3. Proxy

Der Proxy ist als PC-Anwendung implementiert und greift mittels des ScatterWEB .NET SDKs auf die beiden beschriebenen Sensorknotentypen zu. Er stellt nur eine prototypische Implementierung dar, um die Verbindung eines PCs mit dem Sensornetzwerk zu demonstrieren. Sobald eine Verbindung zum Netzwerk aufgebaut ist, werden in einer Liste *(linke Seite von Abbildung 7.13)* alle gefundenen Hauptknoten angezeigt. Sobald einer dieser ausgewählt wird, werden in einer Liste auf der rechten Seite Ereignisse des Sensorknotens dargestellt. Über die Buttons im oberen Teil können der Batteriestatus sowie die Temperatur und Luftfeuchtigkeit abgefragt werden.

Die Anzahl der vom Logger aufgezeichneten Records kann über den Button in der unteren linken Ecke abgefragt werden. Dazu wird auf die öffentliche Variable des Loggers zugegriffen.

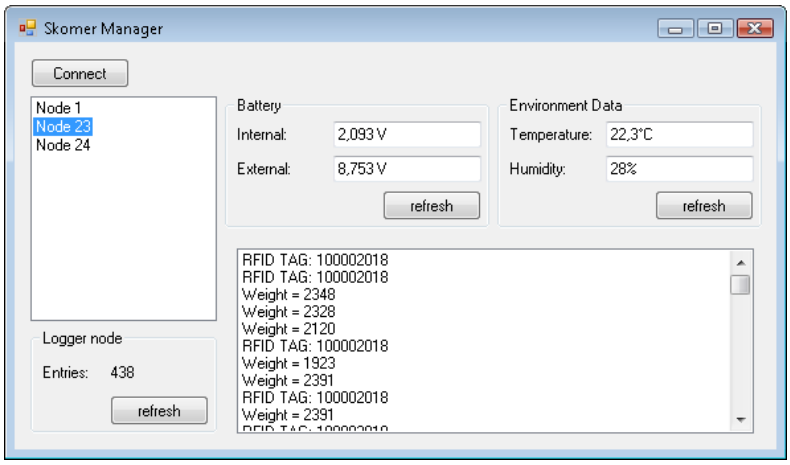


Abbildung 7.13.: PC-Anwendung „Skomer Manager“ zur Fallstudie

7.3.4. Ergebnisse

Diese Fallstudie, bestehend aus zwei verschiedenen Sensorknotenanwendungen sowie einer PC-Anwendung, zeigt, dass mit *Flow* realitätsnahe Anwendungsfälle modelliert und praktisch umgesetzt werden können. Wenn die Firmwarefunktionalität zur Ansteuerung von Hardwareerweiterungen vorhanden ist, kann auf einfache Weise ein *Flow*-Treiber erstellt werden. Dieser Treiber ermöglicht es, die Anwendungslogik graphisch mit den domainspezifischen Sprachen von *Flow* zu spezifizieren. Es ist ohne weiteren Aufwand möglich, die Schnittstellen zu einem Proxy zu definieren und so sehr leicht Verwaltungssoftware für einen PC zu schreiben, ohne spezielle Low-Level-Protokolle implementieren zu müssen. Die Modellierungssprache zur Darstellung der Dataflows ist mächtig genug, um die für diese Fallstudie benötigten Abläufe zu modellieren. Nur an einer Stelle *(um aus der Knoten-Id einen Dateinamen zu erstellen)* musste ein Codeblock und C-Code verwendet werden. Auch die Integration dieses Codes gestaltete sich einfach.

Unabhängig von *Flow* ist die begrenzende Ressource für Sensorknotenanwendungen der verfügbare Programmspeicher. Der Mikrokontroller des MSB-430H besitzt 55 kByte Flashspeicher von dem der Großteil von der Firmware des Sensorknotens und dem Treiber der Skomer Island Hardwareerweiterung belegt wird. Die elf Dataflows, die die oben beschriebenen Aufgaben des Hauptknotens umsetzen, sowie der Infrastrukturcode um auf Anfragen des Proxies reagieren zu können, belegen lediglich 2,8 kByte. Der Speicherverbrauch der aktuellen Softwareversion ist in Abbildung 7.14 dargestellt. Derzeit stehen noch ca. 3 kByte für Erweiterungen zur Verfügung.

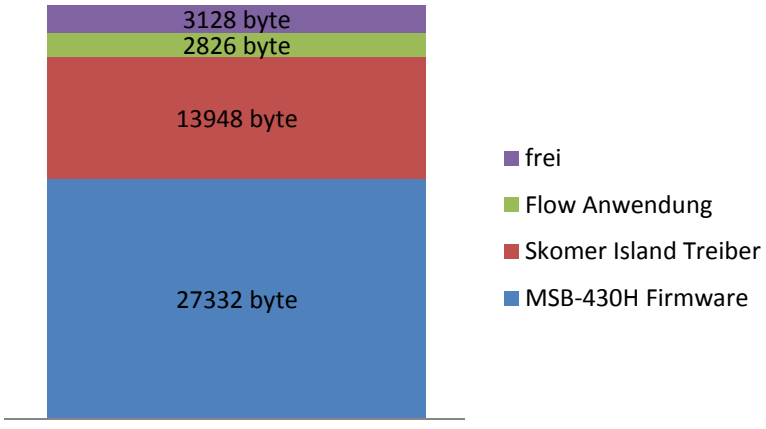


Abbildung 7.14.: Speicherverbrauch der Sensorknotenanwendung zur Fallstudie

8. Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde die Software Factory *Flow* für Embedded Systems entwickelt, die es *(End-)* Anwendern von Sensornetzwerken ermöglicht Software für ihre eingesetzten Sensorknoten zu spezifizieren. Aus diesen Spezifikationen kann – ohne weitere manuelle Schritte – lauffähige Software für die Sensorknoten erzeugt werden, so dass für viele Einsatzzwecke keine Kenntnis der hardwarenahen Programmierung von Mikrocontrollern nötig ist.

Die Spezifikation der Sensorknotenanwendungen erfolgt innerhalb von Visual Studio in drei graphischen domainspezifischen Sprachen, die ebenfalls im Rahmen dieser Diplomarbeit entstanden sind. Diese drei Modellierungssprachen beschreiben unterschiedliche Aspekte einer Sensorknotenanwendung: Die *Hardwaredescription* gibt vor, welche Hardware *(u.a. welche Sensoren und Schnittstellen)* ein Sensorknoten zur Verfügung stellt. In einem *Datastructure*-Modell definiert der Anwender globale Datenstrukturen, die innerhalb der Anwendung aber auch zur Funkkommunikation verwendet werden. Durch mehrere *Dataflows* gibt der Anwender schließlich das Verhalten vor, indem er ereignisgesteuerte Datenflüsse modelliert. Dazu wurden die Modelle so entworfen, dass sie technische Details vor dem Anwender verbergen. Beispielsweise wird auf einer Ebene von Ereignissen, Sensordaten und Funkpaketen gearbeitet, anstatt Interrupts, Callbacks, C-Datentypen und Bytearrays zu verwenden. Bewusst wurde auf die Verwendung von universellen Modellierungssprachen wie UML verzichtet, da nur so eine Umgebung geschaffen werden konnte, die sich nahtlos in die Domäne des Anwenders integrieren kann.

Die Menge an domainspezifischen Sprachen wird durch weitere Tools ergänzt, um eine vollständige Software Factory bereitzustellen. Sämtliche Elemente sind in Visual Studio integriert, so dass alle Arbeiten, vom Anlegen eines neuen Projektes bis zum Aufbringen der fertigen Software auf die Sensorknoten, innerhalb dieser Entwicklungsumgebung durchgeführt werden können. Die Editoren für die Sprachen sind in die Entwicklungsumgebung eingefasst und um Werkzeuge zur einfacheren Modellierung und zur Fehlererkennung erweitert. Dazu wurden die Microsoft DSL Tools verwendet, die allerdings an einigen Stellen um grundlegend benötigte Funktionalität erweitert werden mussten. Des Weiteren wurde Visual Studio um neue Befehle ergänzt, die es dem Anwender erlauben die Funktionen von *Flow* zu verwenden. Auch der Codegenerator, der aus den Modellen C-Code erzeugt, ist innerhalb der Entwicklungsumgebung verfügbar, so dass tatsächlich von einer integrierten Entwicklungsumgebung *Flow* gesprochen werden kann.

Neben dem Verhalten der Sensorknoten, können in den Modellen auch Schnittstellen definiert werden, mit der ein PC auf das Sensornetzwerk bzw. die einzelnen Sensorknoten zugreifen kann. Dazu wird, wenn gewünscht, zusätzlicher Code in den Sensorknotenanwendungen generiert. Eine Proxy-Bibliothek kann erzeugt werden, über die ein PC-Programmierer, ohne Wissen über das Sensornetzwerk, auf dieses zugreifen kann. Dieser Proxy enthält typisierte Methoden, um mit der konkreten Sensorknotenanwendung interagieren zu können.

Während des Entwurfs von *Flow* und allen Werkzeugen wurde großer Wert darauf gelegt, nicht nur eine einzige Hardwareplattform anzusprechen. Die domainspezifischen Sprachen sind so allgemein gehalten, dass sie keine Annahmen über die verwendete Hardware machen. Insbesondere beschreiben erst die Hardwaredescription-Modelle wie die verwendete Hardware aussieht. Um *Flow* anwenden zu können wird daher immer auch eine kompatible Hardwareplattform benötigt. Eine solche Hardwareplattform wurde als Referenzimplementierung für die ScatterWeb MSB-430H Sensorknoten und das MSB-430S Erweiterungsboard erstellt. Somit kann *Flow* in Verbindung mit dieser Plattform und der entsprechenden Hardware praktisch eingesetzt werden. Einige Beispielanwendungen zeigen die grundsätzliche Funktionalität und beschreiben, wie die Modelle zur Programmierung von Sensorknoten verwendet werden. Neben diesen Beispielanwendungen wurde eine Fallstudie, die ein konkretes Szenario nachbildet, mit *Flow* implementiert und zeigt somit, dass *Flow* auch für komplexere Anwendungsfälle nutzbar ist.

8.1. Ausblick

Während der begrenzten Bearbeitungszeit dieser Diplomarbeit wurde die Version 1.0 von *Flow* fertiggestellt aber auch viele weitere Ideen geboren, die noch nicht umgesetzt wurden. Mit der vorliegenden Version 1.0 stellen sowohl die domainspezifischen Sprachen als auch die Werkzeuge um *Flow* herum und *Flow* selbst ein vollständiges System dar, das in der Praxis eingesetzt werden kann. Als nächstes wäre es nun wichtig, aus der Praxis Erfahrungen zu sammeln. Dabei sollte zum einen das Verhalten der Software an sich, aber auch die Art, in der Anwender mit *Flow* arbeiten, beobachtet werden. Wichtige Fragen die zu beantworten sind, wären:

- Können die Anwender jedes gewünschte Verhalten mit *Flow* abbilden, oder sind sie für gewisse Aufgaben in ihren Modellierungsmöglichkeiten eingeschränkt?
- Unterstützen die Modelleditoren die Anwender ausreichend, oder benötigen sie an gewissen Stellen weitere Hilfsmittel?
- Sind gewisse, oft durchgeführte Arbeitsschritte innerhalb von *Flow* zu umständlich, so dass diese durch die Entwicklungsumgebung unterstützt werden sollten?

Schon während des Entwurfs von *Flow*, der Entwicklung und der Anwendungen z.B. zu Testzwecken, beim Erzeugen der Beispielanwendungen und beim Erstellen der Fallstudie sind Ideen entstanden, die in zukünftigen Versionen umgesetzt werden könnten, für die allerdings im Rahmen der Version 1.0 nicht genügend Zeit zur Verfügung stand. Diese Ideen stellen allesamt Erweiterungen dar, die sich nahtlos in die bestehenden Konzepte einfügen und in Zukunft integriert werden können.

Es besteht eine gewisse Trennung, welche Entitäten die Hardwaredescription und die Datastructures bereitstellen können, doch schon jetzt können in beiden Modellen Variablen *(in der Hardwaredescription durch den Hardwarehersteller und in den Datastructures durch den Anwender)* definiert werden. Es erscheint sinnvoll, dem Hardwarehersteller zu erlauben, auch Records und Codeblöcke in einer Hardwareplattform oder einem Treiber bereitzustellen. So könnten auch hardwareunabhängige Treiber z.B. mit mathematischen Funktionen entstehen, die lediglich Codeblöcke zur Verwendung im Dataflow enthalten. Die Codeblöcke, die ein Anwender in Dataflows erstellt, können derzeit nur innerhalb dieses einen Dataflows verwendet werden. Man könnte sich vorstellen, diese Codeblöcke in anderen Dataflows wiederverwendbar zu machen. Es wäre ebenfalls nützlich, auch durch Dataflows Unterprogramme beschreiben zu können, die aus anderen Dataflows aufgerufen werden. Dazu wäre ein neues Shape innerhalb der Dataflows denkbar, das ähnlich wie das Codeblock-Shape Ein- und Ausgänge besitzt, dem aber statt Quellcode ein weiterer Dataflow hinterlegt ist.

Derzeit läuft die Kommunikation zwischen Sensorknoten über das Verschicken und Empfangen von Records ab, für die Kommunikation zwischen einem PC und einem Sensorknoten werden öffentliche Variablen, Output Events und Funktionsaufrufe verwenden. Diese Schnittstellen sollen in Zukunft erweitert werden, so dass auch Sensorknoten auf Output Events und Funktionsaufrufe von anderen Sensorknoten reagieren können. In diesem Zusammenhang wäre es notwendig, globale Datenstrukturen zwischen mehreren Sensorknotenanwendungen explizit definieren zu können, so dass verschiedene Sensorknotenprojekte auf dieselben Definitionen zugreifen können.

Die Modellierung der Dataflows ist stets typisiert, d.h. jedem Element und jeder Kante ist ein Datentyp zugeordnet. Zwei Elemente können nur dann miteinander verbunden werden, wenn die Datentypen übereinstimmen. Dies ist eine wichtige Eigenschaft, um sicher lauffähige Software zu generieren. Da es derzeit keine kompatiblen Kanten unterschiedlichen Typs gibt, sollte die Sensorknotenfirmware durchgängig die gleichen Datentypen für z.B. alle Zahlen verwenden. In Zukunft wäre es wünschenswert, das *Flow*-Typsystem so zu erweitern, dass kompatible Typen angegeben werden können, so dass die Firmware beispielsweise 8, 16 und 32-Bit Integer sowie Gleitkommazahlen anbieten kann. Dazu muss aber *Flow* bekannt sein, wie Typumwandlungen durchgeführt werden bzw. an welchen Stellen dies nicht möglich ist.

Es wäre ebenfalls wünschenswert in einer der folgenden Versionen einen weiteren zentralen Arbeitsschritt in die *Flow*-Entwicklungsumgebung zu integrieren. Die Software-

verteilung könnte mittels einer weiteren domainspezifischen Sprache spezifiziert werden. In einer solchen Sprache könnten die Sensorknoten eines Netzwerkes dargestellt sein und ihnen die verschiedenen Sensorknotenprojekte innerhalb der Solution zugeordnet werden. Basierend auf dieser Beschreibung könnte die Software im Netzwerk verteilt oder Skripte zum Programmieren der Sensorknoten generiert werden.

Damit *Flow* nicht nur als weitere Programmierumgebung für die ScatterWeb-Sensorknoten zur Verfügung steht, wäre es wünschenswert, weitere Hardwareplattformen zu *Flow* kompatibel zu machen. Dazu muss lediglich eine dünne Abstraktionsschicht auf eine vorhandene Firmware aufgesetzt werden und diese *(automatisiert)* in ein Hardwaredescription-Modell überführt werden.

Ein weiterer Schritt für die Verbreitung von *Flow* wäre es, statt nur auf Visual Studio auch auf die kostenlose Visual Studio Shell aufzusetzen. Dadurch wäre es bei geringem Mehraufwand möglich, *Flow* auch an Anwender zu verteilen, die keine Visual Studio Lizenz besitzen.

Auch wenn bereits viele Ideen für die Weiterentwicklung von *Flow* existieren, so stellt bereits die Version 1.0 von *Flow* eine vollständige Software Factory dar, um Sensorknotenanwendungen graphisch auf einem hohen Abstraktionsniveau ereignisorientiert und datengetrieben zu modellieren. Durch die Referenzplattform für die MSB-430H Sensorknoten kann *Flow* sofort eingesetzt werden, um Software für Sensornetzwerke dieser Plattform zu entwickeln.

Glossar

Assembly

.NET-Assemblies stellen die Einheiten dar, in denen kompilierte .NET-Programme oder .NET-Bibliotheken verteilt werden können. Assemblies können als `exe`- oder `dll`-Datei vorkommen und enthalten ausführbaren .NET-Code.

CASE

Computer-Aided Software Engineering. Bezeichnet die Herangehensweise, spezielle Computerprogramme zur Erstellung von Softwarearchitektur zu benutzen. Meist werden dabei Werkzeuge zur UML Modellierung eingesetzt.

COM

Component Object Model. Ein auf Schnittstellen basierendes Objektmodell, 1993 von Microsoft entworfen, das es unter Windows erlaubt, sprachunabhängig Programmteile wiederzuverwenden. Es ist im Ansatz objektorientiert, kennt aber nur Schnittstellenvererbung und keine Basisklassen.

COM-Interopt

COM-Interopt ist der Teil des .NET-Frameworks, der es erlaubt, aus .NET-Anwendungen transparent auf COM-Objekte zuzugreifen *(und umgekehrt)*. Natürlich kann dadurch nur die technische Hürde überwunden werden, beim Programmieren sind die Paradigmen von COM allerdings sehr oft noch sichtbar.

Guid

Globally Unique Identifier: Eine 128 Bit lange Zahl, die meist in der Form `{B7CC3122-A5CD-409D-B933-DCD21D0F3B44}` angegeben wird. Es existieren Algorithmen die Guids so erzeugen, dass sie mit sehr großer Wahrscheinlichkeit einmalig sind.

Literaturverzeichnis

[BHJ+05] Bézivin, Jean ; Hillairet, Guillaume ; Jouault, Frédéric ; Kurtev, Ivan ; Piers, William: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In: *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*. San Diego, California, USA, 2005

[BKLS07] Baar, Michael ; Köppe, Enrico ; Liers, Achim ; Schiller, Jochen: Poster and Abstract: The ScatterWeb MSB-430 Platform for Wireless Sensor Networks. In: *SICS Contiki Hands-On Workshop*. Kista, Sweden, März 2007

[Bly06] Blywis, Bastian: *Entwurf und Implementierung einer modularen und grafisch orientierten Entwicklungsumgebung mit eingebetteter Firmware für den Sensorknoten MSB430*, Freie Universität Berlin, Diplomarbeit, Dezember 2006. – 109 S.

[CJKW07] Cook, Steve ; Jones, Gareth ; Kent, Stuart ; Wills, Alan C.: *Domain Specific Development with Visual Studio DSL Tools*. Addison-Wesley Longman, Amsterdam, 2007. – 576 S. – ISBN 0321398203

[CLRS01] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms*. 2nd Edition. B&T, 2001. – 1184 S. – ISBN 0262032937

[CLZ06] Cheong, Elaine ; Lee, Edward A. ; Zhao, Yang: Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks / EECS Department, University of California, Berkeley. 2006 (UCB/EECS-2006-15). – Technical Report

[Eva03] Evans, Eric J.: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1. Auflage. Addison-Wesley Longman, Amsterdam, 2003. – 560 S. – ISBN 0321125215

[GP92] Green, T. R. G. ; Petre, M.: When visual programs are harder to read than textual programs. In: *Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*. Hungary, 1992, S. 167–180

[GS06] Greenfield, Jack ; Short, Keith: *Software Factories. Moderne Software-Architekturen mit SOA, MDA, Patterns und agilen Methoden*. 1. Auflage. Mitp-Verlag, 2006. – 490 S. – ISBN 3826616103

[Hen06] Hentrich, Benjamin: *Entwurf und Implementierung eines modellgetriebenen architekturzentrierten Eclipse-Frameworks für ScatterWeb*, Freie Universität Berlin, Diplomarbeit, Dezember 2006. – 140 S.

[HI94] Hirakawa, Masahito ; Ichikawa, Tadao: Visual Language Studies - A Perspective. In: *Software - Concepts and Tools* 15 (1994), Nr. 2, S. 61–67

[Hil07] Hillebrandt, Thomas: *Untersuchung und Simulation des Zeit- und Energieverhaltens eines MSB430-H Sensornetzwerkes*, Freie Universität Berlin, Diplomarbeit, Dezember 2007. – 88 S.

[HKB99] Heinzelman, Wendi R. ; Kulik, Joanna ; Balakrishnan, Hari: Adaptive protocols for information dissemination in wireless sensor networks. In: *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–142–9, S. 174–185

[IGE00] Intanagonwiwat, Chalermek ; Govindan, Ramesh ; Estrin, Deborah: Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–197–6, S. 56–67

[Jes05] Jeschke, Frank: *SoftWIRE für ScatterWeb*, Freie Universität Berlin, Studienarbeit, April 2005

[Kam08] Kamenzky, Nicolai: *Entwicklung einer modellgetriebenen Testumgebung zur Unterstützung des Testprozesses von Anwendungen drahtloser Sensornetze*, Freie Universität Berlin, Diplomarbeit, Februar 2008. – 131 S.

[KSLB03] Karsai, G. ; Sztipanovits, J. ; Ledeczi, A. ; Bapty, T.: Model-integrated development of embedded software. In: *Proceedings of the IEEE* 91 (2003), Nr. 1, S. 145–164. – ISSN 0018–9219

[LC02] Levis, Philip ; Culler, David: Maté: a tiny virtual machine for sensor networks. In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002. – ISBN 1–58113–574–2, S. 85–95

[NFH+08] Naumowicz, Tomasz ; Freeman, Robin ; Heil, Andreas ; Calsyn, Martin ; Hellmich, Eric ; Brändle, Alexander ; Guilford, Tim ; Schiller, Jochen: Autonomous Monitoring of Vulnerable Habitats using a Wireless Sensor Network. In: *REALWSN'08 Workshop on Real-World Wireless Sensor Networks*. Glassgow, UK, April 2008

[Pie05] Pietsch, Thomas: *Entwurf und Implementierung einer grafischen Programmierumgebung für Sensorknoten in einem Funknetzwerk*, Freie Universität Berlin, Diplomarbeit, Mai 2005. – 150 S.

[RM99] Rodoplu, Volkan ; Meng, Teresa H.: Minimum energy mobile wireless networks. In: *IEEE Journal on Selected Areas in Communications* 17 (1999), S. 1333–1344

[Sad07] Sadilek, Daniel A.: Prototyping Domain-Specific Languages for Wireless Sensor Networks. In: *ATEM 2007: 4th International Workshop on (Software) Language Engineering*. Nashville, Oktober 2007

[SCC+06] Simon, Doug ; Cifuentes, Cristina ; Cleal, Dave ; Daniels, John ; White, Derek: Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In: *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, 2006. – ISBN 1–59593–332–6, S. 78–88

[Sch97] Schiffer, Stefan: *Visuelle Programmierung. Grundlagen und Einsatzmöglichkeiten*. Addison Wesley Verlag, 1997. – 416 S. – ISBN 382731271X

[Sne95] Snell, Monica: Analysts Predict $3.79 Billion Market for Visual Development Tools by 1999. In: *Computer* 28 (1995), Nr. 3, S. 8–9. – ISSN 0018–9162

[SVEH07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erweiterte Auflage. Dpunkt Verlag, 2007. – 441 S. – ISBN 3898644480

[Tex06] Texas Instruments (Hrsg.): *MSP430x15x, MSP430x16x, MSP430x161x Mixed Signal Microcontroller*. SLAS368E. Dallas, TX, USA: Texas Instruments, August 2006. `http://focus.ti.com/docs/prod/folders/print/msp430f1612.html`

[Tex08] Texas Instruments (Hrsg.): *CC1100: Low-Cost Low-Powet Sub- 1 GHz RF Transceiver*. SWRS038C. Dallas, TX, USA: Texas Instruments, Mai 2008. `http://focus.ti.com/docs/prod/folders/print/cc1100.html`

[Tho04] Thomas, Dave: MDA: Revenge of the Modelers or UML Utopia? In: *IEEE Software* 21 (2004), Nr. 3, S. 15–17. – ISSN 0740–7459

[TWS06a] Terfloth, Kirsten ; Wittenburg, Georg ; Schiller, Jochen: FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks. In: *Proceedings of the First International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE '06)*. New Delhi, India, Januar 2006

[TWS06b] Terfloth, Kirsten ; Wittenburg, Georg ; Schiller, Jochen: Rule-Oriented Programming for Wireless Sensor Networks. In: *Proceedings of the Euro-American Workshop on Middleware for Sensor Networks (EAWMS '06)*. San Francisco, California, USA, Juni 2006

[ZSLM04] Zhang, Pei ; Sadler, Christopher M. ; Lyon, Stephen A. ; Martonosi, Margaret: Hardware design experiences in ZebraNet. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. Baltimore, MD, USA : ACM, 2004. – ISBN 1–58113–879–2, S. 227-238

A. Metamodelle

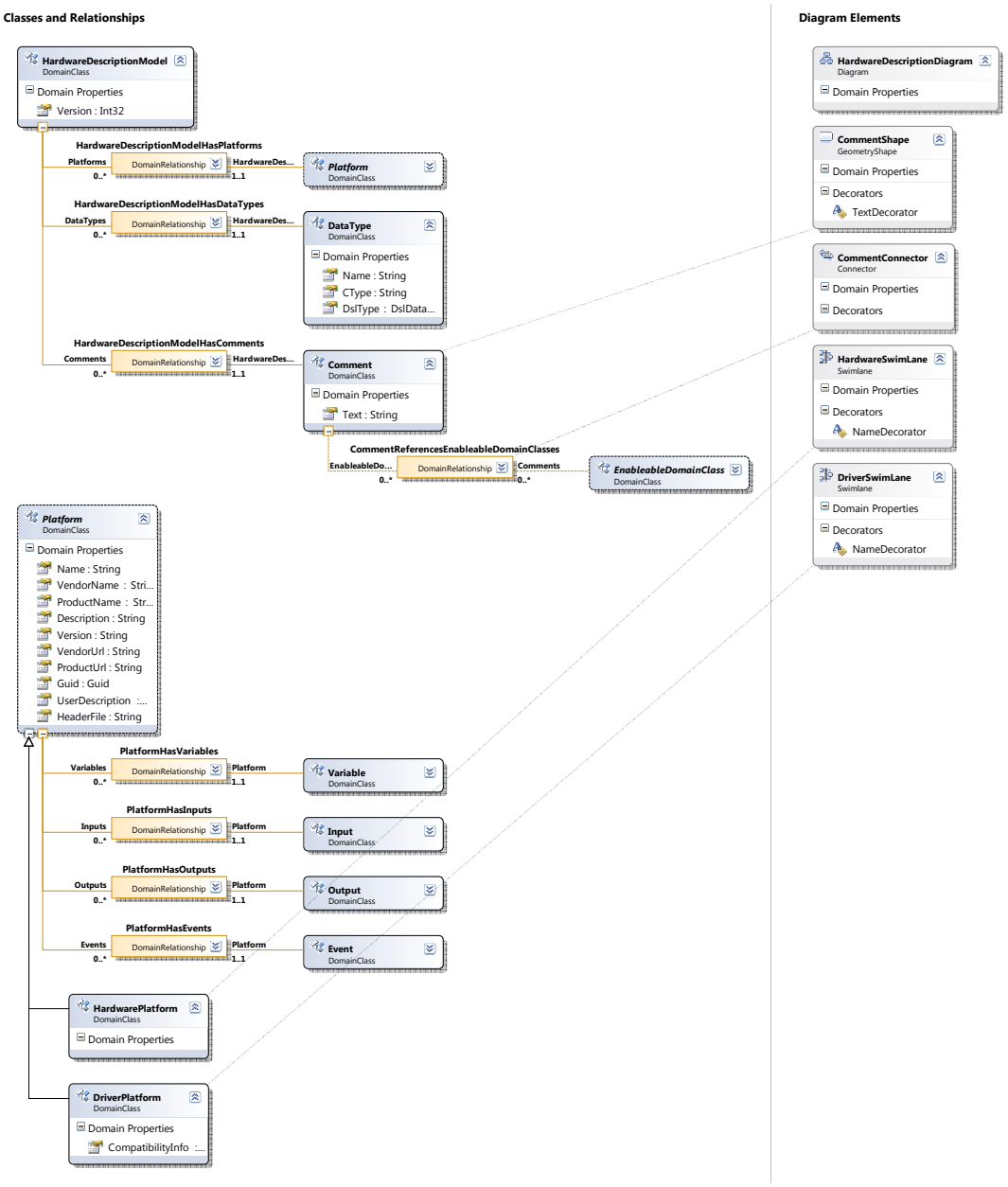


Abbildung A.1.: Hardwaredescription-DSL Metamodell *(Teil 1/3)*

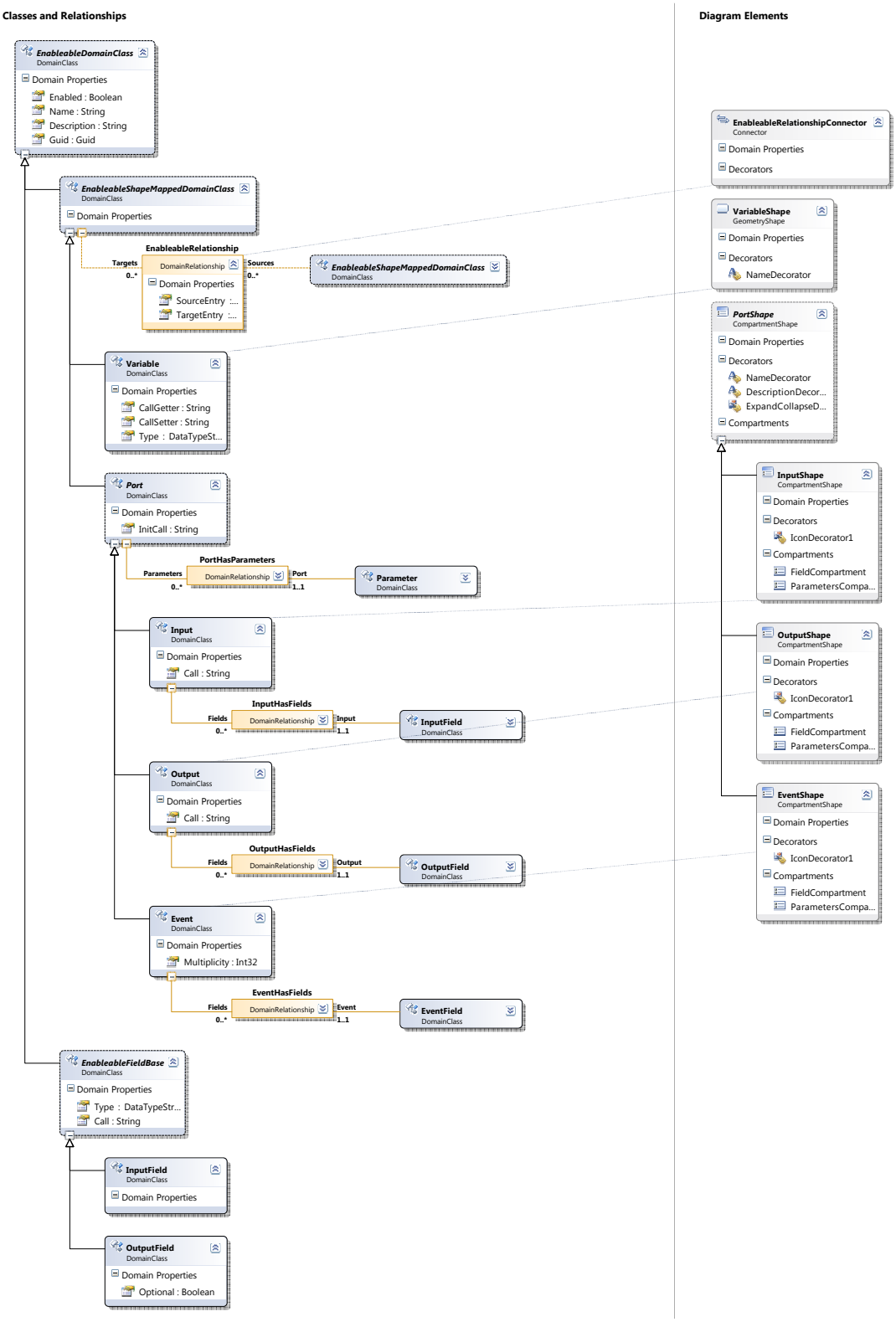


Abbildung A.2.: Hardwaredescription-DSL Metamodell *(Teil 2/3)*

Abbildung A.3.: Hardwaredescription-DSL Metamodell *(Teil 3/3)*

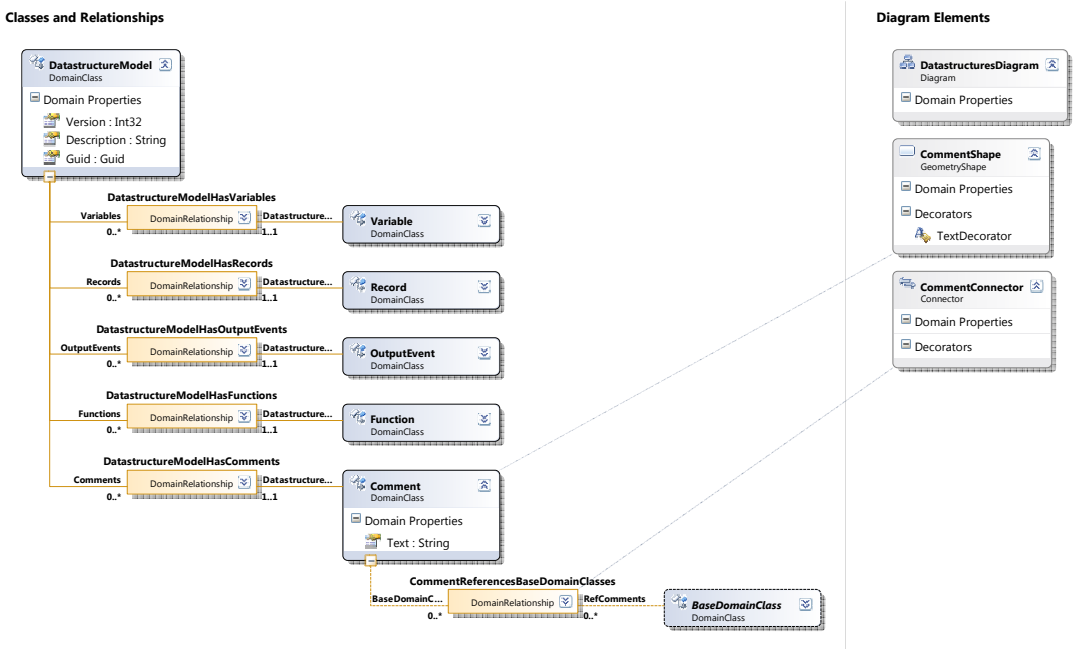


Abbildung A.4.: Datastructures-DSL Metamodell *(Teil 1/2)*

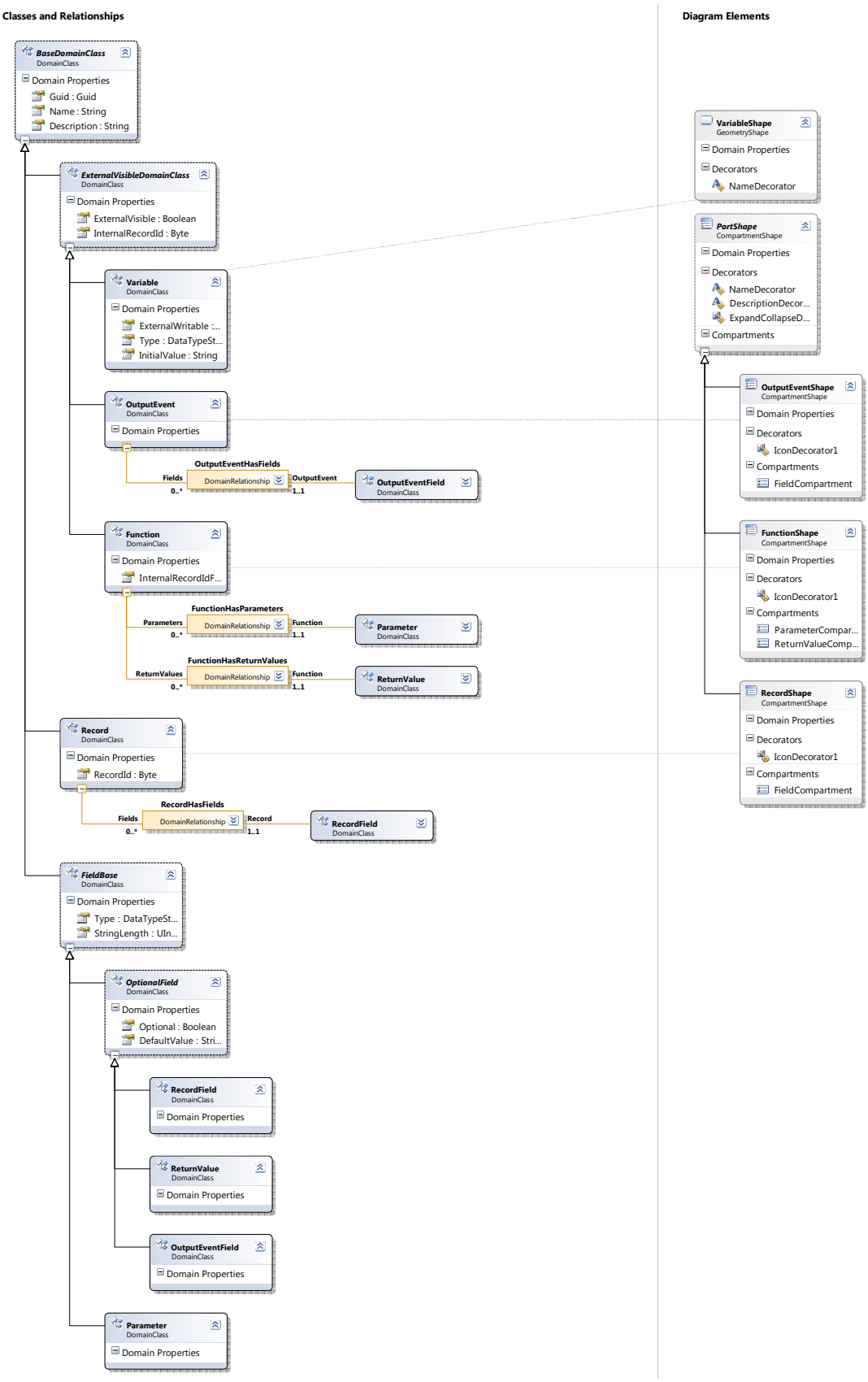


Abbildung A.5.: Datastructures-DSL Metamodell *(Teil 2/2)*

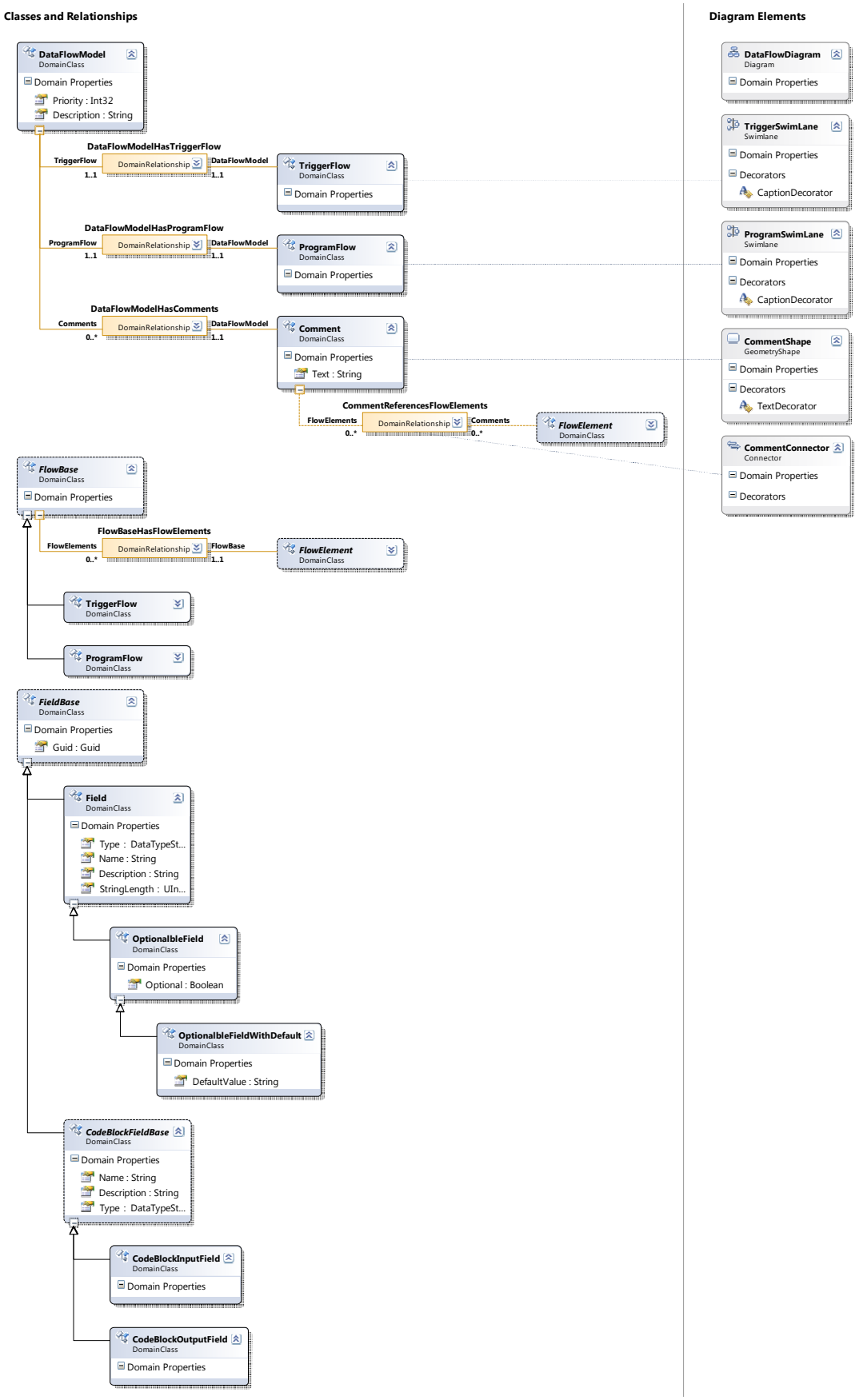


Abbildung A.6.: Dataflow-DSL Metamodell *(Teil 1/3)*

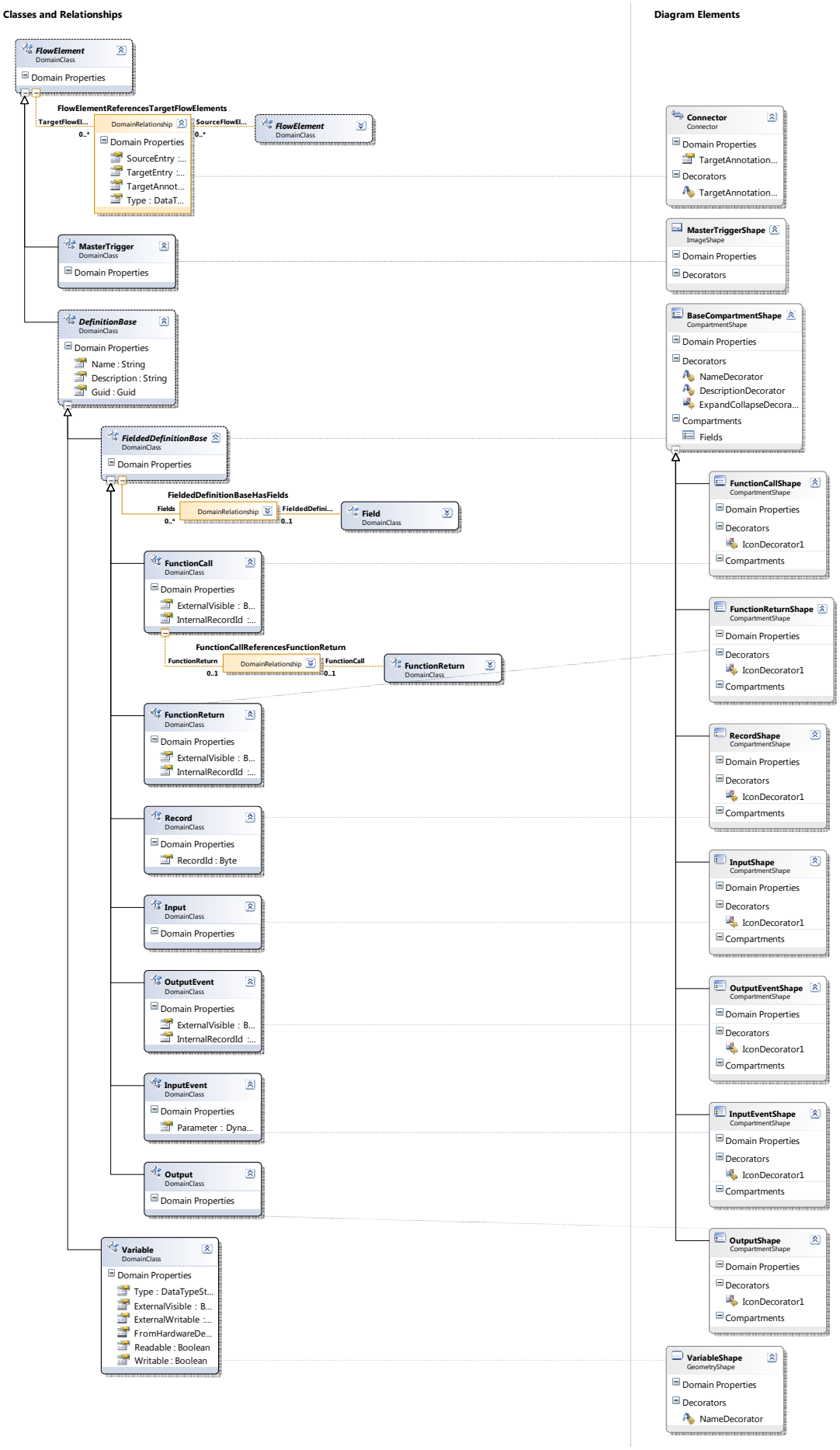


Abbildung A.7.: Dataflow-DSL Metamodell *(Teil 2/3)*

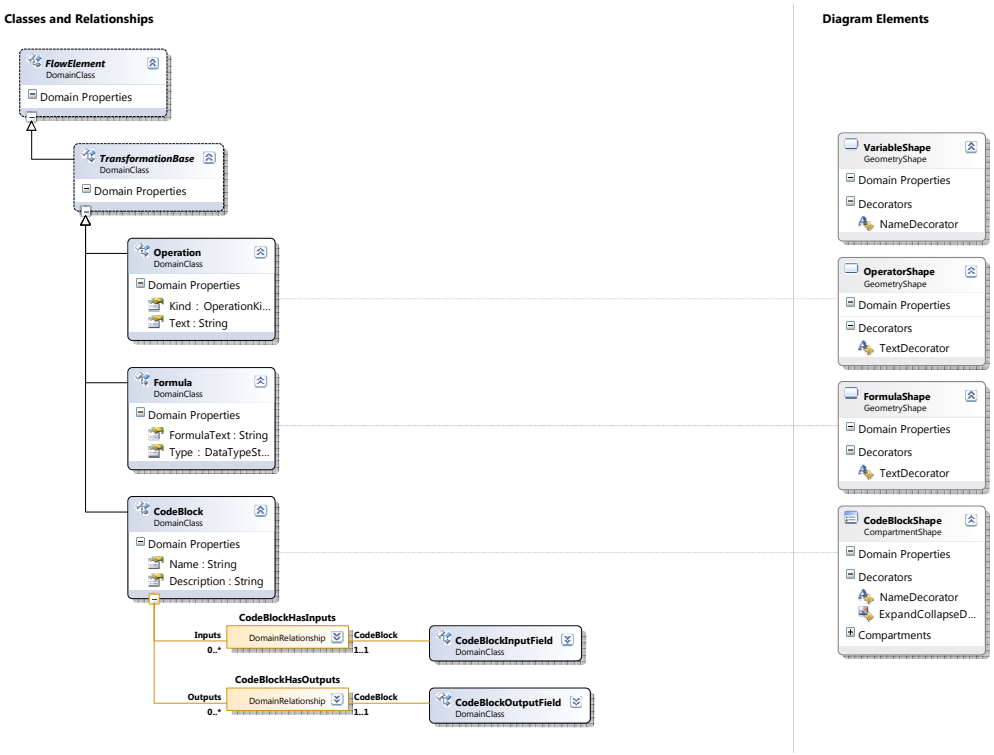


Abbildung A.8.: Dataflow-DSL Metamodell *(Teil 3/3)*

B. Quellcodestruktur

In den folgenden Abschnitten wird die Verzeichnisstruktur der DSL-Projekte von *Flow* beschrieben. In Kapitel 6.2 wurde bereits auf die Art und Weise der Strukturierung eingegangen. Hier soll für jedes Verzeichnis angegeben werden, welche Funktionalität durch die Dateien in ihm implementiert ist. Zusätzlich ist der Umfang in Codezeilen[1] *(Lines of Code, LOC)* angegeben. Auch wenn LOC nicht immer eine aussagekräftige Metrik ist, so gibt sie doch einen Überblick über den Umfang der implementierten Features im Vergleich zueinander.

B.1. Hardwaredescription-DSL

Verzeichnis	**Beschreibung**	**LOC**
`HardwareDescription`		
`HardwareDescriptionDsl`		
`GeneratedCode`	Generierter Quellcode der DSL Tools.	18.983
`UserCode`		
`CallParameter`	Viele Domain Classes besitzen eine `call`-Property, um für den Codegenerator Informationen zu speichern. Die dort verwendeten Datenstrukturen sind hier definiert.	173
`CanMergePlatform`	Verhindert, dass der Anwender dem Modell beliebig Plattformen hinzufügen kann.	5
`Comments`	Erlaubt es, den Kommentartext mehrzeilig anzuzeigen und enthält Logik für die Kanten der Kommentare.	24
`CreateDiagram`	Behandelt das automatische Erzeugen von Plattformen wenn ein neues Diagramm angelegt wird.	19

[1] Zum Messen der Codezeilen wurde der *LinesOfCodeWichtel* von Andreas Berl *(http://www.andreas-berl.de/linesofcodewichtel)* verwendet. Kommentarzeilen und Zeilen mit weniger als drei sichtbaren Zeichen *(z.B. eine einzelne geschweifte Klammer)* wurden nicht mitgezählt.

DataService	Stellt einige weitere Daten für den HardwareDescriptionDataService bereit.	18
DataTypes	Verwaltung von Datentypen, die der Anwender im Modell anlegen kann.	53
DisabledFieldIcon	Anzeige eines Icons für deaktivierte Felder.	38
DisabledShapeColor	Deaktivierte Shapes grau anzeigen.	61
EnableableRelationship	Ermöglicht das Anlegen von Kanten zwischen Feldern und Shapes.	157
GuidCreation	Erzeugt für jedes neue Element eine Guid.	24
ModelCreator	Erstellt aus der eingelesenen Firmware ein Hardwaredescription-Modell.	434
ModelVersion	Verwaltet die Versionsnummer des Modells.	17
Optional	Verhindert das Deaktivieren von nicht-optionalen Feldern von Output-Ports.	12
Restrictions	Sorgt dafür, dass der Editor zu diesem Modell unter gewissen Bedingungen *(z.B. wenn der Anwender statt des Hardwarehersteller damit arbeitet)* nur einen Teil aller Möglichkeiten anbietet.	205
SwimLaneHeader	Entfernt die Köpfe der verwendeten Swimlanes.	8
ToString	Fügt einigen Domain Classes eine ToString()-Methode hinzu.	4
ValidationRules	siehe Anhang C	388
HardwareDescriptionDslPackage		
GeneratedCode	Generierter Quellcode der DSL Tools.	472
Commands	Command-Handler für die Kontextmenüeinträge „Import Driver...“ und „Show Connectors“.	141

OptionPages	Die Options Page, die zur Konfiguration im Visual Studio „Options“ Dialog für *Flow* angezeigt wird.	30
Services	Die beiden Services, die von diesem Package angeboten werden: FirmwareParserService und HardwareDescriptionDataService.	133
UserCode		
GlobalValidation	Integriert die globale Validierung in den Editor.	32
Restrictions	Sorgt dafür, dass der Editor zu diesem Modell unter gewissen Bedingungen nur einen Teil aller Möglichkeiten anbietet.	74
HardwareDescriptionData		
Code	Service Interfaces und Datenklassen für den HardwareDescriptionDataService.	100
GeneratedCode	Generierte Datenklassen.	279
	Summe generiert:	**19.734**
	Summe handgeschrieben:	**2.150**

B.2. Datastructures-DSL

Verzeichnis	**Beschreibung**	**LOC**
Datastructures		
DatastructuresDsl		
GeneratedCode	Generierter Quellcode der DSL Tools.	13.933
UserCode		
Comments	Erlaubt es, den Kommentartext mehrzeilig anzuzeigen und enthält Logik für die Kanten der Kommentare.	24
DataService	Stellt einige weitere Daten für den DatastructureDataService bereit.	16

DataTypes	Ermöglicht den Zugriff auf die in der Hardwaredescription definierten Datentypen und initialisiert neue Variable-Shapes.	11
GuidCreation	Erzeugt für jedes neue Element eine Guid.	12
ModelVersion	Verwaltet die Versionsnummer des Modells.	17
RecordId	Vergibt neuen Shapes automatisch eine noch freie Record-Id.	27
StringLength	Zeigt für den Datentype „String“ in Feldern eine weitere Property „String Length“ an.	12
ToString	Fügt einigen Domain Classes eine `ToString()`-Methode hinzu.	4
ValidationRules	siehe Anhang C	206
DatastructuresDslPackage		
GeneratedCode	Generierter Quellcode der DSL Tools.	457
DataService	Bietet den `DatastructureDataService` an.	36
UserCode		
GlobalValidation	Integriert die globale Validierung in den Editor.	32
IntegrateBaseDataTypes	Bemerkt beim Öffnen eines Modells Änderungen an der zugrundeliegenden Hardwaredescription und passt das aktuelle Modell daran an.	31
DatastructuresData		
Code	Service Interfaces und Datenklassen für den `DatastructureDataService`.	33
GeneratedCode	Generierte Datenklassen.	184
	Summe generiert:	**14.574**
	Summe handgeschrieben:	**461**

B.3. Dataflow-DSL

Verzeichnis	Beschreibung	LOC
DataFlow		
DataFlowDsl		
GeneratedCode	Generierter Quellcode der DSL Tools.	22.531
CodeBlockEditor	Schnittstellendefinition für den CodeBlockEditor-Service.	11
UserCode		
CanDelete	Verhindert das Löschen gewisser Modellelemente.	25
CodeBlock	Aussehen und Verhalten des CodeBlock-Shapes.	23
Comments	Erlaubt es, den Kommentartext mehrzeilig anzuzeigen und enthält Logik für die Kanten der Kommentare.	8
CompartmentHeader	Benennt die Header der Compartment-Shapes für Funktionen um.	14
CompartmentMapping	Steuert das Anlegen von Kanten im Modell.	195
ConnectorAnnotations	Zeigt die Annotationen von Kanten nur dann an, wenn die Kante auf ein Formel-Shape zeigt.	15
CreateDiagram	Behandelt das automatische Erzeugen der beiden Bereiche und des Master-Triggers wenn ein neues Diagramm angelegt wird.	7
DataTypes	Ermöglicht den Zugriff auf die in der Hardwaredescription definierten Datentypen. Ermittelt für das Formel-Shape bei Änderungen den Datentyp.	123
FormulaShape	Zeigt den Text in Formel-Shapes mehrzeilig an.	8
HighlightConnectors	Ändert die Farbe der Kanten, wenn sich die Maus über ihnen befindet.	7

IntegrateBaseChanges	Sucht beim Öffnen eines Modells nach Änderungen an den zugrundeliegenden Modellen und passt das aktuelle Modell daran an.	233
MergeInSwinLane	Steuert, welche Elemente in welchem Bereich *(Trigger bzw. Dataflow)* abgelegt werden dürfen.	88
Operation	Definiert die verschiedenen Operation-Shapes.	43
SwimLaneHeader	Entfernt die Köpfe der verwendeten Swimlanes.	8
ToolboxCreation	Erzeugt Symbole in der Toolbox für die Operation-Shapes.	38
ToString	Fügt einigen Domain Classes eine `ToString()`-Methode hinzu.	16
TypeDescriptors	Registriert die `TypeDescriptor`s für diverse Klassen.	14
ValidationRules	siehe Anhang C	404
DataFlowDslPackage		
GeneratedCode	Generierter Quellcode der DSL Tools.	371
Commands	Command-Handler für den Kontextmenüeintrag „Open CodeBlock Editor...“ der Codeblöcke.	47
UserCode		
CanDelete	Verhindert das Löschen gewisser Modellelemente.	6
ConnectorAnnotations	Zeigt die Annotationen von Kanten nur dann an, wenn die Kante auf ein Formel-Shape zeigt.	5
GlobalValidation	Integriert die globale Validierung in den Editor.	34
IntegrateBaseChanges	Bemerkt beim Öffnen eines Modells Änderungen an der zugrundeliegenden Hardwaredescription oder den Datastructures und passt das aktuelle Modell daran an.	96

`ToolboxCreation`	Erzeugt Symbole in der Toolbox für alle Elemente der zugrundeliegenden Hardwaredescription und den Datastructures.	366
	Summe generiert:	**22.902**
	Summe handgeschrieben:	**1.834**

B.4. Quellcodestatistik

In Kapitel 6 wurden die einzelnen Assemblies von *Flow* beschrieben. In der folgenden Tabelle sind diese Assemblies mit ihrer Anzahl an Codezeilen – differenziert nach handgeschriebenem und generiertem Code – aufgelistet. Diese Tabelle und die folgenden Diagramme sollen nur einen sehr groben Überblick geben, wie groß der Anteil der einzelnen Komponenten am gesamten System ist. Lediglich die LOC zu zählen unterschlägt natürlich u.a. den Aufwand, der für die Erstellung der Metamodelle aufgebracht wurde, obwohl diese Metamodelle eine zentrale Rolle für *Flow* spielen.

Des Weiteren erkennt man, dass der Großteil *(83%)* des Quellcodes durch die DSL Tools generiert wurde. Ohne den Einsatz einer solchen Software Factory wäre es nicht möglich gewesen *Flow* in der gegebenen Zeit zu implementierten. Durch den generierten Code musste weniger Zeit in die Implementierung der Editoren und der Benutzerschnittstelle investiert werden und es stand mehr Zeit für das Design der domainspezifischen Sprachen, des Codegenerators und für die Umsetzung der *Flow*-spezifischen Komponenten zur Verfügung.

Assembly	**LOC handgeschrieben**	**LOC generiert**
`HardwareDescriptionDsl`	1.640	18.983
`HardwareDescriptionDslPackage`	410	472
`HardwareDescriptionData`	100	279
`DatastructuresDsl`	329	13.933
`DatastructuresDslPackage`	99	457
`DatastructuresData`	33	184
`DataFlowDsl`	1.280	22.531
`DataFlowDslPackage`	554	371
`ServiceInterfaces`	52	-
`FlowCommonTypes`	255	-
`Firmwareparser`	839	-
XML Schemas und Transformationen	227	-
`GlobalValidation`	469	-
`UiSupportPackage`	1.619	-
`CodeGenerator`	952	-
Text Templates	953	-
`JaDAL`	1.240	-
`VSXTools`	406	-
Summe:	**11.457**	**57.210**

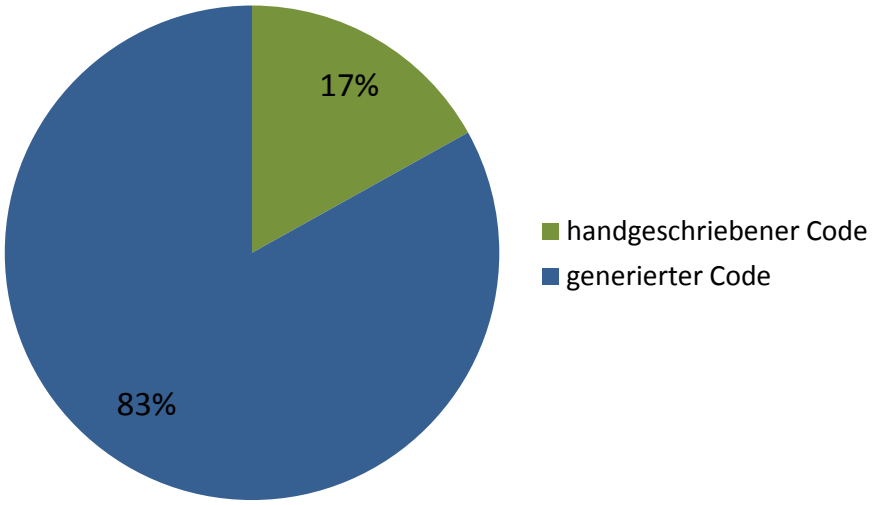


Abbildung B.1.: Verhältnis von handgeschriebenem zu generiertem Code

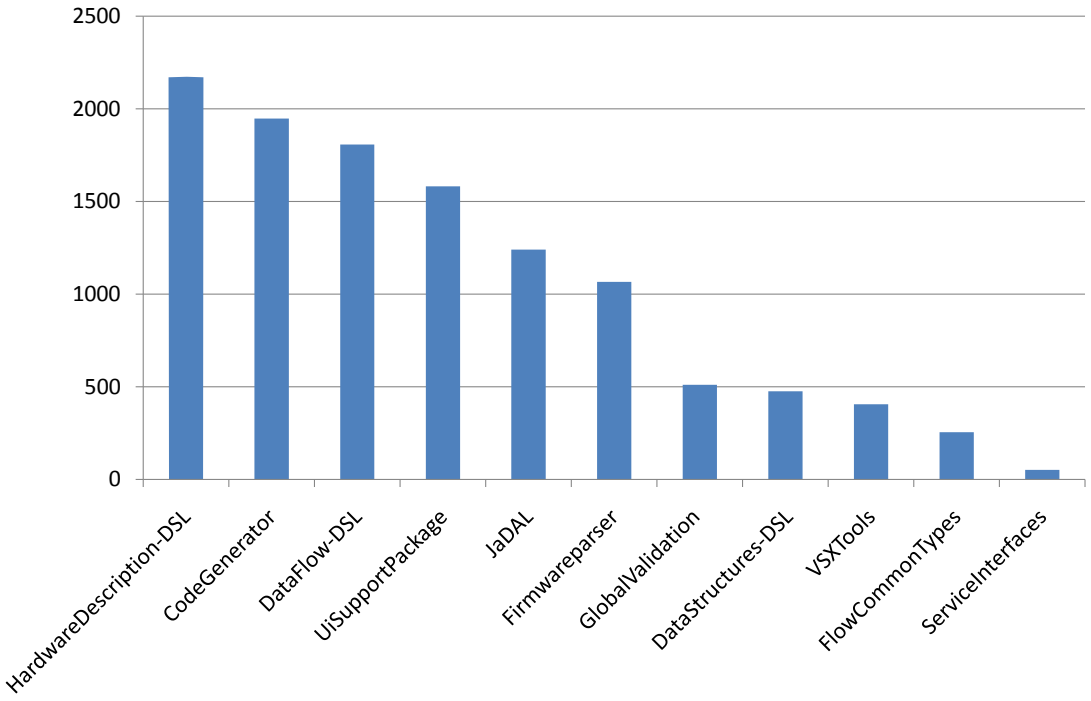


Abbildung B.2.: Handgeschriebene LOC der einzelnen Komponenten von Flow

B.5. Flow Entwicklungssystem

Um *Flow* weiterentwickeln zu können, wird Visual Studio 2008 sowie das Visual Studio SDK benötigt *(siehe Kapitel 3.3)*. Zusätzlich ist die Erweiterung *TTxGen*[2] erforderlich, die sich ebenfalls auf der CD zur Diplomarbeit im Verzeichnis `3rdParty` befindet.

Alle Projekte sind zu der Solution `Flow.sln` zusammengefasst. Obwohl diese Solution sämtlichen Quellcode von *Flow* enthält und beim Kompilieren die Visual Studio Integration Packages in der Registry bei Visual Studio bekannt gibt, müssen einige weitere Einträge in der Registry manuell erstellt werden. Bei diesen Einträgen müssen die Pfade so angepasst werden, dass sie auf die entsprechenden Verzeichnisse und Dateien des Entwicklungssystems verweisen. Dies betrifft natürlich nur ein Entwicklungssystem; auf dem Computer eines Anwenders werden sämtliche Registryeinträge durch das *Flow*-Setupprogramm erzeugt.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\9.0Exp\Configuration\
    TextTemplating\IncludeFolders\.tt]
"IncludeBenjaminSchroeterFlow"="C:\\Flow\\MainCode\\CodeGenerator\\
    TextTemplates"

[HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\9.0Exp\Configuration\
    TextTemplating\DirectiveProcessors\VsProjectFileDirectiveProcessor]
"Class"="BenjaminSchroeter.Dsl.DirectiveProcessors.
    VsProjectFileDirectiveProcessor"
"CodeBase"="C:\\Flow\\MainCode\\Tools\\JaDAL\\bin\\Debug\\JaDAL.dll"

[HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\9.0Exp\Configuration\
    TextTemplating\DirectiveProcessors\XmlFileDirectiveProcessor]
"Class"="BenjaminSchroeter.Dsl.DirectiveProcessors.
    XmlFileDirectiveProcessor"
"CodeBase"="C:\\Flow\\MainCode\\Tools\\JaDAL\\bin\\Debug\\JaDAL.dll"
```

Abbildung B.3.: Registry-File *(`.reg`)* um ein *Flow*-Entwicklungssystem einzurichten

Flow-Setup-Skript

Das Installationsprogramm von *Flow* wird mittels NSIS *(siehe Kapitel 3.4)* erstellt. Das entsprechende Skript nutzt zwei Erweiterungen in Form von Plugins *(Textreplace Plugin und UAC Plugin)*. Diese Plugins befinden sich auf der beigelegten CD im Verzeichnis `3rdParty` und müssen installiert werden, bevor eine neue `FlowSetup.exe` erzeugt werden kann.

[2] http://code.msdn.microsoft.com/TTxGen

C. Validierungsregeln

Um Modelle nach der statischen Semantik zu prüfen und so sicherzustellen, dass sie korrekt modelliert sind, wird bei den DSL Tools ein integriertes Validierungs-Framework eingesetzt *(vgl. [CJKW07, Kapitel 7])*. Dazu werden Validierungsregeln als Methoden innerhalb der Domain Classes implementiert. Diese Methoden überprüfen eine oder mehrere Bedingungen und produzieren bei Bedarf Fehlermeldungen, die dem Anwender mitgeteilt werden. Die Tabellen in den folgenden Anhängen C.1 bis C.3 listen alle Validierungsregeln der drei DSLs auf. Für jede Methode werden die Fehlermeldungen, die sie erzeugen kann, sowie eine Beschreibung angegeben.

Einige projektweite Überprüfungen können allerdings nicht mit diesem Validierungs-Framework umgesetzt werden. Stattdessen wurde der `GlobalValidationService` *(siehe Kapitel 6.5)* eingeführt. Die Regeln die dieser Service überprüft sind in Anhang C.4 aufgelistet.

Auch der Firmware-Parser *(siehe Kapitel 6.4)* führt nach der Bearbeitung eine Überprüfung des Quellcodes durch. Die entsprechenden Regeln sind in Anhang C.5 zu finden.

C.1. Hardwaredescription-DSL

Die folgenden Regeln prüfen die Korrektheit der Namen verschiedener Elemente in einem Hardwaredescription-Modell:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`EnableableDomainClass .ValidateName()`	The name of a <objecttype> must not be empty.	Leere Namen bei Elementen im Modell sind nicht erlaubt.
`FieldBase.ValidateName()`	”	Leere Namen bei Feldern und Parametern sind nicht erlaubt.
`DataType.ValidateName()`	The name of a DataType must not be empty.	Leere Namen bei Datentypen sind nicht erlaubt.
`HardwareDescriptionModel .ValidateNames()`	The Name (<name>) of the following elements must be unique: <elements>	Eindeutigkeit der Datentyp-Namen.
`Platform.ValidateNames()`	”	Eindeutigkeit der Namen von Input-Ports, Output-Ports, Events und Variablen.
`Port.ValidateParameter()`	”	Eindeutigkeit der Parameter-Namen.
`Input.ValidateFields()` `Output.ValidateFunction ParameterUniqueness()` `Event.ValidateFields()`	”	Eindeutigkeit der Feld-Namen.

Die folgenden Regeln prüfen, ob alle Elemente mit Datentypen versehen sind und ob diese Datentypen korrekt definiert sind:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`FieldBase.ValidateType()` `EnableableFieldBase` `.ValidateType()`	The Type of the Field '<name>' must not be empty.	Für jedes Feld muss ein Datentyp gesetzt sein.
`Variable.ValidateType()`	The Type of the Variable '<name>' must not be empty.	Für jede Variable muss ein Datentyp gesetzt sein.
`DataType.ValidateCType()`	The C Type of a DataType must not be empty.	Der C-Typ einer Datentypdefinition darf nicht leer sein.
`EnableableFieldBase` `.ValidateType()` `FieldBase.ValidateType()`	The Type '<type>' of the Field '<name>' is not defined in this Hardware Description.	Es wird ein Datentyp verwendet, der nicht mehr in der Hardwaredescription enthalten ist. Dies kann vorkommen wenn Datentypen gelöscht werden.
`Variable.ValidateType()`	The Type '<type>' of the Variable '<name>' is not defined in this Hardware Description.	"
`Parameter.ValidateValue()`	For the Parameter '<name>' of a <parenttype> you must not use a DataType that maps to a Record.	Parameter können nicht vom Typ „Record“ sein, da für diesen Typ vom Anwendern kein Wert eingegeben werden kann.

Die folgenden Regeln prüfen, ob die Plattformen korrekt im Modell verwendet wurden. Da *Flow* für die Verwaltung der Plattformen zuständig ist, sollten diese Regeln immer erfüllt sein.

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`HardwareDescriptionModel .ValidatePlatformCount()`	The Hardware Description must contain exactly one Hardware Platform.	In einer Hardwaredescription-Datei darf nur genau eine Hardwareplattform definiert sein.
`HardwareDescriptionModel .ValidatePlatformCount()`	The Driver Description must contain exactly one Driver Platform.	In einer Driverdescription-Datei darf nur genau ein Treiber enthalten sein.
`HardwareDescriptionModel .ValidatePlatformCount()`	The Driver Description must not contain any Hardware Platform.	”

In der folgenden Tabelle sind verschiedene weitere Regeln angegeben:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`Parameter.ValidateValue()`	The Value (’<value>’) of Parameter ’<name>’ is not a boolean value. Please use ’True’ or ’False’.	Für einen Parameter wurde ein Wert eingegeben, der nicht zum Datentyp des Parameters kompatibel ist.
`Parameter.ValidateValue()`	The Value (’<value>’) of Parameter ’<name>’ is not a numeric value.	”
`Input.ValidateFields()` `Output.ValidateFields()`	The Input / Output ’<name>’ must contain at least on filed.	Input- und Output-Ports benötigen immer mindestens ein Feld.
`EnableableRelationship .ValidateEnabledFlags()`	’<Element1>’ and ’<element2>’ must not both be enabled since an Enableable Relationship exists.	Überprüft die Bedingungen, die durch die Kanten im Hardwaredescription-Modell vorgegeben sind *(siehe Abschnitt 5.1.6)*.

Die folgenden Regeln prüfen, ob die angegebenen Methoden und Methodenparameter der einzelnen Elemente korrekt und konsistent gefüllt sind. Diese Informationen werden vom Firmware-Parser gefüllt und vom Codegenerator genutzt. Normalerweise sollten hier keine Fehler auftreten.

Da die Struktur dieser Parameter aus Platzmangel nicht innerhalb der Diplomarbeit beschrieben wurde, werden hier die Regeln nur aufgelistet, nicht aber detailliert beschrieben.

Validierungsregel	**Fehlermeldung**

`Event.ValidateFunctionParameter Uniqueness()`
`Input.ValidateFunctionParameter Uniqueness()`
`Output.ValidateFields()`
	The Call Properties of Event '<name>' are invalid. Some Function Parameter Positions are used more than once or the Function Parameter are not counting consistently from 1.

`Variable.ValidateCalls()`	The Call Properties of Variable '<name>' are both empty. Please provide at least a Getter or a Setter (or both).
`EnableableFieldBase.ValidateCall()`	The Call Property of Field '<name>' of '<parentname>' is invalid. You must provide the position in the parameter list or a function name but not both.
`Input.ValidateCall()` `Output.ValidateCall()`	The Call Property of Input / Output '<name>' is invalid. You must provide a Function Name for the Shape or one for every single Field.
`Parameter.ValidateCall()`	The Call Property of Parameter '<name>' of '<parentname>' does not contain the position in the parameter list.
`EventField.ValidateCall()`	The Call Property of Parameter '<name>' of '<parentname>' is invalid. You must provide only the position in the parameter list.
`Parameter.ValidateCall()`	The Call Property of Parameter '<name>' of '<parentname>' is invalid. You should only provide the position in the parameter list.

`OutputField.ValidateOptional()`	The Field '<name>' of Output '<parentname>' must not be Optional since no Function Call is defined in the Call Property.
`Variable.ValidateCalls()`	The Get / Set Call Property of Variable '<name>' is invalid. Please provide only the Function Name.
`Port.ValidateInitCall()`	The InitCall Property of <objecttype> '<name>' is invalid. Please provide a Function Name since you have defined an enabled position or parameter for this shape.
`Port.ValidateInitCall()`	The InitCall Property of <objecttype> '<name>' is invalid. You can provide a Function Name or a Function Name and 'enabled' Parameter.
`EnableableFieldBase .ValidateInitCall()`	The InitCall Property of '<parentname>' is invalid. Please provide a Function Name since you have defined an enabled position on field '<name>'.
`Event.ValidateMultiplicity()`	The Multiplicity of the Event '<name>' must be equal or greater than 1.

C.2. Datastructures-DSL

Die folgenden Regeln prüfen die Korrektheit der Namen verschiedener Elemente der Datastructure-Modelle:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`BaseDomainClass .ValidateName()`	The name of a <objecttype> must not be empty.	Leere Namen bei Elementen des Modells sind nicht erlaubt.
`DatastructureModel .ValidateNames()`	The name (<name>) of the following elements must be unique: <elements>	Eindeutigkeit der Namen von Variablen, Records, Output Events und Funktionen.
`Record.ValidateFields()` `OutputEvent .ValidateFields()`	”	Eindeutigkeit der Feld-Namen.
`Function.ValidateFields()`	”	Eindeutigkeit der Namen von Parametern und Rückgabewerten der Funktionen.

In der folgenden Tabelle sind die Regeln aufgelistet, die Records und Record-Ids prüfen:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`Record .ValidateFieldCount()`	The record ’<name>’ must contain at least one field.	Records müssen mindestens ein Feld enthalten.
`DatastructureModel .ValidateRecordIds()`	The RecordId (<id>) of the following elements must be unique: <records>	Jedes Record muss eine eindeutige Id haben.
`DatastructureModel .ValidateInternalRecordIds()`	The Internal RecordId (<id>) of the following elements must be unique: <elements>	Die interne Record-Id von Variablen, Output Events und Funktionen muss eindeutig sein.

`Function` `.ValidateInternalIds()`	The two Internal RecordIds of the function '<name>' must not be the same.	Funktionen benötigen zwei interne Record-Ids *(eine für den Funktionsaufruf und eine für die Rückgabewerte)*, die verschieden sein müssen.

Die folgenden Regeln prüfen, ob alle Elemente mit Datentypen versehen sind und diese Datentypen korrekt verwendet werden:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`FieldBase.ValidateType()` `Variable.ValidateType()`	The Type of Field / Variable '<name>' must not be empty.	Für jedes Feld sowie die Parameter und Return Values von Funktionen und Variablen muss ein Datentyp gesetzt sein.
`FieldBase.ValidateType()` `Variable.ValidateType()`	The Type ('<type>') of the Field / Variable '<name>' is not defined in the Hardware Description.	Die im Datastructure-Modell verwendeten Datentypen müssen in der Hardwaredescription definiert sein. Dieser Fehler kann nur auftreten, wenn die Modelle nicht synchron sind oder die Hardwaredescription im Projekt fehlt.
`Variable` `.ValidateExternalType()`	A Variable ('<name>') of Record or Other Data Type must not be external visible or external writable.	Variablen, die einen Record aufnehmen, können nicht über den Proxy öffentlich sichtbar gemacht werden.

Die Regeln der folgenden Tabelle überprüfen Werte, die der Anwender als Standard- oder Initialwert angegeben hat:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`OptionalField` `.ValidateDefaultValue()` `Variable` `.ValidateInitialValue()`	The Default / Initial Value (’<value>’) of Field ’<name>’ is not a boolean value. Please use ’True’ or ’False’.	Für boolesche Felder und Variablen: Gültigkeit des Standardwertes.
`OptionalField` `.ValidateDefaultValue()` `Variable` `.ValidateInitialValue()`	The Default / Initial Value (’<value>’) of Field ’<name>’ is not a numeric value.	Für numerische Felder und Variablen: Gültigkeit des Standardwertes.
`OptionalField` `.ValidateDefaultValue()`	The Field ’<name>’ cannot be optional, since of its Data Type.	Für Felder, die Records enthalten und Variablen vom Typ Record darf kein Standardwert angegeben werden.
`Variable` `.ValidateInitialValue()`	The Variable ’<name>’ cannot have an Initial Value, since of its Data Type.	”

C.3. Dataflow-DSL

Um die Dataflows auswerten zu können, existieren gewisse Vorgaben, welche Elemente in den Trigger- und Dataflow-Bereichen verwendet werden dürfen *(siehe Abschnitt 5.3.3)*. Diese Bedingungen werden durch die folgenden Regeln überprüft:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`TriggerFlow .ValidateInputEventCount()`	At least one Event or Function Call is needed for the trigger part of this model.	Der Trigger-Bereich benötigt mindestens ein Event oder eine Funktion.
`TriggerFlow .ValidateEventUniqueness()`	The Event '<name>' must only have one instance within the trigger part of this model.	Dasselbe Event ist nur einmal in Trigger-Bereich zugelassen.
`DataFlowModel .ValidateEvents()`	The Event in the Trigger Flow and Data Flow must be the same Event.	Im Dataflow-Bereich dürfen keine neuen Events auftauchen, die nicht auch im Trigger-Bereich existieren.
`DataFlowModel .ValidateEvents()`	Only one event is allowed in the Data Flow.	Sollen Daten aus Events im Dataflow-Bereich verwendet werden, so darf nur ein Event im gesamten Dataflow vorkommen.
`DataFlowModel .ValidateEvents()`	When using events in the Data Flow only one event is allowed it the Trigger Flow.	"
`TriggerFlow .ValidateInputEventCount()`	Only one Function Call is allowed in the trigger part of this model.	Werden Funktionen im Trigger-Bereich verwendet, so darf nur eine einzige Funktion und keine weiteren Events vorkommen.
`TriggerFlow .ValidateInputEventCount()`	Only one Function Call or Events are allowed in the trigger part of this model, not both.	"

`FunctionCall .ValidatePosition() FunctionReturn .ValidatePosition()`	The empty Function '<name>' must not be present in the Data Flow, only in the Trigger Flow.	Nur wenn Funktionen des Trigger-Bereichs Parameter enthalten, dürfen diese Parameter auch im Dataflow-Bereich angezeigt werden.
`FunctionCall .ValidatePosition() FunctionReturn .ValidatePosition()`	The Function '<name>' used in the Data Flow cannot be found in the Trigger Flow.	"

Die Kanten definieren im Dataflow das Verhalten und steuern so die einzelnen Elemente an. Die folgende Tabelle gibt die Regeln an, die bei der Modellierung der Kanten geprüft werden:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`MasterTrigger .ValidateInputs()`	At least one Input Connection is needed for the Master Trigger in the Trigger Flow.	Der Master-Trigger muss durch mindestens einen „Impuls“ ausgelöst werden.
`InputEvent.ValidateOutgoingConnections()`	The Event '<name>' must not have outgoing connections from both the Head and the Elements.	Im Trigger-Bereich kann entweder nur das Event oder seine Daten als „Impuls“ für den Master-Trigger verwendet werden.
`Variable.ValidateInput OutputCount()`	The Variable '<name>' is not readable / writable. You must not create any outgoing / incoming connections.	Prüft ein- und ausgehende Kanten von Variablen.
`FlowElementReferences TargetFlowElements .ValidateType()`	The connection from '<name1>' to '<name2>' is invalid, since the Datatypes of Source ('<type1>') and Target ('<type2>') are not the same.	Kanten sind nur zwischen Feldern vom gleichen Datentyp zulässig.

`Output` `.ValidateMissingInputs()` `OutputEvent` `.ValidateMissingInputs()` `Record` `.ValidateMissingInputs()` `FunctionReturn` `.ValidateMissingInputs()`	The Field '<name>' of <parenttype> '<parentname>' must be set by an incomming connection.	Gewisse Felder müssen zwingend gesetzt werden, wenn die entsprechenden Shapes verwendet werden.
`Output` `.ValidateMissingInputs()`	The Output '<name>' must have at least one incomming connection.	Output-Ports ohne eingehende Kanten dürfen im Modell nicht vorkommen, da sie keine Bedeutung für die Ausführung haben.
`CodeBlock` `.ValidateMissingInputs()`	The Field '<name>' of Codeblock '<parentname>' must be set by an incomming connection.	Wenn ein Codeblock verwendet wird, muss jeder Parameter durch eine eingehende Kante gesetzt werden.
`Operation` `.ValidateInputCount()`	The <type> Operation must not have more than <number> incomming connections.	Die booleschen Operatoren haben je nach Typ Einschränkungen für die Anzahl der eingehenden Kanten.
`Operation` `.ValidateInputCount()`	The <type> Operation needs at least <number> incomming connections.	"
`Record` `.ValidateMissingInputs()`	The Record '<name>' can only have an incomming Connection to the Head or to the Fields, not both.	Records können entweder als Einheit gesetzt oder ihre Felder einzeln gefüllt werden. Beides zugleich ist nicht möglich.
`Record` `.ValidateMissingInputs()`	The Record '<name>' in the Trigger part of the Dataflow must be set by a connection to the head. Connections into the fields are not allowed.	...im Trigger-Bereich dürfen die Felder niemals gesetzt werden.

Die Elemente und Kanten der Dataflows bilden Graphen, die den Bedingungen der folgenden Tabelle genügen müssen:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`FlowBase .ValidateNoCircle()`	Some Elements in the Trigger / Dataflow Part are forming an illegal circle.	Kreisfreiheit der Graphen.
`TriggerFlow.ValidateEvent GraphComponentsOnlyOne Event()`	Each sub graph of the tigger part must contain only one event.	Im Trigger-Bereich dürfen nicht mehere Events gemeinsam zu einem „Impuls“ für den Master-Trigger werden.
`TriggerFlow.ValidateEvent GraphComponentsLeadTo Trigger()`	The <elementtype> '<name>' is not connected with the rest of the trigger part of the model. It must be connected with the master trigger in some way.	Alleinstehende Elemente im Trigger-Bereich sind nicht zugelassen.

Einige Regeln überprüfen weitere Bedingungen, die im Normalfall immer erfüllt sind, da der Editor dies entsprechend steuert:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`DataFlowModel .ValidateModelBasics()`	Trigger and Data Flow must not be null.	Es muss immer genau ein Trigger- und ein Dataflow-Bereich existieren.
`FlowElementReferences TargetFlowElements .ValidateGuids()`	The Source / Target Guid of the Connection can not be found in the Source / Target Shape '<elementname>': <guid>	Die Kanten müssen eindeutig den Shapes und Feldern zugeordnet werden können.
`FlowElementReferences TargetFlowElements .ValidateGuids()`	The Source / Target of the Connection is not a valid Shape.	”

Bei der Verwendung von Formel-Shapes hat der Anwender mehr Freiheiten, die durch Regeln eingeschränkt werden müssen, als bei den übrigen Shapes:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`FlowElementReferences TargetFlowElements .ValidateAnnotations()`	The annotation of the connection to the Formula '<text>' must not be empty.	Die eingehenden Kanten der Formel-Shapes erhalten eine Annotation *(siehe Abschnitt 5.3.2)*. Dieser Wert darf nicht leer sein.
`FlowElementReferences TargetFlowElements .ValidateAnnotations()`	The annotation name '<name>' is not a valid identifier.	...des Weiteren muss dieser Wert einen gültigen C-Parameternamen darstellen.
`Formula.ValidateFormula()`	*individuelle Fehlermeldung*	*Die eingegebene Formel wird syntaktisch überprüft und falls nötig eine entsprechende Fehlermeldung ausgegeben.*

Codeblöcke sind die einzigen Shapes, die der Anwender im Dataflow frei konfigurieren kann *(alle übrigen Shapes sind in den anderen Modellen definiert)*. Daher müssen für Codeblöcke weitere Regeln überprüft werden:

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`CodeBlock.ValidateCName()`	The Name of a Code Block must not be empty.	Der Name eines Codeblocks darf nicht leer sein.
`CodeBlock.ValidateCName()`	The Name '<name>' of a Code Block is not a valid identifier.	...und muss einen gültigen C-Namen darstellen.
`DataFlowModel .ValidateCodeblockNames()`	The Name (<name>) of the following elements must be unique: <elements>	Des Weiteren muss der Name innerhalb des Dataflows eindeutig sein.
`CodeBlockFieldBase .ValidateCName()`	The Name of the Field of the Code Block '<name>' must not be empty.	Der Name eines Parameters oder Return Values darf nicht leer sein.
`CodeBlockFieldBase .ValidateCName()`	The Name '<name>' of a Field of the Code Block '<name>' is not a valid identifier.	...und muss einen gültigen C-Namen darstellen.
`CodeBlock.ValidateNames()`	The Name (<name>) of the following elements must be unique: <elements>	Des Weiteren muss der Name innerhalb des Codeblocks eindeutig sein.
`CodeBlock .ValidateFieldsCount()`	The CodeBlock '<name>' must have at least one Parameter or one Return Value.	Codeblöcke ohne Parameter und Return Values können im Dataflow nicht verwendet werden, daher sind sie nicht zugelassen.
`CodeBlockFieldBase .ValidateType()`	The Type of the Code Block Field '<name>' must not be empty.	Die verwendeten Datentypen im Codeblock müssen in der Hardwaredescription definiert sein.
`CodeBlockFieldBase .ValidateType()`	The Type ('<type>') of the Code Block Field '<name>' is not defined in the Hardware Description.	"

C.4. Global Validation

Durch den `GlobalValidationService` *(siehe Kapitel 6.5)* werden einige Regeln geprüft, die über das Validierungs-Framework der DSL Tools hinausgehen. Jede Regel ist in einer Klasse, die von `GlobalValidationRule` erbt, implementiert. In der folgenden Tabelle sind diese Klassen sowie die von ihnen produzierten Fehlermeldungen mit einer kurzen Beschreibung aufgeführt.

Validierungsregel	**Fehlermeldung**	**Beschreibung**
`FileExist`	The file '<name>' does not exist.	Prüft, ob alle Modelle des Projektes existieren.
`DataFlowFilenameUniqueness`	The project contains more than one file with the name '<name>': <paths>	Auch in Unterverzeichnissen dürfen zwei Modelle nicht den gleichen Namen haben, da dies zu Namenskoflikten bei der Codegenerierung führt.
`ExactlyOneHardwareInfo`	The project '<name>' does not contain any Hardwaredescription file (.hardwareinfo).	In jedem Projekt muss genau ein Hardwaredescription-Modell enthalten sein.
`ExactlyOneHardwareInfo`	The project '<name>' contains more than one Hardwaredescription file (.hardwareinfo): <list of models>.	"
`IntegrateBaseChanges`	Changes detected in the Hardwaredescription or Datastructures used by '<modelname>'. Please open the file, review these changes and save it again.	Ein Dataflow-Modell ist nicht mehr synchron zu den Definitionen der anderen Modelle. Dies kann zwar automatisch korrigiert werden, aber der Anwender muss dies manuell überprüfen *(siehe Abschnitt 6.2.1)*.
`DatastructuresUniqueness`	More than one Datastructure Model defines a <elementtype> named '<name>': <list of models>.	Elemente können in mehreren Datastructure-Modellen innerhalb des Projektes definiert werden, allerdings müssen sie über alle Modelle hinweg einen eindeutigen Namen haben.

`DatastructuresUniqueness`	More than one Datastructure Model defines an Element with the RecordId <id>: <list of models>.	...sowie eindeutige Record-Ids.
`DataFlowEventCount`	# Models are subscribing the Event '<name>' but only # are allowed: <list of models>.	Bei einigen Events ist die Anzahl der Abonnenten begrenzt.

C.5. Firmware-Parser

Der Firmware-Parser *(siehe Kapitel 6.4)* überprüft die Funktionen, ihre Signaturen und die Annotationen nach dem Parsen, um sicherzustellen, dass aus diesen Daten ein konsistentes Hardwaredescription-Modell erzeugt werden kann. Für diese Prüfungen wurde ein Framework mit Validierungsregeln entwickelt. Die einzelnen Regeln sind thematisch zusammengefasst und als eine Klasse, die von `SemanticCheckBase` erbt, umgesetzt. Die folgende Tabelle enthält diese Regeln, mit den von ihnen erzeugten Fehlermeldungen samt Beschreibung.

Regel	**Fehlermeldung**	**Beschreibung**
001a	The function <function>() is missing one of the following mandatory annotations: @getVariable, @setVariable, @input, @inputInit, @output, @outputInit, @event, @eventCallback.	Wenn eine Funktion eine *Flow*-Annotation trägt, dann muss auch genau eine der angegebenen Annotationen enthalten sein.
001b	At function <function>() the @<annotation> annotation is missing a name.	...und einen Namen angegeben haben.
002a	The element name '<name>' is used for more than one of input, output, variable and event. Please use unique names: <functions>.	Der Name aller Elemente muss eindeutig sein und darf nicht mit mehreren verschiedenen Annotationen verwendet werden.
002b	For event '<name>' a function with @event annotation is missing.	Für Events müssen immer genau zwei Funktionen existieren: eine mit `@event`- und eine mit `@eventCallback`-Annotation.
002c	For event '<name>' a function with @eventCallback annotation is missing.	"
002d	For event '<name>' more than one function uses the @event annotation: <functions>.	Die Event-, Variablen- und Init-Annotation dürfen mit einem Namen nur bei einer Funktion verwendet werden.

002e	For event '<name>' more than one function uses the @eventCallback annotation: <functions>.	"
002f	More than one @getVariable function is defined for '<name>': <functions>.	"
002g	More than one @setVariable function is defined for '<name>': <functions>.	"
002h	For the input port '<name>' more than one function uses the @inputInit annotation: <functions>.	"
002j	For the output port '<name>' more than one function uses the @outputInit annotation: <functions>.	"
002i	For the input port '<name>' at least one function with @input annotation is missing: <functions>.	Zu jeder `@inputInit` / `@outputInit`-Funktion muss auch eine `@input` / `@output`-Funktion existieren.
002k	For the output port '<name>' at least one function with @output annotation is missing: <functions>.	"
002l	The type of the parameter in both functions for '<name>' must be the same: <functions>.	Die Getter und Setter für Variablen müssen vom gleichen Typ sein.
003	The file annotation @firmwarename or @drivername is missing.	Auf Dateiebene muss genau eine der beiden Annotationen `@firmwarename` und `@drivername` vorhanden sein.
004	Do not use both @firmwarename and @drivername in one file.	"
005	The annotation @field should not be used with the @<annotation> function <name>().	`@field` darf nicht zusammen mit `@event`, `@eventCallback`, `@getVariable`, `@setVariable`, `@inputInit` und `@outputInit` auf Funktionsebene verwendet werden.
006	@multiplicity is only valid for an @event function, not for <name>().	`@multiplicity` darf nur bei `@event`-Funktionen verwendet werden.

008	No additional annotation for @<annotation> function <name>() necessary!	`@getVariable`- und `@setVariable`-Funktionen benötigen keine weiteren Annotationen.
020a	The signature of @getVariable function <name>() is wrong. It should be: bool <name>(T*) for some type T.	Für Funktionen zum Lesen und Schreiben von Variablen ist die Signatur vorgegeben.
020b	The signature of @setVariable function <name>() is wrong. It should be: bool <name>(T) for some type T.	”
021	The last parameter of the @event function <name>() must be a function pointer 'void*' without any further annotations.	Der letzte Parameter einer `@event`-Funktion muss ein Funktionspointer *(`void*`)* sein, um einen Callback registrieren zu können.
022a	The type of parameter '<name>' in function <name>() must be 'bool'.	Alle Funktionsparameter mit der Annotation `@enabled` oder `@fieldEnabled` müssen vom Typ `bool` sein.
022b	The type of parameter '<name>' in function <name>() must not be a pointer.	Stellt die Korrektheit weiterer Signaturen sicher.
022c	The type of parameter '<name>' in function <name>() must be a pointer.	”
023	The function <name>() must have exactly one parameter.	”
050a	The parameter <name> of function <name>() must not have any annotations.	Stellt sicher, dass alle benötigten, aber keine verbotenen Annotationen an den Parametern von Funktionen vorhanden sind.
050b	The parameter <name> of function <name>() is missing one of the following mandatory annotations: <annotation>.	”
051	The parameter <name> of <name>() must not have a @default annotation.	Nur Funktionsparameter mit `@parameter`-Annotation dürfen eine `@default`-Annotation haben.

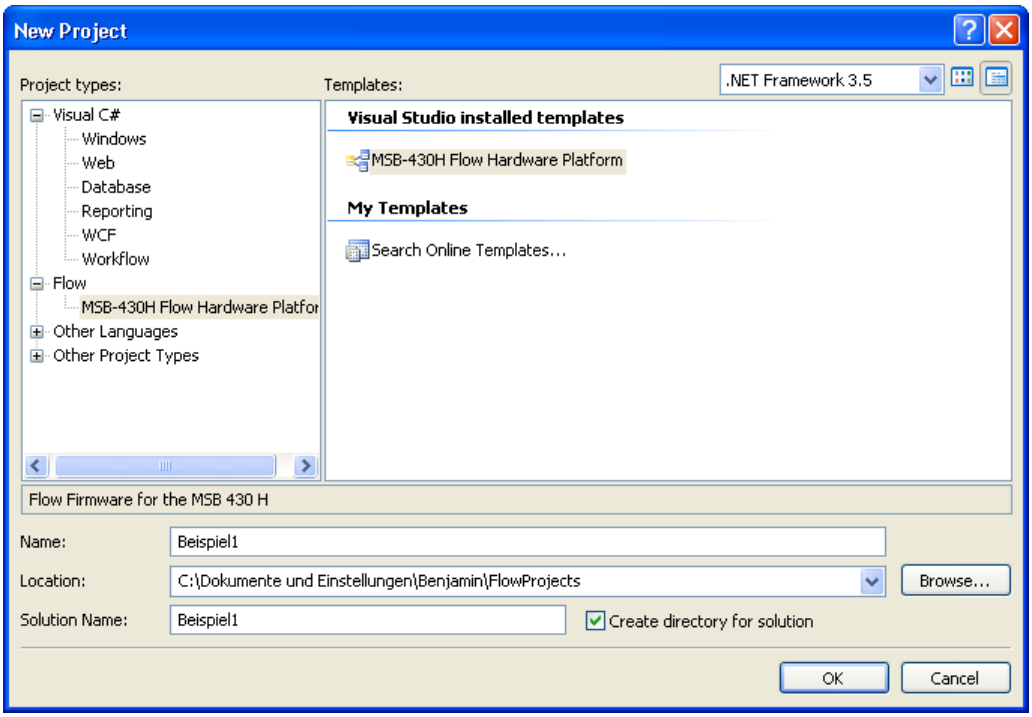
D. Anwenderhandbuch

Im Folgenden soll beschrieben werden, wie *Flow* innerhalb von Visual Studio eingesetzt wird, um Sensorknotenanwendungen zu modellieren. Primär soll auf die Bedienung und Elemente der Benutzeroberfläche eingegangen werden und nicht auf die Modellierung in den verschiedenen Sprachen, da dies bereits im Kapitel 5 beschrieben ist.

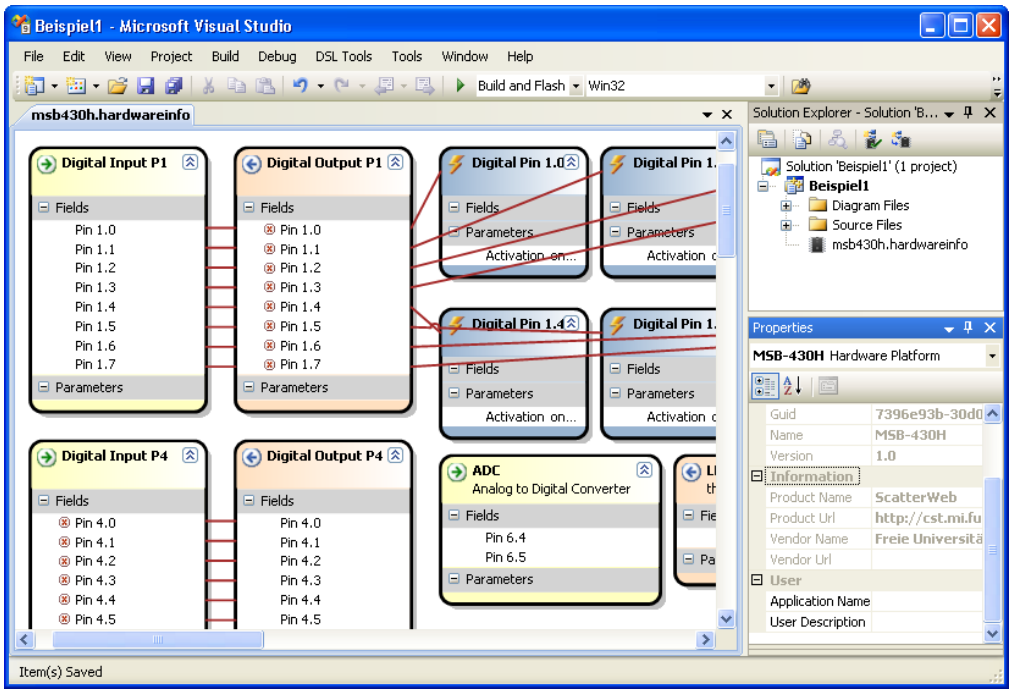
Um die Beispiele der nächsten Seiten ausführen zu können, wird ein lauffähiges System zur Sensorknotenentwicklung mit Visual Studio 2008 und *Flow* benötigt. Des Weiteren sollten *(für die Funkkommunikation)* mehrere Sensorknoten vom Typ MSB-430H und das Erweiterungsboard MSB-430S zur Verfügung stehen.

Beispielanwendung I

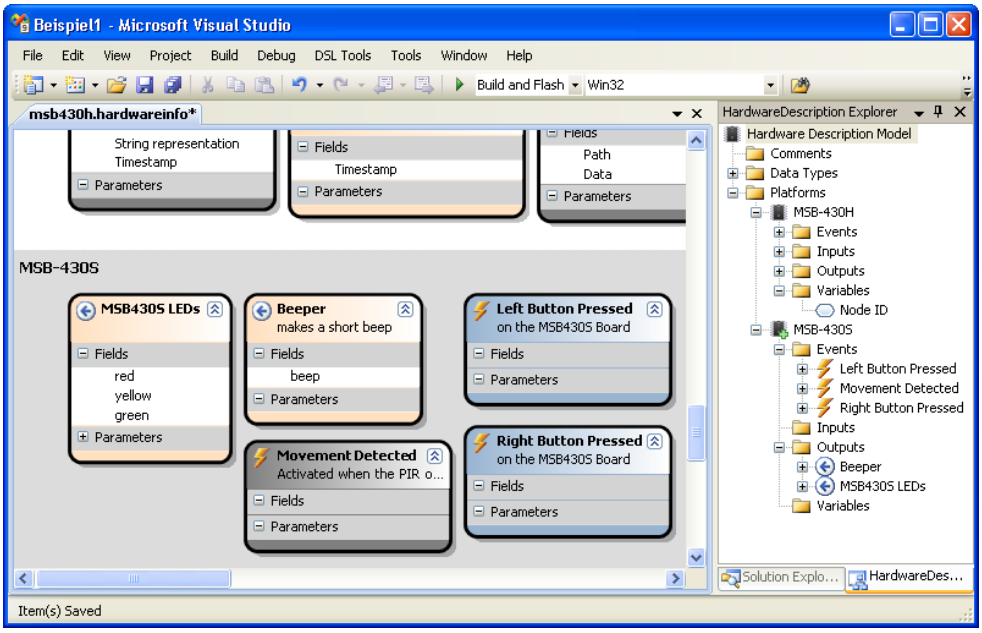
Nachdem die MSB-430H-Plattform auf dem Entwicklungsrechner installiert wurde, steht in Visual Studio ein neuer Projekttyp bereit, der über den New Project Dialog erzeugt werden kann. Für das erste Beispielprogramm wird ein solches Projekt mit dem Namen „Beispiel1“ angelegt.



Das neue Projekt enthält bereits die Hardwaredescription der Sensorknotenhardware *(vollständig in Abbildung 7.1 auf Seite 85 dargestellt)*.



Da für diese Beispielanwendung die Erweiterung MSB-430S verwendet werden soll, muss der entsprechende Treiber geladen werden. Dazu bietet dieses Modell im Kontextmenü[1] den Befehl „Import Driver...“ an. Nachdem der Befehl angeklickt wurde, muss die Datei `MSB-430S Driver.flowdrv` ausgewählt werden. Das Modell wird dadurch um die Elemente des Treibers erweitert. Da dabei auch Kanten angelegt werden, die das Modell unübersichtlich machen können, empfiehlt es sich, ebenfalls über das Kontextmenü mit dem Befehl „Show Connectors“, die Kanten auszublenden.

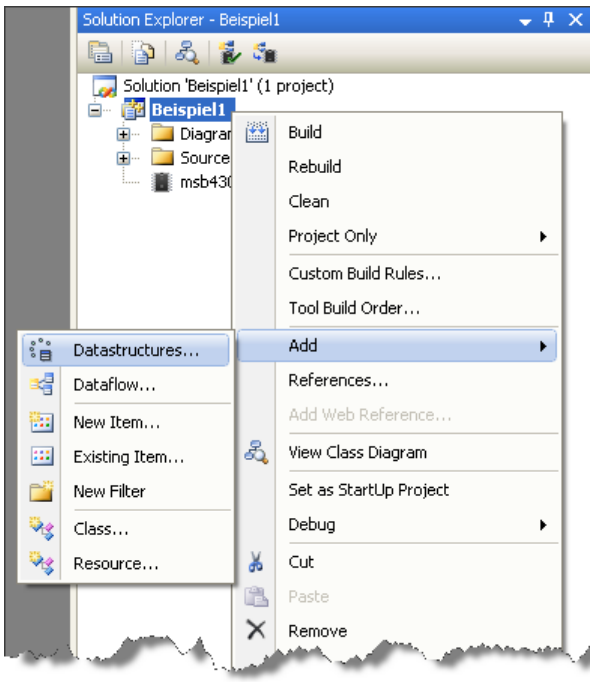


Alle Elemente der Hardwaredescription sind auch im „Hardwaredescription Explorer“ auf der rechten Seite zu sehen, der ebenfalls über das Kontextmenü des Modells geöffnet werden kann.

Die Sensorknotenanwendung soll beim Drücken eines Tasters die grüne LED umschalten. Wird der andere Taster gedrückt und die grüne LED leuchtet, soll der Beeper kurz aktiviert werden. Um zu signalisieren, dass der Sensorknoten aktiv ist, soll währenddessen eine weitere LED ständig blinken. Da für diese Anforderungen viele Elemente der Hardwaredescription nicht benötigt werden, werden sie bereits in diesem Modell über das Visual Studio Properties Window deaktiviert *(enabled = false)*. Dies sind:

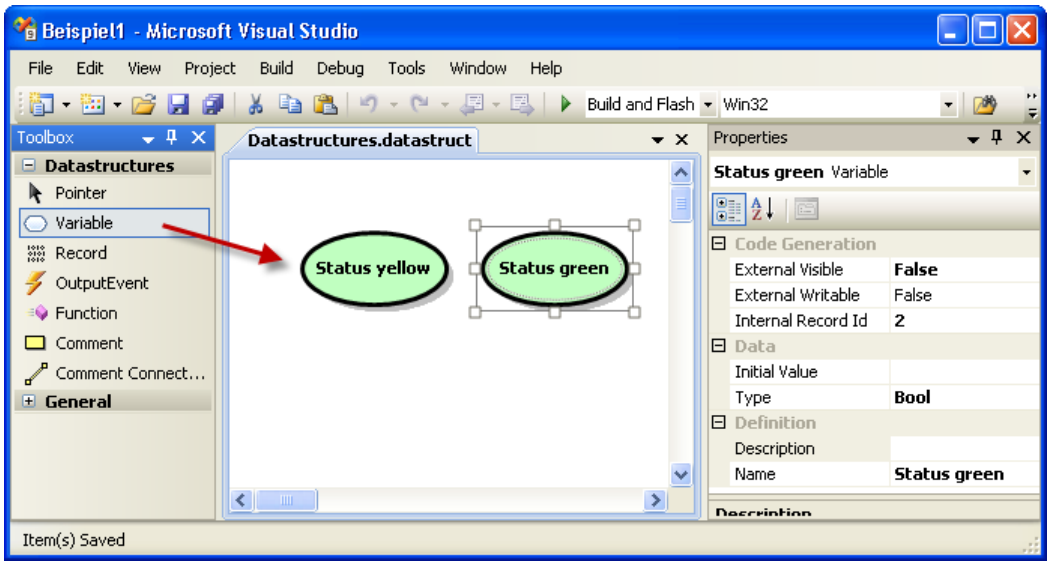
- alle digitalen I/O-Ports sowie die Events
- der analoge ADC-Port
- die Elemente der seriellen Schnittstelle und die Real Time Clock
- die „Write File“-Output-Ports
- der Bewegungsmelder des MSB-430S

Da für die beiden LEDs im weiteren Verlauf der aktuelle Zustand benötigt wird, muss dieser in Variablen gespeichert werden. Dazu wird im Projekt ein neues Datastructure-Modell erstellt.

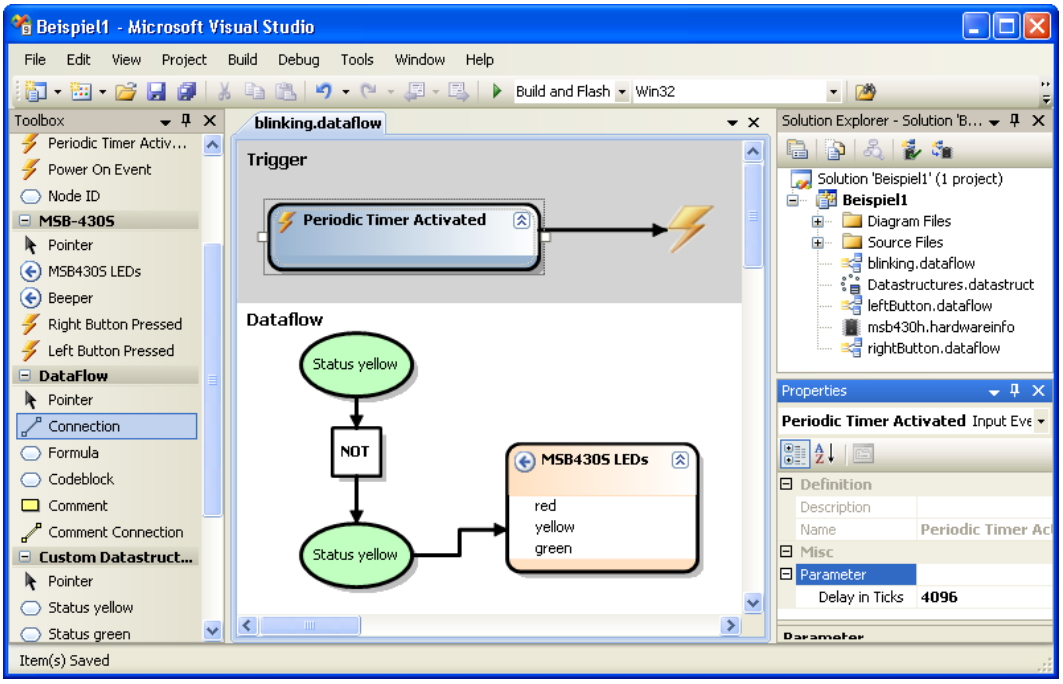


[1] Das Kontextmenü kann durch Rechtsklick auf der Zeichenfläche *(dem Hintergrund)* des Modells aufgerufen werden.

Diesem Modell können aus der Toolbox auf der linken Seite Elemente per drag-and-drop hinzugefügt werden. Für die Beispielanwendung werden zwei Variablen benötigt. Mittels der Eigenschaften im Properties Window können diese Variablen konfiguriert werden. Ihnen wird der angezeigte Name gegeben und der Datentyp „Bool“ ausgewählt.



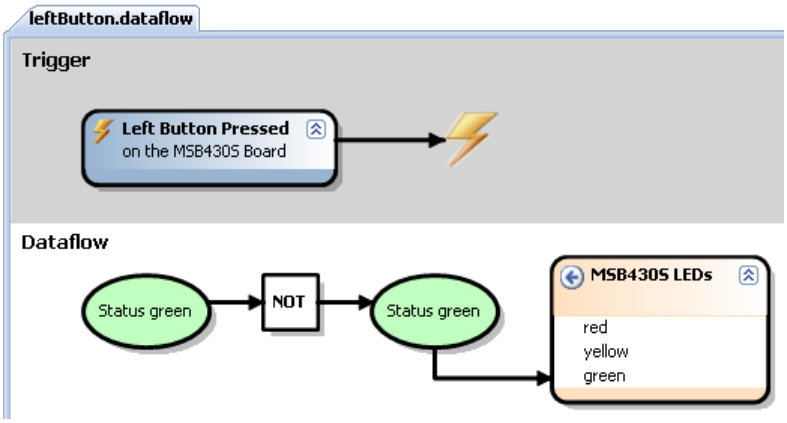
Nachdem nun die Hardware und die Datastructures modelliert und konfiguriert sind, kann begonnen werden, das Verhalten der Anwendung zu definieren. Dazu wird *(so wie bei den Datastructures)* ein neues Dataflow-Modell mit dem Namen „blinking“ angelegt. Dieser erste Dataflow soll dafür sorgen, dass die gelbe LED blinkt. Dazu werden die Elemente aus der Toolbox wie folgt auf dem Modell positioniert und mittels des „Connection“-Tools verbunden:



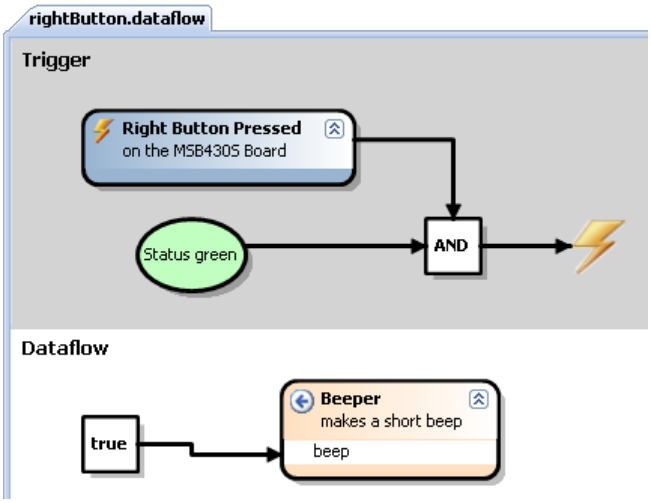
Das „Periodic Timer Activated“-Event im Trigger-Bereich gibt immer, wenn es ausgelöst wird, einen „Impuls“ an den Master-Trigger, so dass der Dataflow ausgeführt wird. Das Event besitzt eine Eigenschaft „Delay in Ticks“, die über das Properties Window auf 4096, was einer halben Sekunde entspricht, eingestellt wird.

Im Dataflow-Bereich wird zuerst die Variable „Status yellow“ gelesen, negiert und dann wieder geschrieben. Zum Schluss wird der Wert der Variable über die gelbe LED des „MSB430S LEDs“-Output-Ports angezeigt, wodurch sie in einem Rhythmus von 0,5 Sekunden blinkt.

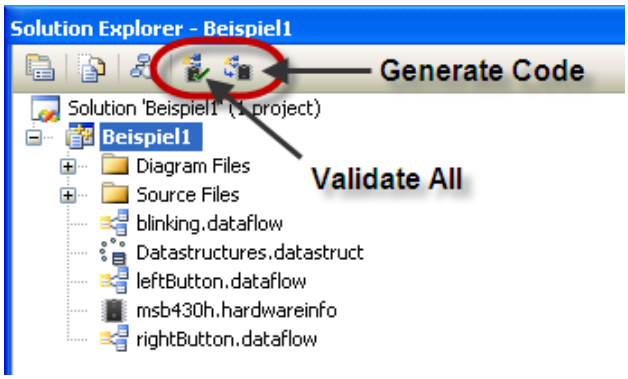
Der zweite Dataflow mit dem Namen „leftButton“ funktioniert analog, wird allerdings von einem anderen Event ausgelöst.



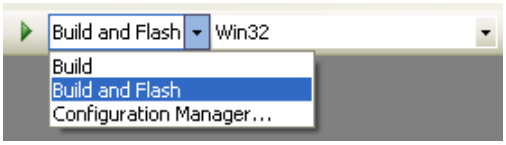
Der dritte und letzte Dataflow „rightButton“ reagiert nun auf den anderen Taster, aber führt den Dataflow nur dann aus, wenn auch die Variable „Status green“ gesetzt ist. Der Dataflow-Bereich selbst ist trivial, da hier lediglich der Beeper aktiviert werden muss.



Nachdem die Anwendung vollständig modelliert ist, kann zunächst geprüft werden, ob die Modelle korrekt gebildet sind und sich fehlerfreier Code generieren lässt. Dies kann durch den linken Button[2] im Solution Explorer durchgeführt werden. Wenn Fehler gefunden werden, werden diese in der Error List von Visual Studio angezeigt. Ist dahingegen die Validierung fehlerfrei verlaufen, so kann mit dem rechten Button der Quellcode für die Sensorknoten generiert werden[3].



Zu guter Letzt kann der Code kompiliert und auf die Sensorknoten geflasht werden. Dazu stehen zwei Konfigurationen bereit, die entweder nur den Code kompilieren *(„Build“)* oder ihn im Anschluss über ein USB-JTAG-Interface auf den Sensorknoten aufbringen *(„Build and Flash“)*. Diese Konfigurationen können über die Symbolleiste von Visual Studio ausgewählt werden. Während des Kompilierens[4] wird die Ausgabe des Compilers im Visual Studio Output Window angezeigt. Sollten Fehler auftreten, so sind diese im Output Window und der Error List zu sehen.

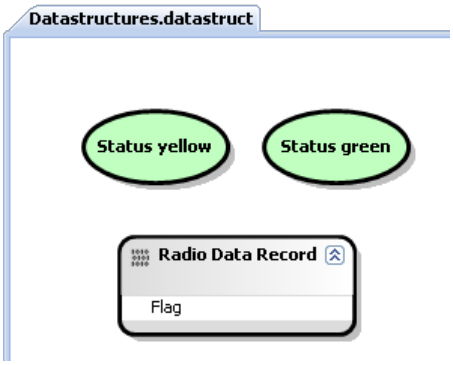


[2] Die *Flow*-Buttons im Solution Explorer sind nur sichtbar, wenn ein *Flow*-Projekt ausgewählt ist.
[3] Jedesmal, wenn die Modelle verändert wurden, muss der Quellcode erneut generiert werden, da ansonsten veralteter Code auf die Sensorknoten programmiert würde.
[4] Das Kompilieren kann entweder über das Menü oder über die Tastenkombination `Shift+Strg+B` gestartet werden.

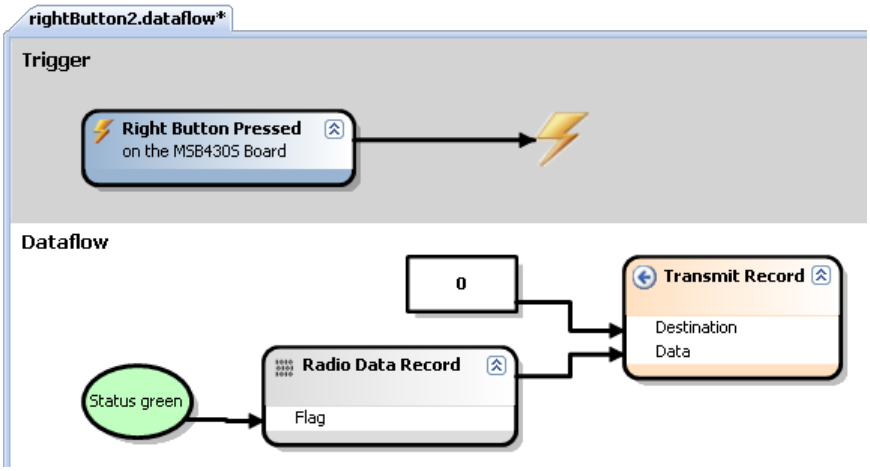
Beispielanwendung II

Das zweite Beispiel erweitert die Anwendung von oben um Funkkommunikation. Immer wenn der rechte Taster gedrückt wird, soll per Broadcast eine Nachricht verschickt werden. Diese Nachricht enthält als Daten den Zustand des grünen LEDs des Senders. Wenn ein anderer Sensorknoten eine solche Nachricht empfängt, prüft dieser die übertragenen Daten und falls das Flag gesetzt ist, gibt auch er einen Pfeifton aus.

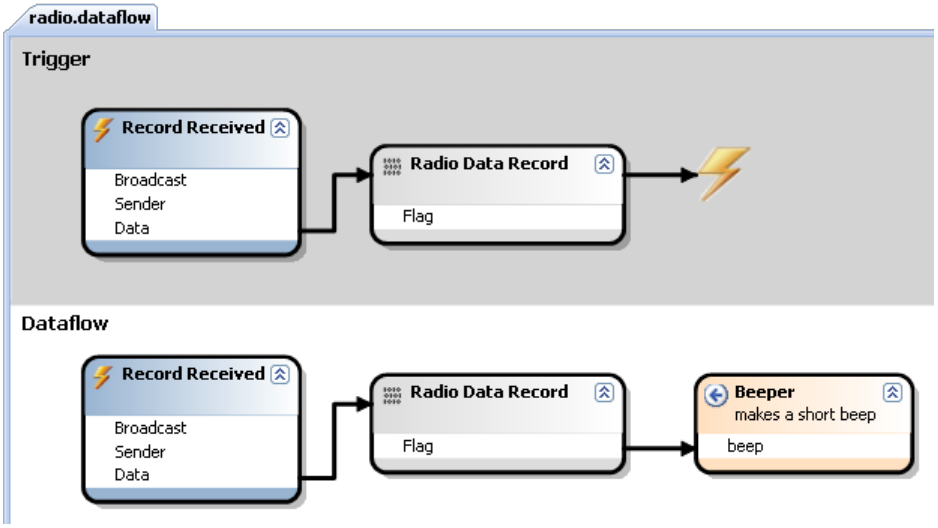
Für die Funkübertragung wird ein Record mit nur einem Feld benötigt. Dies kann im bereits vorhandenen Datastructure-Modell durch drag-and-drop aus der Toolbar hinzugefügt werden. Anschließend lässt sich dem Record über sein Kontextmenü ein Feld hinzufügen. Im Properties Window kann der Name und der Datentyp *(„Bool“)* des Feldes eingestellt werden.



Ein neuer Dataflow „rightButton2“ soll, wenn der Taster betätigt wird, ein Record erstellen und versenden. Ein zweiter Dataflow ist notwendig, da der Dataflow „right-Button“ aus dem Beispiel oben nur dann ausgeführt wird, wenn die Variable „Status green“ gesetzt ist, aber nun bei jedem Tastendruck ein Paket verschickt werden soll. Im Dataflow wird die Variable „Status green“ in ein Record von Typ „Radio Data Record“ verpackt und mittels des „Transmit Record“-Output-Ports an die Broadcastadresse 0 verschickt. Um die Adresse angeben zu können wird ein Formel-Shape verwendet, in dem die Zahl eingetragen ist.



Auf diese Funkpakete soll nun in einem weiteren Dataflow reagiert werden. Wenn ein Funkpaket empfangen wird prüft der Tigger-Bereich dieses Modells, ob das Event ein Record vom Typ „Radio Data Record“ geliefert hat. Wenn dem so ist, wird ein „Impuls“ zum Master-Trigger gegeben und der Dataflow ausgeführt. Im Dataflow wird ein Record mit den Daten aus dem Event gefüllt und das Flag an den Beeper weitergegeben. Der Beeper wird nur dann einen Pfeifton ausgeben, wenn das Flag im Record `TRUE` ist.



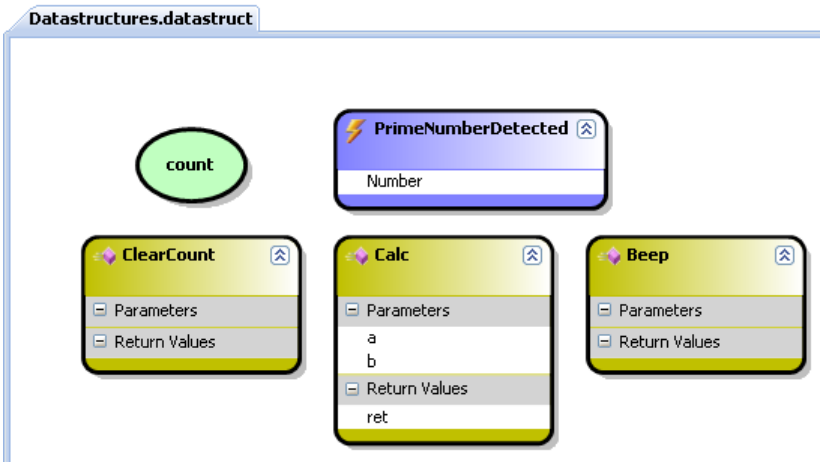
Wie oben beschrieben, kann nun auch aus dieser Anwendung Code generiert und auf mehrere Sensorknoten verteilt werden. Da in diesem Beispiel alle Knoten gleichberechtigt sind und die Kommunikation mittels Broadcasts durchgeführt wird, sollten die Knoten problemlos zusammenarbeiten.

Beispielanwendung III

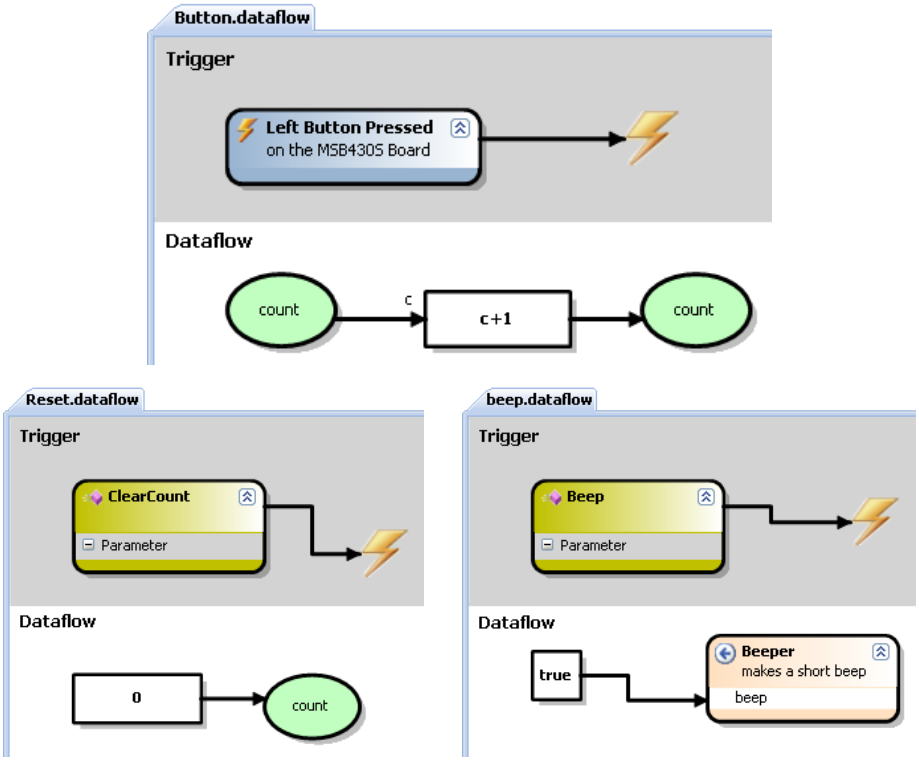
Das folgende Beispiel zeigt, wie *Flow*-Sensorknoten unter Verwendung des ScatterWeb .NET SDKs mit einem PC zusammenarbeiten können. Dazu wird eine Sensorknotenanwendung mit *Flow* modelliert, aber auch der Code der PC-Anwendung vorgestellt.

Die Sensorknoten sollen einen Zustand in Form einer numerischen Variable besitzen, der sich durch Ereignisse *(Tastendrücke)* verändert. Der PC soll auf diese Variable lesend zugreifen und sie mittels eines Funktionsaufrufs auf Null setzen können. Um weitere Möglichkeiten der remote Funktionsaufrufe zu demonstrieren, soll eine Berechnung mit zwei Argumenten auf den Sensorknoten durchgeführt werden, so dass der PC die Berechnung anstoßen kann und das Ergebnis geliefert bekommt. Außerdem soll der PC den Beeper des Sensorknotens aktivieren können. Normalerweise werden Sensornetzwerke eingesetzt, weil die Sensorknoten eine gewisse Intelligenz besitzen und so nicht permanent von einer Basisstation ferngesteuert werden müssen. Um auch dieses Einsatzgebiet zu demonstrieren, soll der Sensorknoten immer dann ein Output Event an den PC auslösen, wenn die numerische Variable ihren Wert ändert und eine Primzahl erreicht.

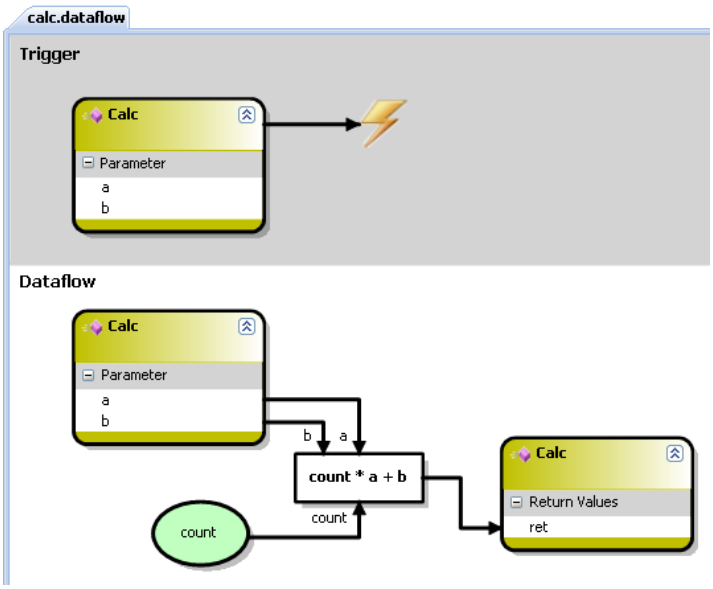
Wie bei den vorhergehenden Beispielanwendungen wird ein neues Sensorknotenprojekt angelegt. Auch bei diesem Beispiel wird der Treiber für die Erweiterung MSB-430S in die Hardwaredescription geladen und ein Datastructure-Modell angelegt. In diesem Modell ist die numerische Variable *count*, die drei beschriebenen Funktionen, die von einem PC aufgerufen werden können, sowie das Output Event zu sehen.



Die ersten drei Dataflow-Modelle sind den der anderen Beispiele sehr ähnlich. Wird der Taster betätigt, so wird die Variable um eins erhöht und beim Funktionsaufruf `ClearCount()` vom PC wird sie auf Null gesetzt. Der Funktionsaufruf `Beep()` aktiviert den Beeper.

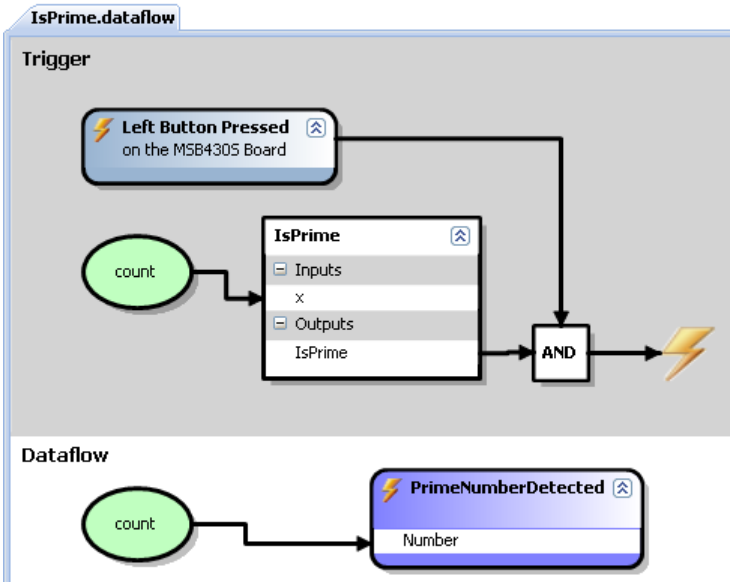


Der Funktionsaufruf `Calc()` vom PC verwendet die mit dem Funktionsaufruf gesendeten Argumente *a* und *b* sowie den Wert der Variable *count*, um mittels eines Formel-Shapes das Ergebnis zu berechnen. Dieses wird als Rückgabewert der Funktion an den PC geschickt.



Der letzte Dataflow wird immer dann ausgeführt, wenn der Taster gedrückt wird und die Variable *count* durch die Veränderung im oben beschriebenen Dataflow nun eine Primzahl darstellt. Da diese Prüfung nicht ohne weiteres in einem Dataflow darzustellen ist, wird hier ein Codeblock im Trigger-Bereich verwendet. Ein Codeblock kann aus der Toolbox dem Diagramm hinzugefügt und dort mit Ein- und Ausgängen bestückt werden. Durch Doppelklick auf den Codeblock wird ein C-Code-Editor geöffnet, in dem der Code aus Abbildung D.1 eingetragen werden muss. Dieser Code prüft, ob der Parameter `x` eine Primzahl ist und setzt den Parameter `IsPrime` als Rückgabe entsprechend.

Wenn der Codeblock den Dataflow mittels des Master-Triggers aktiviert, wird das Output Event `PrimeNumberDetected` ausgelöst, auf das ein PC reagieren kann.



Diese wenigen Dataflows implementieren das öffentliche Interface des Sensorknotens, der so auf Anfragen eines PCs reagieren bzw. Events an den PC schicken kann. Die Anwendung kann, wie bereits beschrieben, kompiliert und auf Sensorknoten programmiert werden.

```
/* This file contains user code for the
   Codeblock 'IsPrime'
   from the Model 'IsPrime' */

// the following line must match exactly the definition in the model
// please do not change this line:
void IsPrime_IsPrime(uint32_t x, bool* IsPrime)
{
    if (x < 2) {
        *IsPrime = false;
        return;
    }
    else if (x == 2) {
        *IsPrime = true;
        return;
    }
    else if (x % 2 == 0) {
        *IsPrime = false;
        return;
    }
    else {
        int i=3;
        for (; i*i <= x; i+=2) {
            if (x%i == 0) {
                *IsPrime = false;
                return;
            }
        }
        *IsPrime = true;
        return;
    }
}
```

Abbildung D.1: Quellcode des Codeblocks „IsPrime“

PC-Anwendung

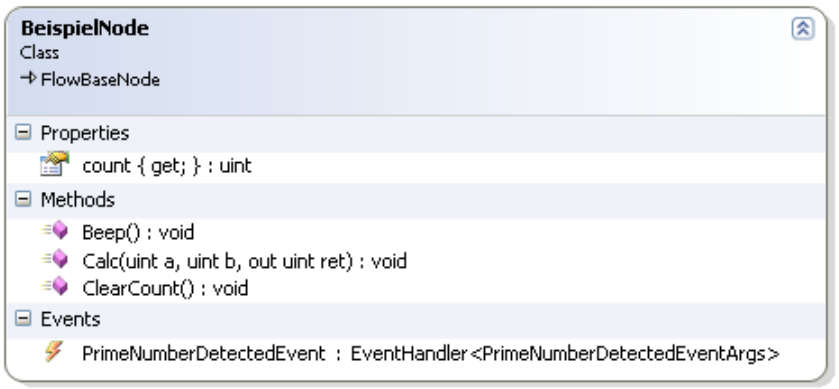
Im nächten Schritt soll eine Windows-Anwendung entstehen, die mit diesen Sensorknoten kommuniziert. Zunächst wird in der Solution ein neues Projekt *(C# Windows Forms Application)* angelegt. Um in diesem Projekt das ScatterWeb .NET SDK zu verwenden, müssen dem Projekt drei Referenzen auf die folgenden Assemblies hinzugefügt werden. Diese Assemblies befinden sich in einem Unterverzeichnis der *Flow*-Installation:

- `ScatterWeb.API.dll`
- `ScatterWeb.SDK.dll`
- `ScatterWeb.SDK.Nodes.Flow.FlowBaseNode.dll`

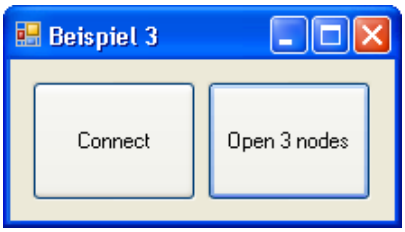
Für den oben beschriebenen Sensorknoten soll ein Proxy generiert werden. Dieser Proxy wird ebenfalls dem Projekt hinzugefügt indem eine Textdatei mit der Endung `.tt` angelegt und mit dem folgenden Code gefüllt wird:

```
<#@ ProjectFile processor="VsProjectFileDirectiveProcessor"
        FileName="..\Beispiel3\Beispiel3.vcproj" #>
<#@ include file="proxy.tt" #>
```

Unter Umständen muss der relative Pfad zum Sensorknotenprojekt angepasst werden. Es ist durchaus möglich, einer Anwendung auf diese Weise mehrere Proxies hinzuzufügen, die den Code für verschiedene Sensorknotenprojekte erzeugen. Durch die Dateiendung `.tt` wird beim Speichern der Datei automatisch der Proxycode erzeugt, der im Solution Explorer unterhalb dieser Datei als C#-Code gefunden werden kann. Die folgende Abbildung zeigt das öffentliche Interface des generierten Proxies, in dem die oben modellierten Elemente wiedergefunden werden können.



Bevor nun mit dieser Klasse gearbeitet werden kann, ist ein wenig Infrastrukturcode notwendig, der mit einem lokal am PC angeschlossenen Sensorknoten, der ein Gateway zum gesamten Sensornetzwerk darstellt, kommunizieren kann. Beim Starten der Anwendung wird das `MainForm` geöffnet, das lediglich zwei Buttons enthält.

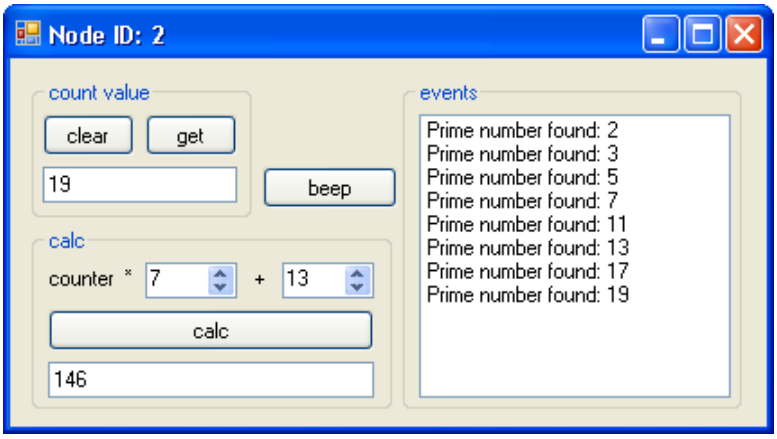


Wenn der Button „Connect“ betätigt wird *(siehe Abbildung D.2, Zeile 17)*, wird der Port zur seriellen Kommunikation ermittelt, ein `NetworkManager` instanziert und nach Sensorknoten gesucht. Wenn Knoten gefunden wurden, wird der zweite Button „Open x nodes“ freigegeben. Mit diesem Button *(Zeile 41)* wird für jeden Knoten im Netzwerk ein `NodeForm` erzeugt und angezeigt. Der in Abbildung D.2 gezeigte Code kann in ähnlicher Art und Weise auch in anderen Anwendungen verwendet werden, auch wenn hier auf Fehlerbehandlung verzichtet wurde, um die relevanten Stellen hervorzuheben.

```
using System;
using System.Linq;
using System.Windows.Forms;
using ScatterWeb.SDK;
using ScatterWeb.SDK.Nodes;
using ScatterWeb.API.Messaging.Channels;
using Flow.SensorNodes;

namespace WindowsFormsApplication1 {
 public partial class MainForm : Form {
   NetworkManager netman;

   public MainForm() {
     InitializeComponent();
   }

   private void buttonConnect_Click(object sender, EventArgs e) {
     // Use the HardwareFinder to get the COM-Port for communication
     ScatterWeb.API.Platform.HardwareFinder hwf = new HardwareFinder();
     string port = hwf.FindConntectedHardware(System.IO.Ports.SerialPort
         .GetPortNames());
     if (String.IsNullOrEmpty(port)) return;

     // create a Channel and a NetworlManager
     Channel channel = new SerialPortChannel(sport);
     netman = new ScatterWeb.SDK.NetworkManager(channel);

     // announce the Nodetype
     netman.RegisterNodeType<BeispielNode>();

     // Open the network
     netman.Open();

     // enable the second button if any nodes available
     int c = netman.Network.OfType<BeispielNode>().Count();
     if (c > 0) {
       this.buttonOpen.Text = string.Format("Open {0} nodes", c);
       this.buttonOpen.Visible = true;
     }
   }

   private void buttonOpen_Click(object sender, EventArgs e) {
     // get all BeispielNode nodes form the NetworkManager
     foreach (BeispielNode n in netman.Network.OfType<BeispielNode>()) {
       // and open a NodeForm for every node
       NodeForm f = new NodeForm(n);
       f.Show();
     }
   }

   private void MainForm_FormClosing(object s, FormClosingEventArgs e) {
     // close the NetworkManager when the application is closing
     if ( netman != null && netman.IsOpen )
       netman.Close();
   }
 }
}
```

Abbildung D.2: Quellcode von `MainForm`

Der `NetworkManager` erzeugt für jeden Sensorknoten abhängig von der installierten Anwendung eine spezialisierte Klasse. Für alle Sensorknoten der oben modellierten Sensorknotenanwendung wird eine Instanz der von Proxy generierten `BeispielNode`-Klasse erzeugt. Diese wird in der Schleife des zweiten Buttons an den Konstruktor des `NodeForms` übergeben *(Zeile 45)*, das in der folgenden Abbildung dargestellt ist:



Auf die Elemente des Forms soll nicht detailliert eingegangen werden, da sie lediglich Funktionen des Sensorknotens wiederspiegeln. Der relevante Code dieses Forms ist in Abbildung D.3 gezeigt und besteht meist nur aus wenigen Zeilen, um auf die Benutzer- und Sensorknotenereignisse zu reagieren.

Die Buttons rufen entweder Funktionen in der Sensorknoten-Klasse auf *(`Beep()` in Zeile 33, `CleartCount()` in Zeile 38 und `Calc()` in Zeile 52)* oder fragen die Property *count* ab *(Zeile 43)*. Alle Aufrufe bewirken, dass der entsprechende Dataflow des Sensorknotens ausgeführt wird. Wenn für eine Funktion Rückgabewerte definiert sind blockiert sie solange, bis der Knoten eine Antwort liefert oder ein Timeout auftritt. Bei den Funktionen ohne Rückgabewerte *(`Beep()` und `CleartCount()`)* wird die Ausführung fortgesetzt, sobald das Gateway das Record entgegengenommen hat.

Für das Output Event wird eine EventHandler-Methode registriert, was im Beispiel im Konstruktor des Forms in Zeile 19 geschieht. Innerhalb dieser Methode muss lediglich beachtet werden, dass das Form nicht direkt manipuliert werden darf, da dieser Aufruf aus einem anderen Thread heraus geschieht. Daher ist eine Konstruktion mittels `this.Invoke()` wie in Zeile 25 gezeigt notwendig.

Über diese Anwendung kann nun transparent auf das Sensornetzwerk zugegriffen werden. Es ist irrelevant, ob der Knoten mit dem man kommuniziert direkt an der Schnittstelle des PCs angeschlossen oder über Funk erreichbar ist. Lediglich die Reaktionszeit variiert deutlich.

```
using System;
using System.Windows.Forms;
using Flow.SensorNodes;

namespace WindowsFormsApplication1 {
 public partial class NodeForm : Form {
   BeispielNode node;

   public NodeForm() {
     InitializeComponent();
   }

   public NodeForm(BeispielNode node) : this() {
     // store the node in a global variable
     this.node = node;
     // display the NodeId in the title of the form
     this.Text = "Node ID: " + node.NodeID;
     // and register for the event
     this.node.PrimeNumberDetectedEvent += new EventHandler<BeispielNode
         .PrimeNumberDetectedEventArgs>(node_PrimeNumberDetectedEvent);
   }

   // the Event from the Sensornode
   void node_PrimeNumberDetectedEvent(object sender, BeispielNode.
       PrimeNumberDetectedEventArgs e) {
       // Invoke is required for the event
       this.Invoke((Action)delegate() {
       // show the data of the event in the listbox
       this.listBox1.Items.Add("Prime number found: " + e.Number);
     });
   }

   private void buttonBeep_Click(object sender, EventArgs e) {
     // let the node beep when the beep button is pressed
     node.Beep();
   }

   private void buttonClear_Click(object sender, EventArgs e) {
     // call the ClearCount function on the node
     node.ClearCount();
   }

   private void buttonGet_Click(object sender, EventArgs e) {
     // query the variable from the node
     uint c = node.count;
     this.textBoxCount.Text = c.ToString();
   }

   private void buttonCalc_Click(object sender, EventArgs e) {
     // call the Calc function of the sensor node with two parameters
     uint x;
     node.Calc((uint)this.numA.Value, (uint)this.numB.Value, out x);
     // and show the return value in the textbox
     this.textBoxCalcResult.Text = x.ToString();
   }
 }
}
```

Abbildung D.3: Quellcode von `NodeForm`

E. Inhalt der CD

Dieser Diplomarbeit ist eine CD beigefügt. Auf der CD ist diese Arbeit als `pdf`-Datei aber auch *Flow* im Quellcode und als ausführbare Erweiterung für Visual Studio enthalten. Auch die *Flow*-Hardwareplattformen und die *Flow*-Treiber, die als Referenzplattform bzw. Fallstudie implementiert wurden, sind auf der CD zu finden. Sollte die CD nicht zur Verfügung stehen, so kann der Inhalt auch unter `http://flow.irgendwie.net` heruntergeladen werden.

Die folgende Liste gibt die Struktur der CD wieder:

Datei bzw. Verzeichnis	**Beschreibung**
`Diplomarbeit Benjamin Schröter.pdf`	Dieses Dokument.
`Flow Anwenderhandbuch.pdf`	Anhang D dieser Arbeit als seperates Dokument.
`FlowSetup.exe`	Installationsprogramm um *Flow* zu installieren.
`MSB-430H FlowPlatformSetup.exe`	Installationsprogramm um die Hardwareplattform MSB-430H für *Flow* zu installieren.
`MSB-430H FlowPlatformSetup without network.exe`	ebenso, allerdings ohne den Netzwerk-Stack.
`MSB-430S Driver.flowdrv`	Treiber für das MSB-430S Erweiterungsboard als *Flow*-Treiberdatei.
`SkomerBoard Driver.flowdrv`	Treiber für die Hardwareerweiterung der Fallstudie aus Kapitel 7.3 als *Flow*-Treiberdatei.
`Flow`	Der gesamte Quellcode von *Flow*. Die Solution `Flow.sln` kann mit Visual Studio 2008 geöffnet werden und enthält alle Projekte die in Anhang B beschrieben sind. In Anhang B.5 sind weitere Details angegeben, um *Flow* weiterentwickeln zu können.

`FlowProjects`	In diesem Verzeichnis sind fünf *Flow*-Projekte enthalten:
`MSB-430H`	Hardwareplattform für den MSB-430H-Sensorknoten samt annotierter Firmware und Hardwaredescription-Modell. Dieses Projekt erzeugt die Datei `MSB-430H FlowPlatformSetup.exe`.
`MSB-430H without network`	ebenso, allerdings ohne den Netzwerk-Stack. Dieses Projekt erzeugt die Datei `MSB-430H FlowPlatformSetup without network.exe`.
`MSB-430S`	*Flow*-Treiberplattform für das MSB-430S Erweiterungsboard. Dieses Projekt erzeugt die Datei `MSB-430S Driver.flowdrv`.
`SkomerBoard`	*Flow*-Treiberplattform für die Hardwareerweiterung der Fallstudie. Dieses Projekt erzeugt die Datei `SkomerBoard Driver.flowdrv`.
`SkomerIslandApp`	Der Quellcode der Fallstudie aus Kapitel 7.3. In der Solution sind die beiden beschriebenen Sensorknotenprojekte sowie die PC-Anwendung enthalten.
`ScatterWeb .NET SDK 2.0`	Der Quellcode des ScatterWeb .NET SDK 2.0 *(Betaversion) (siehe Abschnitt 3.1.1)*.
`Setup`	Alle Dateien und Skripte um die Datei `FlowSetup.exe` zu erstellen.
`Flow.Integration`	Bibliotheken um in Visual Studio neue Projektvorlagen anlegen zu können. Dieser Code stammt aus dem ScatterWeb .NET SDK und wurde von Tomasz Naumowicz für die Verwendung in *Flow* angepasst.
`3rdParty`	Verschiedene Programme und Bibliotheken die von *Flow* zur Entwicklungs- oder Laufzeit verwendet werden.