Diplomarbeit

Informatik

Analyse nebenläufiger Programme unter intransitiven Sicherheitspolitiken

Alexander Reinhard

RWTH Aachen
Fachgruppe Informatik
Arbeitsgruppe Entwurf und Analyse sicherer Softwaresysteme
Erstprüfer: Prof. Dr.-Ing. Heiko Mantel
Zweitprüfer: Prof. Dr.-Ing. Felix Freiling

Abgabetermin: 01. Juni 2006

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 29. Mai 2006

(Alexander Reinhard)

Kurzfassung

Die programmiersprachenbasierte Sicherheit behandelt den Informationsfluss durch Computerprogramme. Vertraulichkeit bedeutet hier, dass der Informationsfluss zwischen den für ein Programm zugänglichen Informationsressourcen nach einer Flusspolitik begrenzt ist.

Deklassifikation ist eine kontrollierte Verletzung der normalen Flusspolitik. Diejenige Deklassifikation, die auf der *intransitive noninterference* basiert, nutzt dafür eine intransitive Flusspolitik.

Sicherheitstypsysteme sind Mechanismen um Informationsflussbedingungen an Programme zu überprüfen, ohne diese auszuführen. In dem Entwicklungsteil dieser Arbeit wird eine bestehende Anwendung erweitert, die mit Sicherheitstypsystemen Programme der *Multi-Threaded While Language* (MWL) analysiert. Diese Erweiterung umfasst neben Sprachelementen zur Synchronisation auch Deklassifikation basierend auf *intransitive noninterference*.

Intransitive noninterference beschränkt, „wo“ Deklassifikation auftreten darf. In dem theoretischen Teil dieser Arbeit werden für MWL Flussbedingungen definiert und untersucht, die beschränken „was“ deklassifiziert wird und „wer“ deklassifizieren darf. Diese Kombination und die Zusammenhänge der Flussbedingungen werden untersucht, um alle Aspekte der Deklassifikation abzudecken.

Abstract

Language-based security addresses information flow by computer programs. Here, confidentiality is the limitation of information flow between information resources accessible to a program. The flow is limited according to a flow policy.

Declassification is a controlled infringement of the normal flow policy. Declassification based on intransitive noninterference uses an intransitive flow policy.

Security type systems are mechanisms to check programs for compliance with information flow conditions without the need to execute them. In the development part of this thesis, we extend an existing application for analyzing programs of the Multi-Threaded While Language(MWL) with security type systems. This extension covers language elements for synchronization and declassification according to intransitive noninterference.

Intransitive noninterference limits ”where” declassification may occur. In the theoretical part of this thesis, we define and analyze flow conditions for MWL limiting ”what” is declassified and ”who” is allowed to declassify. We analyze the combination and the relationship of the flow conditions to cover all aspects of declassification.

Danksagungen

Ich danke sehr herzlich Herrn Prof. Dr.-Ing. Heiko Mantel für die Einführung in das Forschungsgebiet, für die Möglichkeit dieses interessante Thema zu bearbeiten, für die fachlichen und methodischen Ratschläge und auch sonst für die lehr- und hilfreiche Betreuung.

Weiter danke ich Herrn Prof. Dr.-Ing. Felix Freiling für die Motivation, mich mit theoretischer Informationssicherheit zu befassen und für seine Bereitschaft der Zweitprüfer dieser Arbeit zu sein.

Außerdem danke ich Frau Dipl. Inf. Christina Pöpper, die mir bereitwillig zu ihrer Diplomarbeit Fragen beantwortete und mir LaTeX-Makros zur Verfügung stellte, die ich insbesondere für den Satz von Typregeln dankbar verwendet habe.

Inhaltsverzeichnis

1. Einleitung **11**
1.1. Motivation . . . 11
1.2. Einordnung . . . 12
1.3. Struktur der Arbeit . . . 13
1.4. Konventionen . . . 14

2. Grundlagen **15**
2.1. Sicherheitspolitiken . . . 15
2.2. Die „Multi-Threaded While Language“ . . . 17
2.2.1. Syntax und allgemeine Semantik . . . 17
2.2.2. Deklarationen und Arrays . . . 20
2.2.3. Synchronisation . . . 21
2.2.4. Deklassifikation . . . 21
2.3. Sicherheitsbedingungen . . . 22
2.4. Sicherheitstypsysteme . . . 32

3. Erweiterung des EIFAs **37**
3.1. Ursprüngliches System: EIFAv1 . . . 38
3.2. Geänderte Anforderungen . . . 42
3.2.1. Auswahl der Sicherheitstypsysteme durch Benutzer . . . 43
3.2.2. MWL-Syntax . . . 43
3.2.3. Spezifikation der Sicherheitpolitik . . . 44
3.3. Änderungen . . . 44
3.3.1. Paket checker . . . 45
3.3.2. Paket policy . . . 46
3.3.3. Paket ast . . . 47
3.3.4. Paket mwlParser . . . 51
3.3.5. Paket visitor . . . 51
3.4. Schluss . . . 57
3.4.1. Bewertung . . . 57
3.4.2. Ausblick . . . 60

4. *Intransitive Noninterference* und *Delimited Release* **61**
4.1. Motivation . . . 61
4.2. Starke Sicherheit[dr] . . . 62
4.2.1. Grundlagen . . . 63
4.2.2. Definition . . . 63
4.2.3. Zusammensetzbarkeit . . . 68
4.2.4. Verletzung der Konservativität . . . 74
4.2.5. Kompatibilität zur starken Sicherheit[in] . . . 74
4.3. Automatische Analyse mit einem Sicherheitstypsystem . . . 77
4.3.1. Sicherheitstypsystem . . . 77

4.3.2. Möglichkeit der Implementierung im EIFA 86
4.4. Schluss 87
4.4.1. Bewertung 87
4.4.2. Ausblick 89
4.4.3. Alternative Ansätze für das „was“ der Deklassifikation 92

5. *Intransitive Noninterference* und *Robust Declassification* 93
5.1. *Robust Declassification* für starke passive Angreifer 94
5.2. *Robust Declassification* durch *Intransitive Noninterference* 96
5.3. Schluss 100
5.3.1. Schwächere passive Angreifer 100
5.3.2. Bewertung und Ergebnis 101
5.3.3. Ausblick 101

6. Schluss 103
6.1. Zusammenfassung 103
6.1.1. EIFA 103
6.1.2. *Delimited Release* 103
6.1.3. *Robust Declassification* 104
6.2. Ähnliche Arbeiten 104
6.3. Ausblick 105

Literatur 107

A. Aufgabenstellung 113

B. Regelauflistungen 117
B.1. Semantikregeln 117
B.1.1. Deklarations- und Arrayanweisungen 117
B.1.2. Synchronisationsanweisungen 117
B.2. Sicherheitstypsysteme des EIFA 119

C. Zusätzliche Beweise 123

D. Benutzung des EIFA 133
D.1. Inhalt der CD-ROM 133
D.2. Ausführung des EIFA 133
D.3. Die Beispielprogramme 134
D.4. Neucompilierung des EIFA 135

E. Entwicklungsdokumentation 137
E.1. Funktionalität des EIFAv1 137
E.2. Fehlverhalten des EIFAv1 138
E.3. Nebenanforderungen 139

F. Quelltext-Beispiele **143**
F.1. Grammatikdateien für Scanner und Parser (Paket mwlParser) 143
F.2. Paket ast 149
F.3. Paket checker 153
F.4. Paket policy 161
F.5. Paket visitor 168

1. Einleitung

1.1. Motivation

Immer mehr Geräte sind über Datennetze miteinander verbunden. Mobiltelefone, PDAs, Kameras, Autos und natürlich PCs, in all diesen Geräten werden Programme ausgeführt, die über Datennetze kommunizieren. Viele der in diese Geräten gespeicherten Daten sind schützenswert. Sie sind vertraulich oder man ist darauf angewiesen, dass diese Daten korrekt und verfügbar sind. Doch was die Programme auf diesen Geräten mit den Daten machen, bleibt einem oft verborgen. Gezwungenermaßen vertraut man den Programmen (sie sind *trusted*), aber ob sie tatsächlich vertrauenswürdig (*trustworthy*) sind, weiß man nicht. Man benötigt also Verfahren, mit denen sich automatisch die Sicherheit von Programmen überprüfen lässt.

In dieser Arbeit geht es um Vertraulichkeit, das heißt, dass bestimmte Information nur erfahren darf, wer dazu befugt ist. Hier wird Vertraulichkeit über Informationsflusspolitiken (*information flow policies*, kurz Flusspolitiken) spezifiziert. Im Gegensatz zu Zugriffspolitiken, die nur den Zugriff auf Informationsressourcen beschränken, beschränken Flusspolitiken auch den Informationsfluss nach dem Zugriff. Die Mechanismen um Flusspolitiken durchzusetzen heißen Informationsflusskontrollen, im Gegensatz zu den Zugriffskontrollen.

Man betrachte zum Beispiel ein Finanzprogramm. Es benötigt Zugriff auf eigene, vertrauliche Depot- und Kontodaten. Außerdem kommuniziert es mit einem Webserver, um aktuelle Aktienkursinformation abzurufen. Informationsflusskontrolle kann hier sicherstellen, dass die vertraulichen Daten nicht an den Webserver geschickt werden, auch nicht über sogenannte verdeckte Kanäle (*covered channels*).

Die Forderung, keine vertrauliche Information zu veröffentlichen, ist manchmal zu streng. In dem Beispiel wäre es zu aufwendig alle möglichen Aktienkurse abzufragen. Daher soll das Programm nur die Kurse der Aktien abfragen, die in dem Depot sind. Dazu muss der Webserver wissen, welche Aktien das sind. Das heißt, das Programm deklassifiziert diese eigentlich vertrauliche Information. Man möchte daher die Flusspolitik erweitern, sodass sie diese Deklassifikation erlaubt. Sie soll aber trotzdem ausdrücken, dass nur übertragen wird, welche Aktien im Depot sind und das auch nur bei der Abfrage der Aktienkurse und nur an den Webserver, der die Aktienkurse bereitstellt.

In dieser Arbeit geht es um Verfahren, die automatisch feststellen, ob Programme sicher sind. Dazu muss formal definiert sein, was „sicher“ bedeutet. Deshalb geht es in dieser Arbeit auch um formale Sicherheitsbedingungen. Diese Sicherheitsbedingungen sind Informationsflussbedingung, die mit einer Flusspolitik definiert sind.

Die in dieser Arbeit betrachteten Verfahren überprüfen statisch anhand des Programmquelltextes die Erfüllung der Sicherheitsbedingungen, das heißt ohne das Programm auszuführen. Solche Methoden behandelt die sogenannte programmiersprachenbasierte Sicherheit (*language based security*).

1.2. Einordnung

Diese Arbeit hat zwei Ziele. Zum einen soll eine bestehende Anwendung um neue Anwendungsmöglichkeiten erweitert werden. Diese Anwendung ist der EIFA (*Experimental Information Flow Analyzer*), der Programme auf Sicherheit überprüft. Zum anderen sollen Sicherheitsbedingungen definiert und untersucht werden, die Deklassifikation erlauben.

Christina Pöpper hat den EIFA[1] in einer Diplomarbeit entwickelt (s. [Pöp05]). Der EIFA überprüft Programme in der Modellsprache MWL (*Multi-Threaded While Language*) darauf, ob sie eine Sicherheitsbedingung erfüllen. Diese Sicherheitsbedingung und die implementierten Überprüfungsmechanismen stammen aus [SS00]. Die Sicherheitsbedingung ist über die Semantik von MWL definiert. Die Mechanismen sind sogenannte Sicherheitstypsysteme. Sie ähneln Typsystemen, wie sie zum Beispiel in Compilern verwendet werden, um Datentypsicherheit zu überprüfen.

In dieser Arbeit bedeutet das Überprüfen einer Bedingung nicht, dass diese Bedingung entschieden wird. Ein Verfahren, das eine Bedingung überprüft, versucht zu beweisen, dass die Bedingung erfüllt ist. Eine erfolgreiche Überprüfung bedeutet daher, dass die überprüfte Bedingung erfüllt ist. Aus einer erfolglosen Überprüfung folgt aber nicht, dass die Bedingung nicht erfüllt ist.

MWL erlaubt Nebenläufigkeit in Form von mehreren Threads, die einen gemeinsamen Speicher haben. In [Sab01] hat Sabelfeld eine Sicherheitsbedingung für MWL mit Synchronisationsanweisungen definiert und ein Sicherheitstypsystem dafür angegeben. Im Rahmen der vorliegenden Arbeit wird der EIFA um dieses Sicherheitstypsystem erweitert.

Außerdem wird der EIFA so erweitert, dass er Sicherheitsbedingungen mit komplexeren Flusspolitiken überprüfen kann. Üblicherweise sind Sicherheitsbedingungen als sogenannte *noninterference*-Bedingungen formuliert (s. [SM03]). *Noninterference*-Bedingungen nutzen transitive Flusspolitiken.

In [MS04] führen Mantel und Sands die sogenannte *intransitive noninterference* in die programmiersprachenbasierte Sicherheit ein. Hier hat die Flusspolitik eine intransitive Komponente. Nur bei Ausführung einer MWL-Deklassifikationsanweisung darf Information entlang dieser Komponente fließen. Sie erlaubt also an bestimmten Stellen im MWL-Programm und an bestimmten Stellen in der Flusspolitik Deklassifikation, sonst aber nicht. Im Rahmen der vorliegenden Arbeit wird der EIFA um das Sicherheitstypsystem aus [MS04] erweitert, das die auf *intransitive noninterference* basierende Sicherheitsbedingung überprüft.

Zusätzlich zur Mehrfunktionalität des EIFA bringen diese Erweiterungen auch die Erkenntnis, dass der Entwurf des EIFA die genannten Änderungen unterstützt und welche Änderungen der Entwurf erschwert.

Neben der *intransitive noninterference* gibt es noch andere Ansätze die Deklassifikation zu erlauben. In [MS04] wird festgestellt, dass sich die Ansätze danach einteilen lassen, welchen Aspekt der Deklassifikation sie begrenzen. *Intransitive noninterference* begrenzt „wo“ in der Flusspolitik und „wo“ im Programm Deklassifikation

[1] dieser Name wird in der vorliegenden Arbeit benutzt, zuvor war das Programm noch unbenannt

erlaubt ist. In dem Beispiel mit dem Finanzprogramm hieße das, nur vom Aktiendepot zum Webserver darf deklassifiziert werden und nur bei der Abfrage der Aktienkurse. Ein anderer Aspekt ist, „was“ deklassifiziert werden darf. In dem Beispiel soll nur deklassifiziert werden, welche Aktien im Depot sind, aber zum Beispiel nicht die jeweilige Anzahl. Ein dritter Aspekt ist, „wer“ eine Deklassifikation veranlassen darf. In dem Finanzprogramm darf zum Beispiel der Webserver keine Kursabfrage auslösen können, da er damit die Deklassifikation auslöst. [2]

Einen Überblick über die Ansätze zur Deklassifikation bietet [SS05], wo die Ansätze auch nach den oben genannten Aspekten eingeteilt werden. Dort werden die Aspekte „Dimensionen“ genannt, um die Orthogonalität zu betonen. In der vorliegenden Arbeit wird dieser Begriff übernommen.

Das Ziel ist es, Deklassifikation zu erlauben, aber in allen diesen Dimensionen der Deklassifikation zu begrenzen. Deshalb wird in dieser Arbeit *intransitive noninterference* um einen Ansatz ergänzt, der das „was“ begrenzt, also welche Information ein Programm deklassifiziert. Dazu wird basierend auf *delimited release* aus [SM04] eine Sicherheitsbedingung für MWL definiert. Anschließend wird untersucht, ob die Konjunktion der Sicherheitsbedingungen sinnvoll anwendbar ist. Es stellt sich heraus, dass dies unter bestimmten Bedingungen zutrifft. Außerdem wird ein Sicherheitstypsystem angegeben, das MWL-Programme nach beiden Sicherheitsbedingungen überprüft.

Um noch die Dimension „wer“ abzudecken wird eine weitere Sicherheitsbedingung definiert und untersucht. Diese basiert auf der *robust declassification* aus [MSZ06]. Für diese Sicherheitsbedingung stellt sich heraus, dass sie unter bestimmten Bedingungen mit der *intransitive noninterference* durchgesetzt werden kann.

Diese Arbeit führt also die drei Dimensionen der Deklassifikation zusammen.

1.3. Struktur der Arbeit

Die Arbeit besteht im Kern aus zwei Teilen, einem Entwicklungsteil (Abschnitt 3) und einem theoretischen Teil (Abschnitt 4 und Abschnitt 5).

Der Abschnitt 2 stellt die Definitionen und Konzepte vor, die der Funktionalität des EIFA zugrundeliegen. Das sind die Flusspolitiken, die Sprache MWL mit ihren Erweiterungen, die Sicherheitsbedingungen und die Sicherheitstypsysteme. Diese Definitionen und Konzepte sind aus anderen Arbeiten übernommen ([Pöp05], [Sab01], [MS04]). Neu ist die Kombination zu einer einzigen Sprache mit zugehöriger Sicherheitsbedingung und entsprechendem Sicherheitstypsystem. Außerdem enthält dieser Abschnitt einige MWL-Beispiele, welche die Einsatzmöglichkeiten der Konzepte zeigen.

Der Abschnitt 3 stellt zunächst den Entwurf des EIFA aus [Pöp05] vor. Dann beschreibt er die Entwurfsänderung.

Der Abschnitt 4 enthält die Definition einer Sicherheitsbedingung für MWL, basierend auf der *delimited release* aus [SM04]. Anschließend werden einige Eigenschaften

[2] In [SS05] wird noch ein vierter Aspekt „wann“ unterschieden.

gezeigt und ein Sicherheitstypsystem definiert.

Der Abschnitt 5 enthält die Definition einer Sicherheitsbedingung für MWL, basierend auf der *robust declassification* aus [MSZ06]. Es wird gezeigt, wie sich diese Sicherheitsbedingung beweisen lässt.

Der Abschnitt 6 fasst die einzelnen Ergebnisse zusammen und bietet einen Ausblick.

1.4. Konventionen

Zunächst werden einige Konventionen aufgelistet, denen diese Arbeit folgt.

Englische Ausdrücke Englische Ausdrücke stehen *kursiv* geschrieben. Meistens wird eine deutsche Übersetzung benutzt. Hinter dem ersten Auftreten der deutschen Übersetzung steht der englische Ausdruck in Klammern. Einige Fachausdrücke sind nicht übersetzt, weil sie Namen für Konzepte sind, zu denen es keine oder kaum deutschsprachige Literatur gibt. Das sind insbesondere „*(intransitive) noninterference*“, „*delimited release*“ und „*robust declassification*“.

Nummerierung Sätze, Lemmata und Propositionen sind gemeinsam durchgehend nummeriert. Abbildungen, Definitionen und Beispiele sind jeweils einzeln durchgehend nummeriert.

Übernommene Definitionen und Aussagen Definition, Sätze, Lemmata und Propositionen, die aus anderen Arbeiten übernommen sind, kennzeichnet ein Quellverweis direkt nach der jeweiligen Nummer.

Farben in Programmbeispielen In Programmbeispielen sind manchmal Bezeichner farbig hervorgehoben. Diese Farben bilden die Sicherheitsdomänen der Bezeichner ab (s. Abschnitt 2.1). Die genaue Zuordnung geht jeweils aus dem Kontext hervor, zum Beispiel aus Flusspolitikdiagrammen.

Aber die Zuordnung der Bezeichner zu Sicherheitsdomänen wird auch jeweils erläutert, sodass auf Graustufenkopien dieser Arbeit keine Information fehlt. Die Farben dienen nur der Übersichtlichkeit.

Regelnotation Diese Arbeit enthält mehrere Definitionen von Regelsystemen, das heißt Mengen von Regeln. Die Regeln stehen in folgender Notation: $\frac{\text{Bedingung}}{\text{Schluss}}$. Das heißt, wenn Bedingung erfüllt ist, gilt auch Schluss. Falls eine Regel mehr als eine Einzelbedingung hat, bildet die Konjunktion der Einzelbedingungen die Gesamtbedingung. Elemente die im Schluss vorkommen sind implizit universell quantifiziert. Elemente, die nur unquantifiziert in einer Bedingung vorkommen, sind implizit existentiell quantifiziert. Zum Beispiel:

$$\frac{P(c,b) \quad \forall d : Q(c,d)}{S(a,b)} \qquad \Leftrightarrow \qquad \forall a,b : [\exists c : (P(c,b) \wedge \forall d : Q(c,d))] \Rightarrow S(a,b)$$

2. Grundlagen

2.1. Sicherheitspolitiken

Statt dem allgemeinen Begriff Sicherheitspolitik (*security policy*, oft auch als Sicherheitsrichtlinie übersetzt) benutzt diese Arbeit konkretere Begriffe. In diesem Unterabschnitt werden Flusspolitiken (*flow policies*) und Domänenzuweisungen (*domain assignments*) definiert. Zum Beispiel in [SM04] werden diese beiden Konzepte zusammen *security policy* genannt (sie heißen dort *security lattice* und *security environment*).

Zentrales Element eines formalen Sicherheitsmodells ist eine Flusspolitik. Diese beschreibt den erlaubten Informationsfluss in einem System. In dieser Arbeit bedeutet das, die Flusspolitik beschränkt oder erlaubt Informationsfluss zwischen Sicherheitsdomänen. Sicherheitsdomänen, auch Sicherheitsklassen genannt, sind sich nicht überschneidende Mengen von Informationsressourcen. Im Rahmen der programmiersprachenbasierten Sicherheit sind diese Informationsressourcen die Bezeichner der Sprache, also zum Beispiel Wertevariablen, Arrayvariablen oder Semaphore.

Zu beachten sind zwei Punkte:

1. Die Flusspolitik beschreibt *nicht*, welche Information fließen *muss*. Das heißt diese Arbeit beschränkt sich auf Flusspolitiken, die Vertraulichkeitsanforderungen beschreiben aber keine Verfügbarkeitsanforderungen. Integritätsanforderungen sind im Prinzip mit diesen Flusspolitiken ausdrückbar, allerdings behandelt diese Arbeit diese nur am Rande (s. Abschnitt 5).

2. Die Flusspolitik beschreibt nicht, was es bedeutet, dass Information fließt oder das keine Information fließt. Auf welche Weise Information fließen kann, definiert indirekt die Sicherheitsbedingungen (sogenannte *noninterference*-Bedingungen, s. Abschnitt 2.3).

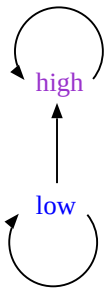


Abbildung 1: Flusspolitik mit zwei Sicherheitsdomänen. Die Pfeile repräsentieren den erlaubten Informationsfluss. Die Farben der beiden Sicherheitsdomänen (s. Abschnitt 1.4) verwendet diese Arbeit durchgehend, auch für komplexere Flusspolitiken

Die einfachste nicht-triviale Flusspolitik enthält zwei Sicherheitsdomänen (s. Abbildung 1). Es gibt eine öffentliche Sicherheitsdomäne „low“, von der Information überall hinfließen darf. Weiter gibt es die vertrauliche Sicherheitsdomäne „high“, von der aus keine Information in die Sicherheitsdomäne „low“ fließen darf. Trotz der Einfachheit lassen sich damit schon viele Anforderungen modellieren. Zum Beispiel implementieren Programme oft einen Kommunikationsendpunkt. Sie empfangen oder

senden einen Teil der Information. Diese Information speichert so ein Programm in Variablenbezeichnern der Sicherheitsdomäne „low“. Andere Information ist vertraulich, diese soll so ein Programm nicht senden. Diese Information speichert so ein Programm in Variablenbezeichnern der Sicherheitsdomäne „high“.

Aufgrund der Einfachheit wird diese Flusspolitik in vielen Veröffentlichungen zur programmiersprachenbasierten Sicherheit benutzt, so auch in [SS00],[Sab01] und [Pöp05].

Eine komplexere Flusspolitik kann beliebig viele Sicherheitsdomänen enthalten. In der einfachsten Form definiert man eine Relation zwischen den Sicherheitsdomänen, die Flussrelation. Die intuitive Bedeutung ist die: genau dann, wenn zwei Sicherheitsdomänen D_1 und D_2 in Flussrelation zueinander stehen, darf im Ablauf des Programms Information von D_1 nach D_2 fließen.

Hier stellt sich die Frage, welche Bedingung eine Flussrelation erfüllen muss, um sinnvoll zu sein. Denning argumentiert in [Den76], dass es nützlich ist wenn die Menge der Sicherheitsdomänen mit der Flussrelation einen (vollständigen) Verband bildet. Für die Zwecke dieser Arbeit genügt aber eine partielle Ordnung. Diese schränkt den Ersteller einer Flusspolitik weniger ein.

Definition 1 ([MS04] MLS-Politik (*MLS Policy*)). Wenn $\mathcal{D}$ eine Menge von Sicherheitsdomänen und $\leq \subseteq \mathcal{D} \times \mathcal{D}$ eine partielle Ordnung sind, dann ist ein Paar $(\mathcal{D}, \leq)$ eine *mehrstufige Flusspolitik (MLS-Politik).*

Falls es eine bezüglich $\leq$ kleinste Sicherheitsdomäne gibt, wird sie *low* genannt, eine größte *high*.

In [MS04] wird eine weitere Art von Flusspolitiken definiert. Sie basiert auf dem Prinzip der *intransitive noninterference* aus [Rus92] von Rushby. Die Motivation dahinter ist, dass sich durch Flusspolitiken mit transitiver Flussrelation manche Anwendungen nicht ausreichend exakt modellieren lassen. Wenn man zum Beispiel festlegen will, dass von einer Sicherheitsdomäne A zu einer Sicherheitsdomäne C Information fließen darf, aber nur wenn sie eine dritte Sicherheitsdomäne B passiert, ist eine transitive Flusspolitik nicht anwendbar.

Definition 2 ([MS04] MLS-Politik mit Ausnahmen (*MLS Policy with Exceptions*)). Wenn $\mathcal{D}$ eine Menge von Sicherheitsdomänen, $\leq \subseteq \mathcal{D} \times \mathcal{D}$ eine partielle Ordnung und $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ sind, dann ist ein Tripel $(\mathcal{D}, \leq, \rightsquigarrow)$ eine *mehrstufige Flusspolitik mit Ausnahmen.*

Die Komponente $\rightsquigarrow$ muss nicht transitiv sein. Mit der zu definierenden Sicherheitsbedingung hat sie für zwei Sicherheitsdomänen D_1 und D_2 mit $D_1 \rightsquigarrow D_2$ die Bedeutung, dass selbst wenn nicht $D_1 \leq D_2$ gilt, unter bestimmten Bedingungen, z.B. die Ausführung einer Deklassifikationsanweisung, Information von D_1 nach D_2 fließen darf.

In dieser Arbeit wird nicht streng zwischen MLS-Politiken mit und ohne Ausnahmen unterschieden und beide werden „MLS-Politik“ genannt. Indem man die Komponente $\rightsquigarrow$ nicht beachtet, kann man eine MLS-Politik mit Ausnahmen als

MLS-Politik ohne Ausnahmen verwenden. Wenn man $\rightsquigarrow$ verwendet ist klar, dass man eine MLS-Politik mit Ausnahmen meint. Das gilt zum Beispiel für die Sicherheitsbedingungen im Abschnitt 2.3: in Definition 5 kann man $\rightsquigarrow$ ignorieren und Definition 9 verwendet $\rightsquigarrow$.

Wir folgt wird der Zusammenhang zwischen Bezeichnern und Sicherheitsdomänen definiert:

Definition 3 ([MS04] Domänenzuweisung (*domain assignment*)). Seien $\mathcal{D}$ eine Menge von Sicherheitsdomänen und $\mathcal{I}$ eine Menge von Bezeichnern. Eine Funktion $\Gamma : \mathcal{I} \mapsto \mathcal{D}$ ist eine *Domänenzuweisung*.

Die Domänenzuweisung wird im nächsten Unterabschnitt benötigt, um die Semantik einer Deklassifikationsanweisung zu definieren und im übernächsten, um Sicherheitsbedingungen zu definieren.

In Programmbeispielen ist die Domänenzuweisung nicht immer explizit angegeben, sondern die Namen der Bezeichner spezifizieren sie. h, h1, h2, bh usw. haben die Sicherheitsdomäne *high*, l, l1, l2, bl usw. haben die Sicherheitsdomäne *low*.

2.2. Die „Multi-Threaded While Language“

MWL wird in [SS00] als eine einfache Programmiersprache eingeführt, um konkrete Sicherheitstypsysteme zu erstellen. In [Pöp05] wird MWL um Deklarationen und Arrays erweitert, um das Sicherheitstypsystem aus [SS00] im EIFA zu implementieren und anschaulichere Beispielprogramme erstellen zu können. In [Sab01] wird MWL um Synchronisationsanweisungen erweitert, um den Einfluss von Synchronisation auf den Informationsfluss zu untersuchen. In [MS04] wird MWL um eine Deklassifikationsanweisung erweitert, um eine Form von intransitiven Informationsfluss zu untersuchen. Da im Rahmen der vorliegenden Arbeit ein Sicherheitstypsystem für MWL mit Synchronisationsanweisungen und Deklassifikation im EIFA implementiert wird, werden zusätzlich alle Erweiterung miteinander kombiniert.

In diesem Abschnitt wird zunächst die Syntax von MWL mit allen hier betrachteten Erweiterungen vorgestellt. Anschließend wird die Semantik vorgestellt und auf die einzelnen Erweiterungen eingegangen.

2.2.1. Syntax und allgemeine Semantik

Mit Zuweisung, Anweisungsfolge, bedingter Anweisung und bedingter Schleife enthält MWL die Standardelemente einer imperativen Programmiersprache. Die Erzeugung neuer Threads ermöglicht die Fork-Anweisung. Die Skip-Anweisung verändert bei ihrer Ausführung keine Daten. Die MWL-Syntax ist durch die Grammatik in Abbildung 2 definiert. C_{ext} steht für die Sprachelemente der Erweiterungen, deren Semantik in den folgenden Abschnitten vorgestellt wird.

$\mathcal{C}$ ist dann eine Menge von MWL-Anweisungen, wenn alle Elemente von $\mathcal{C}$ die Grammatik für C erfüllen. Dabei sind C_{Array} Deklarations- und Arrayanweisungen,

$$
\begin{array}{lll}
C & ::= & \textsf{skip} \mid Id := Exp \mid \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2 \\
 & & \mid C_1 \,;\, C_2 \mid \textsf{fork(} \vec{CD} \textsf{)} \mid \textsf{while } B \textsf{ do } C \mid C_{ext} \\
C_{ext} & ::= & C_{Array} \mid C_{Sync} \mid C_{Declass} \\
C_{Array} & ::= & Id : Type : SecDom \mid Arr : Type \,[\, Exp \,]\!: SecDom \\
 & & \mid Arr \,[\, Exp_1 \,]\!:= Exp_2 \\
C_{Sync} & ::= & Sem \,:\!\textsf{sem}\!: SecDom \mid \textsf{signal(} Sem \textsf{)} \mid \textsf{wait(} Sem \textsf{)} \\
C_{Declass} & ::= & [Id\!:\!=\!Id'] \\
Exp & ::= & Const \mid Id \mid Exp1\ op\ Exp2 \mid Exp_{Array} \\
Exp_{Array} & ::= & Arr[Exp] \mid Arr.\textsf{length}
\end{array}
$$

Abbildung 2: MWL-Grammatik

C_{Sync} Synchronisationsanweisungen und $C_{Declass}$ Deklassifikationsanweisungen. Deklarationsanweisungen dürfen nicht in komplexen Anweisungen (**if** , **while** und **fork**) auftreten. $Type$ ist ein Element aus einer Menge von Datentypen. $SecDom$ ist eine Elemente aus einer Menge von Sicherheitsdomänen $\mathcal{D}$.

In Anweisungsmengen $\mathcal{C}$ ohne Arrayanweisungen, darf keine der Anweisungen Arrayausdrücke (Exp_{Array}) enthalten.

$C, C', C_1, \ldots$ stehen für Anweisungen. $\vec{C}, \vec{C'}, \vec{V}, \ldots$ stehen für Anweisungsvektoren, wobei $\vec{\mathcal{C}} = \bigcup_{n \in \mathbb{N}} \mathcal{C}^n$ die Menge der Anweisungsvektoren für eine Menge von MWL-Anweisungen $\mathcal{C}$ ist. Statt Anweisung oder Anweisungsvektor steht in dieser Arbeit auch Programm, wenn es der Verständlichkeit dient.

Id und Arr sind Bezeichner aus einer Menge von Bezeichnern $\mathcal{I}$. $Const$ ist ein Elemente aus einer Menge von Werten $\mathcal{V}$. Die exakte Definition der Syntax und Semantik von Ausdrücken (in obiger Grammatik Exp und B) ist zur Behandlung von Informationsfluss in dieser Arbeit nicht nötig.

Erst wenn man eine Anwendung (wie den EIFA) implementiert, dem ein Benutzer konkrete MWL Programme übergibt, muss eine exakte Syntax definiert sein (s. Abbildung 17). Für einen Ausdruck steht B statt Exp, wenn verdeutlicht werden soll, dass sich dieser Ausdruck zu einem booleschen Werte auswerten lässt, also $true$ oder $false$.

Hier wird die Semantik für MWL ohne die Erweiterungsanweisungen (C_{ext}) vorgestellt. Ein Zustand s aus einer Zustandsmenge $\mathcal{S}$ ist ein Abbildung von Bezeichnern $Id \in \mathcal{I}$ auf Werte n aus eine Wertemenge $\mathcal{V}$. Der Zustand $[Id = n]s$ ist so definiert, dass für alle Bezeichner Id' gilt: $[Id = n]s(Id') = \begin{cases} s(Id') & Id' \neq Id \\ n & Id' = Id \end{cases}$.

$\langle\!\langle Exp, s \rangle\!\rangle \downarrow n$ bedeutet, dass ein Ausdruck Exp im Zustand s zu n ausgewertet

wird. Diese Auswertung wird als total[3] und atomar angenommen. Wenn der Zustand aus dem Zusammenhang ersichtlich ist, wird n auch Wert von Exp genannt. Als Abkürzung für $\exists n \in \mathcal{V} : \langle\!|Exp_1, s_1|\!\rangle \downarrow n \wedge \langle\!|Exp_2, s_2|\!\rangle \downarrow n$ steht $\langle\!|Exp_1, s_1|\!\rangle \approx \langle\!|Exp_2, s_2|\!\rangle$.

$$\frac{s(Id) = n}{\langle\!|Id, s|\!\rangle \downarrow n} \qquad \frac{}{\langle\!|Const, s|\!\rangle \downarrow Const} \qquad \frac{\langle\!|Exp_1, s|\!\rangle \downarrow n_1 \quad \langle\!|Exp_2, s|\!\rangle \downarrow n_2}{\langle\!|Exp_1 \ op \ Exp_2, s|\!\rangle \downarrow I_{op}(op)(n_1, n_2)}$$

Abbildung 3: Semantische Auswertung von MWL-Ausdrücken. I_{op} ist Teil einer konkreten Semantikdefinition für MWL-Ausdrücke. I_{op} bildet die MWL-Operatoren auf Funktionen über die Wertemenge $\mathcal{V}$ ab.

Ein Programmvektor $\vec{V}$ und ein Zustand s bilden eine Konfiguration $\langle\!|\vec{V}, s|\!\rangle$. Dabei kann man $\vec{V}$ als die aktuell aktiven Threads und s als den aktuellen Speicherinhalt verstehen. Die Menge der Konfigurationen ist $Conf = \vec{\mathcal{C}} \times \mathcal{S}$.

Die operative Semantik wird in zwei Teilen als Smallstep-Semantik[4] definiert. Ein Teil ist die deterministische Smallstep-Semantik für Anweisungen (s. Abbildung 4). $\langle\!|C, s|\!\rangle \rightarrowtriangle \langle\!|\vec{C}, s'|\!\rangle$ bedeutet, dass die Ausführung eines Schrittes eines MWL-Programms die Konfiguration $\langle\!|C, s|\!\rangle$ in die Konfiguration $\langle\!|\vec{C}, s'|\!\rangle$ überführt.

$$\frac{}{\langle\!| \ \mathsf{skip} \ , s|\!\rangle \rightarrowtriangle \langle\!|(), s|\!\rangle} \qquad \frac{\langle\!|Exp, s|\!\rangle \downarrow n}{\langle\!|Id := Exp, s|\!\rangle \rightarrowtriangle \langle\!|(), [Id = n]s|\!\rangle} \qquad \frac{}{\langle\!| \ \mathsf{fork(} \ C\vec{D} \ \mathsf{)} \ , s|\!\rangle \rightarrowtriangle \langle\!|C\vec{D}, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow true}{\langle\!| \ \mathsf{if} \ \ B \ \ \mathsf{then} \ \ C_1 \ \ \mathsf{else} \ \ C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_1, s|\!\rangle} \qquad \frac{\langle\!|B, s|\!\rangle \downarrow false}{\langle\!| \ \mathsf{if} \ \ B \ \ \mathsf{then} \ \ C_1 \ \ \mathsf{else} \ \ C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_2, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow true}{\langle\!| \ \mathsf{while} \ \ B \ \ \mathsf{do} \ \ C, s|\!\rangle \rightarrowtriangle \langle\!|C \ ; \mathsf{while} \ \ B \ \ \mathsf{do} \ \ C, s|\!\rangle} \qquad \frac{\langle\!|B, s|\!\rangle \downarrow false}{\langle\!| \ \mathsf{while} \ \ B \ \ \mathsf{do} \ \ C, s|\!\rangle \rightarrowtriangle \langle\!|(), s|\!\rangle}$$

$$\frac{\langle\!|C_1, s|\!\rangle \rightarrowtriangle \langle\!|(), s'|\!\rangle}{\langle\!|C_1 \ ; \ C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_2, s'|\!\rangle} \qquad \frac{\langle\!|C_1, s|\!\rangle \rightarrowtriangle \langle\!|C_1'\vec{D}, s'|\!\rangle}{\langle\!|C_1 \ ; \ C_2, s|\!\rangle \rightarrowtriangle \langle\!|\langle C_1'; C_2\rangle\vec{D}, s'|\!\rangle}$$

Abbildung 4: Deterministische Smallstep-Semantik der MWL in der Grundform

Ein weiterer Teil ist die nichtdeterministische Smallstep-Semantik für Anweisungsvektoren (s. Abbildung 5). $\langle\!|\vec{C}, s|\!\rangle \rightarrow \langle\!|\vec{C'}, s'|\!\rangle$ bedeutet, dass die Ausführung eines

[3] Konkrete Ausdrücke, wie zum Beispiel eine Division sind nicht unbedingt total, da sie zum Beispiel nicht auf boolesche Werte definiert sind. Hier wird angenommen, dass Programme datentypsicher sind, damit solche Fälle nicht eintreten.

[4] Smallstep-Semantik bedeutet, dass die Semantik jeden Konfigurationsübergang eines Programms definiert. Dies steht im Gegensatz zu einer Bigstep-Semantik, die nur die durch ein Programm berechnete Funktion definiert.

Schrittes $\langle\!|\vec{C}, s|\!\rangle$ in $\langle\!|\vec{C}', s'|\!\rangle$ überführen kann. Die hier definierte Regel bedeutet, dass das ausführende System einen der aktiven Threads auswählt und ausführt.

$$\frac{\langle\!|C_i, s|\!\rangle \twoheadrightarrow \langle\!|\vec{C}, s'|\!\rangle}{\langle\!|\langle C_0 \dots C_{n-1}\rangle, s|\!\rangle \rightarrow \langle\!|\langle C_0 \dots C_{i-1}\vec{C}C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

Abbildung 5: Nichtdeterministische Smallstep-Semantik der MWL in der Grundform

Diese Regeln bilden also Anweisungen auf ein Transitionen zwischen Konfigurationen ab. Über den deterministischen Transitionen wird im Abschnitt 2.3 die Sicherheitsbedingung formuliert.

Es wird angenommen, dass bei der Ausführung eines MWL-Programms jede Transition $\twoheadrightarrow$ eine Zeiteinheit benötigt und atomar ist. Das ist eine starke Annahme. Dabei werden weder die Länge von Ausdrücken, noch eventuelle Geschwindigkeitsunterschiede durch Cache-Mechanismen beachtet.

2.2.2. Deklarationen und Arrays

Deklarationen ($Id : Type : SecDom$ und $Arr : Type$ [Exp]: $SecDom$) ermöglichen dem Programmierer eines MWL-Programms den Datentyp und Sicherheitstyp eines Bezeichners festzulegen. Der Datentyp macht für einen Leser die Bedeutung eines Programms verständlicher. Desweiteren lässt sich ein solches Programm auf Datentypsicherheit[5] prüfen, falls Datentypregeln gegeben sind. Der Sicherheitstyp ermöglicht dem Programmierer die Festlegung einer Domänenzuweisung, über der die Sicherheit des Programms gelten soll.

Die exakten Semantikregeln für Deklarations- und Arrayanweisungen befinden sich im Anhang B.1.1. Eine Deklarationsanweisung verhält sich wie die Zuweisung eines Wertes $default$ an den deklarierten Bezeichner, bzw. an alle Arrayelemente des deklarierten Arrays. $default$ ist ein ausgezeichneter Wert aus der Wertemenge $\mathcal{V}$.

Ein Arraybezeichner Arr wird zu einer indizierten Folge über die Wertemenge $\mathcal{V}$ ausgewertet. Ein Arrayausdruck $Arr.$**length** wird zur Länge des Arrays ausgewertet. Ein Arrayausdruck $Arr[Exp]$ wird zu dem durch den Wert von Exp indizierten Arrayelement ausgewertet. Umgekehrt schreibt eine ausgeführte Arrayzuweisung in das indizierte Element. Falls der Wert von Exp nicht im Indexbereich liegt, wird der Ausdruck zu $default$ ausgewertet. Die Zuweisung ändert in diesem Fall nichts am Zustand (s. Abbildung 32 im Anhang B.1.1).

[5] Um Verwechslungen vorzubeugen wird hier explizit der Begriff „Datentypsicherheit“ anstatt dem üblichen Begriff „Typsicherheit“ (*type safety*) verwendet.

2.2.3. Synchronisation

In [Sab01] werden Semaphore als Synchronisationsprimitive gewählt. Ein Semaphor ist ein spezieller Variablenbezeichner ($Sem \in \mathcal{I}$)[6] mit einem nichtnegativen, ganzzahligen Wert. Nur die Anweisung **signal**(Sem) kann ihn inkrementieren und nur die Anweisung **wait**(Sem) kann ihn dekrementieren. Ein Semaphor Sem ist kein Ausdruck, das heißt ein Programmierer darf Elemente aus der Bezeichnermenge $\mathcal{I}$ in einem Programm nicht als Ausdruck und als Semaphor verwenden. Falls ein Thread einen Semaphor dekrementieren soll und der Wert des Semaphors 0 ist, wartet dieser Thread in Form des blockierenden Wartens (*blocked waiting*).

Zur Definition der Semantik wird in [Sab01] eine markierte deterministische Transition $\xrightarrow{\alpha}$ eingeführt, wobei $\alpha \in \{\odot Sem, \otimes Sem\}$. $\otimes Sem$ bedeutet, dass das Semaphor Sem den ausgeführten Thread blockiert. $\odot Sem$ bedeutet, dass ein von Sem blockierter Thread entblockt werden kann.

Die nichtdeterministische Transition $\rightarrow$ wird nicht um Markierungen ergänzt. Dafür erweitert eine Komponente die Konfiguration, die FIFO-Warteschlangen von wartenden Threads repräsentiert. Eine Konfiguration sieht hier so aus: $\langle\!|\vec{C}, w, s|\!\rangle$, wobei $\vec{C}$ und s wie vorher ein Anweisungsvektor und ein Zustand sind. w bildet Semaphore auf Anweisungsvektoren ab, welche jeweils die Warteschlangen zu dem Semaphor repräsentieren.

Die Semantik für Synchronisationsanweisungen ist im Anhang B.1.2 definiert.

2.2.4. Deklassifikation

Die Deklassifikationanweisung $[Id{:=}Id']$ verhält sich fast wie eine normale Zuweisung. Der einzige Unterschied ist, dass die Semantik die Transitionen dieser Anweisung markiert, um ihr Auftreten der Sicherheitsbedingung zugänglich zu machen (wie oben erwähnt, werden die Sicherheitsbedingungen über den Transitionen formuliert). Die Markierung kennzeichnet die Transition als Deklassifikationsansweisung und enthält die Quell- und Zielsicherheitsdomänen: $\rightarrowtriangle_d^{D_1 \to D_2}$. Zur Unterscheidung werden auch die sonstigen Transitionen mit einem o: $\rightarrowtriangle_o$ markiert. Es ist dann $\rightarrowtriangle = \rightarrowtriangle_o \cup (\bigcup_{\forall D_1, D_2 \in \mathcal{D}} \rightarrowtriangle_d^{D_1 \to D_2})$. Entsprechendes gilt für die nichtdeterministischen Transitionen.

Damit ergeben sich die Ergänzungen für die deterministische (Abbildung 6) und die nichtdeterministische Semantik (Abbildung 7). Man beachte, dass für diese Definition der Semantik eine Domänenzuweisung Γ gegeben sein muss.

Die Idee hinter der Einführung dieser Anweisung ist, dass eine Anweisung $[\mathsf{a}{:=}\mathsf{b}]$ immer erlaubt ist, wenn eine gegebene MLS-Politik $\Gamma(\mathsf{b}) \rightsquigarrow \Gamma(\mathsf{a})$ enthält. Insbesondere auch dann, wenn nach der MLS-Politik $\Gamma(\mathsf{b}) \not\leq \Gamma(\mathsf{a})$.

Wenn man nach dieser Semantik die Transitionen $\rightarrowtriangle$ betrachtet, dann ist die Semantik identisch zur ursprünglichen Semantik. Dabei verhält sich die Deklassifikation wie eine Zuweisung.

[6] In [Sab01] ist Sem ein Element einer speziellen Menge von Semaphorvariablenbezeichnern. Damit muss man aber die Domänenzuweisung formal auf diese Menge erweitern.

$$\frac{\langle\!|Id', s|\!\rangle \downarrow n \quad \Gamma(Id') = D_1 \quad \Gamma(Id) = D_2}{\langle\!|[Id{:=}Id'], s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|(), [Id = n]s|\!\rangle}$$

$$\frac{\langle\!|C_1, s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|(), s'|\!\rangle}{\langle\!|C_1 \,;\, C_2, s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|C_2, s'|\!\rangle}$$

Abbildung 6: Deterministische Smallstep-Semantik der MWL mit Deklassifikation. Sie erweitert die Semantik aus Abbildung 4, wobei dort in den Regeln $\rightarrowtriangle$ durch $\rightarrowtriangle_o$ ersetzt wird.

$$\frac{\langle\!|C_i, s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|\vec{C}, s'|\!\rangle}{\langle\!|\langle C_0 \dots C_{n-1}\rangle, s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|\langle C_0 \dots C_{i-1}\vec{C}C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

Abbildung 7: Nichtdeterministische Smallstep-Semantik der MWL mit Deklassifikation. Sie erweitert die Semantik aus Abbildung 5, wobei dort in den Regeln $\rightarrow$ durch $\rightarrow_o$ ersetzt wird.

2.3. Sicherheitsbedingungen

Jetzt ist definiert, wie man den gewünschten Informationsfluss bzw. dessen Einschränkung spezifiziert und es ist die Programmiersprache MWL mit verschiedenen Erweiterungen definiert. Nun wird definiert, wann ein gegebenes MWL Programm mit einer gegebenen Flusspolitik „sicher“ ist.

Da die Sicherheitsbedingungen über die Semantik definiert sind, gibt es in diesem Abschnitt keine Unterscheidung zwischen MWL mit oder ohne Deklarationen und Arrayausdrücken. Dagegen werden jeweils unterschiedliche Sicherheitsbedingungen für MWL mit Synchronisation und MWL mit Deklassifikation definiert, da hier die Semantik mit unterschiedlich markierten Transitionen definiert ist.

Für das Sicherheitsmodell besteht die Annahme, dass einem Angreifer eine Sicherheitsdomäne D zugeordnet ist und er die Inhalte aller Bezeichner Id mit $\Gamma(Id) \leq D$ sehen kann. Es werden Zustände gleichgesetzt, die sich von einem Angreifer mit der Sicherheitsdomäne D nicht unterscheiden lassen. So abstrahiert eine Äquivalenzrelation über Zuständen die Annahme.

Definition 4 (D-Gleichheit (*D-Equality*))**.** Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $D \in \mathcal{D}$. Zwei Zustände s und s' sind genau dann *D-gleich* (geschrieben $s =_D s'$), wenn

1. $\forall Id \in \mathcal{I} : \Gamma(Id) \leq D \Rightarrow s(Id) = s'(Id)$ und
2. $\forall Arr \in \mathcal{I} : \exists n \in \mathbb{N} : s(Arr) = [Arr_0, \dots, Arr_{n-1}] \wedge s'(Arr) = [Arr'_0, \dots, Arr'_{n-1}]$ (die Längen der Arrays sind jeweils gleich).

Diese Definition ist eine allgemeinere Form der *low*-Gleichheit (*low-equality*) $=_L$ für die binäre Flusspolitik. $=_L$ wird in [Pöp05] verwendet. [7]

Mit dieser D-Gleichheit lässt sich Sicherheit in Worten so ausdrücken: Ein Programm ist sicher, wenn zwei beliebige D-gleiche Startzustände zu für den Angreifer ununterscheidbaren Verhalten führen.

Das ununterscheidbare Verhalten aus D-gleichen Startzuständen wird für MWL ohne Synchronisationsanweisungen wie folgt definiert:

Definition 5 ([MS04] Starke D-Bisimulation (*Strong D-Bisimulation*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen ohne Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Bisimulation* $\cong_D$ die Vereinigung aller symmetrischer Relationen R zwischen Anweisungsvektoren $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ der gleichen Größe ($\vec{V} = (C_1, \ldots, C_n)$ und $\vec{V'} = (C'_1, \ldots, C'_n)$), sodass

$$\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall \vec{W} : \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \twoheadrightarrow \langle\!| \vec{W}, t |\!\rangle$$
$$\Rightarrow \exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \twoheadrightarrow \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D t'$$

Die starke D-Bisimulation ist eine Verallgemeinerung der starken *low*-Bisimulation (*strong low-Bisimulation*) $\cong_L$ für die binäre Flusspolitik (s. [SS00]). $\cong_L$ wird in [Pöp05] verwendet.

Es ist zu beachten, dass die starke D-Bisimulation eine partielle Bisimulation ist. Zum Beispiel gilt $\mathsf{l} := \mathsf{h} \not\cong_{low} \mathsf{l} := \mathsf{h}$. Daher lässt sich mit der starken D-Bisimulation die Sicherheit vor einem Angreifer mit der Sicherheitsdomäne D wie folgt definieren:

Definition 6 ([MS04] Starke D-Sicherheit (*Strong D-Security*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$. Ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ ist genau dann *stark sicher für* D, wenn $\vec{V} \cong_D \vec{V}$.

Ein Programm ist sicher, wenn es für Angreifer mit beliebiger Sicherheitsdomäne sicher ist:

Definition 7 ([MS04] Starke Sicherheit (*Strong Security*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge MWL-Anweisungen. Ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ ist genau dann *stark sicher*, wenn er für alle $D \in \mathcal{D}$ stark sicher ist.

Die Definitionen gelten auch für den Sonderfall der binären Flusspolitik. Da $=_{high}$ die Identitätsrelation über den Zuständen ist, ist die starke *high*-Sicherheit immer erfüllt. Daher folgt aus der starken *low*-Sicherheit direkt die starke Sicherheit.

[7] Allerdings wird in [Pöp05] die zweite Bedingung nicht formal definiert, sondern es wird bei der Einführung von Arrayausdrücken die Aussage gemacht, dass die Länge von Arrays öffentlich ist.

In [Sab01] ist eine Sicherheitsbedingung definiert, die auch das Blocken und Entblocken der Semantik für MWL mit Synchronisationsanweisungen erfasst. Diese Sicherheitsbedingung ist für die binäre Flusspolitik definiert. Sie ist wie die starke Sicherheit aus [SS00] definiert, außer dass in der starken *low*-Bisimulation markierte Transitionen $\overset{\alpha}{\rightarrow}$ mit $\alpha \in \{\epsilon, \otimes Sem, \odot Sem\}$ nur durch identisch markierte Transitionen bisimulierbar sind. Damit deckt diese Bedingung die Annahme ab, dass ein Angreifer das Blocken oder Entblocken (oder äquivalent den Zustand der Warteschlangen) sehen kann.

Diese Sicherheitsbedingung wird zu einer Sicherheitsbedingung für MLS-Politiken verallgemeinert. Dabei ist ebenfalls der einzige Unterschied zur normalen starken Sicherheit (Definition 7), dass markierte Transitionen $\overset{\alpha}{\rightarrow}$ mit $\alpha \in \{\epsilon, \otimes Sem, \odot Sem\}$ nur durch identisch markierte Transitionen bisimulierbar sind.

Definition 8 (Starke D-Bisimulation (*Strong D-Bisimulation*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen mit Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Bisimulation* $\cong_D$ die Vereinigung aller symmetrischer Relationen R zwischen Anweisungsvektoren $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ der gleichen Größe ($\vec{V} = (C_1, \ldots, C_n)$ und $\vec{V'} = (C'_1, \ldots, C'_n)$), sodass

$$\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall \vec{W} : \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \vec{W}, t |\!\rangle$$
$$\Rightarrow \exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D t'$$

wobei $\alpha \in \{\epsilon, \otimes Sem, \odot Sem\}$. Sem ist ein Semaphorbezeichner.

Beispiel 1 (Verarbeitung eines Web-Formulars (Abbildung 8)). Dieses Beispiel basiert auf einem Beispiel aus [Sab01]. Das Beispiel zeigt zum einem, wie sich Threads mit Semaphoren sicher synchronisieren lassen. Zum anderen zeigt es ein *timing leak*, das die Sicherheitsbedingung verletzt.

Dieses und auch weitere Beispiele in dieser Arbeit enthalten Anweisungen, welche als Platzhalter für komplexere und aussagekräftigere Anweisungen dienen. In diesem Beispiel steht `record := forminput` mit der beabsichtigten Bedeutung, dass aus einer neuen Formulareingabe ein Datensatz generiert wird. Außerdem steht `result == 0` statt `result.org == military` (wie in [Sab01]), um zu überprüfen, ob die Eingabe vom Militär kommt. Diese Platzhalter sind so gewählt, dass sie am Informationsfluss nichts ändern. Die beabsichtigte Bedeutung geht jeweils aus einem Kommentar hervor. Der Grund für diese Vereinfachung ist, dass damit der EIFA diese Programme verarbeiten und überprüfen kann.

Hier sei nochmal auf die Einfärbung der Bezeichner hingewiesen, welche die Domänenzuweisung widerspiegelt (s. Abschnitt 1.4). Die Domänenzuweisung steht in der Beschreibung zur Abbildung 8.

Die Arbeit des Programms ist in zwei Threads aufgeteilt. Der erste nimmt die vertraulichen Daten entgegen und erstellt daraus einen Datensatz. Diesen Datensatz gibt er weiter an den zweiten Thread. Dieser verarbeitet die vertraulichen Daten,

```
signal( empty); // zu Beginn ist der Puffer leer
 counter := 0;
 milcounter := 0;
fork(
  // Thread zur Datengewinnung
  while true do
    // Ersatzanweisung zur Erstellung eines neuen Datensatzes:
     record := forminput;
    wait( empty);
     buf := record;
    signal( full)
  end
,
  // Thread zur Datenverarbeitung
  while true do
    wait( full);
     result := buf;
    signal( empty);
    // unsichere Verzweigung, Laufzeiten der
    // Zweige unterschiedlich
    if (result == 0) then // Ersatz fuer Test auf militaerisch
       milcounter := milcounter + 1
    end;
     counter := counter + 1
  end
)
```

Abbildung 8: Verarbeitung eines Webformulars mit mehreren Threads (Beispiel 1). Die Semaphorenbezeichner full und empty, sowie der Variablenbezeichner counter gehören zur Sicherheitsdomäne *low*. Alle anderen Bezeichner gehören zur Sicherheitsdomäne *high*. Das Beispiel ist nicht stark sicher.

zählt aber öffentlich die Datensätze. Zwei Semaphore synchronisieren die Threads: full und empty. Ersteres bedeutet, dass der erste Thread einen neuen Datensatz bereitstellt, letzteres, dass der zweite Thread angefangen hat, einen bereitgestellten Datensatz zu verarbeiten.

Dieses Programm ist nicht stark sicher, da die Verzweigung mit der Bedingung result == 0 einen vertraulichen Variablenbezeichner prüft und die Laufzeit der Zweige unterschiedlich ist. Das heißt die Zahl der militärischen Eingaben, die vertraulich sein sollte, ergibt sich aus der Differenz der tatsächlichen Laufzeit und der Laufzeit, die das Programm ohne eine einzige militärische Eingabe hätte. Wenn diese Verzweigung im **else** -Zweig eine Skip-Anweisung stehen hätte, wäre das Programm stark sicher.

Als Sicherheitsbedingung, die auf der *intransitive noninterference* basiert, wird wieder eine Definitionen aus [MS04] übernommen. Die drei Definitionen folgen dem gleichen Schema, wie die Definitionen für die normale starke Sicherheit.

Definition 9 ([MS04] Starke D-Bisimulation[in] (für *intransitive noninterference*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen ohne Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Bisimulation[in]* $\approx_D^{in}$ die Vereinigung aller symmetrischer Relationen R zwischen Anweisungsvektoren $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ der gleichen Größe ($\vec{V} = (C_1, \ldots, C_n)$ und $\vec{V'} = (C'_1, \ldots, C'_n)$), sodass

$$\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall \vec{W} :$$
$$\begin{bmatrix} \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \rightarrow_o \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow \exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \rightarrow_o \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D t' \end{bmatrix}$$
$$\wedge$$
$$\begin{bmatrix} \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow [\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \\ \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')] \end{bmatrix}$$

Definition 10 ([MS04] Starke D-Sicherheit[in] (für *intransitive noninterference*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$. Dann ist ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ genau dann stark sicher[in] für D, wenn $\vec{V} \approx_D^{in} \vec{V}$.

Definition 11 ([MS04] Starke Sicherheit[in] (für *intransitive noninterference*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Dann ist ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ genau dann stark sicher[in], wenn er für alle $D \in \mathcal{D}$ stark sicher[in] ist.

Genau wie bei der Definition für MWL mit Synchronisationsanweisungen unterscheiden sich hier die Bedingungen für unmarkierte von markierten Transitionen. Doch hier müssen nach Ausführung markierter Transitionen die resultierenden Zustände t und t' nur dann D-gleich sein, wenn mindestens eine von drei Bedingungen zutrifft:

- $D_1 \not\rightsquigarrow D_2$: wenn die Flusspolitik zwischen den Sicherheitsdomänen, zwischen denen die Anweisung deklassifiziert, keine Deklassifikation erlaubt,
- $D_2 \not\leq D$: wenn der Angreifer mit der Sicherheitsdomäne D das Ziel der Deklassifikation nicht sehen kann oder
- $s =_{D_1} s'$: wenn sich die Zustände aus der Sicht der Quellsicherheitsdomäne nicht unterscheiden.

Die starke Sicherheit[in] kann auch wieder für MWL mit Synchronisationsanweisungen definiert werden. Dabei wird wieder verlangt, dass der Bisimulationsschritt einer Transition $\overset{\odot Sem}{\rightarrow}$, $\overset{\otimes Sem}{\rightarrow}$ oder $\rightarrow_o$ der jeweils gleiche Transitionsrelation angehört.

Definition 12 (Starke D-Bisimulation[in] (für *intransitive noninterference*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen mit Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Bisimulation[in]* $\approx_D^{in}$ die Vereinigung aller symmetrischer Relationen R zwischen Anweisungsvektoren $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ der gleichen Größe ($\vec{V} = (C_1, \ldots, C_n)$ und $\vec{V'} = (C'_1, \ldots, C'_n)$), sodass

$$
\begin{aligned}
&\forall s, s', t : \forall i \in \{1 \ldots n\} : \forall \vec{W} : \\
&\left[\begin{array}{c} \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \rightarrow_o \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow \exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \rightarrow_o \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D t' \end{array}\right] \\
&\wedge \\
&\left[\begin{array}{c} \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow \exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \overset{\alpha}{\rightarrow} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D t' \end{array}\right] \\
&\wedge \\
&\left[\begin{array}{l} \vec{V} R \vec{V'} \wedge s =_D s' \wedge \langle\!| C_i, s |\!\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow [\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \rightarrow_d^{D_1 \rightarrow D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \\ \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')] \end{array}\right]
\end{aligned}
$$

wobei $\alpha \in \{\otimes Sem, \odot Sem\}$. Sem ist ein Semaphorbezeichner.

Die Sicherheitsbedingungen sind genauso wie in den Definition 10 und 11 definiert. Folgende beiden Sätze helfen bei der Implementierung des EIFA.

Satz 1 ([MS04]). *Sei Γ eine Domänenzuweisung. Ein MWL-Programm C ohne Deklassifikationsanweisung ist genau dann mit der MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ stark sicher[in], wenn es mit der MLS-Politik $(\mathcal{D}, \leq)$ stark sicher ist.*

Beweis: Für MWL ohne Synchronisationsanweisungen wird dieser Satz in [MS04] bewiesen. Er gilt auch für MWL mit Synchronisationsanweisungen, da die Bedingungen für Synchronisationstransitionen in Definition 12 und Definition 8 äquivalent sind.
□

Satz 2. *Sei Γ eine Domänenzuweisung.*

- *Ein MWL-Programm C ohne Synchronisationsanweisung ist genau dann mit der MLS-Politik $(\mathcal{D}, \leq)$ stark sicher basierend auf Definition 5, wenn es stark sicher basierend auf Definition 8 ist.*
- *Ein MWL-Programm C ohne Synchronisationsanweisung ist genau dann mit der MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ stark sicher[in] basierend auf Definition 9, wenn es stark sicher[in] basierend auf Definition 12 ist.*

Beweis: Wenn es in C keine Synchronisationsanweisungen gibt, muss niemals die Bedingung für Synchronisationstransitionen in der Definition 8 bzw. 12 zutreffen. Die verbleibenden Bedingungen sind äquivalent zu den Bedingungen der Definition 5 bzw. 9. Da die entsprechenden Bisimulationen äquivalent sind, sind auch die darauf basierenden Sicherheitsbedingungen äquivalent.
□

Das bedeutet, wenn man die starke Sicherheit[in] für MWL mit Synchronisations- und Deklassifikationsanweisungen für ein Programm zeigt, zeigt man gleichzeitig auch die anderen Sicherheitsbedingungen, falls in dem Programm keine Synchronisations- bzw. Deklassifikationsanweisungen vorhanden sind.

Ein weiteres Lemma und ein weiterer Satz betrifft ebenfalls den Zusammenhang zwischen den Sicherheitsbedingungen:

Lemma 3. *Seien eine Domänenzuweisung Γ, eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \emptyset)$ und eine Sicherheitsdomäne $D \in \mathcal{D}$ gegeben. Dann gilt $\approx_D^{in} \subseteq \approx_D$.*

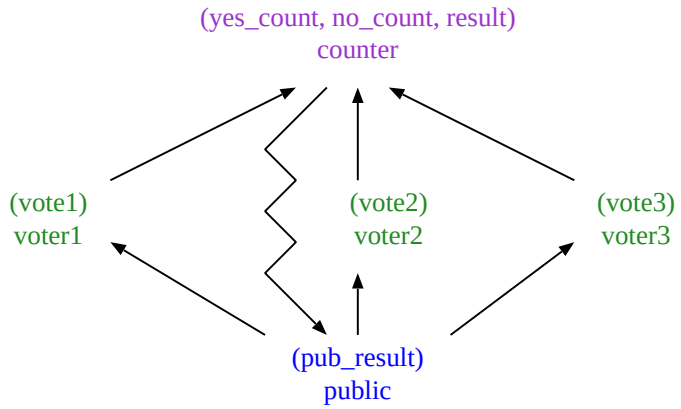
Beweis: Seien $\vec{V}, \vec{V}'$ beliebig mit $\vec{V} \approx_D^{in} \vec{V}'$ gegeben. Das heißt es gibt eine Relation R mit $\vec{V} R \vec{V}'$ und R erfüllt die Bedingung aus Definition 9. Da $\forall D_1, D_2 : D_1 \not\rightsquigarrow D_2$, ist die Bedingung für Deklassifikationstransitionen gleich der für normale Transitionen. Damit erfüllt R auch die Bedingung aus der Definition 5. Damit gilt $\vec{V} \approx_D \vec{V}'$
□

Satz 4. *Seien eine Domänenzuweisung Γ und eine MLS-Politik $(\mathcal{D}, \leq, \emptyset)$ gegeben. Dann gilt: C ist stark sicher[in] $\Rightarrow$ C ist stark sicher.*

Beweis: Der Satz folgt aus Lemma 3.
□

Hier stehen Beispiele verschiedener MWL-Programme und MLS-Politiken, um die starke Sicherheit[in] zu veranschaulichen.

Beispiel 2 (Auszählung einer Wahl (Abbildung 9)**.** Dieses Beispiel nutzt eine MLS-Politik mit Ausnahmen (s. Abbildung 9(a)). Es gibt drei Wähler (modelliert durch die Sicherheitdomänen *voter*1-3) deren Stimmen (modelliert durch die Bezeichner vote1, vote2, vote3) das Programm auszählen soll. Jeder kann „ja“ oder „nein“ stimmen. Das Ergebnis soll der Mehrheit der Stimmen entsprechen und die Wähler sollen das Ergebnis erfahren. Außer dem Ergebnis sollen die Wähler nichts über die anderen Wähler erfahren.



(a) MLS-Politik mit Ausnahme

```
yes_count := 0;
no_count := 0;
// das Zaehlen der Stimmen:
if vote1 then
   yes_count := yes_count + 1
else
   no_count := no_count + 1
end;
if vote2 then
   yes_count := yes_count + 1
else
   no_count := no_count + 1
end;
if vote3 then
   yes_count := yes_count + 1
else
   no_count := no_count + 1
end;
// die Bestimmung des Ergebnisses:
if (yes_count > 1) then
   result := true
else
   result := false
end;
// die Veroeffentlichung des Ergebnisses:
[pub_result := result]
```

(b) MWL-Programm

Abbildung 9: Programm zur Mehrheitsauszählung einer Wahl mit drei Teilnehmern und zwei Wahlmöglichkeiten (Beispiel 2). Die drei Wähler sollen das Ergebnis erfahren, sonst aber keine Information über die anderen Stimmen erhalten. Die Flusspolitik und die Domänenzuweisung der Bezeichner sind in 9(a) dargestellt. Die Bezeichner stehen in Klammern an der jeweiligen Sicherheitsdomäne. Dieses Beispiel ist stark sicher[in].

Das Programm in Abbildung 9(b) ist stark sicher[in] mit der gegebenen MLS-Politik. Es stellt sich die Frage, welchen Nutzen diese Tatsache hat. Es ist nicht direkt aus der MLS-Politik ersichtlich, dass die Wähler außer dem Ergebnis nichts über die anderen Wähler erfahren. Das Programm result:= vote1;[pub_result:= result] erfüllt genauso die Sicherheitsbedingung, verletzt aber die fachliche Anforderung.

Die Lokalisation der Deklassifikation kann nur dabei helfen zu zeigen, dass die fachlichen Sicherheitsanforderungen erfüllt sind. Aus der MLS-Politik ist ersichtlich, dass alle Information, die von einem zu einem anderen Wähler fließt, die einzige Ausnahmeflussrelation passieren muss. Die einzige Deklassifikation findet in der letzten Anweisung entlang der Ausnahmeflussrelation statt. Also muss alle Information zwischen zwei Wählern durch Ausführung dieser Anweisung fließen. Um allerdings die Erfüllung der Anforderung zu beweisen, muss man zeigen, dass der hier deklassifizierte Variablenbezeichner result tatsächlich das Ergebnis enthält, und nicht etwa die Stimme eines einzelnen Wählers.

Im Abschnitt 4 wird eine Sicherheitsbedingung definiert, mit der sich direkt die fachliche Sicherheitsanforderung dieses Beispiels durchsetzen lässt.

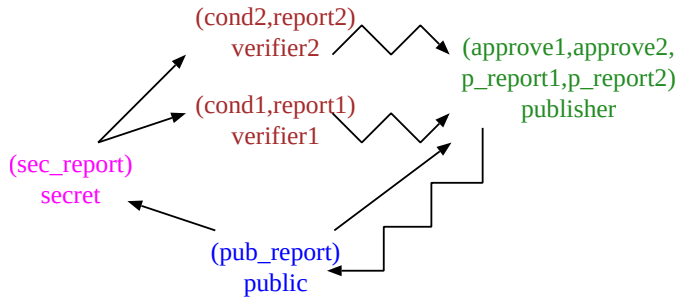
Beispiel 3 (Freigabe nach zweifacher Überprüfung (Abbildung 10)). Dieses Beispiel zeigt, wie man mit *intransitive noninterference* ein schrittweisen Informationsfluss modellieren kann. Das Programm aus Abbildung 10(b) modelliert folgenden Ablauf: zwei Prüfer lesen einen vertraulichen Bericht. Falls beider Prüfer die Freigabe erlauben, veröffentlicht ein Herausgeber den Bericht.

Die MLS-Politik (Abbildung 10(a)) ist so konstruiert, dass der vertrauliche Bericht (Variablenbezeichner sec_report) aus der Sicherheitsdomäne *secret* über einen der Prüfer (Sicherheitsdomänen $verifier1$ und $verifier2$) und dem Herausgeber (Sicherheitsdomäne *publisher*) an die Öffentlichkeit (Sicherheitsdomäne *public*) gelangen kann, aber nicht direkt. Mit einer transitiven Flusspolitik wäre das nicht möglich.

Das Programm ist stark sicher[in]. Besonders zu beachten ist auch, wie die bedingte Deklassifikation realisiert ist. Naheliegend wäre es zum Beispiel statt der ersten If-Anweisung mit anschließender Deklassifikation folgendes zu schreiben:

```
if cond1 then
  [ p_report1 := report1 ]
else
  skip
end
```

Dieses Programm ist aber nicht stark sicher[in], da es zum Beispiel nicht stark *public*-sicher[in] ist. Die Deklassifikationsanweisung ist zur Skip-Anweisung nicht stark *pulic*-bisimular[in]. Das Auftreten einer Deklassifikation ist immer für alle Sicherheitsdomänen sichtbar. Alternativ könnte man im **else** -Zweig, einen neuen, bedeutungslosen Variablenbezeichner dummy1 aus der gleichen Sicherheitsdomäne wie report1 deklassifizieren: [p_report1 := dummy1] Dieses Programmstück, sowie das gesamte Programm entsprechend abgewandelt, sind stark sicher[in].



(a) MLS-Politik mit Ausnahmen

```
// Pruefer 1 liest den Bericht
 report1 := sec_report;
// Pruefer 1 leitet den Bericht weiter, falls seine
// Bedingung zutrifft
if cond1 then
  skip
else
   report1 := 0
end;
[ p_report1 := report1 ];
// Pruefer 1 teilt dem Herausgeber seine Freigabeentscheidung mit
[ approve1 := cond1 ];

// Pruefer 2 liest den Bericht
 report2 := sec_report;
// Pruefer 2 leitet den Bericht weiter, falls seine
// Bedingung zutrifft
if cond2 then
  skip
else
   report2 := 0
end;
[ p_report2 := report2 ];
// Pruefer 2 teilt dem Herausgeber seine Freigabeentscheidung mit
[ approve2 := cond2 ];

// Herausgeber veroeffentlicht den Bericht, falls erlaubt
if( approve1 && approve2 && ( p_report1 == p_report2 ) ) then
  skip
else
   p_report1 := 0
end;
[ pub_report := p_report1 ]
```

(b) MWL-Programm

Abbildung 10: Freigabe nach zweifacher Überprüfung (Beispiel 3). Zwei Prüfer lesen einen vertraulichen Report, den ein Herausgeber nach Freigabe durch die Prüfer veröffentlicht. 10(a) stellt die Flusspolitik und die Domänenzuweisung der Bezeichner dar. Die Bezeichner stehen in Klammern an der jeweiligen Sicherheitsdomäne. Dieses Beispiel ist stark sicher[in].

2.4. Sicherheitstypsysteme

In diesen Abschnitt geht es darum, wie sich automatisch die Sicherheit von MWL-Programmen zeigen lässt. Dazu werden sogenannte Sicherheitstypsysteme vorgestellt. Ein Sicherheitstypsystem ist eine Menge von Regeln. Sie legen fest, ob Ausdrücken und Programmen Typen zugeordnet werden, und wenn ja welche. Der EIFA benutzt mehrere Sicherheitstypsysteme, von denen eines hier definiert wird.

Sicherheitstypsystem für starke Sicherheit[in] Mit dem hier vorgestellten Sicherheitstypsystem kann man die starke Sicherheit[in] von MWL-Programmen zeigen. Es basiert auf dem Sicherheitstypsystem aus [MS04] und wird um Regeln für Arrayausdrücke und -anweisungen, sowie Deklarationsanweisungen ergänzt. Diese sind an Regeln aus einem Sicherheitstypsystem aus [Pöp05] angelehnt. Die Regeln für Ausdrücke sind in Abbildung 11 aufgelistet, die für Anweisungen in Abbildung 12.

Zusätzlich wird dieses Sicherheitstypsystem um Regeln für Synchronisationsanweisungen erweitert (s. Abbildung 13). Diese Regeln sind aus [Sab01] übernommen.

Die weiteren im EIFA implementierten Sicherheitstypsysteme befinden sich im Anhang B.2.

$$\textbf{[Const]}\quad \frac{}{\Gamma \vdash Const : D} \qquad \textbf{[Var]}\quad \frac{\Gamma(Id) = D}{\Gamma \vdash Id : D}$$

$$\textbf{[Op]}\quad \frac{\Gamma \vdash Exp_1 : D_1 \quad \Gamma \vdash Exp_2 : D_2 \quad D_1 \leq D \quad D_2 \leq D}{\Gamma \vdash Exp_1\ op\ Exp_2 : D}$$

$$\textbf{[ArrLen]}\quad \frac{}{\Gamma \vdash Arr\ .\mathsf{length} : D}$$

$$\textbf{[ArrExp]}\quad \frac{\Gamma \vdash Exp : D_1 \quad \Gamma \vdash Arr : D_2 \quad D_1 \leq D \quad D_2 \leq D}{\Gamma \vdash Arr\ [\ Exp\ ] : D}$$

Abbildung 11: Sicherheitstypregeln für Ausdrücke unter einer MLS-Politik. Γ ist eine Domänenzuweisung.

Hauptaussage Das Sicherheitstypsystem ordnet Ausdrücken Sicherheitsdomänen als Typ zu ($\Gamma \vdash Exp : D$), während Programme einfach nur typisierbar sind oder nicht ($\Gamma \vdash \vec{C}$ heißt „$\vec{C}$ ist typisierbar“). Folgender Satz für das Sicherheitstypsystem mit allen Regeln dieses Abschnitts zeigt die Bedeutung von Sicherheitstypsystemen:

Satz 5 (Sicherheit der Analyse). *Seien eine Flusspolitik und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Es gilt für ein Programm $\vec{C} \in \vec{\mathcal{C}}$:*

$$\Gamma \vdash \vec{C} \quad \Rightarrow \quad \vec{C} \textit{ ist stark sicher[in].}$$

[VarDecl] $$\frac{}{\Gamma \vdash Id{:}Type{:}D}$$

[ArrDecl] $$\frac{\Gamma \vdash Exp : low}{\Gamma \vdash Arr{:}Type[Exp]{:}D}$$

[Skip] $$\frac{}{\Gamma \vdash \mathsf{skip}}$$

[Assign] $$\frac{\Gamma \vdash Exp : D \qquad D \leq \Gamma(Id)}{\Gamma \vdash Id{:=}Exp}$$

[ArrAssign] $$\frac{\Gamma \vdash Exp_1 : D_1 \quad \Gamma \vdash Exp_2 : D_2 \quad D_1 \leq \Gamma(Arr) \quad D_2 \leq \Gamma(Arr)}{\Gamma \vdash Arr\ [\ Exp_1\ ]{:=}\ Exp_2}$$

[Declass] $$\frac{\Gamma(Id') \rightsquigarrow \Gamma(Id)}{\Gamma \vdash [Id{:=}Id']}$$

[While$_{low}$] $$\frac{\Gamma \vdash B : low \qquad \Gamma \vdash C}{\Gamma \vdash \mathsf{while}\ B\ \mathsf{do}\ C}$$

[Seq] $$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1\,;C_2}$$

[Fork] $$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash \vec{C_2}}{\Gamma \vdash \mathsf{fork}(C_1\vec{C_2})}$$

[Par] $$\frac{\Gamma \vdash C_0 \quad \ldots \quad \Gamma \vdash C_{n-1}}{\Gamma \vdash (C_0, \ldots C_{n-1})}$$

[If] $$\frac{\Gamma \vdash B : D \quad \Gamma \vdash C_1 \quad \Gamma \vdash C_2 \quad \forall D' \not\geq D : C_1 \approx_{D'} C_2}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2}$$

Abbildung 12: Sicherheitstypregeln für die starke Sicherheit[in] von MWL-Anweisungen. Die Typregeln für Ausdrücke sind in der Abbildung 11 definiert. Γ ist eine Domänenzuweisung. In Implementierungen ersetzt die Bedingung $C_1(R_D)C_2$ die semantische Nebenbedingung $\forall D' \not\geq D : C_1 \approx_{D'} C_2$, wobei $\{R'_D\}_{D' \in \mathcal{D}}$ eine sichere Approximationsrelation ist (s. Definition 13).

[SemDecl] $\dfrac{}{\Gamma \vdash Sem{:}Type{:}D}$

[Wait] $\dfrac{\Gamma(Sem) = low}{\Gamma \vdash \mathsf{wait}(Sem)}$

[Signal] $\dfrac{\Gamma(Sem) = low}{\Gamma \vdash \mathsf{signal}(Sem)}$

Abbildung 13: Sicherheitstypregeln für die starke Sicherheit[in] von Synchronisationsanweisungen. Diese Typregeln erweitern das Sicherheitstypsystem aus der Abbildung 12. Γ ist eine Domänenzuweisung.

Beweis: Der Beweis steht im Anhang C. Er entspricht zum größten Teil dem Beweis dieses Satzes aus [MS04] für MWL ohne Arrays, Deklarations- und Synchronisationsanweisungen. □

Es ist zu beachten, dass die Umkehrung des Satzes im allgemeinen nicht gilt.

Wenn man die Regeln [Signal] und [Wait], bzw. [Declass] weglässt, sind nur noch Programme ohne Synchronisationsanweisung bzw. ohne Deklassifikationsanweisung typisierbar. Nach den Sätze 1 und 2 kann man auf diese Weise mit dem Sicherheitstypsystem auch die anderen Sicherheitsbedingungen aus Abschnitt 2.3 beweisen.

Eigenschaften der Sicherheitstypregeln für Ausdrücke Folgendes Lemma formalisiert die Bedeutung der Sicherheitstypregeln für Ausdrücke:

Lemma 6 ([MS04]). *Falls* $\Gamma \vdash Exp : D$, *dann gilt* $\forall s =_D s' \Rightarrow \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$.

Beweis: Seien $s =_D s'$ und $\Gamma \vdash Exp : D$ beliebig. Der Beweis erfolgt durch Induktion über den Aufbau der Ausdrücke.

$Exp = Const$: Nach der Semantik gilt $\langle\!| Const, s |\!\rangle \downarrow Const \wedge \langle\!| Const, s' |\!\rangle \downarrow Const$.

$Exp = Arr.\mathsf{length}$:
$\langle\!| Arr.\mathsf{length}, s |\!\rangle \approx \langle\!| Arr.\mathsf{length}, s' |\!\rangle$ nach der Definition von $=_D$.

$Exp = Id$: Nach der Sicherheitstypregel gilt $\Gamma(Id) = D$. Nach der Definition von $=_D$ gilt damit $\langle\!| Id, s |\!\rangle \approx \langle\!| Id, s' |\!\rangle$.

$Exp = Exp_1\ Op\ Exp_2$: Nach der Sicherheitstypregel gibt es $D_1, D_2 \leq D$ mit $\Gamma \vdash Exp_1 : D_1$ und $\Gamma \vdash Exp_2 : D_2$. Nach der Definition 4 gilt mit $s =_D s'$ auch $s =_{D_1} s'$ und $s =_{D_2} s'$. Nach Induktionsvoraussetzung gilt damit $\langle\!| Exp_1, s |\!\rangle \approx \langle\!| Exp_1, s' |\!\rangle$ und $\langle\!| Exp_2, s |\!\rangle \approx \langle\!| Exp_2, s' |\!\rangle$. Nach der Semantik ist die Auswertung von Exp eine Funktion der Auswertungen von Exp_1 und Exp_2. Damit folgt $\langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$.

$Exp = Arr[Exp']$: Analog zum vorhergehenden Fall.

□

Eine weitere Aussage hilft bei der Implementierung des Sicherheitstypsystems.

Proposition 7. *Seien eine MLS-Politik* $(\mathcal{D}, \leq)$*, eine Domänenzuweisung* Γ *und ein* $D \in \mathcal{D}$ *gegeben. Es gilt* $\exists D' \leq D : \Gamma \vdash Exp : D'$ *genau dann, wenn für alle atomaren Teilausdrücke* Exp_{atom} *von* Exp *(Arraylängen, Konstanten, Bezeichner)* $\exists D'' \leq D : \Gamma \vdash Exp_{atom} : D''$ *gilt.*

Beweis: Der Beweis lässt sich durch Induktion über den Aufbau der Ausdrücke führen. Für atomare Ausdrücke ist die Aussage in beide Richtungen erfüllt, da der einzige enthaltene atomare Ausdruck der Ausdruck selbst ist.

Seien ein $D \in \mathcal{D}$ und ein Ausdruck $Exp = Exp_1\ Op\ Exp_2$ (analog $Exp = Arr[Exp]$) gegeben. Dann sind die atomaren Teilausdrücke von Exp genau die atomare Teilausdrücke von Exp_1 oder von Exp_2. Es werden nacheinander beide Richtungen gezeigt:

$\Rightarrow$ Es gelte $\Gamma \vdash Exp : D'$ für ein $D' \leq D$. Nach der Sicherheitstypregel gibt es $D_1, D_2 \leq D'$ mit $\Gamma \vdash Exp_1 : D_1$ und $\Gamma \vdash Exp_2 : D_2$. Nach Induktionsvoraussetzung gilt für alle atomaren Teilausdrücke Exp_{atom} in Exp_1 oder Exp_2: $\exists D'' \leq D' : \Gamma \vdash Exp_{atom} : D''$. Da $D' \leq D$ und da dies genau die atomaren Teilausdrücke von Exp sind, ist diese Richtung bewiesen

$\Leftarrow$ Es gelte $\exists D'' \leq D : \Gamma \vdash Exp_{atom} : D''$ für alle atomaren Teilausdrücke von Exp. Dann gilt auch $D'' \leq D$ für alle atomaren Teilausdrücke von Exp_1 und von Exp_2. Nach Induktionsvoraussetzung gibt es $D_1, D_2 \leq D$ mit $\Gamma \vdash Exp_1 : D_1$ und $\Gamma \vdash Exp_2 : D_2$ Nach der Typregel für Ausdrücke gilt $\Gamma \vdash Exp : D$. Mit $D \leq D$ ist diese Richtung bewiesen.

□

Damit kann man die Sicherheitsdomänen aller atomaren Teilausdrücke eines Ausdrucks Exp sammeln und mit einer Sicherheitsdomäne D vergleichen, um $\exists D' \leq D : \Gamma \vdash Exp : D'$ zu zeigen.

Sichere Approximationsrelation Das Sicherheitstypsystem aus Abbildung 12 ist genaugenommen noch kein richtiges Typsystem, denn die Regel [If] hat eine semantische Nebenbedingung. Um das Sicherheitstypsystem anwendbar zu machen, ersetzt man diese semantische Nebenbedingung durch eine sogenannte sichere Approximationsrelation. Diese ist wie folgt definiert:

Definition 13 ([MS04] Sichere Approximationsrelation (*Safe Approximation Relation*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Eine Familie $\{R_D\}_{D \in \mathcal{D}}$ von Relationen über Anweisungen ist dann eine *sichere Approximationrelation*, wenn für beliebige, stark sichere[in] $C, C' \in \mathcal{C}, D \in \mathcal{D}$ gilt:

$$C(R_D)C' \Rightarrow \forall D' \not\geq D : C \cong_{D'} C'.$$

$\forall D' \not\geq D : C \cong_{D'} C'$ bedeutet: aus der Sicht aller Sicherheitsdomänen D', die D nicht sehen können, verhalten sich die beiden Programme C und C' ununterscheidbar. Ein Beispiel für eine sichere Approximationsrelation ist folgende (aus [MS04], hier um die Regel für Arrayzuweisungen ergänzt):

Definition 14 ([MS04] Nicht D-sichtbare Gleichheit (*Non D-Visible Equality*)). Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Die *nicht D-sichtbare Gleichheit* $\{\sim_D\}_{D\in\mathcal{D}}$ ist folgendermaßen definiert: Für jedes D ist $\sim_D$ die kleinste Kongruenzrelation über Anweisungen (transitiv, reflexiv, symmetrisch und geschlossen unter der Konstruktion der Sprache), welche die folgenden Regeln erfüllt:

$$\frac{\Gamma(Id) \geq D}{Id{:=}Exp \sim_D \textsf{skip}} \qquad \frac{\Gamma(Arr) \geq D}{Arr[Exp_1]{:=}Exp_2 \sim_D \textsf{skip}}$$

Satz 8 ([MS04]). *Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Die Relationsfamilie $\{\sim_D\}_{D\in\mathcal{D}}$ ist eine sichere Approximationsrelation.*

Beweis: Der Beweis des Satzes steht im Anhang C. □

Im Abschnitt 4.3.1 wird eine ähnliche Approximationsrelation für eine neue Sicherheitsbedingung definiert.

Transformierende Sicherheitstypsysteme Es gibt auch Sicherheitstypsysteme, die Anweisungen einen Typ zuweisen. Eines ist im EIFA implementiert, ein sogenanntes transformierendes Sicherheitstypsystem. Der Typ einer Anweisung enthält eine Transformation der Anweisung, die sich bis auf Verzögerungen genauso verhält, wie die Anweisung selbst. Dabei werden manche Laufzeitlecks entfernt. Das heißt, hier ist nicht die typisierte Anweisung stark sicher, sondern die Transformation. Die Details sind für diese Arbeit nicht relevant. Zur Abgrenzung nennt man die bisherigen Sicherheitstypsysteme auch „nichttransformierende (*non-transforming*)“ Sicherheitstypsysteme. Ein im EIFA implementiertes transformierendes Sicherheitstypsystem ist in der Abbildung 39 im Anhang B.2 auflistet.

3. Erweiterung des EIFAs

Die Sicherheitstypsysteme aus dem vorherigen Abschnitt ergeben mit $\Gamma \vdash C$ eine algorithmisch entscheidbare Eigenschaft von MWL-Programmen, aus der die Sicherheit der MWL-Programme folgt. Um die Eigenschaften der Sicherheitstypsysteme und der Sicherheitsbedingungen zu untersuchen und zu demonstrieren, wurde im Rahmen von [Pöp05] eine Anwendung entwickelt. Sie wird hier *Experimental Information Flow Analyzer* (kurz EIFA) in der Version 1 genannt. Im EIFA Version 1 sind zwei Sicherheitstypsysteme für die starke Sicherheit mit einer binären Sicherheitspolitik implementiert, ein nichttransformierendes und ein transformierendes (s. Abbildungen 38 und 39 im Anhang). Im Rahmen der vorliegenden Arbeit wurde der EIFAv1 erweitert, indem weitere Sicherheitstypsysteme aus dem vorherigen Abschnitt implementiert wurden.

So wie MWL als einfache Beispielsprache zur Definition konkreter Sicherheitstypsystemen dient, kann der Entwurf des EIFA aufzeigen, welche Strukturen sich grundsätzlich für die Implementierung von Sicherheitstypsysteme eignen. Von besonderem Interesse ist dabei, wie änder- und erweiterbar der Entwurf des EIFA Version 1 ist. Es stellt sich heraus, dass der Entwurf die Änderung und Erweiterung von Sicherheitstypsystemen unterstützt, aber Änderungen an der Sprache erschwert.

Konventionen Bevor der Entwurf beschrieben wird, stehen hier einige Konventionen, denen dieser Abschnitt folgt.

„EIFA Version 1“ heißt der EIFA, wie er in [Pöp05] erstellt wurde. „EIFA Version 2“ heißt der EIFA, wie er in der vorliegenden Arbeit aus dem EIFA Version 1 entstanden ist. „EIFA Version 1“ wird mit „EIFAv1“ abgekürzt. „EIFA Version 2“ wird nur mit „EIFA“ abgekürzt.

MWL steht für MWL mit Deklarationen und Arrayanweisungen. Wenn MWL mit weiteren Erweiterungen gemeint ist, steht das explizit geschrieben.

Die Entwurfsbeschreibung des EIFAv1 und die der Entwurfsänderungen für den EIFA nutzt unter anderem einige Diagramme. Diese führen die Beziehungen zwischen Klassen auf. Die Notation der Diagramme orientiert sich an der UML-Notation:

- Durchgezogene Pfeile mit Dreieck als Spitze repräsentieren die „erbt von“-Beziehung zwischen Klassen.
- Unterbrochene Pfeile mit Dreieck als Spitze repräsentieren die „implementiert“-Beziehung zwischen einer Klasse und einer Schnittstelle.
- Durchgezogene Pfeile mit normaler Pfeilspitze repräsentieren die „ist assoziiert zu“-Beziehung. Dies umfasst hier auch die in UML mögliche Komposition und Aggregation.
- Boxen stehen für Klassen und Schnittstellen. Die Namen von abstrakten Klassen und Schnittstellen stehen im Gegensatz zu den Namen konkreter Klassen kursiv geschrieben.

3.1. Ursprüngliches System: EIFAv1

Dieser Abschnitt beschreibt den Entwurf des EIFAv1, wie er im Rahmen von [Pöp05] implementiert wurde. Aber hier wird zunächst kurz die Funktionalität beschrieben: Der Benutzer startet den EIFAv1 von einer Kommandozeilenschnittstelle und übergibt ihm dabei ein MWL-Programm mit einem Dateinamen als Parameter. Der EIFAv1 wendet zuerst das nichttransformierende Sicherheitstypsystem an. Wenn das MWL-Programm nicht typisierbar ist, wendet der EIFAv1 das transformierende Sicherheitstypsystem an. Eine exakte Verhaltensbeschreibung des EIFAv1 aus der Sicht des Benutzers findet sich im Anhang E.1.

Der EIFAv1 arbeitet in zwei oder drei Hauptschritten, die funktional voneinander unabhängig sind. Im ersten Schritt liest der EIFAv1 das zu überprüfende MWL-Programm ein und generiert aus der Textrepräsentation (Quelltext) des Programms eine Repräsentation in Form eines abstrakten Syntaxbaumes (*abstract syntax tree*, kurz AST). In dem zweiten Schritt überprüft der EIFAv1, ob das MWL-Programm in Form des AST sicher ist, das heißt, ob es gemäß der Sicherheitstypsysteme typisierbar ist. Falls der EIFAv1 das MWL-Programm mit dem transformierenden Sicherheitstypsystem transformiert hat, generiert der EIFAv1 in einem dritten Schritt aus dem transformierten Programm in AST-Repräsentation das transformierte Programm in Textrepräsentation und schreibt diese in eine Datei.

Entsprechend ist der EIFAv1 in vier Komponenten aufgeteilt. Klassenpakete repräsentieren diese Komponenten:

1. Paket ast : enthält Klassen, aus denen sich die Datenstruktur, ein AST, zusammensetzen lässt.

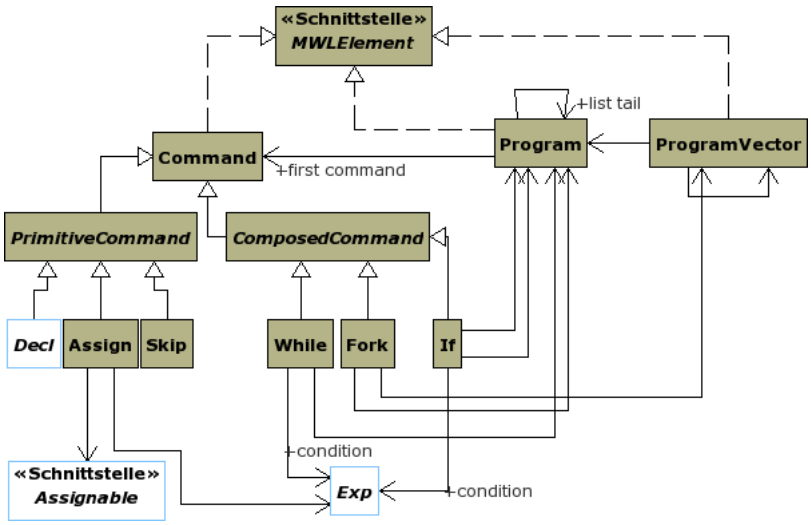
2. Paket mwlParser : enthält Klassen, welche die Funktionalität für den ersten Schritt implementieren, die Generierung eines AST aus der Textrepräsentation eines MWL-Programms

3. Paket visitor : enthält Klassen, welche die Funktionalität für den zweiten und dritten Schritt implementieren. Das heißt Klassen, deren implementierte Algorithmen auf einem AST arbeiten.

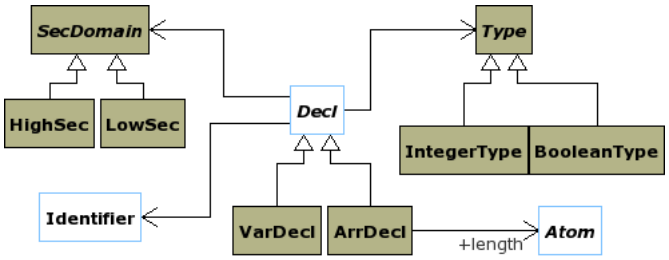
4. Paket checker : enthält Klassen, die den Programmablauf des EIFAv1 steuern

Das Paket ast, die Datenstruktur Die Klassenstruktur des Pakets ast stellt die Abbildung 14 dar. Zu beachten ist, dass die sequentielle Komposition von Anweisungen (C_1;C_2) in dieser Struktur nicht von der Klasse Command abgeleitet ist, obwohl Command MWL-Anweisungen repräsentiert. Die sequentielle Komposition ist durch die Klasse Program repräsentiert, welche eine verkettete Liste von Anweisungen (Command) implementiert.

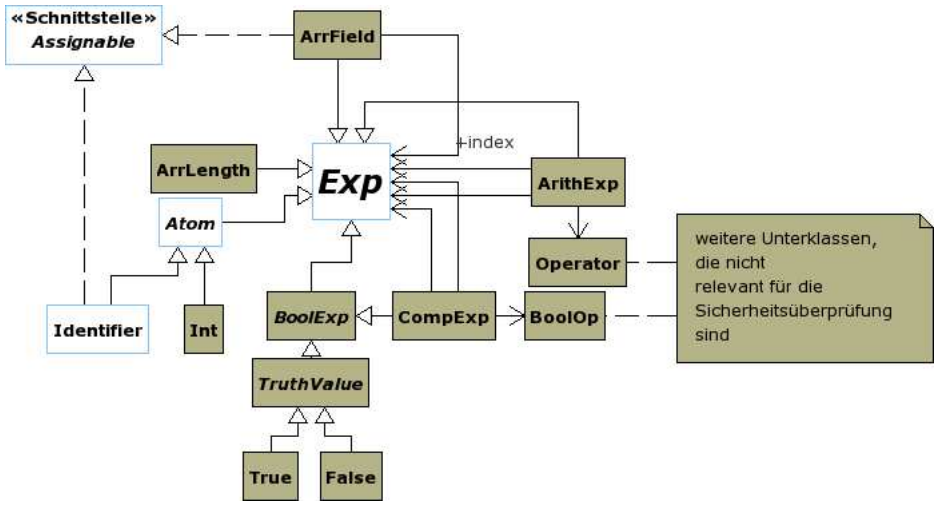
Das Besucher Entwurfsmuster Als einzige eigene Funktionalität stellen die Klassen des Pakets ast eine clone -Methode bereit, die eine Kopie des jeweiligen Objekts



(a) Klassen zur Repräsentation von Anweisung



(b) Klassen zur Repräsentation von Deklarationen



(c) Klassen zur Repräsentation von Ausdrücken

Abbildung 14: Klassenstruktur des Pakets `ast` im EIFAv1. Um die Übersichtlichkeit zu erhöhen, ist das Diagramm in drei Teile geteilt. Helle Boxen stehen für Klassen oder Schnittstellen, die in mehr als einem der drei Teile vorkommen. Rollennamen sind an die Assoziationen geschrieben, wenn sie die Verständlichkeit erhöhen ohne die Übersichtlichkeit zu verringern.

herstellt. Die assoziierten Objekte, also in diesem Fall die Objekte weiter unten im AST, kopiert sie dabei ebenfalls (tiefes Klonen, *deep cloning*).

Weitere Funktionalität lässt sich den Klassen des Pakets ast in Form des sogenannten Besucher-Entwurfsmusters (kurz Besuchermuster, *visitor design pattern*, s. [GHJV95]) hinzufügen. Das heißt, die Klassen des Pakets ast, auf die weitere Algorithmen zugreifen sollen, bieten eine Methode an, die einen sogenannten Besucher entgegennimmt. Diese Methode benachrichtigt den übergebenen Besucher, wie er das Objekt verarbeitet soll. Konkret implementieren die AST-Klassen die Java-Methode accept(visitor.Visitor) folgendermaßen:

```
Object accept(visitor.Visitor v) {
  return v.visit(this);
}
```

Dabei ist Visitor eine Schnittstelle des Pakets visitor, welche für jede dieser accept(visitor.Visitor) implementierenden Klassen eine Methode visit bereitstellt. Da von dem Objekt this die exakte Klasse bekannt ist, wird auch genau die visit-Methode für diese Klasse aufgerufen.

Um einen Algorithmus zu implementieren, der auf einem AST arbeitet, kann ein Programmierer die Schnittstelle Visitor implementieren (solche Implementierungen heißen im Besuchermuster „konkrete Besucher“ *concrete visitor*). Für weitere Details zum Einsatz des Besuchermusters im EIFAv1 s. [Pöp05].

Hier wird diese Art Algorithmen zu implementieren mit zwei Alternativen verglichen. So wird erläutert welche Vorteile es hat, das Besuchermusters im EIFAv1 (und EIFA) einzusetzen. Dass diese Vorteile tatsächlich zum Tragen kommen, stellt sich bei der Erweiterung des EIFAv1 zum EIFA heraus (s. Abschnitt 3.3).

Die eine Alternative ist, statt einem konkreten Besucher mit visit-Methoden für die AST-Klassen zu implementieren, in jeder der Klassen des AST eine Methode für den Algorithmus zu implementieren. Hierfür ist die Klassenstruktur einfacher, aber der Programmierer muss für jeden neuen Algorithmus jeder Klasse eine neue Methode hinzufügen. Außerdem ist die Implementierung des Algorithmus über die Klassen verteilt.

Die andere Alternative wäre es, den Algorithmus wie auch im Beobachtermuster außerhalb der Klassen des AST zu implementieren, ohne dass die Klassen des AST eine accept-Methode bereitstellen. Allerdings müssen Algorithmen die genaue Klasse eines Objekts kennen. Aber wenn sie auf das Objekt über eine Objektvariable der Oberklasse zugreifen, kennen sie nur die Oberklasse. Zum Beispiel ist im AST die Bedingung einer Schleife (Klasse While) ein Ausdruck (Klasse Exp). Ein Algorithmus, der ein Sicherheitstypsystem implementiert, muss den Ausdruck anders behandeln, je nachdem ob er zum Beispiel ein zusammengesetzter Ausdruck oder eine Konstante ist. Bei der Algorithmenimplementierung mit Methoden der AST-Klassen wählt die Laufzeitumgebung die richtige Methode durch Polymorphismus aus. Auch das Beobachtermuster nutzt Polymorphismus: durch Aufruf der accept-Methode. Diese Möglichkeit steht bei der zweiten Alternative nicht zur Verfügung. Hier müsste der Algorithmus durch explizite Abfragen der konkreten Klasse und

bedingte Verzweigungen sozusagen an den objektorientierten Mechanismen vorbei arbeiten.

Das Paket visitor, die Algorithmen Entsprechend dem Besuchermuster implementieren alle Klassen für Algorithmen die Schnittstelle Visitor des Pakets visitor. Die Vererbungsstruktur des Pakets visitor ist in Abbildung 15 dargestellt. NoTransformVisitor und TransformVisitor implementieren das nicht transformierende Sicherheitstypsystem bzw. das transformierende Sicherheitstypsystem. Beide benutzen ExpVisitor. Diese implementiert die Sicherheitstypregeln für Ausdrücke. ToStringVisitor generiert die Textrepräsentation eines MWL-Programms aus seiner AST-Repräsentation. Die abstrakte Klasse AbstractVisitor implementiert alle visit -Methoden als Methoden ohne Effekt und mit leerer Rückgabe (die leere Objektreferenz null). Damit muss nicht jeder konkrete Besucher jede visit -Methode selbst implementieren.

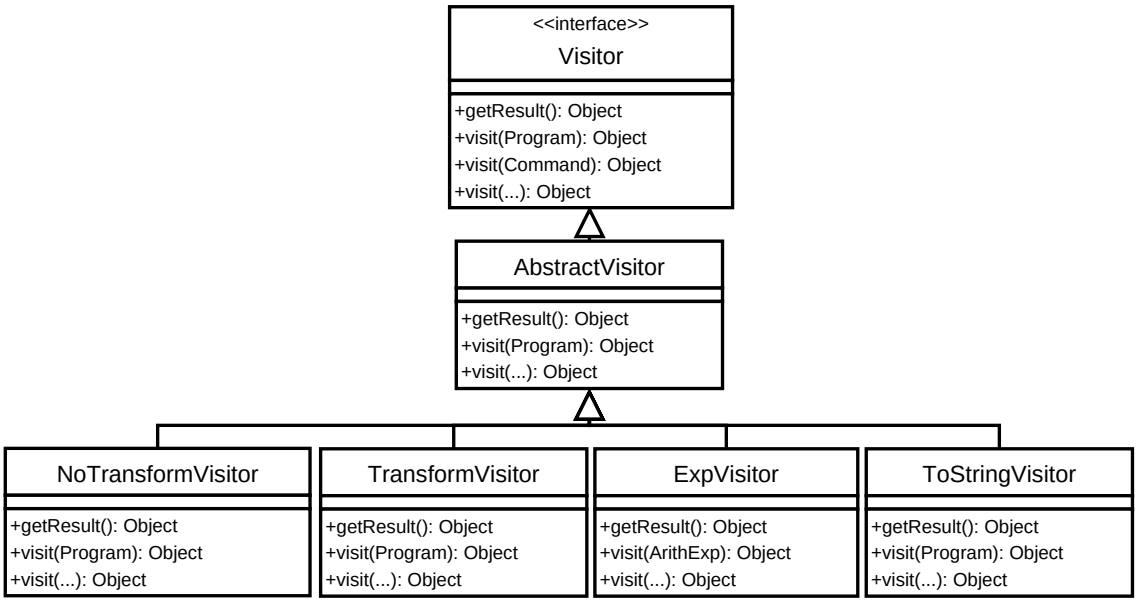


Abbildung 15: Vererbungsstruktur des Pakets visitor im EIFAv1 (übernommen aus [Pöp05])

Wie in der Abbildung zu sehen ist, stellen die Visitor implementierenden Klassen eine Methode getResult bereit. In ihrer Ausführung durchlaufen die Algorithmen den AST über accept - und visit -Aufrufe. Anschließend stellt das jeweilige Besucherobjekt über die getResult -Methode das Ergebnis des ausgeführten Algorithmus zu Verfügung.

Das Paket mwlParser, der Parser Im ersten Schritt seines Ablaufs generiert der EIFAv1 aus der Textrepräsentation eines MWL-Programms die AST-Repräsentation. Diese Funktion ist im Paket mwlParser in zwei Teilschritten implementiert. Zunächst

spaltet eine lexikalische Analyse die Textrepräsentation des MWL-Programms in Token (syntaktische Einheiten) auf. Anschließend generiert eine syntaktische Analyse aus dieser Folge von Token einen AST (s. Abbildung 16).

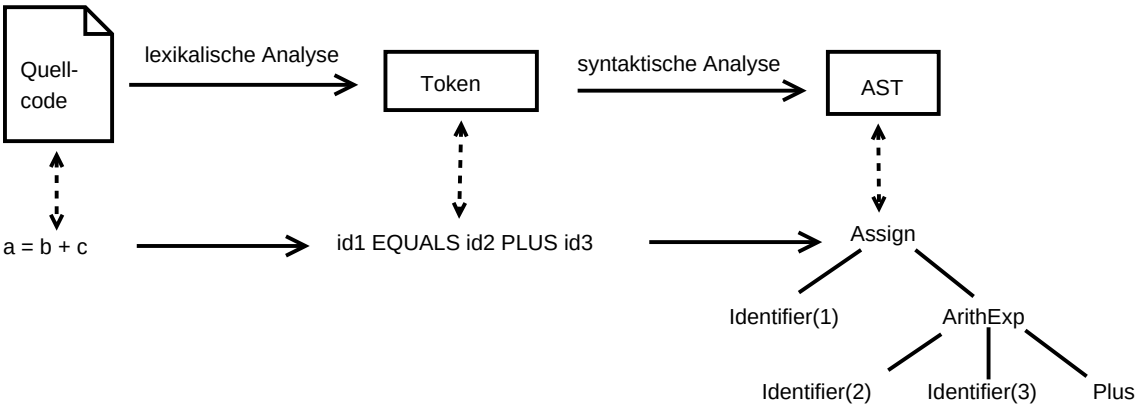


Abbildung 16: Ablauf der Generierung des AST aus dem Quellcode (übernommen aus [Pöp05])

Diese beiden Funktionen stellen die Klassen MWLScanner und MWLParser bereit. Die Quelltexte der beiden Klassen generieren die Werkzeuge JLex und Java CUP [8] aus sogenannten Grammatikdateien.

Das Paket checker, die Steuerung Die Klassen dieses Pakets steuern die Funktionen der Pakete mwlParser und visitor. Die Klasse Analyzer ermöglicht es dem Benutzer den EIFAv1 von der Kommandozeileschnittstelle zu starten, indem sie die main -Methode implementiert. Die Methoden der Klasse Checker nutzen die Funktionen der Pakete mwlParser und visitor. Sie stellen die Eingabedaten zur Verfügung, nehmen die Ergebnisse entgegen und leiten diese weiter. Die beiden Unterklassen CheckerNoTransform und CheckerTransform unterscheiden sich in typsystemspezifischen Abläufen, wie die Erzeugung der entsprechenden Besucherklassen des Pakets visitor oder Generation und Ausgabe der Textrepräsentation des transformierten MWL-Programms.

3.2. Geänderte Anforderungen

Es gibt zwei Arten von Anforderungen, die im EIFA zusätzlich zum EIFAv1 umgesetzt sind. Einerseits gibt es die Anforderungen, die sich direkt aus der Aufgabenstellung der Diplomarbeit ergeben. Sie werden hier Hauptanforderungen genannt und in diesem Abschnitt beschrieben. Andererseits gibt es weniger wichtige Anforderungen, die den EIFA nützlicher machen oder Fehler beheben. Sie werden Nebenanforderungen genannt (s. Anhang E.3).

[8] JLex in der Version 1.2.6 (s. [Ber03]), Java CUP in der Version 0.10k (s. [Hud99])

3.2.1. Auswahl der Sicherheitstypsysteme durch Benutzer

Der allgemeine Ablauf, so wie er im Unterabschnitt 3.1 beschrieben ist, bleibt im Vergleich zum EIFAv1 erhalten. Über Kommandozeilenparameter kann der Benutzer angeben, mit welchem Sicherheitstypsystem der EIFA das MWL Programm überprüft. Der EIFA stellt die beiden Sicherheitstypsysteme aus dem EIFAv1 zur Verfügung, jeweils auch erweitert um die Regeln für Synchronisationsanweisungen. Außerdem stellt er das Sicherheitstypsystem für MWL mit einer MLS-Politik zur Verfügung, optional mit der Regel für Deklassifikationsanweisung oder mit den Regeln für Synchronisationsanweisungen. Die exakten Kombinationen, die der EIFA dem Benutzer zur Auswahl stellt, sind im Anhang B.2 aufgelistet.

3.2.2. MWL-Syntax

Die Grammatik der MWL-Programme, die der EIFA verarbeitet, ist in der Abbildung 17 aufgelistet. Die Grammatik ist unabhängig von dem Sicherheitstypsystem, das der Benutzer auswählt. Sprachelemente, die das gewählte Sicherheitstypsystem nicht behandelt, weist der EIFA als unsicher zurück, denn für diese sind keine Regeln vorhanden.

C	::= **skip**
	| Id : $Type$: $SecDom$ | Arr : $Type$ [Exp]: $SecDom$ | Sem :**sem**: $SecDom$
	| Id := Exp | Arr [Exp_1]:= Exp_2 | [Id:=Id']
	| **if** B **then** C **end** | **if** B **then** C_1 **else** C_2 **end**
	| **while** B **do** C **end** | C_1 ; C_2 | **fork**(C , C_1 , ... , C_n)
	| **signal**(Sem) | **wait**(Sem)
Exp	::= $IntConst$ | Id | Exp_1 Op Exp_2 | Arr .**length** | Arr [Exp]
	| **true** | **false**
op	::= + | − | ∗ | mod | div
	| && | || | <= | < | > | >= | == | !=
B	::= Exp
$Type$	::= **int** | **bool**

Abbildung 17: Grammatik der MWL-Syntax, wie sie im EIFA implementiert ist. Bezeichner Id, Arr bzw. Sem können neben alphanummerischen Zeichen auch „_“ enthalten, allerdings nicht am Beginn des Bezeichners. Das gleiche gilt für Sicherheitsdomänen $SecDom$.

Die Grammatik entspricht fast der des EIFAv1 (s. Abbildung 40 im Anhang). Einige Detailänderungen der Grammatik werden aufgrund von Nebenanforderungen (s. Anhang E.3) durchgeführt. Änderungen, die sich auch im AST wiederspiegeln sind die neuen Anweisungen zur Deklassifikation und Synchronisation sowie eine veränderte Syntax für Ausdrücke. Diese ist im EIFAv1 fehlerhaft (für Details s. Anhang E.3.6).

3.2.3. Spezifikation der Sicherheitpolitik

Der EIFAv1 überprüft nur die Sicherheit von MWL-Programmen bezüglich binärer Flusspolitiken, daher muss der Benutzer nicht die Flusspolitik spezifizieren. Das gilt genauso für den EIFA, falls der Benutzer eine Sicherheitsüberprüfung bezüglich einer binären Flusspolitik wählt. Dagegen muss der Benutzer die Flusspolitik spezifizieren, um ein MWL-Programm bezüglich einer MLS-Politik zu überprüfen. Um die Bedienung des EIFA einfach zu machen, liest der EIFA diese Spezifikation aus der gleichen Datei, aus der er auch das MWL-Programm liest. Falls der Benutzer eine Sicherheitsüberprüfung bezüglich einer MLS-Politik wählt und die vom Benutzer angegebene Datei vor dem MWL-Programm keine MLS-Politik-Spezifikation enthält, schreibt der EIFA auf die Fehlerausgabe der Kommandozeilenschnittstelle „*no security policy declared*“ und beendet sich. Falls der Benutzer eine Sicherheitsüberprüfung bezüglich einer binären Flusspolitik wählt, die angegebene Datei aber trotzdem eine MLS-Politik-Spezifikation enthält, ignoriert der EIFA diese MLS-Politik-Spezifikation.

Die Syntax der MLS-Politik-Spezifikation ist in Abbildung 18 definiert. Der EIFA interpretiert, wenn möglich, die drei Abschnitte als die drei Komponenten der MLS-Politik mit Ausnahmen, bzw. die zwei Abschnitte als die beiden Komponenten der MLS-Politik. Dazu, dass diese Interpretation möglich ist, muss die MLS-Politik-Spezifikation einige Bedingungen erfüllen:

- Die durch $TransFlow$ und $IntransFlow$ spezifizierten Relationen dürfen nur Sicherheitsdomänen in Relation setzen, die auch in $SecDoms$ aufgelistet sind.
- Die durch $TransFlow$ spezifizierte Relation muss reflexiv und transitiv sein.
- Eine der in $SecDoms$ aufgelisteten Sicherheitsdomänen muss zu jedem der in $SecDoms$ aufgelisteten Sicherheitsdomänen in der durch $TransFlow$ spezifizierte Relation stehen.

Die ersten beiden Bedingungen ergeben sich direkt aus den Definitionen der MLS-Politiken. Die letzte Bedingung bedeutet, dass eine Sicherheitsdomäne existieren muss, die der Sicherheitsdomäne „*low*“ der binären Flusspolitik entspricht. Diese Existenz erleichtert die Implementierung der Sicherheitstypsysteme ohne eine Einschränkung der Ausdrucksfähigkeit der Flusspolitik zu bewirken. Falls mindestens eine dieser Bedingungen nicht zutrifft, schreibt der EIFA eine Meldung auf die Fehlerausgabe der Kommandozeilenschnittstelle. Diese beschreibt, welche Anweisung nicht typisierbar ist und warum. Anschließend beendet sich der EIFA.

3.3. Änderungen

Da allgemein die Ausführung des EIFA gegenüber dem EIFAv1 genauso abläuft, kann der Systemaufbau (das heißt auch die Paketstruktur) erhalten bleiben. Der EIFA hat ein zusätzliches Paket `policy`, dessen Klassen die Datenstruktur und Zugriffsalgorithmen für die MLS-Politik implementiert.

MLSSpec	::=	{ *SecDoms* ; *TransFlow* }
		| { *SecDoms* ; *TransFlow* ; *IntransFlow* }
SecDoms	::=	*SecDom* | *SecDom* , *SecDoms*
TransFlow	::=	*SecDom* => *SecDom*
		| *SecDom* => *SecDom* , *TransFlow*
IntransFlow	::=	*SecDom* -> *SecDom*
		| *SecDom* -> *SecDom* , *IntransFlow*

Abbildung 18: Grammatik der Syntax für MLS-Politik-Spezifikationen, wie sie im EIFA implementiert ist. Sicherheitsdomänen *SecDom* bestehen aus alphanummerischen Zeichen einschließlich „_“ und beginnen mit einem Buchstaben.

Die folgenden Unterabschnitte sind den Paketen des EIFA zugeordnet. Klassen ohne Paketbezeichner (KLASSE) in einem der Unterabschnitte sind Teil des Pakets dieses Unterabschnittes. Bei Klassen der jeweils anderen Pakete steht der Paketbezeichner dabei (PAKET.KLASSE).

3.3.1. Paket checker

Grundsätzlich ändert sich die Struktur dieses Pakets gegenüber dem EIFAv1 nicht, da sich die Funktion (Steuerung des Ablaufs und des Datenflusses) ebenfalls nicht ändert. Allerdings ist die Funktion im EIFA komplexer, denn es gibt jetzt mehr Prüfalgorithmen (implementiert in den konkreten Besuchern des Pakets visitor). Diese sind auch nicht festgelegt, sondern der Benutzer kann sie auswählen (s. Abschnitt 3.2.1).

Wie im EIFAv1 steuern die Klassen der Checker -Hierarchie jeweils einen Überprüfungsvorgang. Für die neuen Algorithmen (Überprüfung auf Datentypsicherheit, Überprüfung auf Sicherheit mit MLS-Politik) gibt es neue Klassen in der Hierarchie. Den allgemeinen Ablauf steuert die Oberklasse Checker. Für spezielle Teilaufgaben, wie die Erzeugung der Besucher oder die Verarbeitung der Prüfergebnisse, nutzt sie abstrakte Methoden, die von Unterklassen implementiert sind.[9]

Die Klasse Analyzer, die es ermöglicht den EIFA von der Kommandozeilenschnittstelle zu starten, verarbeitet nun die Kommandozeilenargumente zur Sicherheitstypsystemauswahl. [10]

Eine wichtige neue Klasse ist TypeDeclarations. Sie ist eine unveränderbare Abbildung von Bezeichnern (ast.Identifier) auf Deklarationen (ast.Decl). Objekte dieser Klasse repräsentiert insbesondere Domänenzuweisungen: die einem Bezeichner zugewiesene Deklaration enthält die Sicherheitsdomäne des Bezeichners.

[9] Diese Aufgabenteilung nennt man Schablonenmuster (*template pattern*, s. [GHJV95]).
[10] Dazu benutzt Analyzer die Bibliothek *JSAP* (s. [Lam05])

3.3.2. Paket policy

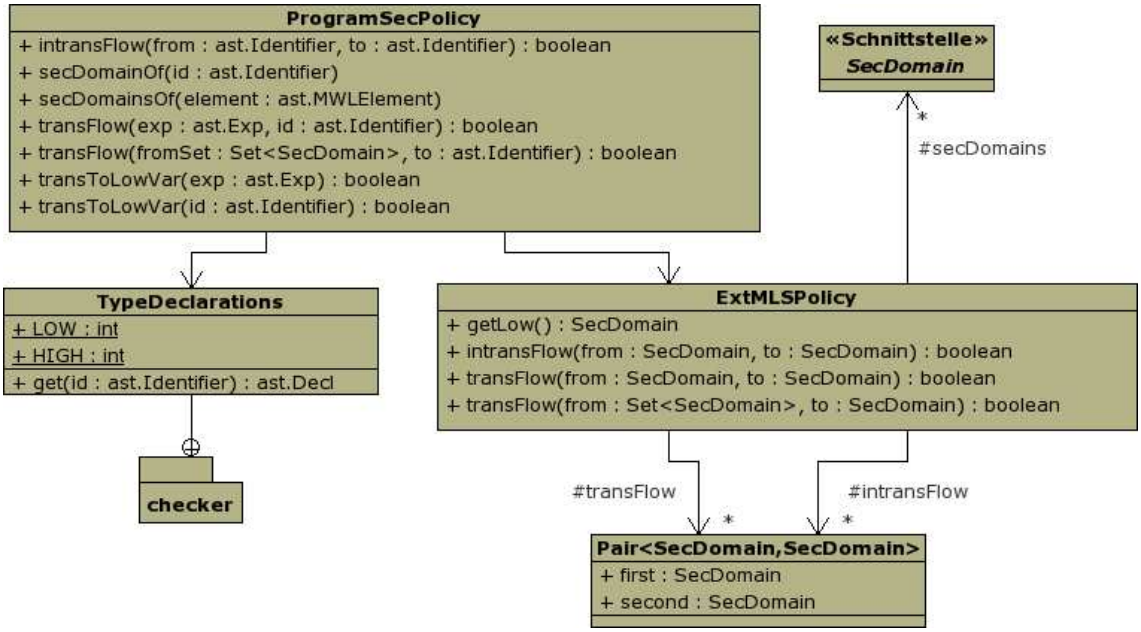


Abbildung 19: Klassenstruktur der wichtigsten Klassen des Pakets policy. Der Pfeil mit dem Kreis als Spitze bedeutet, dass TypeDeclarations aus dem Paket checker ist.

Dieses Paket enthält zwei wichtige Klassen. Objekte der Klasse ExtMLSPolicy repräsentieren MLS-Politiken ohne oder mit Ausnahmen, je nachdem ob man $\rightsquigarrow$ ignoriert oder nicht. Diese Klasse stellt Methoden bereit, um die Flussrelationen zwischen Sicherheitsdomänen abzufragen. Desweiteren bietet sie Zugriff auf das *low*-Element. Der Konstruktor dieser Klasse überprüft die im Abschnitt 3.2.3 aufgeführten Anforderungen an die MLS-Politik. Eine Menge von Sicherheitsdomänen (Schnittstelle SecDomain) und zwei Mengen von Sicherheitsdomänenpaaren repräsentieren die drei Komponenten einer MLS-Politik $(\mathcal{D}, \leq, \rightsquigarrow)$.

Die Klasse ProgramSecPolicy stellt Methoden bereit, um Anfragen zu beantworten, die den Fluss zwischen Sprachelementen betreffen. Die Bedingungen der Sicherheitstypregeln sehen zum Beispiel (für die Zuweisung) so aus: $\Gamma \vdash Exp : D \quad D \leq \Gamma(Id)$. Anders ausgedrückt, es ist gefragt, ob die MLS-Politik zusammen mit der Domänenzuweisung Datenfluss von Exp nach Id erlaubt. Zum Beispiel implementiert die Methode transFlow(Exp, Id) die genannte Bedingung. Dagegen implementiert intransFlow(Id', Id) die Bedingung $\Gamma(Id') \rightsquigarrow \Gamma(Id)$. ProgramSecPolicy repräsentiert damit eine Flusspolitik auf syntaktischer Ebene.

Dabei repräsentieren ein Objekt der Klasse ExtMLSPolicy die MLS-Politik und ein Object der Klasse checker.TypeDeclarations die Domänenzuweisung. ProgramSecPolicy integriert und kapselt diese beiden Klassen. Abgesehen von ProgramSecPolicy ist der EIFA von ihnen unabhängig. Wie sich später herausstellt (s. Abschnitt 4.3.2),

fasst ProgramSecPolicy genau die Bedingungsabfragen zusammen, in denen sich das Sicherheitstypsystem für die starke Sicherheit[dr] (für *delimited release*, s. Abschnitt 4.3) von dem Sicherheitstypsystem für die starke Sicherheit[in] (für *intransitive noninterference*) unterscheidet.

Objekte mit der Schnittstelle SecDomain repräsentieren Sicherheitsdomänen. [11]

3.3.3. Paket ast

Die Klassenstruktur des Pakets ast im EIFA unterscheidet sich aus vier Gründen gegenüber der Struktur im EIFAv1[12]:

1. Es gibt neue Klassen für die neuen Anweisungen:
   - Semaphordeklaration (SemDecl, s. Abbildung 20)

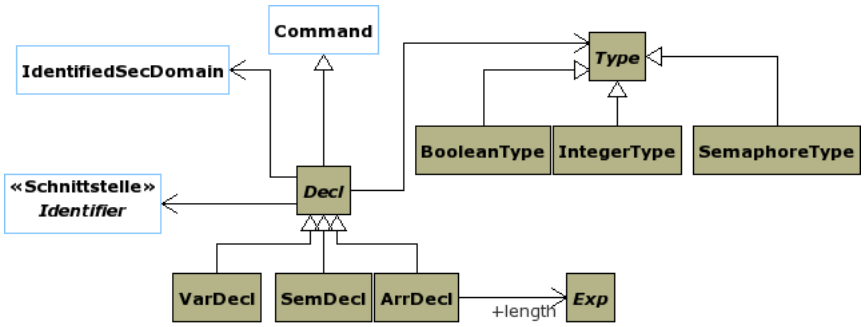


Abbildung 20: Klassenbeziehungen der Klasse Decl im EIFA. Zu den Änderungen gegenüber EIFAv1 vergleiche mit Abbildung 14(b).

   - Signal- und Wait-Anweisungen (Signal, Wait, s. Abbildung 21(a))
   - Deklassifikationsanweisung (IntranDeclass, s. Abbildung 21(b))

2. Die Struktur für Ausdrücke ist anders (s. Abbildung 22), da sich die Syntax unterscheidet (s. Abbildungen 40 und 17). Man beachte die Unterklassen von ComposedExpression. Diese erleichtern die Überprüfung auf Datentypsicherheit (s. Anhang E.3.7). Der Überprüfungsalgorithmus muss erkennen können, welche Datentypanforderungen an eine zu überprüfende Operation bestehen. Dazu fassen die drei Unterklassen von ComposedExpression Operationen zusammen, bei denen die gleichen Anforderungen an die Datentypen bestehen. Zum Beispiel steht CompareOperation für Vergleiche von ganzen

[11] Die einzige Anforderung an Sicherheitsdomänen ist, dass sie identifiziert werden können. Da in Java zwei beliebige Objekt mit der equals-Methode auf Gleichheit überprüfbar sind, stellt die Schnittstelle SecDomain an implementierende Klassen keine Anforderungen.

[12] In den Diagrammen dieses Abschnitts gilt: Helle Boxen stehen für Klassen oder Schnittstellen, die in mehr als einem der Diagramme vorkommen.

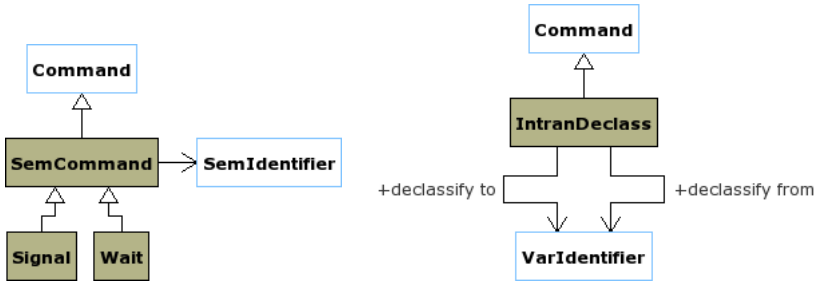


Abbildung 21: Klassen zur Repräsentation von Synchronisations- und Deklassifikationsanweisungen im EIFA

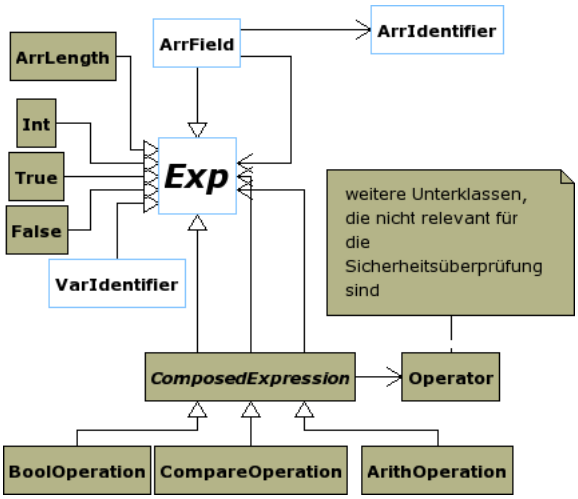


Abbildung 22: Klassen zur Repräsentation von Ausdrücken im EIFA. Zu den Änderungen gegenüber EIFAv1 vergleiche mit Abbildung 14(c)

Zahlen (==,>=,<=). Die Struktur ermöglicht es in dem konkreten Besucher, der die Datentypsicherheit überprüft, anhand der exakten Klasse eines ComposedExpression -Objekts die korrekten Anforderungen auszuwählen. Die exakte Klasse kennt er, da jeweils eine eigene visit -Methode existiert. Jede der drei visit -Methoden enthält nur die jeweils zu überprüfende Bedingung.

Da so die visit -Methoden einfach sind, enthält der AST-Entwurf diese Lösung. Allerdings zeigt die Erfahrung mit dieser Lösung, dass sie problematisch ist. Aufgrund der Änderung am AST müssen auch die anderen konkreten Besucherklassen für alle drei Unterklassen von ComposedExpression die visit -Methoden implementieren. Außerdem erschwert diese Struktur, Anforderungsänderungen an die Datentypsicherheit zu implementieren. Falls sich die Anforderungen ändern, muss eine Änderung am AST durchgeführt werden (Hinzufügen einer neuen Unterklasse zu ComposedExpression), was weitere Änderungen an den Besucherklassen nachsich zieht. Oder die Algorithmen zur Feststellung der Datentypsicherheit müssen die Einteilung in die drei Unterklassen umgehen. Aber damit wäre diese Einteilung unnötig.

Der Kern des Problems dieser Lösung ist, dass sie die Repräsentationen der Syntax und der Semantik (die Datenstrukturen und die Algorithmen) nicht klar trennt. Syntaktisch sind die drei Unterklassen unnötig. Alle Information befinden sich eine Stufe tiefer im AST, in den Operatoren. Ein besserer Lösung wäre daher, die Anforderungen an die Datentypen aufgrund der exakten Klasse des jeweiligen Operatorobjekts zu bestimmen. Um die Klasse eines gegebenen Operatorobjekts zu bestimmen gibt es zwei Möglichkeiten. Zum einem könnte man eine visit -Methode nutzen, wenn man den Operatorklassen eine accept -Methode hinzufügen würde. Zum anderen könnte man die Gleichheit (equals -Methode) nutzen, da Operatoren keine Struktur haben und die Gleichheit nur von der Klasse abhängt.

3. Um syntaktische Unterschiede explizit zu machen, werden zum einen Unterklassen für Variablenzuweisung und Arrayzuweisung von der Klasse Assign abgespaltet (s. Abbildung 23). Dadurch ist die Schnittstelle Assignable (s. Abbildung 14(c)) unnötig. Die explizite Unterscheidung dieser zwei Anweisungen ist sinnvoll, da die Sicherheitstypregeln aller im EIFA implementierter Sicherheitstypsysteme die beiden Anweisungen unterscheiden.

   Zum anderen ist Identifier im EIFA eine Schnittstelle (s. Abbildung 24). Die implementierenden Klassen entsprechen den syntaktischen Kontexten, in denen Bezeichner stehen können (Arraybezeichner, Variablenbezeichner, ...). Diese Unterscheidung hilft zum Beispiel bei der Überprüfung, ob ein in einem Programm im Array-Kontext verwendeter Bezeichner tatsächlich als Arraybezeichner deklariert ist. Die neue Klasse IdentifiedSecDomain (Bezeichner für Sicherheitsdomänen) implementiert policy.SecDomain. Sie ist also eine Sicherheitsdomäne, die allein über ihren Namen identifiziert wird. Die beiden Klassen HighSec und LowSec, welche im EIFAv1 die syntaktischen Sicherheitsdomä-

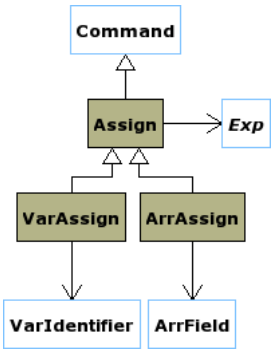


Abbildung 23: Klassen zur Repräsentation von Zuweisungen im EIFA. Zu den Änderungen gegenüber EIFAv1 vergleiche mit Abbildung 14

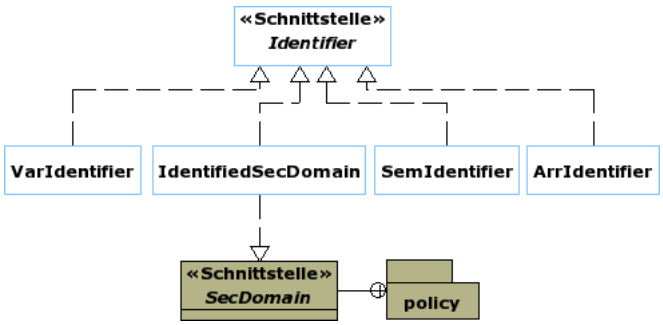


Abbildung 24: Klassen zur Repräsentation von Bezeichnern im EIFA.

nen repräsentieren, sind im EIFA durch IdentifiedSecDomain ersetzt.

4. Die Klassen und Schnittstellen im Paket ast des EIFAv1, die zwar die AST-Klassen strukturieren, aber im EIFAv1 keine Bedeutung haben (das heißt niemals wird eine Methode einer Objektreferenz dieser Klasse bzw. Schnittstelle aufgerufen) sind im EIFA nicht mehr vorhanden oder verändert. Die Klassen PrimitiveCommand, ComposedCommand und Atom (s. Abbildung 14) gibt es im EIFA nicht mehr. Die Klasse ComposedCommand wäre sinnvoll, wenn sie die Verwaltung von Teilanweisungen und rekursive Methodenaufrufe implementieren würde (also das Kompositum-Entwurfsmuster instantiieren würde). Ersteres implementieren aber die Unterklassen selbst, letzteres implementieren die konkreten Besucher. Die Schnittstelle MWLElement ist in der Bedeutung dahingehend erweitert, dass sie alle „besuchbaren" Objekte umfasst. Das heißt alle Klassen, die diese Schnittstelle implementieren, implementieren die accept -Methode des Besuchermusters.

3.3.4. Paket mwlParser

Die Klassen dieses Pakets (bzw. die Grammatik-Quelldateien der Klassen) unterscheiden sich hauptsächlich aus zwei Gründen gegenüber dem EIFAv1. Erstens müssen die Klassen dieses Pakets MWL-Programme der geänderten Grammatik verarbeiten und auf einen AST mit der geänderten Struktur abbilden. Dies erfordert keine Änderung des Konzepts. Zweitens können die MWL-Quelldateien nun auch eine Politik-Spezifikation enthalten (s. Abschnitt 3.2.3). Die Klasse MWLParser stellt im Gegensatz zu der des EIFAv1 eine Methode getPolicy zur Verfügung. Nachdem MWLParser eine MWL-Programmdatei mit einer Politik-Spezifikation geparst hat, gibt diese Methode ein Objekt der Klasse policy.ExtMLSPolicy zurück, welches der spezifizierten Politik entspricht.

Alternativ hätte MWLParser eine Repräsentation der Politik zusammen mit der AST-Repräsentation über die normale Parserergebnisrückgabe zur Verfügung stellen können. Diese Veränderung wäre aber aus der Sicht von Programmteilen, die keine Politikspezifikation verwenden, nicht unsichtbar

Weitere Veränderungen Ergeben sich aus Nebenanforderungen (Kommentare in MWL-Dateien, „_" in Bezeichnern und Zeilen- Spaltenangabe bei Syntaxfehlermeldungen, s. Anhang E.3). [13]

3.3.5. Paket visitor

Auch die allgemeine Struktur dieses Pakets bleibt erhalten. Die dreischichtige Struktur aus Besucherschnittstelle, Standardimplementierung und konkrete Besucher wird vom EIFAv1 übernommen (s. Abbildung 15). Dabei ist zu beachten, dass Visitor in

[13] Aus Gründen, welche die hier nicht erläuterte Softwareentwicklung betreffen, ist der Scanner-Generator *JLex* durch *JFlex* in der Version 1.4.1 ersetzt (s. [Kle04]). *JFlex* verarbeitet alle Eingabedatei, die auch *JLex* verarbeitet.

ASTVisitor und AbstractVisitor in DefaultASTVisitor umbenannt werden, da diese Namen die jeweilige Schnittstelle bzw. Klasse besser beschreiben.

Standardbesucher Als eine Änderung stellt DefaultASTVisitor eine Methode handleNoRuleMatch bereit, die eine Misserfolgsbehandlung implementiert. Damit muss nicht jeder konkreter Besucher selbst die Misserfolgsbehandlung durchführen (wie im EIFAv1).

Der wichtigste Unterschied zwischen den Klassen AbstractVisitor des EIFAv1 und DefaultASTVisitor des EIFA ist, dass DefaultASTVisitor jeden visit -Aufruf als Fehlschlag behandelt (also handleNoRuleMatch aufruft), während AbstractVisitor bei visit -Aufrufen nichts macht. Das bedeutet, mit DefaultASTVisitor sind alle Sprachelemente nicht typisierbar und werden erst durch überschreiben der visit -Methoden typisierbar. Dagegen sind mit AbstractVisitor alle Sprachelemente typisierbar und werden erst durch Überschreiben der visit -Methoden nicht typisierbar.

Diese Änderung ist zum einen als „defensive Programmierung“ sinnvoll, da so in abgeleiteten Besucherklassen vom Programmierer vergessene Methoden mit größerer Wahrscheinlichkeit auffallen. Zum anderen erleichtert dieses Standardverhalten die modulare Implementierung konkreter Besucher.

Besucher für MWL mit Synchronisation unter der binären Flusspolitik Zu sehen ist der Nutzen dieses Standardverhaltens zum Beispiel an den konkreten Besuchern SyncNoTransformVisitor und SyncTransformVisitor, welche die Sicherheitstypsysteme für die starke Sicherheit mit Synchronisationsansweisungen unter der binären Flusspolitik implementieren. Die Klassen sind von NoTransformVisitor bzw. TransformVisitor abgeleitet und erben damit auch die Implementierung der Sicherheitstypregeln für die starke Sicherheit ohne Synchronisationsanweisungen (s. Abbildung 38 bzw. 39 im Anhang). Die beiden neuen Besucher selbst implementieren nur noch die Sicherheitstypregeln für Synchronisationsanweisungen, indem sie die standardmäßig zurückweisenden visit -Methoden von DefaultASTVisitor überschreiben. NoTransformVisitor und TransformVisitor überschreiben diese dagegen nicht und weisen Synchronisationsanweisungen korrekt als „nicht sicher“ zurück.

Besucher für MWL-Anweisungen unter einer MLS-Politik Um nun die Sicherheitstypsysteme für MLS-Politiken zu implementieren, kann man wie im vorherigen Absatz die Vererbung nutzen. Das heißt, man ordnet die konkreten Besucher in einer Vererbungsstruktur wie in Abbildung 25 an. Dabei implementiert MLSVisitor das Sicherheitstypsystem für eine MLS-Politik und MWL ohne Deklassifikations- und Synchronisationsanweisungen. SyncMLSVisitor implementiert dazu die Sicherheitstypregeln für Synchronisationsanweisungen, DeclassMLSVisitor implementiert stattdessen die Sicherheitstypregeln für Deklassifikationsanweisungen. SyncDeclassMLSVisitor vereinigt alle diese Sicherheitstypregeln. Dabei unterscheiden sich SyncMLSVisitor und SyncDeclassMLSVisitor nur in der direkten Oberklasse. Man hat also die Regeln für Synchronisationsanweisungen doppelt implementiert.

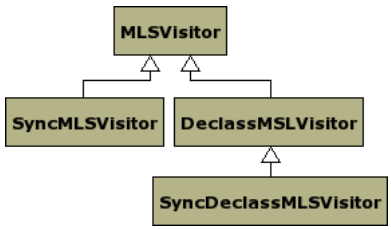


Abbildung 25: Besucherhierarchie für MWL mit MLS-Politik (Mehrfachverwendung durch Vererbung). Entspricht nicht dem Entwurf des EIFA.

Wenn man noch weitere Regeln für neue Sprachelemente ermöglichen würde, müsste man für jede Regelkombination eine eigene Besucherklasse erstellen. Damit würde die Zahl der Klassen exponentiell mit der Zahl der optionalen Regelsätze steigen. Dabei wären die Methoden für gleiche Regeln immer gleich implementiert.

Genau dieser Fall, eine Explosion der Unterklassen aufgrund vieler Kombinationen, wird in [GHJV95] als Anwendbarkeit des sogenannten Dekorierermusters (*decorator pattern*) genannt. Das Dekorierermuster ist eine Alternative zur Vererbung. Hierbei ersetzen eine dekorierende Klasse (genannt Dekorierer) und eine dekorierte Klasse (genannt Komponente) die abgeleitete Klasse und die Basisklasse der Vererbung. Folgende Unterschiede weist das Dekorierermuster gegenüber der Vererbung auf:

- Dekorierer und Komponente implementieren explizit eine gemeinsame Schnittstelle und haben so eine gemeinsame Signatur. Dagegen erbt eine abgeleitete Klasse die Signatur ihrer Basisklasse.
- Das Dekoriererobjekt und das Komponentenobjekt sind nicht identisch, aber das Komponentenobjekt ist mit dem Dekoriererobjekt assoziiert. Dagegen gibt es bei der Vererbung nur ein Objekt, dass gleichzeitig der abgeleiteten Klasse und der Basisklasse angehört.
- Der Dekorierer ruft explizit die Methoden der assoziierten Komponente auf, die er nicht selber implementiert. Die abgeleitete Klasse erbt automatisch nicht überschriebene Methoden der Basisklasse.

Die wichtige Folge dieser Unterschiede ist, dass beim Dekorierermuster die konkrete Klasse der Komponente nicht festgelegt ist, sondern nur die Schnittstelle. Das heißt jede Klasse, die diese Schnittstelle implementiert kann Komponente im Dekorierermuster sein. Dagegen ist bei der Vererbung die Basisklasse durch den Entwurf statisch festgelegt.

Genau dieser Vorteil löst das Problem, denn die Besucher für Deklassifikation und Synchronisation, DeclassMLSNoTransformVisitor und SyncMLSNoTransformVisitor, müssen damit jeweils nur einmal als Dekorierer implementiert werden. Der Besucher MLSNoTransformVisitor implementiert das Sicherheitstypsystem für MWL mit MLS-Politik aber ohne Synchronisation und Deklassifikation. Ein Objekt der Klasse SyncMLSNoTransformVisitor kann jetzt entweder direkt ein Objekt der

Klasse MLSNoTransformVisitor dekorieren oder es kann ein Objekt der Klasse DeclassMLSNoTransformVisitor dekorieren, welches wiederum ein Objekt der Klasse MLSNoTransformVisitor dekoriert.

In Abbildung 26 ist die komplette Instantiierung des Dekorierermuster im EIFA zu sehen. MlsAstVisitor ist die Komponentenschnittstelle. Das heißt Klassen, die

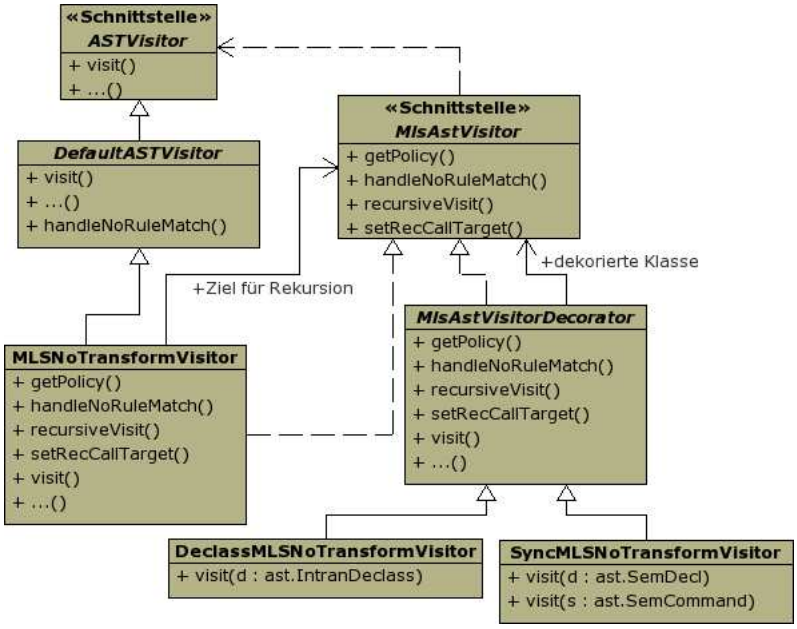


Abbildung 26: Besucherstruktur des EIFA für MWL mit MLS-Politik (Mehrfachverwendung der Regelimplementierungen durch Dekorierermuster). MlsAstVisitorDecorator ruft in allen Methoden die entsprechende Methode der dekorierten Klasse auf. Die Schnittstelle ASTVisitor und die abstrakte Klasse DefaultASTVisitor gehören nicht zur Instantiierung des Dekorierermusters selbst.

diese Schnittstelle implementieren, können nach dem Dekorierermuster erweitert (dekoriert) werden. Zum Beispiel ist MLSNoTransformVisitor eine konkrete Komponente, kann also dekoriert werden. Die Verarbeitung eines visit -Aufrufs stellt die Abbildung 27 anhand von drei Beispielen dar. Die beiden konkreten Dekorierer DeclassMLSNoTransformVisitor und SyncMLSNoTransformVisitor implementieren die Weiterleitung der Methodenaufrufe nicht selber, sondern überlassen das einer gemeinsamen abstrakten Basisklasse MlsAstVisitorDecorator. Deshalb ist diese mit der Komponentenschnittstelle MlsAstVisitor assoziiert.

Hier werden noch die von MlsAstVisitor spezifizierten Methoden erläutert. Das sind zum einen die visit -Methoden von ASTVisitor und handleNoRuleMatch, welche oben beschrieben wird.

Die anderen Methoden sind aufgrund einer der oben genannten Eigenschaften nötig: ein Dekoriererobjekt und sein dekoriertes Komponentenobjekt sind nicht iden-

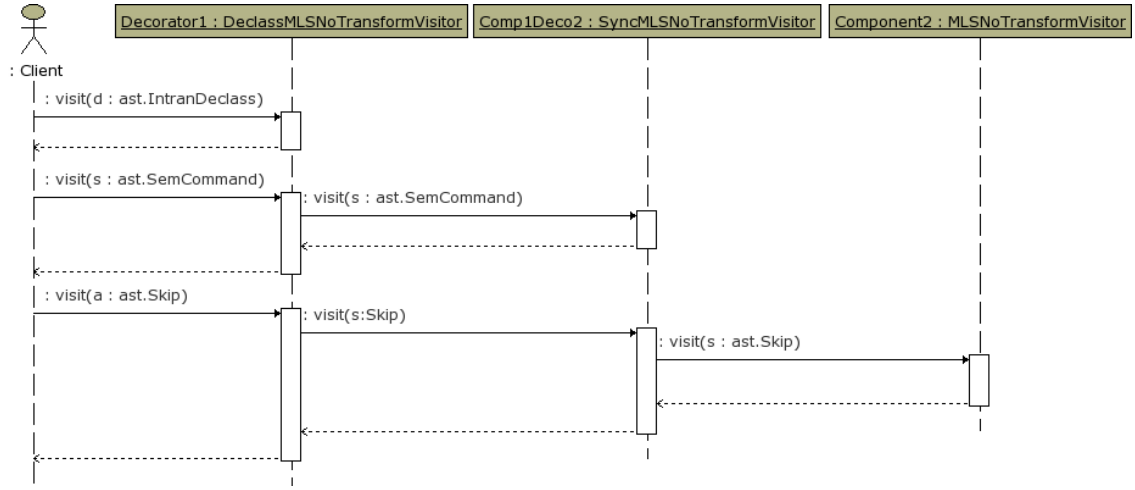


Abbildung 27: Sequenzdiagramm für visit -Aufrufe mit dem Dekorierermuster. Dargestellt sind von oben nach unten die visit -Aufrufe für eine Deklassifikationsanweisung, eine Synchronisationsanweisung und eine Skip-Anweisung. Dabei dekoriert Decorator1 die Komponente Comp1Deco2 und Comp1Deco2 die Komponente Component2. Zur Notation: oben stehen die Objekte mit Namen und Klasse, *Client* ist ein Aufrufer, ein durchgezogener Pfeil repräsentiert einen Methodenaufruf, ein Streifen repräsentiert eine Methodenausführung, ein unterbrochener Pfeil repräsentiert eine Methodenrückkehr.

tisch. Das heißt zum einem, sie haben jeweils eigene Attribute, sodass das Dekoriererobjekt nicht direkt auf die Attribute des Komponentenobjekts zugreifen kann. Deshalb bietet die Schnittstelle MlsAstVisitor die Methode getPolicy an, die ein Objekt der Klasse policy.ProgramSecPolicy liefert. Damit können ein Dekoriererbesucher und dessen Komponentenbesucher die gleiche MLS-Politik benutzen.

Zum anderen ist zu beachten, dass wenn das Komponentenobjekt Methodenaufrufe an sich selbst stellt, diese nicht an den Dekorierer gestellt sind. Allerdings müssen diejenigen visit -Methoden, die Typregeln für zusammengesetzte Sprachelemente implementieren, rekursiv visit -Methoden aufrufen[14]. Diese rekursiven Aufrufe darf der Besucher nicht an sich selbst stellen, sondern er muss sie an den dekorierenden Besucher stellen. Deshalb hat MLSNoTransformVisitor eine Assoziation zu MlsAstVisitor, die auf ein Dekoriererobjekt zeigt. Die Methode recursiveVisit ruft er auf, um rekursive visit -Aufrufe an das Dekoriererobjekt zu stellen.

Man betrachte als Beispiel das Programm signal(s);i:=0. Es werde mit einem Objekt der Klasse SyncMLSNoTransformVisitor (hier *SyncV* genannt) überprüft, welches ein Objekt der Klasse MLSNoTransformVisitor (hier *MlsV* genannt) dekoriert. Das heißt *SyncV* leitet bis auf die Aufrufe für Synchronisationsanweisungen alle visit -Aufrufe an *MlsV* weiter. Die sequentielle Komposition des Programms

[14]Genaugenommen rufen sie die visit -Methoden indirekt über die accept -Methode auf.

signal(s);i:=0 ist im AST ein Objekt der Klasse ast.Program. Also leitet $SyncV$ den visit -Aufruf an $MlsV$ weiter. Zur Überprüfung erfordert die Typregel rekursiv das Programm **signal**(s) zu überprüfen. Dessen visit -Methode ist aber in SyncMLSNoTransformVisitor implementiert. Damit diese Methode aufgerufen wird, muss $MlsV$ den rekursiven Aufruf an $SyncV$ stellen. Dazu nutzt es die Methode recursiveVisit. Diese stellt den visit -Aufruf an das assoziierte Objekt der Schnittstelle MlsAstVisitor, in diesem Fall $SyncV$.

Die Methode setRecCallTarget macht einem Komponentenobjekt das Dekoriererobjekt bekannt. Der Konstruktor von MlsAstVisitorDecorator ruft setRecCallTarget auf, um die Zielklasse (das zu konstruiernde Dekoriererobjekt selbst) für rekursive Aufrufe des Komponentenobjekts zu setzen.

Für einen Entwickler, der dem Sicherheitstypsystem Regeln für neue Sprachelemente hinzufügen oder bestehende Regeln ersetzen will, verringert der Einsatz des Dekorierermuster den Aufwand beträchtlich. Erstens muss jede Regel nur in genau einer Klasse implementiert sein, unabhängig von den möglichen Regelkombinationen. Mit Vererbung benötigt man für jede Regelkombination eine eigene Klasse. Zweitens sind die Klassen für neue Regeln sehr einfach. Zum Beispiel enthält DeclassMLSNoTransformVisitor neben der visit -Methode für Deklassifikationsanweisungen nur noch den Konstruktor. Selbst dieser ist einfach, denn er ruft nur den Konstruktor der Oberklasse MlsAstVisitorDecorator auf.

Es bleibt noch anzumerken, dass wenn es nur um die Erfüllung der Anforderung ginge, der vorgestellte Einsatz des Dekorierermusters für den EIFA zu aufwendig wäre. Der EIFA hat nur zwei optionale Regelsätze, daher macht sich der exponentielle Anstieg der Kombinationsanzahl nicht bemerkbar. Die Doppelimplementierung der Sicherheitstypregeln bei der Kombination durch Vererbung (s. Abbildung 25) ist weniger aufwendig, als die zusätzliche abstrakte Dekoriererklasse zu implementieren.

Der Einsatz des Dekorierermusters begründet sich damit, dass der EIFA nicht nur die Anforderungen erfüllen soll, sondern es soll auch den Entwurf selbst erforscht werden. Der Entwurf soll auch für mehr als zwei optionale Regelsätze geeignet sein. Schon zwei zusätzliche optionale Regelsätze würden mit der Kombination durch Vererbung sieben zusätzliche Besucherklassen benötigen, statt zwei zusätzliche mit dem Dekorierermuster.

Besucher für MWL-Ausdrücke unter einer MLS-Politik Die Sicherheitstypregeln für Ausdrücke unter einer MLS-Politik (s. Abbildung 11) sind in der Klasse SecDomCollect implementiert. Dieser konkrete Besucher liefert bei Aufruf einer visit -Methode für einen Ausdruck die Menge aller Sicherheitsdomänen atomarer Teilausdrücke zurück, die in dem Ausdruck vorkommen. Mit dieser Menge kann man nach Proposition 7 Bedingungen der Form $\Gamma \vdash Exp : D' \quad D' \leq D$ für Ausdrücke Exp überprüfen (s. Abschnitt 3.3.2), wenn man $D'' \leq D$ für alle D'' in dieser Menge überprüft.

Weitere Änderungen Zur Vollständigkeit stehen hier noch einige aus der Sicht des Entwurfs einfache Änderungen.

- Mit der Struktur des Pakets `ast` muss auch die Aufteilung der bestehenden `visit` -Methoden geändert werden.
- Der konkrete Besucher `IdTypeVisitor` implementiert die Überprüfung eines MWL-Programms auf Datentypsicherheit (aufgrund einer Nebenanforderung, s. Anhang E.3). Er erbt direkt von `DefaultASTVisitor` und unterscheidet sich nicht wesentlich von den Besuchern, die Sicherheitstypsysteme implementieren.
- Der Erfolg oder Misserfolg der Besucher wird nicht mehr über das Ergebnis (Methode `getResult`) festgestellt. Stattdessen werfen die Besucher bei Misserfolg eine Ausnahme, bei Erfolg nicht[15]. Dies ermöglicht es dem Besucher mehr Information über den Misserfolg an den Aufrufer zu melden, indem die Ausnahme diese Information enthält. Außerdem sind dadurch die Besucher zustandslos, was das Testen erleichtert.

3.4. Schluss

3.4.1. Bewertung

Insgesamt lässt sich die anfangs gestellte Frage, ob der Entwurf des EIFAv1 die Änderungen und Erweiterungen im Rahmen dieser Arbeit unterstützt hat, mit „ja“ beantworten. Auch die neuen Teile im Entwurf des EIFAs erhöhen zusätzlich die Änder- und Erweiterbarkeit des EIFA.

Funktionalität Zunächst ist festzustellen, dass die Anforderungen an den EIFA (s. Abschnitt 3.2) umgesetzt sind. Insbesondere prüft der EIFA die MWL-Programme der Beispiele aus Abschnitt 2.3 korrekt. Das Programm aus Beispiel 1 erkennt der EIFA als unsicher, kann es aber in ein stark sicheres Programm transformieren. Die Programme aus den Beispielen 2 und 3 erkennt der EIFA als stark sicher[in].

Änderbarkeit und Erweiterbarkeit Zentrales Elemente des EIFAv1-Entwurfs ist der Einsatz des Besuchermusters. Tatsächlich ermöglicht es neue Sicherheitstypsysteme zu ergänzen, ohne die Datenstrukturen (AST) oder die bestehenden Sicherheitstypsysteme und Algorithmen zu ändern.

Ein weiterer Vorteil des Besuchermusters ist, dass alle Methoden für ein Sicherheitstypsystem über eine Schnittstelle zugänglich sind, statt dass die Methoden über mehrere Klassen verteilt sind. Diese eine Schnittstelle lässt sich mit einem Dekorierer nach dem Dekorierermuster erweitern.

[15] Genaugenommen rufen die Besucher bei Misserfolg `handleNoRuleMatch` (s. oben) auf, welches die Ausnahme wirft. Durch überschreiben dieser Methode lässt sich das ursprüngliche Verhalten wiederherstellen.

In dieser Arbeit wird diese Möglichkeit genutzt, um Typregelsätze durch das Dekorierermuster zu kombinieren. Mit diesem Entwurf ist der Aufwand um Typregelsätze zu ergänzen gering. Er besteht darin, die neuen Typregeln genau einmal als Methoden zu implementieren und diese in einem Klassenrumpf mit einem Konstruktor zusammenzufassen. Das ist unabhängig davon, wie viele andere optionale Typregelsätze es gibt. Den Klassenrumpf erstellt eine Entwicklungsumgebung wie *Eclipse Java Development Tools (JDT)* (s. [Fou06]) automatisch. Daher könnte man höchstens noch an der Implementierung der einzelnen Methoden Aufwand einsparen.

Was der Einsatz des Besuchermusters im EIFAv1 nicht unterstützt, sind Änderungen am AST. Das Besuchermuster macht zwar die Datenstrukturen unabhängig von den Algorithmen, aber nicht umgekehrt. Dabei sind weniger die neuen Anweisungen ein Problem. Dafür muss man nur `visit`-Methoden ergänzen. Aber auch jede andere Änderung am AST erzwingt Änderungen an den `visit`-Methoden.

Dieser Nachteil des Besuchermusters sollte eigentlich für Änderungen und Erweiterungen des EIFAv1 kein großes Problem sein, da die MWL-Syntax bis auf Erweiterungen stabil ist und der AST nur die Syntax abbilden sollte. Im Rahmen der Änderungen zum EIFA wurden trotzdem einige Änderungen am AST durchgeführt. Zum einem ist in den Anforderungen die Syntax von Ausdrücken geändert. Diese Änderungen musste der AST abbilden. Zum anderen sind die AST-Klassen nicht vollständig algorithmenunabhängig definiert. Zum Beispiel benötigt keine der Algorithmen im EIFAv1 Wissen über die exakte Klasse eines Operatorobjekts. Deshalb bieten die Operatorklassen keine `accept`-Methoden an. Zusätzlich wurden im EIFA AST-Klassen erstellt, die es erleichtern bestimmte Algorithmen zu implementieren (die Unterklassen von `ast.ComposedExpression`, s. Abschnitt 3.3.3). Es stellte sich aber heraus, dass der Folgeaufwand der Änderungen am AST größer ist, als der eingesparte Aufwand bei der Algorithmenimplementierung.

Die Nachteile des Dekorierermusters, wie es im EIFA eingesetzt ist, sind zum einen die Komplexität des Musters, die größer als zum Beispiel bei der Vererbung ist (vergleiche die Abbildungen 25 und 26). Insbesondere der Aufwand um rekursive Aufrufe zu ermöglichen zeigt dies. Allerdings spricht diese größere Komplexität nicht unbedingt gegen den Einsatz des Dekorierermusters. Für den Entwickler, der neue Sicherheitstypregeln ergänzen will, ist die größere Komplexität nicht sichtbar.

Testbarkeit Die Betrachtung der Änder- und Erweiterbarkeit muss noch durch die Betrachtung der Testbarkeit ergänzt werden. Ein besser testbares System kann man einfacher automatisiert testen. Jedesmal, wenn man das System geändert oder erweitert hat, kann man die automatisierten Tests durchführen. Damit verringert man die Gefahr Fehler zu erzeugen, die sich auf bereits bestehende Teile des Systems auswirken.

Zunächst wird der EIFA betrachtet, dessen Algorithmenimplementierung gut testbar ist. Die Aussagen treffen im allgemeinen auch auf den EIFAv1 zu. Doch der EIFAv1 hat einige Entwurfseigenschaften, welche die Testbarkeit verringern. Diese werden anschließend betrachtet.

Im allgemeinen verschlechtert der Einsatz des Besuchermusters die Testbarkeit, da es doppelt Polymorphismus nutzt. Zusätzlich gibt es im EIFA das Dekorierermuster, das die Auswahl der auszuführenden visit -Methode steuert. Da aber weder die AST-Klassen, noch die Besucherklassen änderbare Attribute enthalten, arbeiten die visit -Methoden rein funktional. Das heißt, das Ergebnis eines visit -Aufrufs ist unabhängig von vorhergehenden visit -Aufrufen. Die visit -Methoden haben aufgerufen mit dem gleichen AST-Objekt und auf das gleiche Besucherobjekt immer das gleiche Ergebnis. Mit dieser Tatsache genügt es, nur jede visit -Methode einzeln zu testen. Aufruffolgen erfordern keine weiteren Test. [16]

Die Besucherklassen des EIFAv1 sind weniger gut testbar. Ein kleineres Problem ist, dass die Besucherobjekte nicht zustandslos sind, denn das Ergebnis bleibt im jeweiligen Besucherobjekt gespeichert. Was aber die Testbarkeit wesentlich erschwert ist, dass einige der Besucherklassen direkt statische Methoden der java.lang.System -Klasse aufrufen. Zum einem geben sie auf diese Weise Meldungen auf der Kommandozeilenschnittstelle aus. Zum anderen beenden sie auf diese Weise die Anwendung. Diese Methodenaufrufe sind für automatisierte Tests schwer zugänglich. Im EIFA dagegen ist die Steuerung des Programmablaufs und des Datenflusses vollständig Aufgabe der Klassen im Paket checker. Die Besucherklassen geben die Ergebnisse nur an den Aufrufer der Algorithmen zurück, was im EIFA die checker -Klassen sind. Bei den automatischen Tests steuern die Testprogramme den Ablauf und den Datenfluss. Sie ersetzen also die Klassen des Pakets checker, was durch die klare Aufgabentrennung möglich ist.

Ergebnis Insgesamt lässt sich das Ergebnis feststellen, dass der Entwurf des EIFAv1 die Erweiterungen im Rahmen dieser Arbeit unterstützt hat. Der Schwerpunkt der Erweiterungen lag auf neuen Sicherheitstypsystemen statt auf Änderungen der Sprache MWL, daher überwiegten die Vorteile des Besuchermusters.

Die Ergänzung des Entwurfs um das Dekorierermuster, das die Besucherklassen organisiert, unterstützt ebenfalls die Erweiterung der Sicherheitstypsysteme. Insbesondere bei der Kombination vieler Regelsätze verringert es den Aufwand sehr. Die Komplexität des Musters selbst ist größer, als zum Beispiel für die Vererbung, was aber für die Erweiterungen unsichtbar ist. Nur für den EIFA, wie er im Rahmen dieser Arbeit entwickelt wurde, erspart der Einsatz des Dekorierermusters keinen Aufwand, da es nur zwei optionale Erweiterungen des Sicherheitstypsystems gibt. Doch mit zwei weiteren optionalen Erweiterungen würden schon fünf Klassen gegenüber der Erweiterung mit Vererbung eingespart.

Es spricht nichts dagegen, dass man die Architektur des EIFA für andere Sprachen als MWL verwenden könnte. Mehr als die Architektur lässt sich aber nicht wiederverwenden. Man könnte im EIFA nicht ohne großen Aufwand die Sprache austauschen. Dafür würde man neue Datenstrukturen benötigen. Wie es vom Besuchermuster bekannt ist (s. [GHJV95]) und wie sich auch bei der Erweiterung des

[16] Im Rahmen dieser Arbeit wurden für alle implementierten Sicherheitstypsysteme automatisierte Tests implementiert.

EIFAv1 zum EIFA bestätigte, hängen die Besucherklassen stark von den Datenstrukturen ab. Man muss sie daher für neue Datenstrukturen neu erstellen. Wenn man mehrmals Sicherheitstypsysteme für neue Sprachen implementieren will, könnten andere Ansätze besser geeignet sein (s. unten).

3.4.2. Ausblick

Wie sich herausgestellt hat, ist der EIFA dafür geeignet neue Sicherheitstypregeln für MWL zu implementieren, eventuell mit Spracherweiterungen. Mögliche Sicherheitstypsysteme gibt es einige. Zum Beispiel wird in [SM02] ein Sicherheitstypsystem für MWL mit Sende- und Empfangsanweisungen vorgestellt. Oder in [KM05] wird ein Sicherheitstypsystem vorgestellt, das Unifikation nutzt, um Laufzeitlecks in bedingten Verzweigungen zu entfernen. Diese Sicherheitstypsysteme im EIFA zu implementieren würde es erlauben zu demonstrieren, dass diese Sicherheitstypsysteme funktionieren. Insbesondere wäre es auch sinnvoll ein transformierendes Sicherheitstypsystem für MLS-Politiken zu implementieren, das Laufzeitlecks entfernt. Denn für einen Programmierer ist es umständlich selber Anweisungen zu zählen und entsprechend Skip-Anweisungen einzufügen, um die Laufzeiten von Programmzweigen gleich zu machen. Solch ein transformierendes Sicherheitstypsystem müsste noch entwickelt werden, denn bestehende (s. [SS00], [KM05]) sind nur mit binären Flusspolitiken anwendbar.

Einige von Sicherheitstypsystemen unabhängige Erweiterungen wären sinnvoll, um bessere Beispielprogramme mit dem EIFA überprüfen zu können. Zum einem könnte man mit mehr Datentypen (zum Beispiel Fließkommazahlen, Zeichen, Zeichenketten) sinnvollere Programme schreiben. Da für die Sicherheitsbedingungen keine Annahmen an die Datentypen gemacht werden, betrifft diese Erweiterung auch nicht die Sicherheitstypsysteme. Zum anderen könnte man benutzerdefinierte Operatoren zulassen. Wie im Beispiel 1 zu sehen, stehen manchmal vereinfachte Ausdrücke in Beispielprogrammen, die nicht die gewünschte Bedeutung haben. Zum Beispiel steht `record :=forminput`, obwohl `record :=generaterecord(forminput)` mit einem anwendungsspezifischen Operator `generaterecord` gemeint ist. Für die Sicherheitsbedingung und damit für die Sicherheitsanalyse macht das keinen Unterschied. Für einen Leser aber schon.

Oben wird erwähnt, dass für mehrmals zu ändernde Sprachen andere Ansätze besser geeignet sein könnten Sicherheitstypsysteme zu implementieren. Zum Beispiel könnte das Werkzeug *antlr* (s. [Par06]) geeignet sein. Es kann nicht nur den Scanner und den Parser aus Grammatikdateien generieren, sondern auch den AST. Zusätzlich ermöglicht es *antlr* Baumparser (*treeparser*) aus einer Art Grammatikdateien zu generieren. Damit könnte man die Sicherheitstypsysteme implementieren. Da mit *antlr* mehr Teile des Systems automatisch generierbar sind, könnten sich Sicherheitstypsysteme für neue Sprachen schneller implementieren lassen. Aber es wäre zu untersuchen, ob sich Sicherheitstypsysteme auch so modular wie im EIFA aufbauen lassen und ob die Implementierungen ähnlich gut testbar sind.

4. *Intransitive Noninterference* und *Delimited Release*

Ziel dieses Abschnitts ist, das „was“ der Deklassifikation mit dem „wo“ der Deklassifikation zu kombinieren. Daher wird zunächst eine neue Sicherheitsbedingung für MWL definiert. Sie erlaubt Deklassifikation, aber begrenzt, „was“ ein Programm deklassifizieren kann. Das steht im Gegensatz zur starken Sicherheit[in] (basierend auf *intransitive noninterference*), die begrenzt, „wo“ ein Programm deklassifizieren darf (s. Abschnitt 1.2). Anschließend wird ein Sicherheitstypsystem vorgestellt, mit dem sich die neue Sicherheitsbedingung für MWL-Programme zeigen lässt. Insbesondere ist das Sicherheitstypsystem präzise genug, um sinnvolle, stark sichere[in] Programme zu typisieren, was durch Beispiele belegt wird. Man kann es daher zusammen mit dem Sicherheitstypsystem für die starke Sicherheit[in] anwenden.

Die wichtigsten Ergebnisse dieses Abschnitts sind, dass man mit der Kombination beider Sicherheitsbedingungen das „was“ und das „wo“ der Deklassifikation abdecken kann und dass man die Erfüllung der Kombination automatisch überprüfen kann.

Dieser Abschnitt ist folgendermaßen aufgebaut. Im Abschnitt 4.1 wird dargestellt, warum man ergänzend zur starken Sicherheit[in] eine Sicherheitsbedingung für das „was“ der Deklassifikation benötigt. Im Abschnitt 4.2 wird so eine Sicherheitsbedingung definiert. Diese ähnelt der starken Sicherheit und kombiniert eine Bisimulation mit Ideen aus [SM04] (s. Abschnitt 4.2.2). Es werden Zusammensetzbarkeitseigenschaften dieser Bedingung gezeigt, wie sie auch für die starke Sicherheit[in] gelten (s. Abschnitt 4.2.3). Um die Eigenschaften zu verdeutlichen, greift dieser Abschnitt nochmal die Beispiele aus Abschnitt 2 auf. Die Zusammensetzbarkeitseigenschaften ermöglichen es im Abschnitt 4.3 ein Sicherheitstypsystem zu definieren. Durch dieses Sicherheitstypsystem kann man für MWL-Programme zeigen, dass sie die neue Sicherheitsbedingung erfüllen (s. Satz 17).

Dieser Abschnitt betrachtet MWL ohne Synchronisationsanweisungen, da das technisch einfacher ist. Die Ergebnisse sollten sich auf MWL mit Synchronisationsanweisungen übertragen lassen, wenn man Synchronisationstransitionen in den Definitionen berücksichtigt (s. Abschnitt 4.4.2).

4.1. Motivation

Hier wird begründet, warum man zusätzlich zur starken Sicherheit[in] eine weitere Sicherheitsbedingung für Deklassifikation benötigt.

Indem man die starke Sicherheit[in] benutzt, legt die Flusspolitik fest, zwischen welchen Sicherheitdomänen („wo“ in der Flusspolitik, s. Definition 2) der Informationsfluss durch Deklassifikation stattfinden darf. Der Ersteller eines MWL-Programms legt fest, durch welche Anweisung des Programms („wo“ in dem Programm, s. Abschnitt 2.2.4) Deklassifikation stattfinden darf. Durch die Sicherheitsdomänen der beiden Bezeichner einer Deklassifikationsanweisung sind die Lokalisationen in der Flusspolitik und in dem Programm verknüpft. „Was“ spielt dabei nur insofern eine

Rolle, dass eine Deklassifikationstransition nur die Information deklassifizieren kann, die aktuell in Bezeichnern steckt, deren Sicherheitsdomäne die Deklassifikationsquelle ist.

Im Beispiel 2 bezieht sich die fachliche Sicherheitsanforderung darauf, welche Information das Programm deklassifizieren darf, also „was“. *Intransitive noninterference* in Form der starken Sicherheit[in] kann bei der Argumentation helfen, dass das Programm den Sicherheitsanforderungen entspricht. Allerdings zeigt Beispiel 2 auch ein Programm (result:= vote1;[pub_result:= result]), das genauso die starke Sicherheit[in] mit der dort eingesetzten MLS-Politik erfüllt, aber nicht die fachliche Sicherheitsanforderung. Beispiel 6 in diesem Abschnitt greift dieses Programm nochmal auf und wendet darauf die neue Sicherheitsbedingung an.

Zunächst aber wird folgendes Beispiel betrachtet:

Beispiel 4. Die Frage ist, welche Information deklassifiziert die Laufzeitumgebung, wenn sie die Anweisung [l := h] dieses Programms ausführt:

```
if bh then
   h := h1
else
   h := h2
[ l := h ]
```

Sie deklassifiziert den Wert des Bezeichners h zum Zeitpunkt der Ausführung dieser Anweisung. Dieser Wert ist abhängig von den Startwerten der Bezeichner bh, h1 und h2.

Hier ist es für den Leser des Programms direkt ersichtlich, „was“ dieses Programm deklassifiziert. Die Anweisungen könnten aber über ein längeres Programm verteilt sein, sogar über mehrere Threads.

Ein Sicherheitsbedingung, die das „was“ behandelt, kann diesen Informationsfluss berücksichtigen.

4.2. Starke Sicherheit[dr]

Folgende Grundidee lässt sich nutzen, um zu spezifizieren, welche Information ein Angreifer lernen darf, wenn er die Ausführung eines Programms beobachtet. Man geht davon aus, dass der Angreifer schon zu Beginn alle zu deklassifizieren erlaubte Information kennt. Er darf keine weitere Information lernen, wenn er die Ausführung des Programms beobachtet. Um das Wissen eines Angreifers über Zustände zu modellieren, wird im Abschnitt 2.3 die Äquivalenzrelation $=_D$ vorgestellt. Ein Ansatz um zu modellieren, „was“ der Angreifer zusätzlich weiß, ist diese Äquivalenzrelation zu verkleinern. Zum Beispiel kann man nur noch Zustände in Relation zueinander setzen, welche die gleiche Parität eines vertraulichen Variablenbezeichners haben. Damit modelliert man, dass der Angreifer diese Parität kennt.

Dieses Modell wurde schon in [Coh78] als sogenannte *selective dependency* verwendet. In [SS05] werden weitere Deklassifikationskonzepte aufgeführt, welche dieses Modell verwenden. Er wird dort *PER (partial equivalence relation) model* genannt

Hier wird die *delimited release* aus [SM04] betrachtet. Für *delimited release* wird in [SM04] ein Sicherheitstypsystem vorgestellt, das den in dieser Arbeit betrachteten Sicherheitstypsystemen ähnelt. Alternative Ansätze, die in Erwägung gezogen wurden, sind *abstract noninterference* aus [GM04] und eine Ansatz aus [Low04], der die deklassifizierbare Informationsmenge begrenzt. Allerdings gibt es für beide Ansätze keine mit Sicherheitstypsystemen vergleichbare Überprüfungsverfahren. Weiteres dazu steht im Abschnitt 4.4.2.

4.2.1. Grundlagen

Delimited release unterscheidet sich von anderen Ansätzen basierend auf dem *PER model* darin, wie man die deklassifizierbare Information spezifiziert. Da Sicherheitsbedingungen über die Semantik von Programmen definiert sind, spezifizieren andere Ansätze auch die zu deklassifizierende Information rein semantisch. Dadurch ist die automatische Überprüfung auf Sicherheit schwer, es gibt für keinen anderen Ansatz basierend auf dem *PER model* einen automatischen Überprüfungsalgorithmus (s. [SM04]).

Mit *delimited release* spezifiziert man die deklassifizierbare Information über Ausdrücke der Sprache (hier MWL), also syntaktisch. Wie in [SM04] werden diese Ausdrücke *escape-hatches*[17] genannt. Die Idee für die Sicherheitsbedingung ist folgende: wenn zwei Zustände in allen sichtbaren Bezeichnern und *escape-hatches* die gleichen Werte haben, muss sich das Programm gestartet aus diesen beiden Zuständen ununterscheidbar verhalten, soweit es sichtbar ist. Im Gegensatz zu einer normalen *noninterference*-Bedingung, wie zum Beispiel der starken Sicherheit, muss das sichtbare Verhalten nicht ununterscheidbar sein, wenn die sichtbaren Variablenbezeichner die gleichen Werte haben, aber eine *escape-hatch* in den Zuständen unterschiedliche Werte hat.

In [SM04] wird *delimited release* nur für sequentielle Programme definiert. Die dort gegebene Definition drückt folgendes aus: ein Programm ist dann sicher, wenn für jede Sicherheitsdomäne das Programm gestartet aus zwei beliebigen Zuständen, in denen die Werte aller für die Sicherheitsdomäne sichtbaren Bezeichner und *escape-hatches* gleich sind, nach Ausführung des Programms alle für diese Sicherheitsdomäne sichtbaren Bezeichner gleich sind.

Hier folgt die Definition einer *delimited release*-Sicherheitsbedingung für MWL und eine MLS-Politik.

4.2.2. Definition

Die Sicherheitsbedingung für *delimited release* mit MLS-Politik und Nichtdeterminismus durch Nebenläufigkeit wird definiert, indem eine Komponente in der Definition der starken Sicherheit verändert wird. Eine von einer *escape-hatch*-Menge abhängi-

[17] Hier wird der englische Ausdruck statt den möglichen Übersetzungen „Ausstiegsluke“ oder „Notluke“ genutzt, s. Abschnitt 1.4

ge Zustandsäquivalenz $=_D^{\mathcal{H}}$ ersetzt in der Definition der starken D-Bisimulation (s. Definition 5) die Zustandsäquivalenz $=_D$.

Zunächst folgt die Definition der möglichen *escape-hatch*-Mengen:

Definition 15 (*escape-hatches*). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Dann ist $\mathcal{H} \subseteq \{(D', Exp) | D' \in \mathcal{D} \wedge Exp \text{ ist MWL-Ausdruck}\}$ eine Menge von *escape-hatches*.

Intuitiv bedeutet die Existenz einer *escape-hatch* (D', Exp), dass der Ausdruck Exp nach D' deklassifizierbar ist. Wenn das Deklassifikationsziel aus dem Kontext ersichtlich oder irrelevant ist, werden auch nur Ausdrücke als *escape-hatches* bezeichnet. In dieser Arbeit gehören die *escape-hatches* zur Flusspolitik, da sie einen erlaubten Informationsfluss spezifizieren

Wie folgt wird die oben genannte Zustandsäquivalenz definiert:

Definition 16 ($D, \mathcal{H}$-Gleichheit). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{H}$ eine Menge von *escape-hatches* und sei $D \in \mathcal{D}$. Zwei Zustände s und s' sind genau dann *$D, \mathcal{H}$-gleich* (geschrieben $s =_D^{\mathcal{H}} s'$), wenn

1. $s =_D s'$
2. $\forall (D', Exp) \in \mathcal{H}: \quad D' \leq D \Rightarrow \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$

Es lässt sich folgende Eigenschaft feststellen:

Lemma 9. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{H} = \emptyset$ und sei $D \in \mathcal{D}$. Dann gilt*

$$\forall s, s': \quad s =_D s' \quad \Leftrightarrow \quad s =_D^{\mathcal{H}} s'$$

Beweis: Seien s, s' beliebige Zustände und $\mathcal{H} = \emptyset$. Dann ist die zweite Bedingung aus Definition 16 immer erfüllt. Also gilt $s =_D^{\mathcal{H}} s'$ genau dann, wenn die erste Bedingung aus Definition 16 gilt. Diese ist $s =_D s'$. □

Definition 17 (Starke D-Bisimulation[dr] (für *delimited release*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$, eine Domänenzuweisung Γ und eine Menge von *escape-hatches* $\mathcal{H}$ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Bisimulation[dr]* $\approx_D^{dr}$ die Vereinigung aller symmetrischen Relationen R zwischen Anweisungsvektoren $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ der gleichen Größe ($\vec{V} = (C_1, \ldots, C_n)$ und $\vec{V'} = (C_1', \ldots, C_n')$), sodass

$$\begin{aligned}\forall s, s', t: \forall i \in \{1 \ldots n\}: \forall \vec{W}: \vec{V} R \vec{V'} \wedge s =_D^{\mathcal{H}} s' \wedge \langle\!| C_i, s |\!\rangle \twoheadrightarrow \langle\!| \vec{W}, t |\!\rangle \\ \Rightarrow \exists \vec{W'}, t': \langle\!| C_i', s' |\!\rangle \twoheadrightarrow \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} R \vec{W'} \wedge t =_D^{\mathcal{H}} t'\end{aligned}$$

Lemma 9 erweitert sich auf diese Bedingung.

Lemma 10. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Seien $\mathcal{H} = \emptyset$, $D \in \mathcal{D}$ und $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$. Dann gilt: $\vec{V} \approx_D \vec{V'} \quad \Leftrightarrow \quad \vec{V} \approx_D^{dr} \vec{V'}$.*

Beweis: Die Definition 17 für $\approx_D^{dr}$ unterscheidet sich von der Definition 5 für $\approx_D$ nur in der in der Zustandsgleichheit $=_D^{\mathcal{H}}$ bzw. $=_D$. Daher ist mit Lemma 9 das Lemma 10 bewiesen.
□

Es folgen die Sicherheitsdefinitionen:

Definition 18 (Starke D-Sicherheit[dr] (für *delimited release*))**.** Seien eine MLS-Politik $(\mathcal{D}, \leq)$, eine Domänenzuweisung Γ und eine Menge von *escape-hatches* $\mathcal{H}$ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$. Ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ ist genau dann *stark sicher[dr] für* D, wenn $\vec{V} \approx_D^{dr} \vec{V}$.

Definition 19 (Starke Sicherheit[dr] (für *delimited release*))**.** Seien eine MLS-Politik $(\mathcal{D}, \leq)$, eine Domänenzuweisung Γ und eine Menge von *escape-hatches* $\mathcal{H}$ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Ein Anweisungsvektor $\vec{V} \in \vec{\mathcal{C}}$ ist genau dann *stark sicher[dr]*, wenn er für alle $D \in \mathcal{D}$ stark sicher[dr] ist.

Die starke Sicherheit ist ein Sonderfall der starken Sicherheit[dr]:

Satz 11. *Seien eine MLS-Politik* $(\mathcal{D}, \leq)$ *und eine Domänenzuweisung* Γ *gegeben. Sei* $\mathcal{H} = \emptyset$*. Dann gilt für einen beliebigen Anweisungsvektor* $\vec{V} \in \vec{\mathcal{C}}$*:*
$\vec{V}$ *ist genau dann stark sicher, wenn* $\vec{V}$ *stark sicher[dr] ist.*

Beweis: Mit $\mathcal{H} = \emptyset$ und $\vec{V} = \vec{V}'$ wendet man Lemma 10 an. □

Die Definitionen nutzen die allgemeine Transition $\twoheadrightarrow$. Deklassifikationsanweisungen verhalten sich bezüglich der Transition $\twoheadrightarrow$ genauso wie Zuweisungen (s. Abschnitt 2.2.4). Daher kann man die Definitionen auch auf Programme anwenden, die Deklassifikationsanweisungen enthalten.

Man betrachte das Ausgangsbeispiel: Beispiel 4. Wenn man $\mathcal{H} = \emptyset$ setzt (also normale starke Sicherheit wählt), dann ist das Programm nicht sicher, da aufgrund der Deklassifikationsanweisung $[\ \mathsf{l} := \mathsf{h}]$ direkt Information von *high* nach *low* fließt. Wenn man $\mathcal{H} = \{(low, \mathsf{h}), (low, \mathsf{h1}), (low, \mathsf{h2}), (low, \mathsf{bh})\}$ setzt, dann ist das Programm stark sicher[dr]. Sobald eine oder mehrere der *escape-hatches* fehlen, gilt das nicht mehr.

***Escape-Hatches* nach der Originaldefinition aus [SM04]** In der Definition von *delimited release* aus [SM04] werden die *escape-hatches* anders spezifiziert als hier. Sie sind dort nicht explizit Teil der Flusspolitik, sondern sie sind Teil des MWL-Programms. Das heißt es gibt einen Deklassifikationsausdruck $\mathsf{declassify}(Exp, D)$. Dieser unterscheidet sich semantisch nicht vom Ausdruck *Exp*. Als *escape-hatches* betrachtet die Sicherheitsbedingung alle solche im MWL-Programm vorkommenden Ausdrücke.

Wie oben argumentiert, ist eine externe *escape-hatch*-Menge sinnvoll, da die *escape-hatches* Informationsfluss erlauben und somit zur Flusspolitik gehören. Aber letzten Endes geben technische Unterschiede den Ausschlag, welche Variante man wählt. In [SM04] ermöglicht die Spezifikation der *escape-hatches* eine einfachere Überprüfung

der Sicherheitsbedingung. Dort schränkt das Sicherheitstypsystem die Deklassifikation auf diese Deklassifikationsanweisungen ein und verbietet die Zuweisung an darin vorkommende Bezeichner. Da per Definition jeder deklassifizierte Ausdruck eine *escape-hatch* ist, muss das nicht weiter überprüft werden.

Dagegen wird in dieser Arbeit die starke Sicherheit[dr] mit der starken Sicherheit[in] kombiniert, für die es schon Deklassifikationsanweisungen gibt. Im Gegensatz zu den Deklassifikationsausdrücken aus [SM04] eignen sich diese nicht als Spezifikation von *escape-hatches*, da sie nur Bezeichner deklassifizieren (s. Abschnitt 4.2.5).

Vergleich der starken Sicherheit[dr] mit der Sicherheit aus [SM04] Die Sicherheitsbedingung aus [SM04] ist in der vorliegenden Arbeit nicht formal definiert. Aber man kann plausibel machen, dass aus der starken Sicherheit[dr] eines sequentiellen MWL-Programms auch die Sicherheitsbedingung aus [SM04] folgt, abgesehen von der Spezifikation der *escape-hatches*. Die Sicherheitsbedingung aus [SM04] verlangt, dass ein aus zwei $D, \mathcal{H}$-gleichen Zuständen gestartetes Programm in D-gleichen Zuständen terminiert, falls es terminiert. Seien ein sequentielles, stark sicheres[dr] Programm und zwei beliebige $D, \mathcal{H}$-gleiche Zustände gegeben. Die aus den beiden Zuständen gestarteten Programmabläufe ergeben zwei Zustandsfolgen. Wenn man stellenweise die Zustandspaare der Zustandsfolgen betrachtet, folgt aus der starken Sicherheit[dr], dass jedes Zustandspaar $D, \mathcal{H}$-gleich ist. Das gilt auch für die Endzustände, falls sie existieren. Damit ist erfüllt, was die Sicherheitsbedingung aus [SM04] fordert: die Endzustände sind D-gleich, falls sie existieren.

Beispiel 5 (Beispiele aus [SM04]). Hier stellen Beispiele aus [SM04] dar, wie man die starke Sicherheit[dr] nutzen kann und in welchen Fällen sich von der Originaldefinition von *delimited release* aus [SM04] unterscheidet. Die Flusspolitik ist hier immer binär. Die Domänenzuweisung ergibt sich aus den Bezeichnernamen: „h“ als letzter Buchstabe für *high*, „l“ als letzter Buchstabe für *low*.

Das Programm des ersten Beispiels berechnet den öffentlichen Durchschnitt von vertraulichen Variablen:

```
avgl := ( h1 + h2 + h3 + h4) div 4
```

Mit der *escape-hatch* `( h1 + h2 + h3 + h4) div 4` ist dieses Programm stark sicher[dr]. Dagegen ist das Programm

```
h1 := h4; h2 := h4; h3 := h4;
avgl := ( h1 + h2 + h3 + h4) div 4
```

nicht stark sicher[dr]. Wenn man die ersten drei Zuweisungen ausführt, fließt zusätzliche vertrauliche Information in die *escape-hatch*, sodass die letzte Zuweisung `h4` statt dem Durchschnitt deklassifiziert. Bei diesem Beispiel unterscheidet sich die Sicherheitsbedingung aus [SM04] nicht von der starken Sicherheit[dr].

Ein weiteres Beispiel modelliert eine Operation auf einer elektronischen Brieftasche. Der Inhalt `h` der Brieftasche ist vertraulich, der Preis eines Produkts `kl` und das bisher ausgegebene Geld `l` sind öffentlich. Die *escape-hatch* ist `h>= kl`, also die Bedingung, ob der Inhalt noch ausreicht, um das Produkt zu kaufen.

if h>= kl **then** h:= h− kl ; l:= l+ kl **end**

Das Programm ist nach der Sicherheitsbedingung aus [SM04] sicher, da der Fluss der beiden Zuweisungen nach der binären Flusspolitik erlaubt ist und der Ablauf nur von der *escape-hatch* abhängt. Das Programm ist aber nicht stark sicher[dr], denn mit h:= h− kl fließt zusätzliche Information in die *escape-hatch*. Man betrachte zwei Zustände s und s' so, dass

- $s(\,\mathsf{l}\,) = s'(\,\mathsf{l})$
- $s(\,\mathsf{kl}\,) = s'(\,\mathsf{kl})$
- $s(\,\mathsf{h}) \geq 2s(\,\mathsf{kl})$
- $2s'(\,\mathsf{kl}\,) > s'(\,\mathsf{h}\,) \geq s'(\,\mathsf{kl}\,)$.

Der Wert der *escape-hatch* ist in beiden Zuständen gleich, nämlich *true*. Es gilt also $s =^{H}_{low} s'$. Wenn man h:= h− kl in beiden Zuständen ausführt, bleibt im Zustand s der Wert der *escape-hatch* gleich, im Zustand s' ändert er sich zu $false$.

Man könnte aber einen Variablenbezeichner newh einführen und statt der unsicheren Zuweisung die Zuweisung newh:= h− kl einsetzen. Damit wäre das Programm stark sicher[dr]. Aber man könnte so das Programm nicht sinnvoll mehrmals hintereinander ausführen. Allerdings wäre auch die erste Version des Programms nicht mehr sicher nach der Sicherheitsbedingung aus [SM04], wenn man es mehrmals hintereinander ausführt.

Anmerkung zu *delimited release* mit Deklarationsanweisungen Die starke Sicherheit[dr] angewandt auf Deklarationen entspricht nicht ganz dem Sinn, in dem in dieser Arbeit Deklarationen in Beispielprogrammen verwendet werden. Zum Beispiel ist nach der Definition der Semantik für Deklarationen die Deklaration h1:**int**:high nicht stark sicher[dr], falls es die *escape-hatch* h1 + h2 gibt. Zwei Zustände, in denen der Wert dieser Summe gleich, aber der Wert von h1 unterschiedlich ist, haben nach der Ausführung unterschiedliche Werte der Summe. Es fließt also Information in die *escape-hatch*. Das ist ein Problem, denn in dieser Arbeit modellieren Deklarationen auch Eingabeparameter, dieser Informationsfluss ist also gewünscht.

Das Problem lässt sich von zwei Seiten lösen. Zum einem, indem man eine neue, markierte Transition einführt, die semantisch eine Deklaration repräsentiert. Die Sicherheitsbedingung könnte bei dieser Transition den Informationsfluss in *escape-hatches* erlauben. Zum anderen, indem man Eingabeparameter nicht in Deklarationen, sondern explizit vor das MWL-Programm schreibt.

Diese Arbeit geht nicht weiter auf Deklarationen ein, außer im Ausblick (s. Abschnitt 4.4.2). Da Deklaration in MWL-Programmen nicht in Teilprogrammen auftreten dürfen (außer in der sequentiellen Komposition), fließt außer dem Startwert keine Information in die deklarierten Bezeichner. Da Deklarationen von Bezeichnern nur auftreten dürfen, bevor diese Bezeichner verwendet werden, sind die Werte der Bezeichner vor der Deklaration der Bezeichner auch nicht relevant für den Ablauf

des Programms. Es macht also keine Unterschied, ob die Bezeichner (und damit auch die *escape-hatches*) ihren Wert implizit vor Start des Programms zugewiesen bekommen oder zu Beginn des Programms durch Deklarationen.

Auf jeden Fall ist *delimited release* eng damit verbunden, wie bei der Ausführung eines MWL-Programms die Startwerte von Bezeichnern zustande kommen. Interessant wäre es, *delimited release* in Kombination mit weiteren Ansätzen zur Deklaration, lokalen Variablen, expliziten Eingabe- oder Funktionsparametern und Eingabe-/Ausgabeanweisungen zu betrachten (für einen Ausblick s. Abschnitt 4.4.2).

4.2.3. Zusammensetzbarkeit

Die Zusammensetzbarkeitseigenschaften einer Sicherheitsbedingung sind nützlich, um Programme auf Sicherheit zu überprüfen. Die Definition von starker Sicherheit[dr] orientiert sich an der Definition der normalen starken Sicherheit. Daher lassen sich die Zusammensetzbarkeitseigenschaften (s. [MS04]) auch ohne große Schwierigkeiten übertragen.

Die Beweise benötigen folgende Relation:

Definition 20 (*D-bisimulation up to* $\approx_D^{dr}$). Seien eine MLS-Politik $(\mathcal{D}, \leq)$, eine Domänenzuweisung Γ und eine Menge von *escape-hatches* $\mathcal{H}$ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$. Eine symmetrische Relation R über Anweisungen aus $\mathcal{C}$ ist eine *D-bisimulation up to* $\approx_D^{dr}$, wenn

$$\begin{aligned}\forall C, C', s, s', C_1, \ldots, C_n, t : CRC' \wedge s =_D^{\mathcal{H}} s' \wedge \langle\!|C, s|\!\rangle \twoheadrightarrow \langle\!|C_1, \ldots, C_n, t|\!\rangle \\ \Rightarrow \exists C_1', \ldots, C_n', t' : \begin{bmatrix} \langle\!|C', s'|\!\rangle \twoheadrightarrow \langle\!|C_1', \ldots, C_n', t'|\!\rangle \\ \wedge \quad \forall i \in \{1, \ldots, n\} : C_i (R \cup \approx_D^{dr})^+ C_i' \wedge t =_D^{\mathcal{H}} t' \end{bmatrix}\end{aligned}$$

Dabei ist $(\cdot)^+$ der transitive Abschluss.

Nützlich ist diese Bisimulation mit folgender Eigenschaft:

Proposition 12. *Sei eine starke D-Bisimulation[dr]* $\approx_D^{dr}$ *gegeben. Wenn* R *eine D-bisimulation up to* $\approx_D^{dr}$ *ist, dann gilt* $R \subset \approx_D^{dr}$.

Beweis: Seien eine starke D-Bisimulation[dr] $\approx_D^{dr}$ und eine *D-bisimulation up to* $\approx_D^{dr}$ R gegeben. Es sei definiert:

$$\begin{aligned}S = \{ (\,(C_1, \ldots, C_k)\,,\, (C_1', \ldots, C_k')\,) | \\ k \in \mathbb{N} \wedge \forall i \in \{1, \ldots, k\} : C_i, C_i' \in \mathcal{C} \wedge C_i (R \cup \approx_D^{dr})^+ C_i' \}\end{aligned}$$

Das ist genau die Relation, in der die Anweisungsvektoren in der Implikation der Definition 20 stehen.

Durch die Definition von S gilt $R \subseteq S$. Noch zu zeigen ist $S \subseteq \approx_D^{dr}$. Damit folgt $R \subseteq \approx_D^{dr}$.

Seien $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ mit $\vec{V} S \vec{V'}$ gegeben. Nach Definition ist S symmetrisch und $\vec{V} = (C_1, \ldots, C_k), \vec{V'} = (C'_1, \ldots, C'_k)$ sind gleichlang. Seien $i \in \{1, \ldots, k\}, s, s', t, \vec{W} = (C^*_1, \ldots, C^*_n)$ mit $s =^{\mathcal{H}}_D s'$ und $\langle\!|C_i, s|\!\rangle \twoheadrightarrow \langle\!|\vec{W}, t|\!\rangle$ gegeben. Es ist zu zeigen:

$$\exists \vec{W'}, t' : \langle\!|C'_i, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W'}, t'|\!\rangle \wedge \vec{W} S \vec{W'} \wedge t =^{\mathcal{H}}_D t'$$

Die Aussage lässt sich durch Induktion über den $(R \cup \approx^{dr}_D)^+$-Abstand m von C_i und C'_i zeigen, wobei m die kleinste Länge einer Folge $(E_0, \ldots, E_m)$ mit $E_0 = C_i, E_m = C'_i$ und $\forall j \in \{0, \ldots, m-1\} : E_j (R \cup \approx^{dr}_D) E_{j+1}$ ist. Diese existiert, da $C_i (R \cup \approx^{dr}_D)^+ C'_i$. Die ersten beiden Fälle sind der Induktionsanfang:

$m = 1, C_i R C'_i$: Die Aussage folgt direkt aus der Definition 20.

$m = 1, C_i \approx^{dr}_D C'_i$: Nach der Definition von $\approx^{dr}_D$ gilt $\exists \vec{W'}, t' : \langle\!|C'_i, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W'}, t'|\!\rangle \wedge \vec{W} \approx^{dr}_D \vec{W'} \wedge t =^{\mathcal{H}}_D t'$. Da $\approx^{dr}_D \subseteq S$, gilt auch $\vec{W} S \vec{W'}$.

$\exists t'', \vec{W''} : (\langle\!|E_{m-1}, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W''}, t''|\!\rangle \wedge \vec{W} S \vec{W''} \wedge t =^{\mathcal{H}}_D t''), E_{m-1} R C'_i$: Da $s' =^{\mathcal{H}}_D s'$ gilt nach Definition 20: $\exists \vec{W'}, t' : \langle\!|C'_i, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W'}, t'|\!\rangle \wedge \vec{W''} S \vec{W'} \wedge t'' =^{\mathcal{H}}_D t'$. Da S und $=^{\mathcal{H}}_D$ transitiv sind, gilt auch $\vec{W} S \vec{W'}$ und $t =^{\mathcal{H}}_D t'$.

$\exists t'', \vec{W''} : (\langle\!|E_{m-1}, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W''}, t''|\!\rangle \wedge \vec{W} S \vec{W''} \wedge t =^{\mathcal{H}}_D t''), E_{m-1} \approx^{dr}_D C'_i$: Da $s' =^{\mathcal{H}}_D s'$ folgt aus der Definition von $\approx^{dr}_D$: $\exists \vec{W'}, t' : \langle\!|C'_i, s'|\!\rangle \twoheadrightarrow \langle\!|\vec{W'}, t'|\!\rangle \wedge \vec{W''} \approx^{dr}_D \vec{W'} \wedge t'' =^{\mathcal{H}}_D t'$. Da $\approx^{dr}_D \subseteq S$, gilt auch $\vec{W''} S \vec{W'}$. Da S und $=^{\mathcal{H}}_D$ transitiv sind, gilt auch $\vec{W} S \vec{W'}$ und $t =^{\mathcal{H}}_D t'$.

□

Es folgen die Zusammensetzbarkeitseigenschaften.

Lemma 13. *Seien eine starke D-Bisimulation[dr]* $\approx^{dr}_D$ *und MWL-Programme bzw. Programmvektoren so gegeben, dass* $C_1 \approx^{dr}_D C'_1$, $C_2 \approx^{dr}_D C'_2$ *und* $\vec{C} \approx^{dr}_D \vec{C'}$. *Dann gilt auch*

1. $C_1 \,;\, C_2 \;\approx^{dr}_D\; C'_1 \,;\, C'_2$
2. $\textsf{fork}(C_1 \vec{C}) \;\approx^{dr}_D\; \textsf{fork}(C'_1 \vec{C'})$
3. $[\forall s =^{\mathcal{H}}_D s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle] \Rightarrow$
   a) $\textsf{if}\ B\ \textsf{then}\ C_1\ \textsf{else}\ C_2 \;\approx^{dr}_D\; \textsf{if}\ B\ \textsf{then}\ \; C'_1\ \textsf{else}\ \; C'_2$
   b) $\textsf{while}\ B\ \textsf{do}\ C_1 \;\approx^{dr}_D\; \textsf{while}\ B\ \textsf{do}\ C'_1$

Beweis: Es wird jeweils eine Relation R mit den Voraussetzungen definiert und gezeigt, dass diese eine *D-bisimulation up to* $\approx^{dr}_D$ ist. Mit Proposition 12 folgt dann $R \subseteq \approx^{dr}_D$.

Für 1.: Sei $R = \{(C_1;C_2, C_1';C_2') | C_1 \approx_D^{dr} C_1' \wedge C_2 \approx_D^{dr} C_2'\}$. Es gelte $C_1;C_2 R C_1';C_2'$, $s =_D^{\mathcal{H}} s'$ und $\langle\!| C_1;C_2, s |\!\rangle \rightarrowtriangle \langle\!| (C_1^*, \ldots, C_n^*), t |\!\rangle$ für ein $n \in \mathbb{N}$. Es ist zu zeigen:

$$\exists (C_1'^*, \ldots, C_n'^*), t' :$$
$$\langle\!| C_1';C_2', s' |\!\rangle \rightarrowtriangle \langle\!| (C_1'^*, \ldots, C_n'^*), t' |\!\rangle \wedge t =_D^{\mathcal{H}} t' \wedge \forall i \in \{1, \ldots, n\} : C_i^* (R \cup \approx_D^{dr})^+ C_i'^*$$

Nach der Semantik (s. Abbildung 4) gibt es zwei Fälle:

Fall 1 ($\langle\!| C_1, s |\!\rangle \rightarrowtriangle \langle\!| (), t |\!\rangle$): Hier gilt $(C_1^*, \ldots, C_n^*) = C_2$.
Da $C_1 \approx_D^{dr} C_1'$ gibt es ein t' mit $\langle\!| C_1', s' |\!\rangle \rightarrowtriangle \langle\!| (), t' |\!\rangle$ und $t =_D^{\mathcal{H}} t'$. Nach der Semantik gilt $\langle\!| C_1';C_2', s' |\!\rangle \rightarrowtriangle \langle\!| C_2', t' |\!\rangle$. Da $C_2 \approx_D^{dr} C_2'$, erfüllen C_2' und t' die zu zeigende Bedingung.

Fall 2 ($\langle\!| C_1, s |\!\rangle \rightarrowtriangle \langle\!| C_3\vec{D}, t |\!\rangle$): Hier gilt $(C_1^*, \ldots, C_n^*) = (C_3;C_2)\vec{D}$.
Da $C_1 \approx_D^{dr} C_1'$ gibt es $t', C_3', \vec{D'}$ mit $\langle\!| C_1', s' |\!\rangle \rightarrowtriangle \langle\!| C_3'\vec{D'}, t' |\!\rangle$, $C_3\vec{D} \approx_D^{dr} C_3'\vec{D'}$ und $t =_D^{\mathcal{H}} t'$. Nach der Semantik gilt $\langle\!| C_1';C_2', s' |\!\rangle \rightarrowtriangle \langle\!| (C_3';C_2')\vec{D'}, t' |\!\rangle$. Da $C_3 \approx_D^{dr} C_3'$ und $C_2 \approx_D^{dr} C_2'$, gilt $C_3;C_2 R C_3';C_2'$. Da auch $\vec{D} \approx_D^{dr} \vec{D'}$, erfüllen $(C_3';C_2)\vec{D}$ und t' die zu zeigende Bedingung.

Für 2.: Sei $R = \{(\mathsf{fork}(C_1\vec{C}), \mathsf{fork}(C_1'\vec{C'})) | C_1 \approx_D^{dr} C_1' \wedge \vec{C} \approx_D^{dr} \vec{C'}\}$. Es gelte $\mathsf{fork}(C_1\vec{C})\, R\, \mathsf{fork}(C_1'\vec{C'})$ und $s =_D^{\mathcal{H}} s'$. Nach der Semantik gilt $\langle\!| \mathsf{fork}(C_1\vec{C}), s |\!\rangle \rightarrowtriangle \langle\!| C_1\vec{C}, s |\!\rangle$ und $\langle\!| \mathsf{fork}(C_1'\vec{C'}), s' |\!\rangle \rightarrowtriangle \langle\!| C_1'\vec{C'}, s' |\!\rangle$. Da $C_1 \approx_D^{dr} C_1' \wedge \vec{C} \approx_D^{dr} \vec{C'}$, ist R eine *D-bisimulation up to* $\approx_D^{dr}$.

Für 3.a): Sei

$$R = \{(\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, \textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2') \mid$$
$$C_1 \approx_D^{dr} C_1' \wedge C_2 \approx_D^{dr} C_2' \wedge \forall s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle\}.$$

Es gelte $(\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2) R (\textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2')$ und $s =_D^{\mathcal{H}} s'$. Dann gilt $\langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle$. Sei $\langle\!| B, s |\!\rangle \downarrow \mathit{true}$, dann gilt auch $\langle\!| B, s' |\!\rangle \downarrow \mathit{true}$ (die Argumentation für $\langle\!| B, s |\!\rangle \downarrow \mathit{false}$ läuft analog). Dann gilt nach der Semantik

$$\langle\!| \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s |\!\rangle \rightarrowtriangle \langle\!| C_1, s |\!\rangle \text{ und}$$

$$\langle\!| \textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2', s' |\!\rangle \rightarrowtriangle \langle\!| C_1', s' |\!\rangle.$$

Da $C_1 \approx_D^{dr} C_1'$ ist R eine *D-bisimulation up to* $\approx_D^{dr}$.

Für 3.b): Sei

$$R = \{(C_1;\textsf{while } B \textsf{ do } C_2, C_1';\textsf{while } B \textsf{ do } C_2') |$$
$$C_1 \approx_D^{dr} C_1' \wedge C_2 \approx_D^{dr} C_2' \wedge \forall s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle\}$$
$$\cup \{(\textsf{while } B \textsf{ do } C, \textsf{while } B \textsf{ do } C') | C \approx_D^{dr} C' \wedge \forall s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle.\}$$

Für Elemente der ersten Menge lässt sich der Beweis analog zu 1. führen, wobei für die zweite Anweisung der Sequenz die Relation R statt $\approx_D^{dr}$ gilt. Für Elemente der

zweiten Menge lässt sich der Beweis analog zu 3.a) führen. Dabei nutzt man für den Fall $\langle\!|B, s|\!\rangle \downarrow true$ die erste Menge und für den Fall $\langle\!|B, s|\!\rangle \downarrow false$, dass $() \approx_D^{dr} ()$.

Dieser Beweis ist zu großen Teilen aus [MS04] übernommen, wo die Zusammensetzbarkeitseigenschaften für $\approx_D^{in}$ gezeigt werden.
□

Es folgende die Zusammensetzbarkeitseigenschaften bezüglich der starken Sicherheit[dr].

Satz 14. *Wenn die Programme C_1, C_2 und der Programmvektor $\vec{C}$ stark sicher[dr] sind, dann auch:*

1. $C_1;C_2$
2. **fork**$(C_1\vec{C})$
3. **while** B **do** C_1, *falls es die Sicherheitsdomäne low gibt und falls gilt* $\forall s =_{low}^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$
4. **if** B **then** C_1 **else** C_2, *falls* $\forall D \in \mathcal{D} : (\exists s =_D^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx_D^{dr} C_2$

Beweis: 1. und 2. folgen direkt aus Lemma 13. Für 3. verwendet man, dass

$$\forall D \in \mathcal{D} : s =_D^{\mathcal{H}} s' \Rightarrow s =_{low}^{\mathcal{H}} s'.$$

Daher gilt der Vorrausetzung von 3. $\forall D \in \mathcal{D} : \forall s =_D^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$. Das erfüllt für alle $D \in \mathcal{D}$ die Bedingung von Lemma 13.3, also kann man dieses ebenfalls anwenden.

Für 4. zeigt man, dass

$$\begin{aligned} R = \{(&\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2) \mid \\ &C_1, C_2 \text{ sind stark sicher[dr] und} \\ &\forall D \in \mathcal{D} : (\exists s =_D^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx_D^{dr} C_2\} \end{aligned}$$

für alle $D \in \mathcal{D}$ eine *bisimulation up to* $\approx_D^{dr}$ ist. Dann folgt mit Lemma 12 $R \subseteq \approx_D^{dr}$.

Es gelte $(\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2) R (\textsf{if } B \textsf{ then} C_1' \textsf{else } C_2')$, das heißt $C_1' = C_1, C_2' = C_2$. Sei $D \in \mathcal{D}$ beliebig. Zu unterscheiden sind zwei Fälle:

Fall 1 ($\forall s =_D^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$): Damit ist die Bedingung von Lemma 13.3 erfüllt und man kann es mit $C_1 = C_1', C_2 = C_2'$ anwenden, um die Behauptung zu beweisen.

Fall 2 ($\exists s =_D^{\mathcal{H}} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle$): Dann gilt nach Definition $C_1 \approx_D^{dr} C_2$. Da C_1 und C_2 stark sicher[dr] sind, gilt auch $C_1 \approx_D^{dr} C_1$ und $C_2 \approx_D^{dr} C_2$.

Seien $s =_D^{\mathcal{H}} s'$ beliebig. Nach der Semantik gilt

$$\langle\!|\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s|\!\rangle \rightarrowtriangle \langle\!|C, s|\!\rangle \text{ und}$$

$$\langle\!|\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, s'|\!\rangle \rightarrowtriangle \langle\!|C', s'|\!\rangle$$

mit $C \in \{C_1, C_2\}$ und $C' \in \{C_1, C_2\}$. Da $C_1 \approx_D^{dr} C_2$, $C_1 \approx_D^{dr} C_1$ und $C_2 \approx_D^{dr} C_2$, gilt für alle $C, C' \in \{C_1, C_2\}$: $C \approx_D^{dr} C'$. Da auch $s =_D^{\mathcal{H}} s'$, ist R eine *D-bisimulation up to* $\approx_D^{dr}$.

□

Die Zusammensetzbarkeitseigenschaften werden im Abschnitt 4.3 genutzt, um die Sicherheit von Programmen zu überprüfen.

Starke Sicherheit[dr] und starke Sicherheit[in] haben als Sicherheitsbedingungen für Programm mit Nebenläufigkeit eine weitere gemeinsame Eigenschaft, die in [MS04] in einer Seitenbemerkung erwähnt wird. Für stark sichere Anweisungen (bis auf If- und While-Anweisungen mit konstanten Bedingungen) müssen auch alle Teilanweisungen stark sicher sein. Dagegen stellen übliche Sicherheitsbedingungen für sequentielle Programme nicht diese Forderung, zum Beispiel die *delimited release*-Sicherheitsbedingung aus [SM04]. Als Beispiel wird h:=0; l:= h genannt. Dieses Programm erscheint als sequentielles Programm sicher. Aber ein paralleler Thread könnte den Wert von h ändern.

Für die starken Sicherheit[dr] bedeutet das auch, dass ein Programm die zu deklassifizierende Information nicht über mehrere Schritte aus vertraulichen Ausdrücken zusammensetzen kann, wenn es dabei Zwischenwerte in vertraulichen Bezeichnern speichert. Denn diesen könnte ein anderer Thread vertrauliche Information zuweisen. Zum Beispiel ist h:= h1; h:= h+ h2; l:= h mit der *escape-hatch* h1+ h2 nicht stark sicher[dr].

Diese Einschränkung und weitere Eigenschaften zeigen sich anhand der schon bekannten Beispiele:

Beispiel 6 (Auszählung einer Wahl)**.** Man betrachte das Programm aus Beispiel 2 (s. Abbildung 9 für das Programm und die MLS-Politik) mit den *escape-hatches*:

$$\mathcal{H} = \{ (\textit{public}, (\text{vote1} \,\&\&\, \text{vote2}) \,||\, (\text{vote2} \,\&\&\, \text{vote3}) \,||\, (\text{vote1} \,\&\&\, \text{vote3})) \}$$

Intuitiv erfüllt das Programm die fachliche Anforderung „das Programm darf nur die *escape-hatch* in $\mathcal{H}$ deklassifizieren“. Denn wenn [pub_result := result] ausgeführt wird, hat result gerade den Wert dieser *escape-hatch*. Da aber result kein *escape-hatch* ist, ist diese Anweisung nicht stark sicher[dr]. Selbst wenn man result den *escape-hatches* hinzufügt, ist die Sicherheitsbedingung verletzt:

```
if (yes_count > 1) then
   result := true
else
   result := false
end
```

In die *escape-hatch* result fließt vertrauliche Information des Zwischenergebnisses yes_count. Dieses Zwischenergebnis könnte eine Zuweisung in einem parallelen Thread verändern.

Das untenstehende Programm führt die Auszählung in einem Ausdruck durch. Dadurch ist dieses Programm sicher, wenn zusätzlich result ein *escape-hatch* ist.

```
// Berechnung des Ergebnisses
result := ( vote1 && vote2 ) || ( vote2 && vote3 ) ||
          ( vote1 && vote3 );
//Veroeffentlichung des Ergebnisses
[ pub_result := result ]
```

Die noch kürze Version des Programms,

```
    // Berechnung und Ver"offentlichung des Ergebnisses
pub_result := ( vote1 && vote2 ) || ( vote2 && vote3 ) ||
              ( vote1 && vote3 )
```

ist auch ohne die *escape-hatch* result stark sicher[dr], allerdings ist sie nicht mehr stark sicher[in].

Die Flusspolitik (inklusive der *escape-hatches*) lässt sich in dem Sinn präzise bezüglich der fachlichen Anforderung spezifizieren, dass sie kaum mehr Informationsfluss als nötig erlaubt. Nur die *escape-hatch* result ist intuitiv unnötig.

Es fällt auf, dass die starke Sicherheit[in] zur Erfüllung der Sicherheitsanforderung nicht mehr nötig ist. Allein die Tatsache, dass nur die *escape-hatch* mit dem Ergebnis deklassifizierbar ist, erfüllt schon die Sicherheitsanforderung. Das liegt daran, dass man im Beispiel 2 die starke Sicherheit[in] mit wenig Erfolg benutzt, um durchzusetzen welche Information (also „was“) das Programm deklassifiziert. Das Programm result:= vote1;[pub_result:= result], das stark sicher[in] ist, ist nicht stark sicher[dr], denn die erste Anweisung verletzt die Bedingung.

Beispiel 7 (Freigabe nach zweifacher Überprüfung). Man betrachte das Programm aus Beispiel 3 (s. Abbildung 10 für das Programm und die MLS-Politik), welches stark sicher[in] ist. Die Frage ist, wie sich die starke Sicherheit[dr] bei diesem Programm anwenden lässt. Die Deklassifikation der Bezeichner dieses Programms lässt sich kaum einschränken. Folgendes ist eine mögliche *escape-hatch*-Menge, mit der das Programm stark sicher[dr] ist:

$$\begin{aligned}\mathcal{H} = \{&(\mathit{public}, \mathsf{p\_report1}), (\mathit{public}, \mathsf{approve1}),\\ &(\mathit{public}, \mathsf{approve2}), (\mathit{public}, \mathsf{p\_report2}),\\ &(\mathit{public}, \mathsf{cond1}), (\mathit{public}, \mathsf{cond2}),\\ &(\mathit{public}, \mathsf{report1}), (\mathit{public}, \mathsf{report2}),\\ &(\mathit{public}, \mathsf{sec\_report})\}\end{aligned}$$

Das sind alle in dem Programm auftretende Bezeichner. Es lässt sich auch keine der *escape-hatches* entfernen oder ersetzen. Das Problem ist, dass alle Variablenbezeichner für Zwischenwerte auch deklassifiziert werden. Die starke Sicherheit[dr] ist hier trotzdem nicht sinnlos. Dieses Programm könnte Teil eines größeren Programms sein. Es könnte noch andere Bezeichner in der Sicherheitsdomäne *secret* geben, deren Deklassifikation die starke Sicherheit[dr] verhindert. Eine weitere Änderung des

Programms, mit der die starke Sicherheit[dr] noch sinnvoller wäre, ist cond1 und cond2 durch komplexere Ausdrücke zu ersetzen. Die Einzelwerte der Bezeichner, die in den Wert dieser Ausdrücke einfließen, sind damit geschützt.

Insgesamt ist die starke Sicherheit[in] hier wichtiger als im Beispiel 6, denn hier beschränken die fachlichen Sicherheitsanforderungen, wo die Information fließen darf.

4.2.4. Verletzung der Konservativität

In [SS05] werden einige Prinzipien vorgeschlagen, die Deklassifikationskonzepte erfüllen sollten. Während die starke Sicherheit[dr] die anderen erfüllt[18], verletzt sie in gewisser Weise das Prinzip der Konservativität (*conservativity*). Dieses Prinzip ist so formuliert: „Sicherheit von Programmen ohne Deklassifikation ist äquivalent zur *noninterference*“. Dabei ist mit *noninterference* eine Sicherheitsbedingung ohne Deklassifikation gemeint.

Angewandt auf die starke Sicherheit[dr] gibt es zwei Interpretationen dieser Aussage. Die eine ist, Deklassifikation als Verletzung der starken Sicherheit zu verstehen. Damit besagt dieses Prinzip, dass jedes stark sichere Programm auch stark sicher[dr] sein muss. Allerdings ist nach der Definition der starken D-Bisimulation[dr] gegenüber der normalen starken D-Bisimulation der Informationsfluss in die *escape-hatches* eingeschränkt. Die Einschränkung gilt unabhängig davon, ob das Programm diese *escape-hatches* tatsächlich deklassifiziert.

Es ist normalerweise nicht notwendig Programme zu schreiben, bei denen das Problem auftritt. Statt zum Beispiel l:= h; h:= h1; h2:= h zu schreiben, wobei h die einzige *escape-hatch* ist, könnte man auch h3:= h; l:= h; h3:= h1; h2:= h3 schreiben, wobei h3 ein neuer Bezeichner ist.

Für praktische Anwendungen kann man auch die Disjunktion der starken Sicherheit und der starken Sicherheit[dr] als Sicherheitsbedingung verwenden, diese erfüllt die Konservativität. Man muss dabei aber beachten, dass diese Bedingung nicht die Zusammensetzbarkeitseigenschaften erfüllt. Zum Beispiel ist h1:= h2 stark sicher und l:= h1 stark sicher[dr], wenn es die einzige *escape-hatch* h1 gibt. Aber h1:= h2; l:= h1 erfüllt keine der beiden Sicherheitsbedingungen.

Die andere Interpretation der Konservativität ist, dass „ohne Deklassifikation“ im Fall der starken Sicherheit[dr] eine leere *escape-hatch*-Menge bedeutet. Mit dieser Interpretation besagt der Satz 11, dass die starke Sicherheit[dr] die Konservativität erfüllt.

4.2.5. Kompatibilität zur starken Sicherheit[in]

Hier wird die Konjunktion der starken Sicherheit[dr] und der starken Sicherheit[in] betrachtet. Ein Problem dabei ist, dass die Deklassifikationsanweisungen nur die Deklassifikation einzelner Variablenbezeichner erlauben. Das heißt, um tatsächlich

[18] Diese sind so formuliert, dass Sicherheitsbedingungen für das „was“ der Deklassifikation normalerweise die Prinzipien erfüllen (s. [SS05]), so auch die starke Sicherheit[dr]. Um das allerdings exakt zu überprüfen müsste man zunächst die Prinzipien als formale Bedingungen definieren.

Deklassifikation zu erlauben, benötigt man Variablenbezeichner als *escape-hatches*. Statt mit der *escape-hatch* h1 + h2 das Programm [l := h1 + h2] zu schreiben, muss man h := h1 + h2; [l := h] schreiben und die *escape-hatch* h hinzufügen. Zum Beispiel tritt das Problem im Beispiel 6 auf, wo ein Programm geschrieben werden soll, dass nach beiden Bedingungen sicher ist.

Dieses Vorgehen wird unten berücksichtigt, wenn ein Sicherheitstypsystem erstellt wird, das Programme auf starke Sicherheit[dr] überprüft. Die einfachste Möglichkeit das Verbot des zusätzlichen Informationsflusses in *escape-hatches* durchzusetzten ist, keine Zuweisung an Bezeichner zu erlauben, die in einer *escape-hatch* vorkommen. Diese Möglichkeit wird zum Beispiel in [SM04] genutzt, um die dort definierte *delimited-release*-Bedingung zu überprüfen. Die damit verbundene Einschränkung ist für sich nicht zu restriktiv, denn statt den Bezeichner mit der ihm zugewiesenen Information zu deklassifizieren, könnte man auch direkt diese Information deklassifizieren. Doch in Kombination mit der starken Sicherheit[in] ist das nicht möglich. Wie oben gezeigt muss hier Information aus *escape-hatches* in andere *escape-hatches* fließen, die aus einem einzelnen Bezeichner bestehen.

Die starke Sicherheit[in] und die starke Sicherheit[dr] ließen sich noch besser kombinieren, wenn man eine andere Deklassifikationsanweisung ergänzen würde, zum Beispiel $[Id{:=}Exp]$. Der Abschnitt 4.4.2 führt dazu Überlegungen auf.

Hier wird ein Beispiel aufgeführt, in dem beide Sicherheitsbedingungen genutzt werden.

Beispiel 8 (Virusüberprüfung einer E-Mail (Abbildung 28))**.** Das Programm in Abbildung 28(b) modelliert folgenden Ablauf. Ein Virusfilter (Sicherheitsdomäne *filter*) überprüft eine E-Mail (Variablenbezeichner mail) darauf, ob sie einen Virus enthält (repräsentiert durch die Bedingung mail==23 für „E-Mail enthält einen Virus“). Falls nicht, leitet er die E-Mail an den Leser (Sicherheitsdomäne *reader*) weiter. Das Prüfergebnis (Ausdruck mail!=23) veröffentlicht er, sodass zum Beispiel öffentliche Virusstatistiken erstellt werden können.

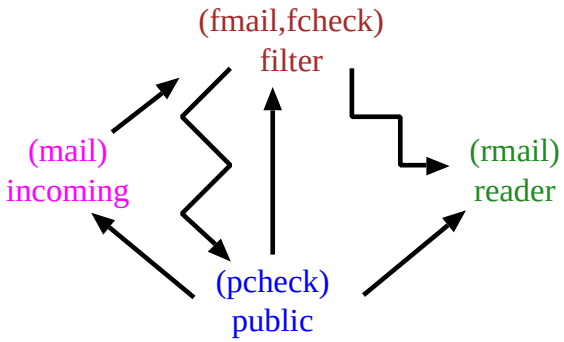
Die MLS-Politik aus Abbildung 28(a) stellt mit der intransitiven Ausnahmeflussrelation sicher, dass Information über die E-Mail nur durch den Filter zu dem Leser und zu der Öffentlichkeit fließen. Der EIFA liefert unter Anwendung des Sicherheitstypsystems für starke Sicherheit[in] (s. Abschnitt 2.4) das Ergebnis, dass das Programm mit dieser MLS-Politik stark sicher[in] ist.

Um einzuschränken, „was“ deklassifiziert wird, benötigt man eine *escape-hatch* Menge. Diese muss (*public*, mail!=23) und (*reader*, mail) enthalten, da dieser Informationsfluss gefordert ist. Zusätzlich muss sie (*public*, fcheck) und (*reader*, fmail) enthalten, damit diese Informationen durch die Deklassifikationsanweisungen fließen können.

Also sei die *escape-hatch*-Menge:

$$\mathcal{H} = \{(public,\ \text{mail!=23}), (public,\ \text{fcheck}), (reader,\ \text{mail}), (reader,\ \text{fmail})\}.$$

Mit $\mathcal{H}$ ist das Programm stark sicher[dr]. Das lässt sich mit einem Sicherheitstypsystem zeigen (s. Beispiel 9), das im nächsten Abschnitt definiert wird.



(a) MLS-Politik mit Ausnahmen

```
// Ueberpruefung, ob E-Mail keinen Virus enthaelt
// mail==23 steht fuer "E-Mail enthaelt Virus"
 fcheck := mail != 23;

// Veroeffentlichung des Pruefergebnisses
[ pcheck := fcheck ];

// Falls kein Virus gefunden wurde,
// wird die Mail weitergeleitet
if fcheck then
   fmail := mail
else
   fmail := 0
end;
[ rmail := fmail ]
```

(b) MWL-Programm

Abbildung 28: Virusüberprüfung einer E-Mail (Beispiel 8). Ein Virusfilter überprüft eine E-Mail darauf, ob sie einen Virus enthält. Falls nicht, leitet er sie an den Leser weiter. Das Prüfergebnis veröffentlicht er. 28(a) stellt die Flusspolitik und die Domänenzuweisung der Bezeichner dar. Die Bezeichner stehen in Klammern an der jeweiligen Sicherheitsdomäne. Dieses Beispiel ist stark sicher[in]. Mit den *escape-hatches* $\mathcal{H}$ (s. Beispiel 8) ist es stark sicher[dr].

4.3. Automatische Analyse mit einem Sicherheitstypsystem

Hier wird ein Sicherheitstypsystem definiert, mit dem man die starke Sicherheit[dr] von MWL-Programmen zeigen kann. Dabei werden die Überlegungen aus dem Abschnitt 4.2.5 berücksichtigt.

4.3.1. Sicherheitstypsystem

Wie im Abschnitt 4.2.5 erläutert, wäre es zu restriktiv, keine Zuweisungen an Bezeichner aus der *escape-hatch*-Menge zu erlauben. Denn dann kann man kaum sinnvolle Programme schreiben, die mit diesem Sicherheitstypsystem typisierbar sind und starke Sicherheit[in] erfüllen. Das Sicherheitstypsystem lässt daher unter bestimmten Bedingungen die Zuweisung an Bezeichnern zu, die zwar als *escape-hatch* vorkommen, aber nicht in einem zusammengesetzten *escape-hatch*-Ausdruck.

Sicherer Datenfluss Eine Kernfrage, die man sich bei der Erstellung eines Sicherheitstypsystems stellen muss, ist, unter welcher Bedingung Information von einem Ausdruck in einen Bezeichner fließen darf, ohne dass dieser Fluss die Sicherheitsbedingung verletzt. Denn zumindest ist diese Bedingung die Vorbedingung der Zuweisungsregel. Das Sicherheitstypsystem zur Überprüfung auf normale starke Sicherheit hat hier die Bedingung $\exists D : \Gamma \vdash Exp : D \wedge D \leq \Gamma(Id)$. Wenn man die Regel [If] mit der nicht-D-sichtbaren Gleichheit als sichere Approximationsrelation betrachtet, spielt diese Bedingung auch dort eine Rolle: Zuweisungen in einem Zweig sind nur dann durch **skip** im anderen Zweig austauschbar, wenn der Bezeichner Id und die Verzweigungsbedingung B die Bedingung $\exists D : \Gamma \vdash B : D \wedge D \leq \Gamma(Id)$ erfüllen.

Damit ist diese syntaktische Flussbedingung als eine Art Parameter identifiziert. Ähnliche Überlegungen werden schon bei der Implementierung des EIFA angestellt. Die Klasse policy.ProgramSecPolicy implementiert solche syntaktischen Bedingungen (s. Abschnitt 3.3.2).

Hier wird dieser Parameter geändert, um aus dem Sicherheitstypsystem zur Überprüfung auf starke Sicherheit eines zur Überprüfung auf starke Sicherheit[dr] zu gewinnen, ohne strukturell das Sicherheitstypsystem zu ändern. Gegenüber einer vollständig neuen Definition des Sicherheitstypsystems ermöglicht dieses Vorgehen

- eine einfachere Definition, denn man benötigt nur vier neue Regeln (s.u. die Abbildung 29, dort fünf Regeln, da [Assign] und [Declass] getrennt sind),
- eine einfache Kombination des neuen Sicherheitstypsystems mit dem für die starke Sicherheit[in] und
- eine leichtere Erweiterung des EIFA um das neue Sicherheitstypsystem (s. Abschnitt 4.3.2).

Das heißt man formuliert zunächst die Bedingung für Zuweisungen. Zur Definition einer praktisch anwendbaren Regel für If-Anweisungen nutzt man wieder eine Approximationsrelation, bei der im Prinzip gegenüber der nicht-D-sichtbaren Gleichheit nur die obige Flussbedingung ersetzt ist (s. Definition 23).

Die Bedingung für die Zuweisung an Bezeichner wird durch folgende Menge definiert:

Definition 21. Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit *escape-hatches* $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Dann ist

$$\mathcal{Z} = \{(Id, Exp) | Id \text{ kommt in keinem zusammengesetzten Ausdruck aus } \mathcal{H} \text{ vor} \\ \wedge\ \forall D \in \mathcal{D} : (D = \Gamma(Id) \vee (D, Id) \in \mathcal{H}) \Rightarrow \\ (\forall s =_D^{\mathcal{H}} s' \Rightarrow \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle)\}$$

Es gelte $\Gamma \vdash Exp : D$ für MWL-Ausdrücke Exp und $D \in \mathcal{D}$ nach den Regeln aus Abbildung 11. Dann ist

$$\mathcal{Z}_{\vdash} = \{(Id, Exp) | Id \text{ kommt in keinem zusammengesetzten Ausdruck aus } \mathcal{H} \text{ vor} \\ \wedge\ \forall D \in \mathcal{D} : (D = \Gamma(Id) \vee (D, Id) \in \mathcal{H}) \Rightarrow \\ (\exists D' \in \mathcal{D} : (\Gamma \vdash Exp : D' \vee (D', Exp) \in \mathcal{H}) \wedge D' \leq D)\}$$

Proposition 15. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit escape-hatches $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Es gelte $\Gamma \vdash Exp : D$ für MWL-Ausdrücke Exp und $D \in \mathcal{D}$ nach den Regeln aus Abbildung 11. Es gelten*

1. $\forall D \in \mathcal{D} : (\exists D' \in \mathcal{D} : (\Gamma \vdash Exp : D' \vee (D', Exp) \in \mathcal{H}) \wedge D' \leq D) \Rightarrow$
   $(\forall s =_D^{\mathcal{H}} s' : \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle)$
2. $\mathcal{Z}_{\vdash} \subseteq \mathcal{Z}$.

Beweis: Die zweite Aussage folgt direkt aus der ersten. Zu zeigen ist noch die erste Aussage. Seien $D \in \mathcal{D}$ und ein MWL-Ausdruck Exp beliebig. Es lassen sich zwei Fälle unterscheiden, in denen die Prämisse zutrifft:

Fall 1 ($\exists D' \in \mathcal{D} : \Gamma \vdash Exp : D' \wedge D' \leq D$): Nach $=_D^{\mathcal{H}} \subseteq =_D$ und Lemma 6 gilt $\forall s =_{D'}^{\mathcal{H}} s' : \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$). Da mit $D' \leq D$ auch $\forall s =_D^{\mathcal{H}} s' : s =_{D'}^{\mathcal{H}} s'$, gilt $\forall s =_D^{\mathcal{H}} s' : \langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$.

Fall 2 ($\exists D' \in \mathcal{D} : (D', Exp) \in \mathcal{H} \wedge D' \leq D$): Seien $s =_D^{\mathcal{H}} s'$ beliebig. nach der Definition von $=_D^{\mathcal{H}}$ gilt $\langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$.

□

$\mathcal{Z}_{\vdash}$ ist unabhängig von den konkret möglichen Operatoren entscheidbar. Ein Algorithmus, der für jedes $D \in D$ eine *escape-hatch* mit kleinerem $D' \in \mathcal{D}$ sucht und $\Gamma \vdash Exp : D'$ überprüft, ist einfach zu erstellen.

Folgende Aussagen und das Sicherheitstypsystem nutzen die Menge $\mathcal{Z}$. Mit $\mathcal{Z}_{\vdash}$ wird $\mathcal{Z}$ approximiert. Da $\mathcal{Z}$ hier nur in Prämissen auftaucht, lässt sie sich hier immer durch $\mathcal{Z}_{\vdash}$ ersetzen. Damit kann man das Sicherheitstypsystem unabhängig von den konkret möglichen Operationen anwenden.

Folgendes Lemma zeigt, wie $\mathcal{Z}$ als Bedingung für eine Sicherheitstypregel nützlich sein kann. Aus den ersten beiden Aussagen ergeben sich direkt die Sicherheitstypregeln für Zuweisungen und Arrayzuweisungen. Die letzten beiden Aussagen lassen sich dazu nutzen wir, eine Approximationsrelation ähnlich der nicht-D-sichtbaren Gleichheit zu definieren.

Lemma 16. *Seien eine MLS-Politik* $(\mathcal{D}, \leq)$ *mit escape-hatches* $\mathcal{H}$ *und einer Domänenzuweisung* Γ *gegeben. Es gilt*

1. $(Id, Exp) \in \mathcal{Z} \Rightarrow Id{:=}Exp$ *ist stark sicher[dr]*
2. $(Arr, Exp_1), (Arr, Exp_2) \in \mathcal{Z} \Rightarrow Arr[Exp_1]{:=}Exp_2$ *ist stark sicher[dr]*
3. $(Id, B) \in \mathcal{Z} \wedge Id{:=}Exp$ *ist stark sicher[dr]*
   $\Rightarrow [\forall D : (\exists s =^{\mathcal{H}}_D s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow Id{:=}Exp \approx^{\mathcal{H}}_D \textsf{skip}]$
4. $(Id, B) \in \mathcal{Z} \wedge Arr[Exp_1]{:=}Exp_2$ *ist stark sicher[dr]*
   $\Rightarrow [\forall D : (\exists s =^{\mathcal{H}}_D s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow Arr[Exp_1]{:=}Exp_2 \approx^{\mathcal{H}}_D \textsf{skip}]$

Beweis:
Zu 1.: Seien $D \in \mathcal{D}$ und $s =^{\mathcal{H}}_D s'$ beliebig.
Nach der Semantik gilt

$$\langle\!| Id{:=}Exp, s |\!\rangle \rightarrowtriangle \langle\!| (), t |\!\rangle \text{ mit } t = [Id = n]s, \langle\!| Exp, s |\!\rangle \downarrow n \text{ und}$$

$$\langle\!| Id{:=}Exp, s' |\!\rangle \rightarrowtriangle \langle\!| (), t' |\!\rangle \text{ mit } t' = [Id = n']s', \langle\!| Exp, s' |\!\rangle \downarrow n'.$$

Da $() \approx^{dr}_D ()$ bleibt nur noch zu zeigen, dass $t =^{\mathcal{H}}_D t'$. Angenommen $t \neq^{\mathcal{H}}_D t'$. Nach der Definition von $=^{\mathcal{H}}_D$ (s. Definition 16) gibt es zwei Fälle.

Fall 1 ($\exists Id' : \Gamma(Id') \leq D \wedge t(Id') \neq t'(Id')$): Mit $s =^{\mathcal{H}}_D s'$ folgt $s(Id') = s'(Id')$. Falls $Id \neq Id'$, gilt $s(Id') = t(Id')$ und $s'(Id') = t'(Id')$, woraus $s(Id') \neq s'(Id')$ folgt, ein Widerspruch. Also gilt $Id = Id'$. Damit ist $n = t(Id) \neq t'(Id) = n'$, also $\langle\!| Exp, s |\!\rangle \not\approx \langle\!| Exp, s' |\!\rangle$.

Aus $(Id, Exp) \in \mathcal{Z}$ und $s =^{\mathcal{H}}_D s'$ folgt mit $\Gamma(Id) \leq D$, dass $\langle\!| Exp, s |\!\rangle \approx \langle\!| Exp, s' |\!\rangle$, ein Widerspruch.

Fall 2 ($\exists (D'', Exp') \in \mathcal{H} : D'' \leq D \wedge \langle\!| Exp', t |\!\rangle \not\approx \langle\!| Exp', t' |\!\rangle$): Falls Id in Exp' vorkommt, folgt aus $(Id, Exp) \in \mathcal{Z}$, dass $Id = Exp'$. Es gilt also $(D'', Id) \in \mathcal{H}$. Dies führt mit der Argumentation aus Fall 1 mit D'' statt $\Gamma(Id)$ zum Widerspruch. Falls Id nicht in Exp' vorkommt, gilt $\langle\!| Exp', s |\!\rangle \approx \langle\!| Exp', t |\!\rangle$ und $\langle\!| Exp', s' |\!\rangle \approx \langle\!| Exp', t' |\!\rangle$. Aus $D'' \leq D$ und $(D'', Exp) \in \mathcal{H}$ folgt mit $s =^{\mathcal{H}}_D s'$ $\langle\!| Exp', s |\!\rangle \approx \langle\!| Exp', s' |\!\rangle$. Daher folgt $\langle\!| Exp', t |\!\rangle \approx \langle\!| Exp', t' |\!\rangle$, ein Widerspruch.

Zu 2.: Die Beweisführung ist analog zu der für 1., nur dass im Fall 1 ($\langle\!| Exp_1, s |\!\rangle \not\approx \langle\!| Exp_1, s' |\!\rangle \vee \langle\!| Exp_2, s |\!\rangle \not\approx \langle\!| Exp_2, s' |\!\rangle$) und ($\langle\!| Exp_1, s |\!\rangle \approx \langle\!| Exp_1, s' |\!\rangle \wedge \langle\!| Exp_2, s |\!\rangle \approx \langle\!| Exp_2, s' |\!\rangle$) folgen, also auch ein Widerspruch. Im Fall 2 verwendet

man nur, dass es ein Exp'' mit $(Id, Exp'') \in \mathcal{Z}$ bzw. $(Arr, Exp'') \in \mathcal{Z}$ gibt, wie zum Beispiel für $Exp'' = Exp_1$

Zu 3.: Seien $D \in \mathcal{D}$ mit $\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$, u, u' beliebige Zustände mit $u =_D^{\mathcal{H}} u'$. Nach der Semantik gilt:

$$\langle\!| Id{:=}Exp, u |\!\rangle \rightarrowtriangle \langle\!| (), t |\!\rangle \text{ für den Zustand } t \text{ und}$$

$$\langle\!| \mathsf{skip}, u' |\!\rangle \rightarrowtriangle \langle\!| (), u' |\!\rangle.$$

Dabei unterscheiden sich u und t nur im Wert von Id. Da $\rightarrowtriangle$ deterministisch ist, sind diese Transitionen die einzigen Transitionen aus den beiden Konfigurationen.

Da $() \approx_D^{dr} ()$ gilt, muss nur noch $t =_D^{\mathcal{H}} u'$ gezeigt werden.

Zunächst wird $u =_D^{\mathcal{H}} t$ gezeigt, indem man $u \neq_D^{\mathcal{H}} t$ annimmt und zu einem Widerspruch führt. Da sich beide Zustände nur im Wert von Id unterscheiden und mit $(Id, B) \in \mathcal{Z}$ Id nicht in einer zusammengesetzten *escape-hatch* vorkommt, gibt es nach Definition von $=_D^{\mathcal{H}}$ eine Sicherheitsdomäne $D' \leq D$ mit $D' = \Gamma(Id)$ oder $(D', Id) \in \mathcal{H}$. Aus $(Id, B) \in \mathcal{Z}$ folgt damit $\forall s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle$, ein Widerspruch. Damit gilt $u =_D^{\mathcal{H}} t$.

Über die Transitivität von $=_D^{\mathcal{H}}$ folgt $t =_D^{\mathcal{H}} u'$.

Zu 4.: Analog zu dem Beweis für 3. da es nicht darauf ankommt, wie sich Arr bzw. Id ändert. Es kommt nur darauf an, dass die Änderung für eine Beobachter mit einem $D \in \mathcal{D}$ für das $\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$ gilt, unsichtbar ist.

□

Sicherheitstypsystem Das Sicherheitstypsystem übernimmt die Sicherheitstypregeln für Ausdrücke aus der Abbildung 11. Auch die Sicherheitstypregeln für Anweisungen übernimmt es vom Sicherheitstypsystem für die starke Sicherheit[in] (s. Abbildung 12), wobei es die Regeln [Assign],[ArrAssign], [Declass] und [If] ersetzt. Die Regeln für Deklarationen übernimmt es nicht (s. Anmerkung zu Deklarationen im Abschnitt 4.2.2).

Die neuen Sicherheitstypregeln für Anweisungen listet die Abbildung 29 auf.

Folgender Satz besagt, dass das Sicherheitstypsystem korrekt ist.

Satz 17. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit escape-hatches $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $\vec{C} \in \vec{\mathcal{C}}$. Dann gilt: $\Gamma \vdash \vec{C} \Rightarrow \vec{C}$ ist stark sicher[dr].*

Beweis: Satz 17 lässt sich durch eine Induktion über die Struktur von Anweisungen zeigen. Das heißt man zeigt, dass wenn man in den Prämissen der Regeln für $\Gamma \vdash C$ die Bedingung „C ist stark sicher[dr]“ einsetzt (Induktionsvoraussetzung), jeweils aus den Prämissen die Konklusion folgt. Man zeigt also, dass jede einzelne Regel korrekt ist.

Die ersten vier Fälle bilden den Induktionsanfang.

[Skip]: Es gilt $\forall s : \langle\!| \mathsf{skip}, s |\!\rangle \rightarrowtriangle \langle\!| (), s |\!\rangle$. Da $\forall D : () \approx_D^{dr} ()$, ist **skip** immer stark sicher[dr].

$$\textbf{[Assign]}\quad \frac{(Id, Exp) \in \mathcal{Z}}{\Gamma \vdash Id{:=}Exp}$$

$$\textbf{[Declass]}\quad \frac{(Id, Id') \in \mathcal{Z}}{\Gamma \vdash [Id{:=}Id']}$$

$$\textbf{[ArrAssign]}\quad \frac{(Arr, Exp_1) \in \mathcal{Z} \quad (Arr, Exp_2) \in Z}{\Gamma \vdash Arr\ [\ Exp_1\ ]{:=}\ Exp_2}$$

$$\textbf{[While}_{hatch}\textbf{]}\quad \frac{\Gamma \vdash (low, B) \in \mathcal{H} \qquad \Gamma \vdash C}{\Gamma \vdash \textsf{while}\ B\ \textsf{do}\ C}$$

$$\textbf{[If]}\quad \frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2 \quad \forall D \in \mathcal{D} : (\exists s =^{\mathcal{H}}_{D} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx^{dr}_{D} C_2}{\Gamma \vdash \textsf{if}\ B\ \textsf{then}\ C_1\ \textsf{else}\ C_2}$$

Abbildung 29: Neue Sicherheitstypregeln für die starke Sicherheit[dr] von MWL-Programmen. Sie ersetzen und ergänzen die Sicherheitstypregeln aus Abbildung 12. Die Sicherheitstypregeln für Ausdrücke sind in der Abbildung 11 definiert. Γ ist eine Domänenzuweisung. Die Bedingung $C_1(R_B)C_2$ ersetzt in Implementierungen die semantische Nebenbedingung der Regel [If], wobei $\{R_{B'}\}_{B' \text{ ist MWL-Ausdruck}}$ eine sichere Approximationsrelation für *delimited release* ist (s. Definition 22).

[Assign]: Folgt direkt aus Lemma 16.1.

[ArrAssign]: Folgt direkt aus Lemma 16.2.

[Declass]: Da eine Deklassifikation sich bezüglich der Transition $\rightarrowtriangle$ wie eine Zuweisung verhält, folgt die Korrektheit aus Lemma 16.1.

[While$_{low}$]: Mit der Regelbedingung $\Gamma \vdash B : low$ folgt $\forall s =_{low} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$ nach Lemma 6. Damit, mit $=^{\mathcal{H}}_{low} \subseteq =_{low}$ und mit der Induktionsvoraussetzung für $\Gamma \vdash C$ ist die Voraussetzung für Satz 14.3 erfüllt, woraus die Korrektheit folgt.

[While$_{hatch}$]: Mit der Regelbedingung $(low, B) \in \mathcal{H}$ folgt nach der Definition von $=^{\mathcal{H}}_{D}$, dass $\forall s =^{\mathcal{H}}_{low} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$. Damit und mit der Induktionsvoraussetzung für $\Gamma \vdash C$ ist die Voraussetzung für Satz 14.3 erfüllt, woraus die Korrektheit folgt.

[Seq]: Mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash C_2$ ist die Voraussetzung für Satz 14 erfüllt, woraus die Korrektheit folgt.

[Fork]: Mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash \vec{C}_2$ ist die Voraussetzung für Satz 14 erfüllt, woraus die Korrektheit folgt.

[Par]: Korrektheit folgt mit der Induktionsvoraussetzung für $\Gamma \vdash C_0, \ldots, \vdash C_{n-1}$ direkt aus Definition von $\approx_D^{dr}$.

[If]: Mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash C_2$ und der Regelbedingung $\forall D \in \mathcal{D} : (\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow C_1 \approx_D^{dr} C_2$ ist die Voraussetzung für Satz 14.4 erfüllt, woraus die Korrektheit folgt.

□

Dieses Sicherheitstypsystem ist aufgrund der semantischen Nebenbedingung der Regel [If] nicht entscheidbar. Aber wie bei dem Sicherheitstypsystem für die starke Sicherheit[in] kann man die Bedingung approximieren (s. Definition 13).

Definition 22 (Sichere Approximationsrelation für *delimited release*). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit *escape-hatches* $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Eine Familie $\{R_B\}_{B \text{ ist MWL-Ausdruck}}$ von Relationen über Anweisungen ist dann eine *sichere Approximationrelation für delimited release*, wenn für beliebige, stark sichere[dr] MWL-Programme C und C' und einem beliebigen MWL-Ausdruck B gilt:

$$C(R_B)C' \Rightarrow \forall D \in \mathcal{D} : (\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow C_1 \approx_D^{dr} C_2$$

Als Beispiel ist analog zur nicht-D-sichtbaren Gleichheit (s. Definition 14) folgende Approximationsrelation definiert:

Definition 23 (Nicht B-sichtbare Gleichheit). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit *escape-hatches* $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Die *nicht B-sichtbare Gleichheit* $\{\sim_B\}_{B \text{ ist MWL-Ausdruck}}$ ist folgendermaßen definiert: Für jedes B ist $\sim_B$ die kleinste Kongruenzrelation über Anweisungen (transitiv, reflexiv, symmetrisch und geschlossen unter der Konstruktion der Sprache), welche die folgenden Regeln erfüllt:

$$\frac{(Id, B) \in \mathcal{Z}}{Id{:=}Exp \sim_B \textsf{skip}} \qquad \frac{(Arr, B) \in \mathcal{Z}}{Arr[Exp_1]{:=}Exp_2 \sim_B \textsf{skip}}$$

Satz 18. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ mit escape-hatches $\mathcal{H}$ und eine Domänenzuweisung Γ gegeben. Die Relationsfamilie $\{\sim_B\}_{B \text{ ist MWL-Ausdruck}}$ ist eine sichere Approximationsrelation für delimited release.*

Beweis: Sei B ein MWL-Ausdruck und seien $C, C' \in \mathcal{C}$ so, dass C, C' stark sicher[dr] sind und $C \sim_B C'$. Es werden explizit die Regeln aufgeführt, nach denen $\sim_B$ konstruiert ist und es wird durch Induktion über die kleinste Zahl der Regelanwendung zur Ableitungen von $C \sim_B C'$ gezeigt, dass $\forall D : (\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow C \approx_D^{dr} C'$. Es gibt jeweils diese kleinste Zahl, ansonsten wäre $\sim_B$ nicht die kleinste Kongruenzrelation.

Sei D beliebig mit $\exists s =_D^{\mathcal{H}} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$. Es lassen sich die Fälle nach der letzten Regel der Ableitung unterscheiden. Die ersten drei Fälle sind der Induktionsanfang.

$$\frac{(Id, B) \in \mathcal{Z}}{Id{:=}Exp \sim_B \textsf{skip}}$$

Unsichtbare Zuweisung :
Da $C = Id{:=}Exp$ stark sicher[dr] ist, $(Id, B) \in \mathcal{Z}$ und
$\exists s =^{\mathcal{H}}_D s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$, folgt $C \cong^{dr}_D C'$ nach Lemma 16.3.

$$\frac{(Arr, B) \in \mathcal{Z}}{Arr[Exp_1]{:=}Exp_2 \sim_B \textsf{skip}}$$

Unsichtbare Arrayzuweisung :
Da $C = Arr[Exp_1]{:=}Exp_2$ stark sicher[dr] ist, $(Arr, B) \in \mathcal{Z}$ und
$\exists s =^{\mathcal{H}}_D s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$, folgt $C \cong^{dr}_D C'$ nach Lemma 16.4.

$$\frac{C = C'}{C \sim_B C'}$$

Reflexivität :
Da C stark sicher[dr] ist, gilt nach Definition der starken Sicherheit[dr], dass $C \cong^{dr}_D C$.

$$\frac{C' \sim_B C}{C \sim_B C'}$$

Symmetrie :
$C \cong^{dr}_D C'$ folgt mit der Induktionsvoraussetzung für $C' \sim_B C$ direkt aus der Symmetrie von $\cong^{dr}_D$.

$$\frac{C \sim_B C'' \quad C'' \sim_B C'}{C \sim_B C'}$$

Transitivität :
$C \cong^{dr}_D C'$ folgt mit der Induktionsvoraussetzung für $C \sim_B C''$ und $C'' \sim_B C'$ direkt aus der Transitivität von $\cong^{dr}_D$.

$$\frac{C_1 \sim_B C_1' \quad C_2 \sim_B C_2'}{C_1;C_2 \sim_B C_1';C_2'}$$

Sequentielle Komp. :
Aus der Induktionsvoraussetzung für $C_1 \sim_B C_1'$ und $C_2 \sim_B C_2'$ folgt mit Lemma 13.1, dass $C_1;C_2 \cong^{dr}_D C_1';C_2'$.

$$\frac{C_1 \sim_B C_1' \quad \vec{C}_2 \sim_B \vec{C}_2'}{\textsf{fork}(C_1\vec{C}_2) \sim_B \textsf{fork}(C_1'\vec{C}_2')}$$

Parallele Komp. (für $\vec{C}_2 \sim_B \vec{C}_2'$ gilt $\sim_B$ punktweise):
Aus der Induktionsvoraussetzung für $C_1 \sim_B C_1'$ und $\vec{C}_2 \sim_B \vec{C}_2'$ folgt mit Lemma 13.2, dass $\textsf{fork}(C_1\vec{C}_2) \cong^{dr}_D \textsf{fork}(C_1'\vec{C}_2')$.

$$\frac{C_1 \sim_B C_1'}{\textsf{while } B' \textsf{ do } C_1 \sim_B \textsf{while } B' \textsf{ do } C_1'}$$

Schleifenkomp. :
Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($\forall s =^{\mathcal{H}}_D s' : \langle\!| B', s |\!\rangle \approx \langle\!| B', s' |\!\rangle$) :
Damit ist die Bedingung für Lemma 13.3 erfüllt und mit der Induktionsvoraussetzung für $C_1 \sim_B C_1'$ folgt $\textsf{while } B' \textsf{ do } C_1 \cong^{dr}_D \textsf{while } B' \textsf{ do } C_1'$.

Fall 2 ($\exists s =^{\mathcal{H}}_D s' : \langle\!| B', s |\!\rangle \not\approx \langle\!| B', s' |\!\rangle$) :
Seien s, s' diese Zustände. Sei $\langle\!| B', s |\!\rangle \downarrow true$ (der Fall $\langle\!| B', s |\!\rangle \downarrow false$ ist symmetrisch). Dann gilt $\langle\!| B', s' |\!\rangle \downarrow false$. Nach der Semantik gilt

$$\langle\!| \textsf{while } B' \textsf{ do } C_1, s |\!\rangle \rightarrowtriangle \langle\!| C_1;\textsf{while } B' \textsf{ do } C_1, s |\!\rangle \text{ und}$$

$$\langle\!|\textsf{while } B' \textsf{ do } C_1, s'|\!\rangle \rightarrowtriangle \langle\!|(), s'|\!\rangle.$$

Da C_1;**while** B' **do** C_1 und () eine unterschiedliche Anzahl an Threads haben, sind sie nicht stark D-bisimular[dr].

Da aber $C =$ **while** B' **do** C_1 stark sicher[dr] ist, ist das ein Widerspruch. Damit kann der Fall 2 garnicht eintreten.

Verzweigungskomp. $\dfrac{C_1 \sim_B C_1' \quad C_2 \sim_B C_2'}{\textsf{if } B' \textsf{ then } C_1 \textsf{ else } C_2 \sim_B \textsf{if } B' \textsf{ do } C_1' \textsf{ else } C_2'}$:

Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($\forall s =_D^{\mathcal{H}} s' : \langle\!|B', s|\!\rangle \approx \langle\!|B', s'|\!\rangle$) :
Analog zu Fall 1 für die Schleifenkomposition.

Fall 2 ($\exists s =_D^{\mathcal{H}} s' : \langle\!|B', s|\!\rangle \not\approx \langle\!|B', s'|\!\rangle$) :
Seien s, s' diese Zustände. Sei $\langle\!|B', s|\!\rangle \downarrow true$ (der Fall $\langle\!|B', s|\!\rangle \downarrow false$ ist symmetrisch). Dann gilt $\langle\!|B', s'|\!\rangle \downarrow false$. Nach der Semantik gilt

$$\langle\!|\textsf{if } B' \textsf{ then } C_1; \textsf{else } C_2, s|\!\rangle \rightarrowtriangle \langle\!|C_1, s|\!\rangle \text{ und}$$

$$\langle\!|\textsf{if } B' \textsf{ then } C_1; \textsf{else } C_2, s'|\!\rangle \rightarrowtriangle \langle\!|C_2, s'|\!\rangle.$$

Da $C =$ **if** B' **then** C_1;**else** C_2 stark sicher[dr] ist, gilt $C \approx_D^{dr} C$. Damit und da $\rightarrowtriangle$ deterministisch ist, gilt $C_1 \approx_D^{dr} C_2$. Analog folgt $C_1' \approx_D^{dr} C_2'$. Aus der Induktionsvoraussetzung für $C_1 \sim_B C_1'$ und $C_2 \sim_B C_2'$ und der Transitivität von $\approx_D^{dr}$ folgen $C_1 \approx_D^{dr} C_1'$, $C_2 \approx_D^{dr} C_2'$, $C_1 \approx_D^{dr} C_2'$ und $C_1' \approx_D^{dr} C_2$.

Seien nun $u =_D^{\mathcal{H}} u'$ beliebig. Nach der Semantik gilt

$$\langle\!|\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, u|\!\rangle \rightarrowtriangle \langle\!|C^*, u|\!\rangle \text{ und}$$

$$\langle\!|\textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2', u'|\!\rangle \rightarrowtriangle \langle\!|C'^*, u'|\!\rangle$$

mit $C^* \in \{C_1, C_2\}$ und $C'^* \in \{C_1', C_2'\}$. Da $C_1 \approx_D^{dr} C_1'$, $C_2 \approx_D^{dr} C_2'$, $C_1 \approx_D^{dr} C_2'$ und $C_2 \approx_D^{dr} C_1'$, gilt für alle $C^* \in \{C_1, C_2\}$ und $C'^* \in \{C_1', C_2'\}$: $C^* \approx_D^{dr} C'^*$. Da auch $u =_D^{\mathcal{H}} u'$, ist die Bedingung für $C \approx_D^{dr} C'$ erfüllt.

Der Beweis ist von dem Beweis für die Eignung einer beobachtbare Äquivalenz aus [KM05] inspiriert.
□

Beispiel 9 (Anwendung des Sicherheitstypsystems auf die Beispiele). Die Programme im Beispiel 6 sind genau dann durch das Sicherheitstypsystem typisierbar, wenn sie stark sicher[dr] sind. Man betrachte zum Beispiel:

```
// Berechnung des Ergebnisses
 result := ( vote1 && vote2 ) || ( vote2 && vote3 ) ||
           ( vote1 && vote3 );
//Veroeffentlichung des Ergebnisses
[ pub_result := result ]
```

Dieses Programm ist nach der Regel für die sequentielle Komposition typisierbar, wenn die beiden Anweisungen typisierbar sind. Da beide Ausdrücke, die auf der rechten Seite stehen, als *escape-hatch* nach *public* vorhanden sind und weder result noch pub_result in einem komplexen *escape-hatch*-Ausdruck vorkommen, sind auch beide Zuweisungen typisierbar.

Im Beispiel 7 wird das Programm aus Abbildung 10 mit einer *escape-hatch*-Menge $\mathcal{H}$ betrachtet, die außer dem Bezeichner pub_report genau alle im Programm vorkommenden Bezeichner mit dem Deklassifikationsziel *public* enthält. Damit ist jedes Bezeichner-Ausdruck-Paar in der Menge $\mathcal{Z}$. Da auch die Längen aller Programmzweige gleich sind, ist dieses Programm typisierbar. Allerdings ist für dieses Beispiel die Approximation $\mathcal{Z}_{\vdash}$ für $\mathcal{Z}$ nicht präzise genug. Diese erlaubt nicht die Verknüpfung von *escape-hatches* durch Operatoren. Solch eine tritt aber in der Bedingung der letzten If-Anweisung auf:

```
approve1 && approve2 && ( p_report1 == p_report2 )
```

Für das Beispiel 8 lässt sich feststellen, dass jedes Bezeichner-Ausdruck-Paar einer Zuweisung oder Deklassifikation in $\mathcal{Z}_{\vdash}$ und somit in $\mathcal{Z}$ ist. Man betrachte als ein Beispiel die erste Anweisung: fcheck := mail != 23 . fcheck kommt in keinem zusammengesetzten *escape-hatch*-Ausdruck vor. Da außerdem die *escape-hatch* (*public*, mail!=23) existiert, gilt (fcheck, mail!=23) $\in \mathcal{Z}_{\vdash}$.

Ebenfalls gilt (fmail, fcheck) $\in \mathcal{Z}_{\vdash}$, womit (fmail:= mail) $\sim_{\text{fcheck}}$ (fmail :=0) folgt. Daher ist auch die Verzweigung typisierbar. Die Typisierbarkeit des kompletten Programms ergibt sich aus der Regel für die sequentielle Komposition.

Kombination der Sicherheitstypsysteme für Deklassifikation Die einfachste Möglichkeit, die starke Sicherheit[in] und die starke Sicherheit[dr] eines Programms zu überprüfen ist, nacheinander beide Sicherheitstypsysteme anzuwenden.

Aufgrund der Ähnlichkeit der Regeln lässt sich auch einfach ein Sicherheitstypsystem erstellen, dass die Sicherheit von Programmen nach beiden Sicherheitsbedingungen zeigen kann: Die Regeln für Ausdrücke sind die aus Abbildung 11. Für Anweisungen gibt es:

- die Regeln [Skip], [While$_{low}$], [Seq], [Fork] und [Par] aus Abbildung 12, welche beide Sicherheitstypsysteme gemeinsam haben.
- die Regeln [Assign] und [Declass] aus Abbildung 12 ergänzt um die Bedingung $(Id, Exp) \in \mathcal{Z}$ bzw. $(Id, Id') \in \mathcal{Z}$.
- die Regel [ArrAssign] aus Abbildung 12 ergänzt um die Bedingungen $(Arr, Exp_1) \in \mathcal{Z}$ und $(Arr, Exp_2) \in \mathcal{Z}$
- die Regel [If] aus Abbildung 12 ergänzt um die Bedingung $\forall D \in \mathcal{D} : (\exists s =^{\mathcal{H}}_{D} s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle) \Rightarrow C_1 \approx^{dr}_{D} C_2$

Dieses Sicherheitstypsystems ist korrekt, da die beiden einzelnen Sicherheitstypsysteme korrekt sind und da die Regelbedingungen dieses Sicherheitstypsystems durch

Konjunktion der jeweiligen Regelbedingungen der einzelnen Sicherheitstypsysteme gebildet sind.

Anzumerken ist, dass die Regel [while$_{hatch}$] nicht zu dem Sicherheitstypsystem gehört, denn sie ist für die starke Sicherheit[in] nicht korrekt.

Auch die Approximationsrelationen für die semantischen Nebenbedingungen der [If]-Regel lassen sich kombinieren: die nicht-B, D-sichbare Gleichheit lässt sich genauso wie die anderen Approximationen definieren (s. Definitionen 14 und 23), nur mit den Regeln:

$$\frac{(Id, B) \in \mathcal{Z} \quad \Gamma(Id) \leq D}{Id{:=}Exp \sim_{B,D} \textsf{skip}} \qquad \frac{(Arr, B) \in \mathcal{Z} \quad \Gamma(Arr) \leq D}{Arr[Exp_1]{:=}Exp_2 \sim_{B,D} \textsf{skip}}$$

Durch die Konjunktion der Bedingungen folgt direkt: $\sim_{B,D} \subseteq \sim_B$ und $\sim_{B,D} \subseteq \sim_D$.

Insgesamt ist das Sicherheitstypsystem für die starke Sicherheit[in] (einschließlich der Approximationsrelation) nur an wenigen Stellen um die Bedingung $(Id, Exp) \in \mathcal{Z}$ ergänzt, um damit auch starke Sicherheit[dr] zu beweisen.

4.3.2. Möglichkeit der Implementierung im EIFA

Nachdem das Sicherheitstypsystem zur Überprüfung auf starke Sicherheit[dr] definiert und untersucht ist, wird hier noch gezeigt, wie man den EIFA um dieses Sicherheitstypsystem erweitern kann. Die Implementierung dieser Änderungen ist nicht im Rahmen dieser Arbeit. Dieser Abschnitt zeigt die Entwurfsänderungen.

Im Abschnitt 4.3.1 wird festgestellt, dass sich die Sicherheitstypsysteme für normale starke Sicherheit (bzw. starke Sicherheit[in]) und für starke Sicherheit[dr] nur in wenigen Bedingungen unterscheiden. Weiter wird festgestellt, dass der EIFA diese Bedingungen in der Klasse ProgramSecPolicy des Pakets policy überprüft, statt direkt in den Besucherklassen für die Sicherheitstypsysteme (s. Abschnitt 3.3.2). Daher genügt es zur Erweiterung des EIFA eine neue Klasse mit der gleichen Schnittstelle wie ProgramSecPolicy zu implementieren. Genaugenommen kann man ProgramSecPolicy zu einer Schnittstelle machen, von der man drei Klassen ableitet (s. Abbildung 30). Die Klasse ProgramInPolicy entspricht der ursprünglichen Klasse ProgramSecPolicy. Aber die Besucherklassen, welche die Sicherheitstypsysteme implementieren (s. Abschnitt 3.3.5) benutzen diese Klasse nicht mehr direkt, sondern sie greifen auf die Schnittstelle ProgramSecPolicy zu. Indem sie hier statt ProgramInPolicy die Klasse ProgramDrPolicy verwenden, überprüfen die Besucherklassen MWL-Programme auf die Erfüllung der starken Sicherheit[dr] statt der starken Sicherheit[in]. Indem sie ProgramInDrPolicy verwenden, überprüfen sie auf die Erfüllung beider Sicherheitsbedingungen.

Die Tabelle 4.3.2 stellt die von den wichtigsten Methoden überprüften Bedingungen gegenüber. intransFlow(Id', Id) entspricht der Bedingung der Regel [Declass], transFlow(Exp, Id) der Bedingung der Regel [Assign] und transToLowVar(Exp) den Bedingungen der Regeln [While$_{low}$] und [While$_{hatch}$].

Zur Vollständigkeit muss noch angemerkt werden, dass eine weitere Änderung nötig ist. Den Entwurf der Klasse, welche die Approximationsrelationen für die se-

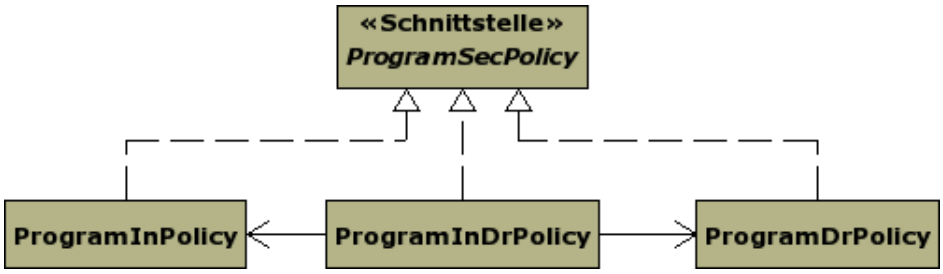


Abbildung 30: Unterteilung der syntaktischen Flusspolitikklassen zur Erweiterung des EIFA um die starke Sicherheit[dr].

	ProgramInPolicy	ProgramDrPolicy
intransFlow(Id', Id)	$\Gamma(Id') \rightsquigarrow \Gamma(Id)$	$(Id, Id') \in \mathcal{Z}$
transFlow(Exp, Id)	$\Gamma \vdash Exp : D \wedge D \leq \Gamma(Id)$	$(Id, Exp) \in \mathcal{Z}$
transToLowVar(Exp)	$\Gamma \vdash Exp : low$	$\Gamma \vdash Exp : low \vee (low, Exp) \in \mathcal{H}$

Tabelle 1: Wichtige Methoden der syntaktischen Flusspolitikklassen. Für jede Klasse-Methode-Kombination ist die jeweils durch die Methode überprüfte Bedingung aufgeführt. Die Methoden der Klasse ProgramInDrPolicy überprüfen jeweils die Konjunktion beider hier aufgeführten Bedingungen.

mantische Nebenbedingung der Regel [If] repräsentiert (diese wird in dieser Arbeit nicht beschrieben), muss man noch ändern, damit im EIFA alle Sicherheitstypsysteme korrekt implementiert sind. Denn im EIFA ist sie wie die nicht-D-sichtbare Gleichheit unabhängig von der Bedingung B einer If-Anweisung, während die nicht-B-sichtbare Gleichheit davon abhängig ist.

Als Ergebnis kann man sagen, dass sich der Entwurf der Repräsentation für die Flusspolitiken bewährt, insbesondere die Einführung der Klasse ProgramSecPolicy. Sie wurde aus der Erkenntnis heraus eingeführt, dass sich die Bedingungen der Sicherheitstypregeln mehrfach darauf beziehen, ob Informationsfluss zwischen Ausdrücken und Variablenbezeichnern erlaubt ist. Als sich herausstellte, dass sich mit der starken Sicherheit[dr] diese Bedingungen ändern, zeigte sich ProgramSecPolicy als die Stelle im EIFA-Entwurf, an der man die Änderungen einfügen kann.

4.4. Schluss

4.4.1. Bewertung

Jetzt sind die starke Sicherheit[dr] definiert und ihre Eigenschaften untersucht. Insbesondere ist untersucht und an Beispielen vorgeführt, wie MLS-Politiken mit *escape-hatches* definiert sein müssen und wie MWL-Programme geschrieben sein müssen, damit diese Programme gleichzeitig stark sicher[in] und stark sicher[dr] sind. Es ist

ein Sicherheitstypsystem definiert, dass MWL-Programme auf starke Sicherheit[dr] überprüft. Dieses Sicherheitstypsystem ist auch nicht zu restriktiv für Programme, die stark sicher[in] sind.

Zusätzliche Sicherheit gegenüber der starken Sicherheit[in] Wie die Beispiele 6 und 8 zeigen, gibt es Programme, für die man die gewünschten fachlichen Sicherheitsanforderungen mit der starken Sicherheit[dr] durchsetzen kann. Insbesondere kann man zum Beispiel für das Programm im Beispiel 6 die fachlichen Sicherheitsanforderungen nicht mit der starken Sicherheit[in] durchsetzten, da beschränkt werden soll, „was“ das Programm deklassifiziert. Die starke Sicherheit[dr] erweitert also tatsächlich die Möglichkeiten fachliche Sicherheitsanforderungen formal zu spezifizieren und damit durchzusetzten.

Umgekehrt besteht nicht die Gefahr, dass man die starke Sicherheit[in] abschwächt, wenn man die Konjunktion der starken Sicherheit[in] und der starken Sicherheit[dr] benutzt.

Bezüglich der Sicherheit besteht also das gewünschte Ergebnis. Die starke Sicherheit[dr] ergänzt sinnvoll die starke Sicherheit[in].

Zusätzliche Anwendbarkeit gegenüber der starken Sicherheit Sicherheitsbedingungen, die Deklassifikation zulassen, definiert man, weil normale *noninterference*-Bedingungen, wie die starke Sicherheit, für viele Anwendungsfälle zu restriktiv sind. Ziel ist es daher, dass die starke Sicherheit[in] weniger restriktiv als die starke Sicherheit ist.

Wie das Beispiel 6 zeigt, gibt es Programme, die stark sicher[dr] sind und bei denen die starke Sicherheit nicht anwendbar ist, weil sie den Anforderungen an die Programmfunktion widerspricht. Die starke Sicherheit[dr] erweitert also im Vergleich zur starken Sicherheit erfolgreich die Anwendungsmöglichkeiten.

Aber umgekehrt stellt sich heraus, dass die starke Sicherheit[dr] nicht die Konservativität erfüllt (s. Abschnitt 4.2.4). Das heißt mit einer gegebenen *escape-hatch*-Menge kann ein Programm gleichzeitig stark sicher und nicht stark sicher[dr] sein, weil durch das Programm in eine *escape-hatch* vertrauliche Information fließt, ohne dass dies die normale Flusspolitik verletzt. Praktisch ist das aber keine Einschränkung. Im Abschnitt 4.2.4 wird vorgeführt, wie sich das Problem vermeiden lässt. Außerdem tritt diese Einschränkung nur auf, wenn man eine *escape-hatch*-Menge vorgibt. Die starke Sicherheit[dr] mit der leeren *escape-hatch*-Menge ist immer äquivalent zur starken Sicherheit (s. Satz 11).

Zu beachten ist, dass die potentiell deklassifizierbare Information beschränkt ist, denn nicht jede Information ist als Ausdruck darstellbar. Weitere Überlegungen dazu befinden sich im Abschnitt 4.4.2.

Insgesamt besteht auch hier das gewünschte Ergebnis. Man kann die Sicherheitsbedingung in mehr Anwendungsfällen einsetzen, als mit der starken Sicherheit alleine.

Überprüfbarkeit Die automatische Überprüfbarkeit mit einem Sicherheitstypsystem ist gezeigt (s. Abschnitt 4.3). Das Sicherheitstypsystem ist dem Sicherheitstypsystem für starke Sicherheit und starke Sicherheit[in] ähnlich. Damit lässt es sich nicht nur parallel zum Sicherheitstypsystem für die starke Sicherheit[in] anwenden, sondern es lässt sich auch einfach ein Sicherheitstypsystem für beide Sicherheitsbedingungen erstellen. Außerdem lässt sich durch die Ähnlichkeit das neue Sicherheitstypsystem einfach in bestehende Anwendungen (wie den EIFA, s. Abschnitt 4.3.2) integrieren.

Kombination mit der starken Sicherheit[in] Im Abschnitt 4.2.5 wird die Kompatibilität der starken Sicherheit[dr] mit der starken Sicherheit[in] betrachtet. Man kann die Konjunktion beider Bedingungen nutzen. Die Beispiele 6, 7 und 8 zeigen Programme, die beide Sicherheitsbedingungen erfüllen. Insbesondere im Beispiel 8 sind beide Sicherheitsbedingungen nötig, um die fachlichen Sicherheitsanforderungen zu erfüllen.

Ein Nachteil ist, dass die Quellvariablenbezeichner der Deklassifikationsanweisungen als *escape-hatches* vorkommen müssen (s. Abschnitt 4.2.5).[19] Das heißt einige Elemente der *escape-hatch*-Menge sind nicht aufgrund von fachlichen Sicherheitsanforderungen darin enthalten, sondern aus technischen Gründen.

Ergebnis Insgesamt ist die starke Sicherheit[dr] im allgemeinen, aber insbesondere auch in Kombination mit der starken Sicherheit[in] nützlich. Zum einen deckt sie Sicherheitsanforderungen ab, die von der starken Sicherheit[in] nicht abgedeckt werden. Zum anderen ist sie für Programme anwendbar, die Information deklassifizieren müssen und daher nicht stark sicher sein können.

Es ist also das Ziel dieses Abschnitts erreicht. Es ist eine Sicherheitsbedingung für das „was“ der Deklassifikation definiert. Diese Sicherheitsbedingung ist für nebenläufige Programme (MWL-Programme) definiert und automatisch überprüfbar. Da die starke Sicherheit[dr] mit der starken Sicherheit[in] kombiniert ist, sind zwei Dimensionen der Deklassifikation abgedeckt, das „wo“ und das „was“.

4.4.2. Ausblick

Dieser Abschnitt stellt noch einige Ideen und Probleme vor, die man weiter erforschen könnte.

***Delimited Release* und die Eingabe** Hinter der starken Sicherheit[dr] und allgemein hinter *delimited release* steckt die Idee, dass ein Programm nur die Information deklassifizieren darf, die schon bei der Eingabe der Bezeichnerwerte in den *escape-hatches* steckt. Wie am Ende des Abschnitts 4.2.2 angemerkt ist, muss man *delimited release* eventuell an neue Eingabemodelle anpassen. In dieser Arbeit wird davon ausgegangen, dass die Eingabewerte schon vor Beginn der Ausführung eines Programms

[19] Außer für die uninteressanten Fälle $\texttt{[d2:=d1]}$ mit $\Gamma(\texttt{d1}) \leq \Gamma(\texttt{d1})$.

vorhanden sind. Aber wenn es zum Beispiel mehrere Prozesse[20] gibt, die über Sende- und Empfangsanweisungen miteinander kommunizieren, ist der Empfang auch eine Art Eingabe. Man könnte erlauben, dass die empfangene Information in *escape-hatches* fließt. Konkret könnte man dazu MWL für verteile Programme betrachten, wie es in [SM02] vorgestellt wird. Dort ist eine starke Bisimulation als Sicherheitsbedingung und ein Sicherheitstypsystem ähnlich wie in dieser Arbeit definiert. Die Kombination von *delimited release* mit der dort definierten starken Bisimulation sollte aufgrund der Ähnlichkeit kein Problem sein. Anschließend wäre die Bedingung wie genannt zu lockern. Interessant ist, welche neuen Möglichkeiten es damit gibt, sinnvolle Programme zu schreiben, aber auch welche neuen Möglichkeiten ein Angreifer hat.

Noch allgemeiner ist die Idee, *delimited release* von den Eingabewerten zu lösen. Man kann eine neue Anweisung einführen, die sich wie eine Zuweisung verhält. Doch im Gegensatz zur Zuweisung, erlaubt diese Anweisung den Informationsfluss in *escape-hatches*.

Weiter wäre es interessant zu untersuchen, wie *delimited release* auf programminterne „Ein-/Ausgaben“ anwendbar ist, das heißt auf Parameter und Rückgabewerte von Funktionen.

Deklassifikationsanweisung mit Ausdrücken Im Abschnitt 4.2.5 wird die Kompatibilität der starken Sicherheit[dr] mit der starken Sicherheit[in] betrachtet. Dabei wird ein Problem angesprochen. Wenn man eine Flusspolitik ohne Bezeichner als *escape-hatches* hat, ist die Konjunktion der beiden Sicherheitsbedingungen zu streng, da MWL-Deklassifikationsanweisungen nur Bezeichner deklassifizieren.

Als Lösung könnte man eine Deklassifikationsanweisungen für Ausdrücke einführen: $[Id{:=}Exp]$. In [MS04] wurde die Deklassifikation auf Bezeichner beschränkt, um nicht versehentlich mehr Information zu deklassifizieren als gewollt. Mit starken Sicherheit[dr] ist in der Flusspolitik spezifiziert, welche Information das Programm deklassifizieren darf, somit fällt diese Gefahr weg.

Allerdings stellt sich die Frage, wie die Semantik einer solchen Deklassifikationsanweisung zu definieren ist. Die starke Sicherheit[in] soll die Quell- und Zielsicherheitsdomänen der Deklassifikation beschränken. Doch welche Sicherheitsdomäne ist die Quelle einer Deklassifikation $[Id{:=}Exp]$? Falls allen Bezeichner in Exp die gleiche Sicherheitsdomäne zugeordnet ist, könnte man diese als Quelle bezeichnen. Aber wenn *escape-hatch*-Ausdrücke Bezeichner mit unterschiedlichen Sicherheitsdomänen enthalten, benötigt man immernoch Variablenbezeichner als *escape-hatches*, um diese Ausdrücke tatsächlich zu deklassifizieren.

Deklassifizierbare Information Sicherheitsbedingungen basierend auf *delimited release* schränken das „was“ der Deklassifikation ein, aber „was“ darf eine *delimited release*-Flusspolitik als deklassifizierbar erlauben? Grundsätzlich hängt das von den

[20] Hier laufen Prozesse im Gegensatz zu Threads auf jeweils einem eigenen Prozessor und besitzen einen eigenen Speicher

möglichen Ausdrücken ab. Wenn es nur einfache Ausdrücke gibt, zum Beispiel nur Bezeichner, dann ist *delimited release* sehr beschränkt. Man dürfte nichtmal den Mittelwert zweier vertraulicher Variablenbezeichner deklassifizieren, ohne auch zu erlauben, die Einzelwerte zu deklassifizieren.

Allgemeiner kann man zum Beispiel mit Hilfe der Komplexitätstheorie überlegen, ob eine bestimmte Information deklassifizierbar ist. Jeder Ausdruck hat eine feste Anzahl an Operatoren. Wenn alle zugehörigen Operatorfunktionen in $O(f(n))$ für eine Funktion f sind, ist auch die durch einen Ausdruck dargestellte Funktion in $O(f(n))$.

Unsichere *Escape-Hatch*-Mengen Ein Problem der starken Sicherheit[dr] ist die größere Komplexität der möglichen Flusspolitiken mit *escape-hatches*.

Erst mit Flusspolitiken sind fachliche Sicherheitsanforderungen formalisiert und damit formal überprüfbar. Daher muss der Ersteller einer Flusspolitik selbst entscheiden, ob diese Flusspolitik die fachlichen Sicherheitsanforderungen korrekt abbildet. Durch die größere Komplexität kann er dabei mehr Fehler machen. Das bedeutet hier, eine *escape-hatch* könnte mehr Informationsfluss erlauben, als gewünscht. Dies kann zum Beispiel durch sehr komplexe Ausdrücke geschehen, insbesondere, wenn es viele MWL-Operatoren gibt. Eine große Gefahr besteht in der Kombination der Information verschiedener *escape-hatches*. Wenn ein Programm zum Beispiel `h1 + h2` und `h1 − h2` deklassifizieren darf, fließen damit die vollständige Information über `h1` und `h2`. Welche Kombinationen von *escape-hatches* welche zusätzliche Information enthält, hängt wieder von den möglichen Operatoren ab. Eine allgemeine Möglichkeit diese Gefahr zu beseitigen ist, keinen Bezeichner in mehr als einem *escape-hatch*-Ausdruck zu erlauben.

Starke Sicherheit[dr] und Synchronisationsanweisungen Die Definitionen 5 für die normale starke D-Bisimulation und 17 für die starke D-Bisimulation[dr] unterscheiden sich nur in der Zustandsäquivalenz $=_D$ bzw. $=_D^{\mathcal{H}}$. Entsprechend Definition 8 für die starke D-Bisimulation mit Synchronisation kann man Definition 17 so modifizieren, dass Synchronisationstransitionen weder untereinander noch mit der normalen Transitionen im Bisimulationsschritt austauschbar sind.

Die Aussagen über die starke Sicherheit[dr] lassen sich auf diese modifizierte Sicherheitsbedingung übertragen. Nur im Beweis für die Zusammensetzbarkeit der starken D-Bisimulation[dr] (s. Lemma 13) muss man einen weiteren Fall betrachten. Eine sequentielle Komposition von Programmen führt nach der Semantik zu einer Synchronisationstransition, wenn das erste Teilprogramm zu einer Synchronisationstransition führt. Dieser Fall lässt sich analog zu dem gleichen Fall für die starke Sicherheit[in] behandeln (s. Beweis für Lemma 23 im Anhang C). Dass die Sicherheitstypregeln für Synchronisationsregeln (s. Abbildung 13) korrekt sind, lässt sich einfach zeigen. Das Verhalten einer Synchronisationsanweisung hängt nur vom Wert ihres Semaphorbezeichners ab. Da den Semaphorbezeichnern die Sicherheitsdomäne *low* zugeordnet sein muss, haben diese in $D, \mathcal{H}$-gleichen Zuständen den gleichenWert.

4.4.3. Alternative Ansätze für das „was“ der Deklassifikation

Neben der *delimited release* wurden noch andere Ansätze für das „was“ der Deklassifikation zur Kombination mit *intransitive noninterference* (in Form der starken Sicherheit[in]) in Erwägung gezogen.

Abstract Noninterference Ein alternativer Ansatz, um das „was“ der Deklassifikation zu spezifizieren, ist die *abstract noninterference* (s. [GM04]). Hier werden sogenannte abstrakte Interpretationen verwendet, um das Wissen eines Angreifers über einen Zustand zu modellieren. Dieser Ansatz ist allgemeiner als das *PER model* (s. [HM05]), das heißt allgemeiner, als das Wissen des Angreifers mit Äquivalenzrelationen zu modellieren. Es ist also nicht direkt eine Alternative zu *delimited release*, denn die Idee hinter *delimited release* ist es, die semantische Äquivalenzrelation syntaktisch zu spezifizieren, um damit die syntaktische Analyse zu erleichtern. Es wäre interessant zu untersuchen, ob sich *delimited release* auf *abstract noninterference* übertragen lässt, um dafür ebenfalls syntaktische Analysen zu erleichtern.

Quantifying Information Flow Weitere Ansätze befassen sich mit der Idee, eigentlich verbotenen Informationsfluss zu erlauben, aber die Informationsmenge zu beschränken. Zum Beispiel wird in [Low04] die Idee vorgestellt, die für den Angreifer sichtbaren Abläufe eines Programms aus für den Angreifer ununterscheidbaren Anfangszuständen zu zählen. Mit einer normalen *noninterference*-Bedingung, wie zum Beispiel die starke Sicherheit, gilt ein Programm nur dann als sicher, wenn es genau einen sichtbaren Ablauf aus ununterscheidbaren Anfangszuständen gibt. Mit der Idee aus [Low04] gilt ein Programm auch dann als sicher, wenn es aus für den Angreifer ununterscheidbaren Anfangszuständen n verschiedene sichtbare Abläufe gibt, wobei $\log n$ die deklassifizierbaren Informationsbits sind. Es wäre interessant, diese Idee auf eine imperative Sprache wie MWL zu übertragen, denn Lowe behandelt eine Prozessalgebra. Zu beachten ist aber, dass der Ansatz in [Low04] nicht direkt Deklassifikation adressiert, sondern unvermeidbare verdeckte Kanäle. Das könnte ein Hindernis zur Kombination mit der starken Sicherheit[in] sein, denn diese beschränkt Deklassifikation auf Deklassifikationsanweisungen. Dadurch sind verdeckte Kanäle verboten.

5. *Intransitive Noninterference* und *Robust Declassification*

Mit der starken Sicherheit[in] basierend auf *intransitive noninterference* und der starken Sicherheit[dr] basierend auf *delimited release* sind zwei Dimensionen der Deklassifikation abgedeckt, das „wo “ und das „was “, wie sie in [MS04] identifiziert werden. Ein dritter Aspekt ist das „wer“. Es soll also eingeschränkt werden, wem es erlaubt ist, die Deklassifikation zu beeinflussen. Dazu erfolgt die Definition einer Sicherheitsbedingung (s. Definition 26), die auf der Idee der *robust declassification* basiert (s. [ZM01], [MSZ06]). Diese Idee ist folgende: Ein Angreifer, der Teile des Programms verändern darf (aktiver Angreifer), kann keine zusätzliche Information gegenüber einem Angreifer lernen, der diese Teile nicht ändern darf (passiver Angreifer).

Nach der Definition wird gezeigt, dass diese Sicherheitsbedingung aus der starken Sicherheit[in] folgt, wenn die Ausnahmen der MLS-Politik bestimmte Bedingungen erfüllen (s. Satz 21). Die Idee dahinter ist, dass man mit der starken Sicherheit[in] gegenüber dem Angreifer die Integrität der Information sicherstellt, von der die Deklassifikation abhängt. In [MSZ06] wird diese Idee in Form eines Sicherheitstypsystems genutzt. Mit Satz 21 wird gezeigt, dass diese Idee auch unabhängig von einem Sicherheitstypsystem funktioniert.

Allerdings ist die neue Sicherheitsbedingung nur ein Schritt dahin, *robust declassification* für Programme mit mehreren Threads zu definieren. Sie ist unter der Annahme definiert, dass ein Angreifer auch ohne Teile des Programms zu verändern sehr viel Information durch Ausführung des Programms lernen kann (s. Abschnitt 5.3.1).

Beispiel 10. Folgendendes Programm ist für eine binäre Flusspolitik mit dem Ausnahmefluss von *high* nach *low* stark sicher[in]. Mit `h1` und `h2` als *escape-hatches* ist es auch stark sicher[dr].

```
if bl then
  [ l := h1 ]
else
  [ l := h2 ]
```

Angenommen, der Angreifer kann Variablenbezeichner mit der Sicherheitsdomäne *low* beeinflussen. So kann er auch bestimmen, ob `h1` oder `h2` deklassifiziert wird.

Es gibt Fälle, in denen diese Möglichkeit gewünscht ist. Wenn man aber verhindern will, dass der Angreifer die Deklassifikation beeinflusst, kann man das durch Erweiterung der Flusspolitik erreichen. Man betrachte die Flusspolitik in Abbildung 31 mit folgendem Programm C_{cond}. Die Sicherheitsdomänen sind jeweils aus dem Variablennamen ablesbar:

```
if blh then
  [ lh := hh1 ]
else
  [ lh := hh2 ]
```

Ein Angreifer, der in einem parallelen Thread Zuweisungen an Variablenbezeichner der Sicherheitsdomäne LL ausführen darf, kann nun nicht mehr beeinflussen, was das Programm deklassifiziert, da nirgendwo eine Zuweisung an blh , hh1 oder hh2 vorkommt.

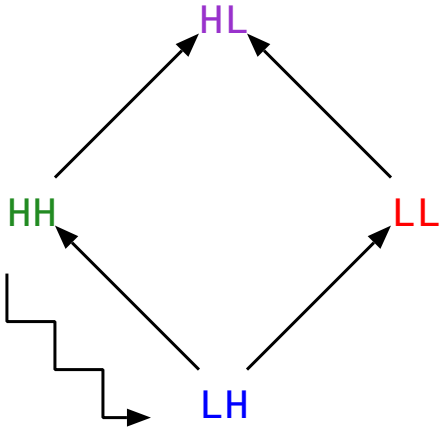


Abbildung 31: Flusspolitik, die „*robust declassification*“ bezüglich eines Angreifers auf LL durchsetzen soll. Der erste Buchstabe einer Sicherheitsdomäne steht für die Vertraulichkeit gegenüber dem Angreifer („H “, wenn er nicht davon lesen kann), der zweite für die Integrität („H “, wenn er nicht dahin schreiben kann).

Falls allerdings die Programme $C_{race1} =$ blh:=**true** und $C_{race2} =$ blh:=**false** als Threads parallel zu C_{cond} laufen, besteht eine Race-Condition. Es gibt Scheduler, mit denen der aktive Angreifer die Auflösung dieser Race-Condition bestimmen kann, wenn er die Schedulerstrategie kennt. Man betrachte zum Beispiel den Programmvektor $(C_A, C_{cond}, C_{race1}, C_{race2})$, wobei C_A das vom Angreifer gewählte Programm ist. Angenommen, der Scheduler führt erst den ersten Thread aus, dann den dritten und dann immer den ersten. Wenn $C_A =$ **skip**, dann führt dieser Scheduler C_{race2} aus, wenn $C_A =$ **skip**;**skip**, dann führt er C_{race1} aus.

Ein passiver Angreifer, der auf $C_A =$ **skip** festgelegt ist, lernt nur hh2 . Ein aktiver Angreifer, der zwischen $C_A =$ **skip** und $C_A =$ **skip**;**skip** wählen kann, bestimmt ob hh1 oder hh2 deklassifiziert wird.

Ein Problem der unten definierten Sicherheitsbedingung ist, dass der genannte Programmvektor trotz der Möglichkeiten des aktiven Angreifers die Sicherheitsbedingung erfüllt. Denn sie ist unter der Annahme definiert, dass schon der passive Angreifer den jeweils auszuführenden Thread wählen kann.

5.1. *Robust Declassification* für starke passive Angreifer

Die Sicherheitsbedingung wird nach einem ähnlichen Schema wie die Sicherheitsbedingung für *robust declassification* aus [MSZ06] definiert.

Das Schema ist folgendes: Zuerst wird eine Menge der möglichen Angriffsprogramme (hier faire Angriffe) und eine sicherheitsdomänenabhängige Relation zwischen Konfigurationen (also Paaren aus einem Programmvektor und einem Speicherzustand) definiert. Die gewünschte Bedeutung dieser Relation ist, dass wenn zwei Konfigurationen in der Relation zueinander stehen, die aus diesen Konfigurationen gestarteten Programmabläufe für die entsprechende Sicherheitsdomäne nicht unterscheidbar sind. Die Definition der Sicherheitsbedingung nutzt diese beiden Definitionen.

Zunächst die Angriffsprogramme:

Definition 24 (Fairer Angriff (*fair attack*))**.** Ein MWL-Programm A ist ein *fairer Angriff* bezüglich einer Sicherheitsdomäne $D \in \mathcal{D}$, wenn es nach folgender Grammatik gebildet wird

$$\boxed{A ::= \ \mathsf{skip} \ \mid Id := Exp \mid A_1 \ ; \ A_2 \mid \ \mathsf{fork}(A\vec{A})},$$

wobei für alle Bezeichner Id, die alleine oder in Ausdrücken auftreten, $\Gamma(Id) = D$ gilt. Die Menge aller MWL-Programme, die diese Grammatik erfüllen, heißt $\mathcal{F}_D$.

Faire Angriffe sind stark sicher:

Lemma 19. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $D \in \mathcal{D}$. Es gilt $\forall A \in \mathcal{F}_D : A$ ist stark sicher.*

Beweis: Sei $A \in \mathcal{F}_D$ beliebig. Der Beweis lässt sich durch Induktion über den Aufbau von A führen, indem man das Sicherheitstypsystem für starke Sicherheit (s. Abschnitt 2.4) anwendet. $A = \mathsf{skip}$ ist immer typisierbar. $A = Id{:=}Exp$ ist typisierbar, da alle Bezeichner die gleiche Sicherheitsdomäne haben. In den beiden anderen Fällen erfüllt die Induktionsvoraussetzung für die Teilprogramme direkt die Regelbedingungen.
□

Wie folgt wird die Konfigurationsrelation definiert:

Definition 25 (starke D-Konfigurationsbisimulation)**.** Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen ohne Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Dann ist die *starke D-Konfigurationsbisimulation* $\simeq_D$ die Vereinigung aller symmetrischen Relationen R über gleichgroße Konfigurationen mit $\langle\!|\vec{V}, s|\!\rangle R \langle\!|\vec{V}', s'|\!\rangle$ genau dann, wenn $s =_D s'$, $\vec{V} = (C_1, \ldots, C_n)$, $\vec{V}' = (C'_1, \ldots, C'_n)$ und

$$\forall i \in \{1, \ldots, n\} : \forall \vec{W}, t : \langle\!|C_i, s|\!\rangle \rightarrowtriangle \langle\!|\vec{W}, t|\!\rangle$$
$$\Rightarrow \exists \vec{W}', t' : \begin{bmatrix} \langle\!|C'_i, s'|\!\rangle \rightarrowtriangle \langle\!|\vec{W}', t'|\!\rangle \wedge \\ \langle\!|C_1 \ldots C_{i-1} \vec{W} \ldots C_n, t|\!\rangle R \langle\!|C'_1 \ldots C'_{i-1} \vec{W}' \ldots C'_n, t'|\!\rangle \end{bmatrix}$$

Diese Bisimulation ist so definiert, dass der Bisimulationsschritt jeweils vom Thread an der gleichen Stelle ausgeführt werden muss. Es wird also ein Angreifer modelliert, der in jedem Schritt den auszuführenden Thread auswählen kann, und das in jeder Ausführung des Programms neu.

Folgender Zusammenhang besteht zwischen der starken D-Bisimulation und der starken D-Konfigurationsbisimulation:

Lemma 20. *Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen ohne Synchronisationsanweisungen. Sei $D \in \mathcal{D}$. Für alle $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ gilt:*

$$\vec{V} \approxeq_D \vec{V'} \Rightarrow \forall s =_D s' : \langle\!|\vec{V}, s|\!\rangle \simeq_D \langle\!|\vec{V'}, s'|\!\rangle$$

Beweis: Sei $D \in \mathcal{D}$ beliebig. Sei $R_D = \{(\langle\!|\vec{V}, s|\!\rangle, \langle\!|\vec{V'}, s'|\!\rangle) | \vec{V} \approxeq_D \vec{V'} \wedge s =_D s'\}$. Es ist zu zeigen, dass $R_D \subseteq \simeq_D$ gilt. Seien $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ und s, s' mit $\langle\!|\vec{V}, s|\!\rangle R_D \langle\!|\vec{V'}, s'|\!\rangle$. Die Symmetrie von R_D, die gleichen Längen von $\vec{V}$ und $\vec{V'}$ und $s =_D s'$ folgen direkt aus der Definition von R_D. Man kann also $\vec{V} = (C_1 \dots C_n), \vec{V'} = (C'_1 \dots C'_n)$ schreiben.

Seien $i \in \{1, \dots, n\}$ und $\vec{W}, t$ beliebig mit $\langle\!|C_i, s|\!\rangle \rightarrowtriangle \langle\!|\vec{W}, t|\!\rangle$. Da $\vec{V} \approxeq_D \vec{V'}$ und $s =_D s'$, gibt es $\vec{W'}, t'$ mit $\langle\!|C'_i, s'|\!\rangle \rightarrowtriangle \langle\!|\vec{W'}, t'|\!\rangle \wedge \vec{W} \approxeq_D \vec{W'} \wedge t =_D t'$. Damit gilt auch $\langle\!|C_1 \dots C_{i-1} \vec{W} \dots C_n|\!\rangle \approxeq_D \langle\!|C'_1 \dots C'_{i-1} \vec{W'} \dots C'_n|\!\rangle$. Daraus folgt

$$\langle\!|C_1 \dots C_{i-1} \vec{W} \dots C_n, t|\!\rangle R_D \langle\!|C'_1 \dots C'_{i-1} \vec{W'} \dots C'_n, t'|\!\rangle.$$

Damit ist die Existenz von $\vec{W'}$ und t' aus der Definition 25 gezeigt.
□

Wie folgt wird die Sicherheitsbedingung definiert:

Definition 26 (D-Sicherheit[rd] (Sicherheit für *robust declassification*)). Seien eine MLS-Politik $(\mathcal{D}, \leq)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Programmen ohne Synchronisationsanweisungen. $C \in \mathcal{C}$ ist für $D \in \mathcal{D}$ *D-sicher[rd]*, falls gilt:

$$\forall s, s' : \forall A, A' \in \mathcal{F}_D : \langle\!|AC, s|\!\rangle \simeq_D \langle\!|AC, s'|\!\rangle \Rightarrow \langle\!|A'C, s|\!\rangle \simeq_D \langle\!|A'C, s'|\!\rangle$$

Das bedeutet, egal welches Angriffsprogramm A für den passiven Angreifer vorgegeben ist, es kann kein aktiver Angreifer das Angriffsprogramm A' so wählen, dass er Konfigurationen unterscheiden kann, die der passive Angreifer nicht unterscheiden kann.

5.2. *Robust Declassification* durch *Intransitive Noninterference*

Folgender Satz stellt eine Beziehung zwischen der starken Sicherheit[in] und der D-Sicherheit[rd] auf. Er ermöglicht über die starke Sicherheit[in] die D-Sicherheit[rd] eines MWL-Programms automatisch zu beweisen. Das Sicherheitstypsystem aus [MSZ06] funktioniert ähnlich. Dort wird die Deklassifikation lokalisiert und nur zwischen bestimmten Sicherheitsdomänen erlaubt.

Satz 21. *Seien eine MLS-Politik mit Ausnahmen* $(\mathcal{D}, \leq, \rightsquigarrow)$*, eine Domänenzuweisung* Γ *und eine Sicherheitsdomäne* $D_A \in \mathcal{D}$ *gegeben. Wenn für alle* $D, D' \in \mathcal{D}$ *mit* $D \rightsquigarrow D'$ *gilt, dass* $D, D' \not\geq D_A$*, dann gilt für alle MWL-Programme* C*:*

$$C \text{ ist stark sicher[in]} \Rightarrow C \text{ ist } D_A\text{-sicher[rd]}$$

Beweis: Seien eine MLS-Politik mit Ausnahmen $(\mathcal{D}, \leq, \rightsquigarrow)$, eine Domänenzuweisung Γ und eine Sicherheitsdomäne $D_A \in \mathcal{D}$ gegeben. Es gelte für alle $D, D' \in \mathcal{D}$ mit $D \rightsquigarrow D'$, dass $D, D' \not\geq D_A$. Seien C ein beliebiges, stark sicheres[in] MWL-Programm, s, s' beliebige Zustände und $A, A' \in \mathcal{F}_{D_A}$ beliebig so gegeben, dass $\langle\!|AC, s|\!\rangle \simeq_{D_A} \langle\!|AC, s'|\!\rangle$.

Falls $\forall D : D \geq D_A$ gilt, gilt auch $\rightsquigarrow = \emptyset$. Nach Satz 4 ist damit C stark sicher. Da A' als fairer Angriff nach Lemma 19 stark sicher ist, ist $A'C$ ebenfalls stark sicher. Da mit $\langle\!|AC, s|\!\rangle \simeq_{D_A} \langle\!|AC, s'|\!\rangle$ auch $s =_{D_A} s'$ gilt, folgt nach Lemma 20 $\langle\!|A'C, s|\!\rangle \simeq_{D_A} \langle\!|A'C, s'|\!\rangle$. Damit ist dieser Fall erledigt.

Im weiteren kann man annehmen, dass $\exists D : D \not\geq D_A$.

Es ist zu zeigen: es lässt sich ein R_{D_A} über Konfigurationen so definieren, dass $R_{D_A} \subseteq \simeq_{D_A}$ und $\langle\!|A'C, s|\!\rangle R_{D_A} \langle\!|A'C, s'|\!\rangle$ gelten. Dann gilt auch $\langle\!|A'C, s|\!\rangle \simeq_{D_A} \langle\!|A'C, s'|\!\rangle$ und der Satz ist bewiesen.

Es bleibt noch R_{D_A} zu definieren sowie $R_{D_A} \subseteq \simeq_{D_A}$ und $\langle\!|A'C, s|\!\rangle R_{D_A} \langle\!|A'C, s'|\!\rangle$ zu beweisen.

Es sei

$$\begin{aligned} R_{D_A} = \{(\langle\!|\vec{W}_A\vec{W}, t|\!\rangle, \langle\!|\vec{W}'_A\vec{W}', t'|\!\rangle) | \\ & \vec{W}_A, \vec{W}'_A \in \vec{\mathcal{F}}_{D_A} \wedge \vec{W}, \vec{W}' \in \vec{\mathcal{C}} \wedge \\ & \vec{W}_A \cong_{D_A} \vec{W}'_A \wedge \vec{W} \cong^{in}_{D_A} \vec{W}' \wedge t =_{D_A} t' \\ & \exists \vec{V}, \vec{V}' \in \vec{\mathcal{C}} : \exists s, s' : \\ & \quad \langle\!|\vec{V}, s|\!\rangle \simeq_{D_A} \langle\!|\vec{V}', s'|\!\rangle \wedge \\ & \quad \forall D' \not\geq D_A : s =_{D'} t \wedge s' =_{D'} t' \wedge \vec{V} \cong_{D'} \vec{W} \wedge \vec{V}' \cong_{D'} \vec{W}'\}. \end{aligned}$$

Da

- A' stark sicher ist,
- C stark sicher[in] sind,
- $s =_{D_A} s'$ (weil $\langle\!|AC, s|\!\rangle \simeq_{D_A} \langle\!|AC, s'|\!\rangle$) und
- $\langle\!|C, s|\!\rangle \simeq_{D_A} \langle\!|C, s'|\!\rangle$ (weil $\langle\!|AC, s|\!\rangle \simeq_{D_A} \langle\!|AC, s'|\!\rangle$),

gilt $\langle\!|A'C, s|\!\rangle R_{D_A} \langle\!|A'C, s'|\!\rangle$. Dabei setzt man für die existenzquantifizierten Programmvektoren $\vec{V}$ und $\vec{V}'$ der Definition das Programm C ein, die existenzquantifizierten Zustände sind s und s'.

Noch zu zeigen: $R_{D_A} \subseteq \simeq_{D_A}$. Seien $\vec{W}_A, \vec{W}, \vec{W}'_A, \vec{W}', t, t'$ mit $\langle\!|\vec{W}_A\vec{W}, t|\!\rangle R_{D_A} \langle\!|\vec{W}'_A\vec{W}', t'|\!\rangle$ gegeben. Es sind folgende vier Punkte zu zeigen:

1. R_{D_A} ist symmetrisch: folgt direkt aus der Definition, alle Bedingungen sind symmetrisch
2. $\vec{W}_A\vec{W}$ und $\vec{W}'_A\vec{W}'$ sind gleichlang: folgt aus $\vec{W} \approx^{in}_{D_A} \vec{W}'$ und $\vec{W}_A \approx_{D_A} \vec{W}'_A$
3. $t =_{D_A} t'$: folgt direkt aus der Definition
4. $\forall \vec{C}, t^* : \langle\!|\vec{W}_A\vec{W}, t|\!\rangle \rightarrow \langle\!|\vec{C}, t^*|\!\rangle \Rightarrow \exists \vec{C}', t'^* : \langle\!|\vec{W}'_A\vec{W}', t'|\!\rangle \rightarrow \langle\!|\vec{C}', t'^*|\!\rangle$, wobei $\langle\!|\vec{C}, t^*|\!\rangle R_{D_A} \langle\!|\vec{C}', t'^*|\!\rangle$ und die Transitionen den Thread der gleichen Stelle ausführen: noch zu zeigen

Seien $\vec{V}, \vec{V}', s, s'$ die nach der Definition von R_{D_A} existierenden Programmvektoren und Zustände. Seien $\vec{W} = (W_1, \ldots, W_{n_1})$, $\vec{W}' = (W'_1, \ldots, W'_{n_1})$, $\vec{V} = (V_1, \ldots, V_{n_1})$, $\vec{V}' = (V'_1, \ldots, V'_{n_1})$, $\vec{W}_A = (W_{A1}, \ldots, W_{An_2})$ und $\vec{W}'_A = (W'_{A1}, \ldots, W'_{An_2})$. Das ist möglich, da die Längengleichheiten der Programmvektoren aus der Definition von R_{D_A} folgen.

Es ist zu zeigen, dass es für alle $\vec{C}, t^*$ mit $\langle\!|\vec{W}_A\vec{W}, t|\!\rangle \rightarrow \langle\!|\vec{C}, t^*|\!\rangle$

- Programmvektoren $\vec{W}^*, \vec{W}'^*, \vec{V}^*, \vec{V}'^* \in \vec{\mathcal{C}}$, $\vec{W}^*_A, \vec{W}'^*_A \in \vec{\mathcal{F}}_{D_A}$ und
- Zustände t'^*, s^*, s'^*

so gibt, dass $\vec{C} = \vec{W}^*_A\vec{W}^*$, $\langle\!|\vec{W}'_A\vec{W}', t'|\!\rangle \rightarrow \langle\!|\vec{W}'^*_A\vec{W}'^*, t'^*|\!\rangle$ (wobei der Thread der gleichen Stelle ausgeführt wird) und dass gilt:

1. $t^* =_{D_A} t'^*$
2. $\vec{W}^* \approx^{in}_{D_A} \vec{W}'^*$
3. $\vec{W}^*_A \approx_{D_A} \vec{W}'^*_A$
4. $\forall D' \not\geq D_A : \vec{V}^* \approx_{D'} \vec{W}^*$
5. $\forall D' \not\geq D_A : \vec{V}'^* \approx_{D'} \vec{W}'^*$
6. $\forall D' \not\geq D_A : s^* =_{D'} t^*$
7. $\forall D' \not\geq D_A : s'^* =_{D'} t'^*$
8. $\langle\!|\vec{V}^*, s^*|\!\rangle \simeq_{D_A} \langle\!|\vec{V}'^*, s'^*|\!\rangle$

Daraus folgt $\langle\!|\vec{W}^*_A\vec{W}^*, t^*|\!\rangle R_{D_A} \langle\!|\vec{W}'^*_A\vec{W}'^*, t'^*|\!\rangle$. Damit ist der vierte der oben genannten Punkte gezeigt.

Dass es die mit „*“ gekennzeichneten Programmvektoren und Zustände gibt, lässt sich mit einer Fallunterscheidung zeigen. Es wird danach unterschieden, ob ein Thread des Angriffsprogramms oder einer des eigentlichen Programms ausgeführt wird. Im zweiten Fall wird weiter unterschieden, ob die Transition eine Deklassifikationstransition ist oder nicht. In jedem der Fälle sind die erfüllten Aussagen (1. – 8.) unterstrichen.

Fall 1: $\exists i \in \{1,\ldots,n_2\} : \vec{W}_A^* = (W_{A1},\ldots,\vec{W}_{Ai}^*,\ldots,W_{An_2}) \wedge \vec{W}^* = \vec{W} \wedge$
$\langle\!|W_{Ai},t|\!\rangle \rightarrowtriangle_o \langle\!|\vec{W}_{Ai}^*,t^*|\!\rangle$ ($\vec{W}_A$ enthält als fairer Angriff keine Deklassifikationsanweisung).

Da $\vec{W}_A \approx_{D_A} \vec{W}'_A$ und $t =_{D_A} t'$, existieren $\vec{W}'^*_{Ai}, t'^*$ mit $\langle\!|W'_{Ai},t'|\!\rangle \rightarrowtriangle_o \langle\!|\vec{W}'^*_{Ai},t'^*|\!\rangle$, $\vec{W}^*_{Ai} \approx_{D_A} \vec{W}'^*_{Ai}$ und $\underline{t^* =_{D_A} t'^*}$. Mit $\vec{W}'^*_A = (W'_{A1},\ldots,\vec{W}'^*_{Ai},\ldots,W'_{An_2})$ folgt daraus $\underline{\vec{W}^*_A \approx_{D_A} \vec{W}'^*_A}$.

Man setze $\vec{W}'^* = \vec{W}'$, $\vec{V}^* = \vec{V}$, $\vec{V}'^* = \vec{V}'$, $s^* = s$ und $s'^* = s'$. Die Aussagen 2., 4., 5. und 8. folgen daher direkt aus der Definition von R_{D_A}.

Da $\vec{W}_A \in \mathcal{F}_{D_A}$, gilt für alle Zielbezeichner Id von Zuweisungen in $\vec{W}_A$: $\Gamma(Id) = D_A$. Daher gilt für alle diese Bezeichner $\forall D' \not\geq D_A : D' \not\geq \Gamma(Id)$. Daraus folgt $\forall D' \not\geq D_A : t^* =_{D'} t \wedge t'^* =_{D'} t'$. Aus der Definition von R_{D_A} folgt $\forall D' \not\geq D_A : s =_{D'} t \ \wedge \ s' =_{D'} t'$. Mit $s^* = s$ und $s'^* = s'$ folgen auch $\underline{\forall D' \not\geq D_A : s^* =_{D'} t^*}$ und $\underline{\forall D' \not\geq D_A : s'^* =_{D'} t'^*}$.

Fall 2: $\exists i \in \{1,\ldots,n_1\} : \vec{W}^* = (W_1,\ldots,\vec{W}_i^*,\ldots,W_{n_1}) \wedge \vec{W}_A^* = \vec{W}_A \wedge$
$\langle\!|W_i,t|\!\rangle \rightarrowtriangle \langle\!|\vec{W}_i^*,t^*|\!\rangle$.

Zunächst setze man $\vec{W}'^*_A = \vec{W}'_A$. Damit folgt $\underline{\vec{W}^*_A \approx_{D_A} \vec{W}'^*_A}$ direkt aus der Definition von R_{D_A}.

Da $\vec{W} \approx^{in}_{D_A} \vec{W}'$ und $t =_{D_A} t'$, existieren $\vec{W}'^*_i$ und t'^* mit $\langle\!|W'_i,t'|\!\rangle \rightarrowtriangle \langle\!|\vec{W}'^*_i,t'^*|\!\rangle$ und $\vec{W}^*_i \approx^{in}_{D_A} \vec{W}'^*_i$. Mit $\vec{W}'^* = (W'_1,\ldots,\vec{W}'^*_i,\ldots,W'_{n_1})$ folgt daraus $\underline{\vec{W}^* \approx^{in}_{D_A} \vec{W}'^*}$.

Da $\forall D' \not\geq D_A : s =_{D'} t \ \wedge \ s' =_{D'} t' \ \wedge \ \vec{V} \approx_{D'} \vec{W} \ \wedge \ \vec{V}' \approx_{D'} \vec{W}'$, gibt es $\vec{V}_i^*, \vec{V}'^*_i, s^*, s'^*$ mit

$$\langle\!|\vec{V}_i,s|\!\rangle \rightarrowtriangle \langle\!|\vec{V}_i^*,s^*|\!\rangle \text{ mit } \forall D' \not\geq D_A : \vec{V}_i^* \approx_{D'} \vec{W}_i^* \text{ und}$$

$$\langle\!|\vec{V}'_i,s'|\!\rangle \rightarrowtriangle \langle\!|\vec{V}'^*_i,s'^*|\!\rangle \text{ mit } \forall D' \not\geq D_A : \vec{V}'^*_i \approx_{D'} \vec{W}'^*_i.$$

Mit $\vec{V}^* = (V_1,\ldots,\vec{V}_i^*,\ldots,V_{n_1})$ und $\vec{V}'^* = (V'_1,\ldots,\vec{V}'^*_i,\ldots,V'_{n_1})$ gelten auch

$$\underline{\forall D' \not\geq D_A : \vec{V}^* \approx_{D'} \vec{W}^*} \text{ und } \underline{\forall D' \not\geq D_A : \vec{V}'^* \approx_{D'} \vec{W}'^*}.$$

Da $\langle\!|\vec{V},s|\!\rangle \simeq_{D_A} \langle\!|\vec{V}',s'|\!\rangle$ und $\langle\!|\vec{V}_i,s|\!\rangle \rightarrowtriangle \langle\!|\vec{V}_i^*,s^*|\!\rangle$ gibt es $\vec{U},u$ mit $\langle\!|\vec{V}'_i,s'|\!\rangle \rightarrowtriangle \langle\!|\vec{U},u|\!\rangle$ und $\langle\!|\vec{V}^*,s^*|\!\rangle \simeq_{D_A} \langle\!|(V'_1,\ldots,\vec{U},\ldots,\vec{V}'_{n_1}),u|\!\rangle$. Da $\rightarrowtriangle$ deterministisch ist, gilt $\vec{U} = \vec{V}'^*_i$ und $u = s'^*$. Also gilt $\underline{\langle\!|\vec{V}^*,s^*|\!\rangle \simeq_{D_A} \langle\!|\vec{V}'^*,s'^*|\!\rangle}$.

Es sind noch die Zustandsgleichheiten zu beweisen, hier lassen sich nach der Art der Transition Fälle unterscheiden.

Fall 2a: $\langle\!|W_i,t|\!\rangle \rightarrowtriangle_o \langle\!|\vec{W}_i^*,t^*|\!\rangle$ oder $\langle\!|W_i,t|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|\vec{W}_i^*,t^*|\!\rangle$ mit $D_1 \not\rightsquigarrow D_2$.
Dann folgen $\underline{t^* =_{D_A} t'^*}$, $\underline{\forall D' \not\geq D_A : s^* =_{D'} t^*}$ und $\underline{\forall D' \not\geq D_A : s'^* =_{D'} t'^*}$ aus den jeweiligen starken Bisimulationen.

Fall 2b: Es gilt $\langle\!|W_i, t|\!\rangle \rightarrow_d^{D_1 \to D_2} \langle\!|\vec{W}_i^*, t^*|\!\rangle$ mit $D_1 \rightsquigarrow D_2$.

Dann gelten $D_1 \not\ge D_A$ und $D_2 \not\ge D_A$. Nach der Definition von R_{D_A} gelten damit auch $s =_{D_1} t$ und $s' =_{D_1} t'$. Daher folgen $\underline{\forall D' \not\ge D_A : s^* =_{D'} t^*}$ und $\underline{\forall D' \not\ge D_A : s'^* =_{D'} t'^*}$ wie im Fall 2a aus den jeweiligen starken Bisimulationen.

Um $\underline{t^* =_{D_A} t'^*}$ zu beweisen, zeigt man nach Definition von $=_{D_A}$, dass $\forall Id : \Gamma(Id) \le D_A \Rightarrow t^*(Id) = t'^*(Id)$. Sei Id beliebig mit $\Gamma(Id) \le D_A$. Es lassen sich zwei Fälle unterscheiden.

$\Gamma(Id) = D_A$: Da $D_2 \not\ge D_A$, gilt $\Gamma(Id) \neq D_2$. Nach der Semantikregel für Deklassifikationsanweisungen werden die Transitionen

$$\langle\!|W_i, t|\!\rangle \rightarrow_d^{D_1 \to D_2} \langle\!|\vec{W}_i^*, t^*|\!\rangle \text{ und } \langle\!|W_i', t'|\!\rangle \rightarrow_d^{D_1 \to D_2} \langle\!|\vec{W}_i'^*, t'^*|\!\rangle$$

von Deklassifikationen

$$[Id' := Id^*] \text{ und } [Id'' := Id^{**}] \text{ mit } \Gamma(Id') = \Gamma(Id'') = D_2$$

hervorgerufen. Also gilt $Id' \neq Id \neq Id''$. Ebenfalls nach der Semantikregel gelten $t^* = [Id' = m_1]t$ und $t'^* = [Id'' = m_2]t'$ für zwei Werte m_1 und m_2. Daher gelten $t^*(Id) = t(Id)$ und $t'^*(Id) = t'(Id)$. Da nach Definition von R_{D_A} $t =_{D_A} t'$, gilt $t(Id) = t'(Id)$. Also gilt $t^*(Id) = t'^*(Id)$.

$\Gamma(Id) \neq D_A$: Da $\Gamma(Id) \le D_A$, gilt $\Gamma(Id) \not\ge D_A$. Daher gelten $s^* =_{\Gamma(Id)} t^*$ und $s'^* =_{\Gamma(Id)} t'^*$, also auch $s^*(Id) = t^*(Id)$ und $s'^*(Id) = t'^*(Id)$. Aus $\langle\!|\vec{V}^*, s^*|\!\rangle \simeq_{D_A} \langle\!|\vec{V}'^*, s'^*|\!\rangle$ folgt $s^* =_{D_A} s'^*$, also gilt auch $s^*(Id) = s'^*(Id)$. Damit gilt $t^*(Id) = t'^*(Id)$.

□

5.3. Schluss

5.3.1. Schwächere passive Angreifer

Bevor die Erkenntnisse des Abschnitts 5 bewertet werden, folgt hier eine Betrachtung, was die D-Sicherheit[rd] für schwächere passive Angreifer aussagt. Das heißt, was sie über passive Angreifer aussagt, die weniger aus der Ausführung von Programmen lernen können als Angreifer, die der starken D-Konfigurationsbisimulation $\simeq_D$ entsprechen. Das trifft zum Beispiel auf Angreifer zu, die den Scheduler für die Ausführung kennen, aber auf diesen Scheduler festgelegt sind.

Dabei stellt man fest, dass die D-Sicherheit[rd] im allgemeinen nur Aussagen über Angreifer macht, deren erlernbare Information über Konfigurationen exakt der starken D-Konfigurationsbisimulation $\simeq_D$ entspricht. Denn wenn der passive Angreifer schwächer ist, gibt es mehr Information, die der aktive Angreifer potentiell zusätzlich lernen kann.

Konkret kann schon der passive Angreifer mit $\simeq_D$ Information lernen, indem er die Auswahl der auszuführenden Threads bestimmt. Damit kann der aktive Angreifer nicht mehr diese Möglichkeit ausnutzen, um noch mehr als der passive Angreifer zu lernen.

Dazu betrachte man das Programm $(C_{cond}, C_{race1}, C_{race2})$ aus Beispiel 10. Es ist mit der Flusspolitik aus der Abbildung 31 stark sicher[in]. Mit Satz 21 folgt, dass das Programm LL-sicher[rd] ist.

Aber wie das Beispiel 10 zeigt, gibt es Scheduler, mit denen die Wahl des Angriffsprogramms beeinflusst, welchen Variablenbezeichner das Programm deklassifiziert. Wenn dieser Scheduler festgelegt ist und der Angreifer den Scheduler kennt, kann der aktive Angreifer mehr als der passive Angreifer lernen.

5.3.2. Bewertung und Ergebnis

Es ist eine neue Sicherheitsbedingung für MWL definiert, die D-Sicherheit[rd]. Diese Sicherheitsbedingung verbietet einem Angreifer auf einer Sicherheitsdomäne D_A Deklassifikation herbeizuführen. Genauer, ein aktiver Angreifer, der sein Angriffsprogramm ändern kann, lernt nicht mehr als ein passiver Angreifer, der sein Angriffsprogramm nicht ändern kann. Vorhandene Bedingungen dieser Art sind nur für sequentielle Programme definiert (s. [MSZ06]).

Die starke Sicherheit[in] kann die D-Sicherheit[rd] durchsetzen (s. Satz 21), daher kann man bestehende Sicherheitstypsysteme zur Überprüfung verwenden. Allerdings hat diese Methode einen Nachteil. Sie kann nicht die D-Sicherheit[rd] für alle Sicherheitsdomänen zeigen. Zum Beispiel ist $[\,\mathsf{l} := \mathsf{h}\,]$ mit $high \rightsquigarrow low$ low-sicher[rd] und $high$-sicher[rd]. Aber die beiden Sicherheitsdomänen stehen in der $\rightsquigarrow$-Flussrelation zueinander. Daher ist die Bedingung von Satz 21 nicht erfüllt.

Ein Nachteil der Sicherheit[rd] ist, dass sie keine Aussagen über schwächere passive Angreifer macht, wie der vorhergehenden Abschnitt erläutert. Die Annahme, dass der passive Angreifer entsprechend $\simeq_D$ Information lernen kann, ist in vielen praktischen Fällen zu stark, da er zum Beispiel auf eine Schedulerstrategie festgelegt ist (s. Abschnitt 5.3.1).

Als Ergebnis ist daher die Aussagekraft der Sicherheit[rd] in vielen praktischen Fällen eingeschränkt. Aber es ist gezeigt, dass grundsätzlich *robust-declassification*-Bedingungen mit Hilfe von *intransitive-noninterference*-Bedingungen durchsetzbar sind. Das heißt, das „wer“ der Deklassifikation im Sinne von aktiver oder passiver Angreifer lässt sich mit dem „wo“ der Deklassifikation begrenzen. Da man das „wo“ mit dem „was“ kombinieren kann (s. Abschnitt 4), sind alle drei Dimensionen der Deklassifikation abgedeckt.

5.3.3. Ausblick

Vergleich mit Sicherheitsbedingung aus [MSZ06] In [MSZ06] wird eine *robust-declassification*-Bedingung für sequentielle Programme definiert. Genaugenommen ist dort die Sicherheitsbedingung keine Bedingung für Programme, sondern für Pro-

grammkontexte. In diesen Programmkontexten fehlen an bestimmtem Stellen Teilprogramme. An diese Stellen können Angriffsprogramme eingesetzt werden. Eine Parallelkomposition wie bei der Definition der D-Sicherheit[rd] ist für sequentielle Programme nicht möglich. Die Definition der D-Sicherheit[rd] ist in diesem Punkt eher von der Sicherheitsbedingung aus [ZM01] inspiriert. Dort läuft der Angriff auch parallel zu dem eigentlichen Programm.

Ein weiterer Unterschied ist die Art der Flusspolitik. In [MSZ06] besteht eine Sicherheitsdomäne aus zwei Komponenten, eine für die Vertraulichkeit, die andere für die Integrität. Diese beiden Komponenten sind jeweils in einem eigenen Verband geordnet, wobei die gesamte Flussrelation das Produkt der beiden Verbände ist. Dieses Produkt der beiden Verbände ist eine MLS-Politik. Aber welche Teile eines Zustands für einen Angreifer sichtbar sind, hängt nur von der Vertraulichkeitskomponente der Sicherheitsdomänen ab.

Die Variante der Flusspolitik in der vorliegenden Arbeit ist technisch einfacher zu handhaben. Aber mit der Bedingung, dass Deklassifikation nur zwischen Sicherheitsdomänen höherer Integrität[21] als der des Angreifers erlaubt ist, sollte sich der Satz 21 auf die Variante aus [MSZ06] übertragen lassen. Denn mit der höheren Integrität ist die Bedingung des Satzes bezüglich des Produktverbands erfüllt.

Die komplexere Variante der Flusspolitik wird in [MSZ06] genutzt, um als Gegensatz zur Deklassifikation ein sogenanntes *endorsement* einzuführen, das die Integrität erhöht. Damit ist es dem Angreifer begrenzt erlaubt, die Deklassifikation zu beeinflussen (s. *qualified robustness* aus [MSZ06]).

Schedulerabhängige *Robust Declassification* Wie im Abschnitt 5.3.1 beschrieben, macht die D-Sicherheit[rd] im Allgemeinen keine Aussagen über schwächere passive Angreifer. Praktisch kommen diese aber in vielen Fällen vor.

Daher benötigt man eine schedulerabhängige Sicherheitsbedingung für *robust declassification*. Zum Beispiel könnte man mit Schedulern wie aus [SS00] eine schedulerabhängige Konfigurationsbisimulation definieren. Diese könnte man statt der starken D-Konfigurationsbisimulation $\simeq_D$ in die Definition der Sicherheitsbedingung einsetzen.

Interessant ist die Frage, ob es für solche schedulerabhängigen Sicherheitsbedingungen ähnliche Aussagen wie Satz 21 gibt. Zumindest verhindert eine MLS-Politik entsprechend der Bedingung von Satz 21 die Beeinflussung der Deklassifikation durch direkten oder indirekten Informationsfluss.

Robust declassification wird in dieser Arbeit nur für MWL ohne Synchronisationsanweisungen betrachtet, denn Synchronisation beeinflusst den Scheduler. Wenn Lösungen für schedulerabhängige Sicherheitsdefinitionen gefunden sind, kann auch versucht werden, eine Sicherheitsbedingung für MWL mit Synchronisation zu definieren.

[21] zu beachten ist, dass „höhere Integrität“ „weiter unten im Integritätsverband“ bedeutet, im Gegensatz zur Vertraulichkeit

6. Schluss

6.1. Zusammenfassung

Diese Arbeit bringt Ergebnisse im Entwicklungsteil (s. Abschnitt 3) und im theoretischen Teil (s. Abschnitte 4 und 5) hervor.

6.1.1. EIFA

Im Entwicklungsteil sind die Hauptergebnisse, dass im EIFA die Erweiterungen implementiert sind und dass der Entwurf diese Änderungen unterstützt hat.

Die Ziele der EIFA-Weiterentwicklung sind erreicht. Der EIFA prüft MWL-Programme gegenüber einer MLS-Politik (ohne oder mit Ausnahmen) und mit Sicherheitstypregeln für Synchronisationsanweisungen und Deklassifikationsanweisungen. Daneben sind Fehler behoben und es lassen sich besser lesbare MWL-Programme prüfen (zum Beispiel mit Kommentaren).

Die Erfahrungen durch die Entwicklung zeigen (s. Abschnitt 3), dass der Entwurf des EIFAv1 den Entwickler dabei unterstützt die Sicherheitstypregeln zu ändern, zu erweitern oder zu ersetzen. Die wichtigste Rolle spielt dabei das Besuchermuster, das die Datenstrukturen (hier der AST) unabhängig von der Implementierung der Sicherheitstypsysteme macht. Für den Fall, dass man viele Regelkombinationen zulassen will, unterstützt einen der Einsatz des Dekorierermusters, das im Rahmen dieser Arbeit für die Sicherheitstypsysteme für MLS-Politiken (s. Abschnitt 3.3.5) eingesetzt wurde. Damit vermeidet man viele Doppelimplementierungen von Sicherheitstypregeln.

Die Erfahrungen durch die Entwicklung zeigen aber auch den Nachteil des Besuchermusters. Das Besuchermuster macht es aufwendiger die Datenstrukturen zu ändern.

Weitere Details zu diesen Ergebnissen stehen im Abschnitt 3.4.1.

6.1.2. *Delimited Release*

Hier sind die Hauptergebnisse die Definition einer Sicherheitsbedingung für MWL (die starke Sicherheit[dr]) und ihre Kombination mit der starken Sicherheit[in]. Die starke Sicherheit[dr] erlaubt Deklassifikation. Aber sie schränkt ein, welche Information deklassifizierbar ist. Die starke Sicherheit[dr] ist mit einem Sicherheitstypsystem überprüfbar. Außerdem kann sie sinnvoll in Konjunktion mit der starken Sicherheit[in] angewendet werden. Man kann also das „wo“ und das „was“ der Deklassifikations beschränken.

In Abschnitt 4.2.2 wird diese Sicherheitsbedingung definiert. Sie nennt sich „starke Sicherheit[dr]“ („starke Sicherheit für *delimited release*“), weil sie ähnlich der starken Sicherheit aus [MS04] definiert ist und diese mit Ideen aus [SM04] zur Spezifikation der deklassifizierbaren Information (*delimited release*) kombiniert.

Beispiele zeigen, dass die starke Sicherheit[dr] sinnvoll eingesetzt werden kann.

Die starke Sicherheit[dr] hat ähnliche Zusammensetzbarkeitseigenschaften, wie die starke Sicherheit (s. Abschnitt 4.2.3). Diese Eigenschaften werden im Abschnitt 4.3 genutzt, um mit wenigen Änderungen des Sicherheitstypsystems für starke Sicherheit ein Sicherheitstypsystem anzugeben, für das typisierbare Programme stark sicher[dr] sind. Aufgrund der wenigen Änderungen ließe sich das Sicherheitstypsystem ohne großen Aufwand im EIFA implementieren.

Die starke Sicherheit[dr] unterscheidet sich von anderen Sicherheitsbedingungen für das „was“ der Deklassifikation darin, dass ein automatischer Prüfmechanismus bekannt ist und dass sie gleichzeitig für nebenläufige Programme definiert ist.

Man kann die starke Sicherheit[dr] unter bestimmten Bedingungen sinnvoll mit der starken Sicherheit[in] kombinieren (s. Abschnitt 4.2.5). Das Sicherheitstypsystem berücksichtigt diese Bedingungen, sodass man sinnvolle MWL-Programme schreiben kann, die nicht nur stark sicher[in] und stark sicher[dr] sind, sondern auch mit beiden Sicherheitstypsystemen typisierbar sind.

Diese Kombination zweier Sicherheitsbedingungen für zwei Dimensionen der Deklassifikation (s. [SS05]) ist ein Schritt zu einer umfassenden Sicherheitsbedingung für Deklassifikation.

6.1.3. *Robust Declassification*

Hier sind die Hauptergebnisse die Definition einer Sicherheitsbedingung für MWL (die D-Sicherheit[rd]) und ihre Durchsetzung mit der starken Sicherheit[in]. Die D-Sicherheit[rd] erlaubt Deklassifikation, verbietet aber einem Angreifer der Sicherheitsdomäne D aktiv die Deklassifikation so zu beeinflussen, dass er mehr lernt als ein passiver Angreifer. Sie begrenzt also, „wer“ die Deklassifikation beeinflussen kann.

Im Abschnitt 5.1 wird diese Sicherheitsbedingung definiert. Sie nennt sich D-Sicherheit[rd] (Sicherheit für *robust declassification*), weil sie ähnlich der *robust-declassification*-Sicherheitsbedingung aus [MSZ06] definiert ist. Allerdings entspricht die D-Sicherheit[rd] nur unter in vielen Fällen unrealistischen Annahmen der gewünschten Bedeutung (s. Abschnitt 5.3.1).

Im Abschnitt 5.2 wird gezeigt, dass die D-Sicherheit[rd] mit bestimmten MLS-Politiken aus der starken Sicherheit[in] folgt.

Mit diesen Ergebnissen gibt es für MWL neben der starken Sicherheit[in] für das „wo“ der Deklassifikation und der starken Sicherheit[dr] für das „was“ der Deklassifikation auch eine Bedingung für das „wer“ der Deklassifikation. Es lässt sich Deklassifikation in allen drei Dimensionen beschränken.

6.2. Ähnliche Arbeiten

Einen Überblick über die programmiersprachenbasierte Sicherheit im Allgemeinen bietet [SM03] und über Deklassifikation im Speziellen bietet [SS05].

Neben den theoretischen Arbeiten gibt es wenige Implementierungen, die wie der EIFA statisch die Sicherheit von Programmen überprüfen. Eine Implementierung überprüft *Jif*, eine Abwandlung von *Java*, nach dem sogenannten *decentralized label*

model (s. [ML00]). Allerdings ist davon die Korrektheit gegenüber einer Sicherheitsbedingung wie *noninterference* nicht bewiesen.

Wie in [SS05] dargestellt, gibt es viele Ansätze zur Deklassifikation. *Abstract noninterference* (s. [GM04]) und Lowes Ansatz, um die deklassifizierte Information zu quantifizieren (s. [Low04]) werden schon im Abschnitt 4.4.2 aufgeführt. Ein weiterer Ansatz für das „was“ der Deklassifikation ist zum Beispiel *relaxed noninterference* (s. [LZ05]). Hier gibt es statt Sicherheitsdomänen Lambdaterme, die den Informationsfluss beschreiben. Sogenannte Aktionen (ebenfalls Lambdaterme) ermöglichen die Deklassifikation. Für *relaxed noninterference* gibt es ein Sicherheitstypsystem.

Die Sicherheitsbedingungen in dieser Arbeit benutzen eine punktweise Bisimulation. Das heißt jedes einzelne Programmpaar zweier Programmvektoren muss bisimular zueinander sein, damit die Programmvektoren bisimular sind. Diese Einschränkung soll dazu dienen, eine probabilistische, schedulerunabhängige Sicherheit zu erreichen (s. [SS00]), obwohl die Sicherheitsbedingung selbst rein possibilistisch formuliert ist. Andere Sicherheitsbedingungen versuchen keine probabilistischen Aussagen zu machen (s. [BC02]). Entsprechend sind auch die Sicherheitstypsysteme für diese Sicherheitsbedingungen weniger restriktiv. Zum Beispiel erlaubt das Sicherheitstypsystem aus [BC02] Schleifenbedingungen ungleich der Sicherheitsdomäne *low*, wenn anschließend in dem Thread der Schleife keine Zuweisung an eine kleinere Sicherheitsdomäne als die der Schleifenbedingung auftritt.

6.3. Ausblick

Dieser Abschnitt fasst zusammen, welche weiteren interessanten Ideen und Aufgaben mit den Themen und Ergebnissen dieser Arbeit verknüpft sind.

Den EIFA könnte man zum einem um weitere Sprachelemente und weitere Sicherheitstypsysteme erweitern. Zum anderen könnte man Ansätze überlegen, wie man noch einfacher Sicherheitstypsysteme für neue und sich mehrmals ändernde Sprachen implementieren könnte. Ein Vorschlag ist, noch mehr Teile des Systems automatisch zu generieren, zum Beispiel mit *antlr* (s. [Par06]). Weitere Details dazu stehen im Abschnitt 3.4.2.

Auf längere Sicht besteht das Ziel, eine Anwendung wie den EIFA für eine angewandte Programmiersprache zu entwickeln, zum Beispiel für Java. Dazu muss man zunächst theoretisch den Einfluss vieler weiterer Sprachelemente auf den Informationsfluss untersuchen, zum Beispiel Klassen und Objekte, Methodenaufrufe mit Polymorphismus und Ausnahmen.

Für *delimited release* beschreibt Abschnitt 4.4.2 weitere interessante Probleme und Ansätze. Am wichtigsten ist es zu untersuchen, wie die starke Sicherheit[dr] mit weiteren, die Eingabe und Ausdrücke betreffenden Sprachelementen zusammenpasst. Insbesondere bei Funktionen und Prozeduren mit lokalen Variablen muss man sich überlegen was es bedeutet, wenn ein Ausdruck eine *escape-hatch* ist. Mit Funktionen muss man auch überlegen ob und wenn, unter welchen Bedingungen, Funktionsaufrufe in *escape-hatches* vorkommen dürfen.

Literatur

[BC02] G. Boudol und C. Ilaria: *Noninterference for concurrent programs and thread systems.* Theoretical Computer Science, 281(1-2):109–130, Juni 2002.

[Ber03] E. Berk: *JLex: A Lexical Analyzer Generator for Java(TM).* http://www.cs.princeton.edu/ appel/modern/java/JLex/, 2003.

[Coh78] E. Cohen: *Information Transmission in Sequential Programs.* In: *Foundations of Secure Computation*, Seiten 297–335. Academic Press, 1978.

[Den76] D. E. Denning: *A lattice model of secure information flow.* Communications of the ACM, 19(5):236–243, 1976.

[Fou06] The Eclipse Foundation: *Eclipse Java Development Tools (JDT).* http://www.eclipse.org/jdt/, 2006.

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, 1995.

[GM04] R. Giacobazzi und I. Mastroeni: *Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation.* In: *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Seiten 186–197. ACM-Press, NY, 2004. Venice, Italy, Januar 14-16,2004.

[HM05] S. Hunt und I. Mastroeni: *The PER of abstract Non-Interference.* In: *Static Analysis Symposium (SAS'05)*, Band 3672 der Reihe *Lecture Notes in Computer Science*, Seiten 171–185. Springer-Verlag, 2005.

[Hud99] S. E. Hudson: *CUP Parser Generator for Java.* http://www.cs.princeton.edu/ appel/modern/java/CUP/, 1999.

[Kle04] G. Klein: *JFlex - The Fast Scanner Generator for Java.* http://jflex.de/, 2004.

[KM05] B. Köpf und H. Mantel: *Eliminating implicit information leaks by transformational typing and unification.* Technischer Bericht 498, ETH Zürich, Oktober 2005.

[Lam05] M. Lamb: *Jsap: the java-based simple argument parser.* http://www.martiansoftware.com/jsap/, 2005.

[Low04] G. Lowe: *Defining information flow quantity.* Journal of Computer Security, 12(3-4):619–653, 2004.

[LZ05] P. Li und S. Zdancewic: *Downgrading policies and relaxed noninterference.* In: *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 158–170, New York, NY, USA, 2005. ACM Press.

[ML00] A. C. Myers und B. Liskov: *Protecting privacy using the decentralized label model.* ACM Trans. Softw. Eng. Methodol., 9(4):410–442, 2000.

[MS04] H. Mantel und D. Sands: *Controlled declassification based on intransitive noninterference.* Technischer Bericht 2004-06, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, 2004.

[MSZ06] A. C. Myers, A. Sabelfeld, und S. Zdancewic: *Enforcing robust declassification and qualified robustness.* Erscheint in: Journal of Computer Security, 2006.

[Par06] T. Parr: *Antlr parser generator.* http://www.antlr.org/, 2006.

[Pöp05] C. Pöpper: *A security analyzer for multi-threaded programs.* Diplomarbeit, ETH Zürich, 2005.

[Rus92] J. Rushby: *Noninterference, transitivity and channel-control security policies.* Technischer Bericht CSL-92-02, SRI International, 1992.

[Sab01] A. Sabelfeld: *The impact of synchronisation on secure information flow in concurrent programs.* In: *Ershov Memorial Conference*, Seiten 225–239, 2001.

[SM02] A. Sabelfeld und H. Mantel: *Static confidentiality enforcement for distributed programs.* In: *Proc. Symp. on Static Analysis*, Band 2477 der Reihe *Lecture Notes in Computer Science*, Seiten 376–394. Springer-Verlag, 2002.

[SM03] A. Sabelfeld und A. C. Myers: *Language-based information-flow security.* IEEE Journal on Selected Areas in Communications, 21(1):5–19, Januar 2003.

[SM04] A. Sabelfeld und A. C. Myers: *A model for delimited information release.* In: *Proceedings of the International Symposium on Software Security (ISSS'03)*, 2004.

[SS00] A. Sabelfeld und D. Sands: *Probabilistic noninterference for multi-threaded programs.* In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW '00)*, Seiten 200–215, Washington - Brussels - Tokyo, Juli 2000. IEEE.

[SS05] A. Sabelfeld und D. Sands: *Dimensions and principles of declassification.* In: *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, Seiten 255–269, 2005.

[ZM01] S. Zdancewic und A. Myers: *Robust declassification.* In: *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, Seiten 15–26, Washington - Brussels - Tokyo, Juni 2001. IEEE.

Abbildungsverzeichnis

1. Flusspolitik mit zwei Sicherheitsdomänen 15
2. MWL-Grammatik 18
3. Semantische Auswertung von MWL-Ausdrücken 19
4. Deterministische Smallstep-Semantik der MWL in der Grundform . . 19
5. Nichtdeterministische Smallstep-Semantik der MWL in der Grundform 20
6. Deterministische Smallstep-Semantik der MWL mit Deklassifikation . 22
7. Nichtdeterministische Smallstep-Semantik der MWL mit Deklassifikation. 22
8. MWL Programm: Verarbeitung eines Webformulars (Beispiel 1) . . . 25
9. MWL-Programm: Mehrheitsauszählung (Beispiel 2) 29
10. MWL-Programm: Freigabe nach zweifacher Überprüfung (Beispiel 3) 31
11. Sicherheitstypregeln für Ausdrücke unter einer MLS-Politik 32
12. Sicherheitstypregeln für die starke Sicherheit[in] von MWL-Anweisungen 33
13. Sicherheitstypregeln die starke Sicherheit[in] von Synchronisationsanweisungen 34
14. Klassenstruktur des Pakets ast im EIFAv1 39
15. Vererbungsstruktur des Pakets visitor im EIFAv1 (übernommen aus [Pöp05]) 41
16. Ablauf der Generierung des AST aus dem Quellcode (übernommen aus [Pöp05]) 42
17. Im EIFA implementierte MWL-Syntax 43
18. Im EIFA implementierte Syntax der MLS-Politik-Spezifikation 45
19. Klassenstruktur der wichtigsten Klassen des Pakets policy. 46
20. Klassenbeziehungen der Klasse Decl im EIFA. 47
21. Klassen zur Repräsentation von Synchronisations- und Deklassifikationsanweisungen im EIFA 48
22. Klassen zur Repräsentation von Ausdrücken im EIFA. 48
23. Klassen zur Repräsentation von Zuweisungen im EIFA. 50
24. Klassen zur Repräsentation von Bezeichnern im EIFA. 50
25. Besucherhierarchie für MWL mit MLS-Politik (Mehrfachverwendung durch Vererbung). 53
26. Besucherstruktur des EIFA für MWL mit MLS-Politik (Mehrfachverwendung durch Dekorierermuster). 54
27. Sequenzdiagramm für visit -Aufrufe mit dem Dekorierermuster 55
28. MWL-Programm: Virusüberprüfung einer E-Mail (Beispiel 8) 76
29. Neue Sicherheitstypregeln für die starke Sicherheit[dr] von MWL-Programmen 81
30. Unterteilung der syntaktischen Flusspolitikklassen 87
31. Flusspolitik, die *robust declassification* durchsetzen soll 94
32. Semantik von Arrayausdrücken 117
33. Deterministische Smallstep-Semantik der MWL mit Deklarationen und Arrayanweisungen 118

34. Deterministische Smallstep-Semantik der MWL mit Synchronisationsanweisungen 118
35. Nichtdeterministische Smallstep-Semantik der MWL mit Synchronisationsanweisungen 119
36. Sicherheitstypregeln für Ausdrücke unter der binären Flusspolitik . . 120
37. Transformierende Sicherheitstypregeln für Synchronisationsanweisungen unter der binären Flusspolitik. 120
38. Nicht-Transformierende Sicherheitstypregeln für MWL-Anweisungen unter der binären Flusspolitik. 121
39. Transformierende Sicherheitstypregeln für MWL-Anweisungen unter der binären Flusspolitik. 122
40. Im EIFAv1 implementierte MWL-Syntax 137

A. Aufgabenstellung

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Diplomarbeit

Analyzing Multi-Threaded Programs under Intransitive Security Policies

Analyse nebenläufiger Programme unter intransitiven Sicherheitspolitiken

Contact: mantel@cs.rwth-aachen.de

Introduction

Protecting the confidentiality of information is an important problem in modern networked information systems. In particular, the use of mobile code creates additional threats.

Imagine the following scenario: You want to run an application from a software supplier that you do not fully trust. The application (e.g. a spreadsheet) might need confidential (*high*) data to perform its task. On the other hand, there might be communication of seemingly uncritical (*low*) data (e.g. a registration process) with the supplier of the software. The question is how to ensure that the program does not "leak" the secret data, neither accidentally (bugs in the program) nor on purpose (a Trojan Horse).

An *Information Flow* analysis is a possible answer to such threats. Its purpose is to check that there is no secret information leaking from high input to low output. Possible leaks include explicit assignments such as in statements like $l := h$ or, more subtly, in **if** $h = 1$ **then** $l := 1$ **else** $l := 0$, where one can draw conclusions on the (secret) value of the high variable h only by observing the (non-secret) value of the low variable l. Recently, so called *Security Typed* Systems have been developed for mechanizing information flow analyzes. For instance, the rules for typing assignments and conditionals read as follows:

$$\frac{exp : \mathrm{low}}{\vdash l := exp} \qquad \frac{exp : \mathrm{low} \quad \vdash C_1 \quad \vdash C_2}{\vdash \textbf{if } exp \textbf{ then } C_1 \textbf{ else } C_2}$$

Informally, the rule on the right hand side can be read: A conditional can be typed if each branch can be typed and the guarding condition is of "low" security type.

Apart from toy examples as above, today's security typed languages help avoiding many subtle leaks and cover a broad range of interesting language constructs, including distributedness, concurrency, synchronization and declassification.

Based on an existing security type system [SS00], Pöpper developed a security analyzer for a multi-threaded while language (short: MWL) in her diploma thesis [Pöp05]. This analyzer was designed with the aim to easily accommodate new language features and changes to the security type system by adapting the implementation. The flexibility of the implementation was tested by implementing a non-transforming variant of the original (transforming) security type system and by extending the type system with typing rules for arrays.

Project Objectives

Core: The objective of this thesis is three-fold:

Firstly, the security analyzer shall be extended to support a richer programming language and a richer security policy language. This extension will make the security analyzer applicable to more interesting example programs. In the first extension, the MWL is extended with primitives for synchronizing threads. The typing rules will be taken from [Sab01]. In the second extension, primitives for declassification will be added and the policy language will be augmented according to [MS04]. This will allow one to define multi-level security policies that permit declassification. Both of these extensions serve as experiments to test: How flexible is the implementation of the security analyzer with respect to such changes?

Secondly, the two extensions (synchronization, declassification) shall be combined. This requires an extension of the underlying theory: An appropriate security condition must be defined, a security type system must be developed, and the soundness of the type system must be shown with respect to the security condition. The security type system shall be implemented and the extended analyzer shall be tested with interesting example programs.

Thirdly, the control of declassification shall be refined. While intransitive noninterference [MS04] controls *where* information can be released, other approaches control *what* information can be released or *who* can initiate a declassification [SS05]. The different approaches to controlling declassification were developed mostly independently and are believed to be fairly orthogonal. The third objective of this thesis is to develop an integrated approach that controls *where as well as what* information can be released. This requires an adaptation of the policy language, the security condition, the type system, and the soundness proof.

Extension 1: After a successful integration of controlling *where* information can be released with controlling *what* information can be released, it would be interesting to see whether it is possible to also integrate a control of *who* can initiate declassification.

Extension 2: In [KM05] a transforming security type system was proposed that uses unification to compute substitutions that eliminate some insecurities in a program. The approach primarily aims at security policies with two security levels, high and low. An extension to policies with more than two domains was sketched, but this extension is rather primitive and appears inefficient. It would be interesting to extend this approach such that it can cope more directly with security policies comprising more than two security domains. This adaptation is a prerequisite for adopting the unification-based approach to security policies that permit controlled declassification.

Extension 3: The language-based variant of intransitive noninterference [MS04] is suitable for policies that explicitly distinguish between the ordering of a security lattice $\leq$ and the downgrading relation $\rightsquigarrow$. This differs from the original definition of intransitive noninterference [Rus92], which supports policies with only a single information-flow relation. It would be interesting to develop a language-based variant of intransitive noninterference that is capable of supporting such policies. Such a variant might provide a suitable basis for supporting more refined policies, which allow one to control intransitive information flow with additional conditions.

Main Activities

Defining a schedule for the entire project

Exploring the adaptability of the security analyzer includes

- determining the extent of the necessary modifications to the security analyzer (parser, data structures, type system, ...),
- implementing the two extensions (declassification, downgrading),
- testing and evaluating the extended tool using suitable example programs, and
- evaluating the adaptability of the security analyzer based on the experiences made.

Integrating the Solutions for Declassification and Synchronization includes

- adapting the security condition,
- developing a suitable security type system and proving its soundness,
- implementing the type system in the security analyzer, and
- testing and evaluating the extended tool using suitable example programs.

Refining the Control of Declassification includes

- analyzing existing approaches with respect to the possibility of integrating them with intransitive noninterference,
- adapting the security condition,
- developing a suitable security type system and proving its soundness, and
- testing and evaluating the extended type system using suitable example programs (possibly after an implementation).

Deliverables

The diploma thesis shall include:

- brief user documentation for the security analyzer
- detailed development documentation for the extensions and modifications of the security analyzer
- detailed presentation of security conditions, type sytems, and soundness proofs as described in the prior sections
- detailed explanation of design choices made (description of alternatives, discussion of their advantages and disadvantages, arguments for the chosen solution, discussion of decision in retrospective)
- description elaborating on insights gained, of open problems identified, and possible extensions of the system or the type system that could be pursued in the future

The tool shall be supplied in electronic form (well-documented Java source code and executable).

A talk at the end shall present the main results of the diploma thesis and could include a tool demonstration.

Supervision

Prof. Dr. Heiko Mantel (mantel@cs.rwth-aachen.de)

References

[KM05] B. Köpf and H. Mantel. Eliminating Implicit Information Leaks by Transformational Typing and Unification. In *Pre-workshop proceedings of FAST'05: Workshop on Formal Aspects in Security and Trust*, 2005.

[MS04] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems, APLAS 2004*, LNCS 3303, pages 129–145, Taipei, Taiwan, November 4–6 2004. Springer-Verlag.

[Pöp05] C. Pöpper. A Security Analyzer for Multi-Threaded Programs. Master's thesis, ETH Zurich, March 2005.

[Rus92] J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.

[Sab01] A. Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239, 2001.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, 2000.

[SS05] A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, 2005.

B. Regelauflistungen

Dieser Abschnitt listet Regelergänzung zu Abschnitt 2 auf, um die Semantik- und Sicherheitstypregeln zu vervollständigen.

B.1. Semantikregeln

Hier sind ergänzend zu den Semantikregeln aus Abschnitt 2.2 die Regeln für Deklarations-, Array- und Synchronisationsanweisungen aufgeführt.

B.1.1. Deklarations- und Arrayanweisungen

Diese Regeln sind aus [Pöp05] übernommen. Ein Arraybezeichner Arr wird zu einer indizierten Folge über der Wertemenge $\mathcal{V}$ ausgewertet. Die Auswertung von Arrayausdrücken ist über die Regeln in Abbildung 32 definiert. $default$ ist ein ausgezeichneter Wert aus der Wertemenge $\mathcal{V}$.[22]

$$\frac{\langle Arr, s\rangle \downarrow [Arr_0, \ldots, Arr_{l-1}]}{\langle Arr\ .\mathsf{length}\ , s\rangle \downarrow l}$$

$$\frac{\langle Arr, s\rangle \downarrow [Arr_0, \ldots, Arr_{l-1}] \qquad \langle Exp, s\rangle \downarrow k \qquad k \in \mathbb{N} \qquad 0 \leq k < l}{\langle Arr\ [\ Exp\ ]\ , s\rangle \downarrow Arr_k}$$

$$\frac{\langle Arr, s\rangle \downarrow [Arr_0, \ldots, Arr_{l-1}] \qquad \langle Exp, s\rangle \downarrow k \qquad (k \notin \mathbb{N} \ \vee\ k \geq l)}{\langle Arr\ [\ Exp\ ]\ , s\rangle \downarrow default}$$

Abbildung 32: Semantik von Arrayausdrücken

Die Semantik von Anweisungen aus MWL in der Grundform wird um die Regeln in der Abbildung 33 erweitert.

B.1.2. Synchronisationsanweisungen

Diese Regeln sind aus [Sab01] übernommen.

Die Semantikregeln für die Synchronisationsanweisungen sind in Abbildung 34 definiert.

Da für MWL mit Synchronisationsanweisungen die Konfigurationen um Warteschlangen erweitert sind, wird die nichtdeterministische Semantik von MWL durch eine neue Semantik ersetzt (definiert in Abbildung 35).

[22] [Pöp05] benutzt hier spezielle Werte für die Wertebereiche der Datentypen **bool** und **int**. Das ist aber solange Datentypsicherheit gegeben ist für die Sicherheit des Programms irrelevant, da dieser Wert keine Information enthält.

$$\frac{}{\langle\!| v : \text{Type:}SecDom, s |\!\rangle \rightarrowtriangle \langle\!| (), [v = default]s |\!\rangle}$$

$$\frac{\langle\!| Exp, s |\!\rangle \downarrow l}{\begin{array}{l}\langle\!| Arr : \textsf{bool}[\ Exp\ ]\text{: } SecDom, s |\!\rangle \rightarrowtriangle \\ \quad \langle\!| (), [Arr_0 = default, \ldots, Arr_{l-1} = default]s |\!\rangle\end{array}}$$

$$\frac{\begin{array}{l}\langle\!| Arr, s |\!\rangle \downarrow [Arr_0, \ldots, Arr_{l-1}] \\ \langle\!| Exp_2, s |\!\rangle \downarrow n \qquad \langle\!| Exp_1, s |\!\rangle \downarrow k \quad k \in \mathbb{N} \quad 0 \leq k < l\end{array}}{\langle\!| Arr\ [\ Exp_1\ ]{:=}\ Exp_2, s |\!\rangle \rightarrowtriangle \langle\!| (), [Arr_k = n]s |\!\rangle}$$

$$\frac{\langle\!| Arr, s |\!\rangle \downarrow [Arr_0, \ldots, Arr_{l-1}] \quad \langle\!| Exp_1, s |\!\rangle \downarrow k \quad (k \notin \mathbb{N} \vee k \geq l)}{\langle\!| Arr\ [\ Exp_1\ ]{:=}\ Exp_2, s |\!\rangle \rightarrowtriangle y\langle\!| (), s |\!\rangle}$$

Abbildung 33: Deterministische Smallstep-Semantik der MWL mit Deklarationen und Arrayanweisungen (erweitert Semantik aus Abbildung 4)

$$\frac{}{\langle\!| Sem : \textsf{sem}\text{: } SecDom, s |\!\rangle \rightarrowtriangle \langle\!| (), [Sem = 0]s |\!\rangle}$$

$$\frac{\langle\!| Sem, s |\!\rangle \downarrow n \quad n > 0}{\langle\!| \textsf{wait}(Sem), s |\!\rangle \rightarrowtriangle \langle\!| (), [Sem = Sem - 1]s |\!\rangle} \qquad \frac{\langle\!| Sem, s |\!\rangle \downarrow n \quad n = 0}{\langle\!| \textsf{wait}(Sem), s |\!\rangle \overset{\otimes Sem}{\rightarrowtriangle} \langle\!| (), s |\!\rangle}$$

$$\frac{}{\langle\!| \textsf{signal}(Sem), s |\!\rangle \overset{\odot Sem}{\rightarrowtriangle} \langle\!| (), s |\!\rangle} \qquad \frac{\langle\!| C_1, s |\!\rangle \overset{\alpha}{\rightarrowtriangle} \langle\!| C_1', s' |\!\rangle \quad \alpha \in \{\odot Sem, \otimes Sem\}}{\langle\!| C_1; C_2, s |\!\rangle \overset{\alpha}{\rightarrowtriangle} \langle\!| C_1'; C_2, s' |\!\rangle}$$

Abbildung 34: Deterministische Smallstep-Semantik der MWL mit Synchronisationsanweisungen (erweitert Semantik aus Abbildung 4, erst Regel gilt nur wenn auch Deklarationen definiert sind).

$$\frac{\langle\!| C_i, s |\!\rangle \rightarrowtriangle \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| \langle C_0 \dots C_{n-1} \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_0 \dots C_{i-1} \vec{C} C_{i+1} \dots C_{n-1} \rangle, w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\otimes Sem}{\rightarrowtriangle} \langle\!| C', s' |\!\rangle \quad w_{Sem} = \vec{D}}{\langle\!| \langle C_0 \dots C_{n-1} \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_0 \dots C_{i-1} C_{i+1} \dots C_{n-1} \rangle, [w_{Sem} = \vec{D} C'] w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot Sem}{\rightarrowtriangle} \langle\!| C', s' |\!\rangle \quad w_{Sem} = C\vec{D}}{\langle\!| \langle C_0 \dots C_{n-1} \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_0 \dots C_{i-1} C' C_{i+1} \dots C_{n-1} C \rangle, [w_{Sem} = \vec{D}] w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot Sem}{\rightarrowtriangle} \langle\!| C', s' |\!\rangle \quad w_{Sem} = ()}{\langle\!| \langle C_0 \dots C_{n-1} \rangle, w, s |\!\rangle \rightarrow \langle\!| \langle C_0 \dots C_{i-1} C' C_{i+1} \dots C_{n-1} \rangle, w, [Sem = Sem + 1] s' |\!\rangle}$$

Abbildung 35: Nichtdeterministische Smallstep-Semantik der MWL mit Synchronisationsanweisungen (ersetzt Semantik aus Abbildung 5).

B.2. Sicherheitstypsysteme des EIFA

Hier sind ergänzend zum Abschnitt 2.4 Sicherheitstypregeln der im EIFA implementieren Sicherheitstypsysteme aufgelistet.

Das sind ein nicht-transformierendes (s. Abbildung 38) und ein transformierendes Sicherheitstypsystem (s. Abbildung 39) aus [Pöp05] für MWL ohne Synchronisationsanweisungen und ohne Deklassifikationsanweisung. Diese Sicherheitstypsysteme sind eine Modifikation des Sicherheitstypsystems aus [SS00]. Sie ermöglichen die starke Sicherheit von Programmen bezüglich einer binären Sicherheitspolitik zu beweisen. Diese beiden sind im EIFAv1 implementiert. Als weiteres Sicherheitstypsystem sind in [Sab01] transformierende Sicherheitstypregeln für Synchronisationsanweisungen definiert (s. Abbildung 37). Die Korrektheit der Sicherheitstypsysteme zeigen jeweils die Autoren in den oben genannten Veröffentlichungen.

Der EIFA stellt dem Benutzer folgende Sicherheitstypsysteme zur Auswahl:

1. nichttransformierendes Typsystem für starke Sicherheit und eine binäre Flusspolitik (Regeln aus den Abbildungen 36, 38), wie auch schon im EIFAv1 implementiert

2. transformierendes Typsystem für starke Sicherheit und eine binäre Flusspolitik (Regeln aus den Abbildungen 36, 39), wie auch schon im EIFAv1 implementiert

3. nichttransformierendes Typsystem für starke Sicherheit mit Synchronisation und eine binäre Flusspolitik (Regeln aus den Abbildungen 36, 38, 13),

4. transformierendes Typsystem für starke Sicherheit mit Synchronisation und eine binäre Flusspolitik (Regeln aus den Abbildungen 36, 39, 37),

5. nichtransformierendes Typsystem für starke Sicherheit ohne Deklassifikationsanweisungen und eine MLS-Politik (Regeln aus den Abbildungen 11, 12, ohne [Declass]),
6. nichtransformierendes Typsystem für starke Sicherheit[in] und eine MLS-Politik (Regeln aus den Abbildungen 11, 12),
7. nichtransformierendes Typsystem für starke Sicherheit ohne Deklassifikationsanweisungen mit Synchronisation und eine MLS-Politik (Regeln aus den Abbildungen 11, 12, 13, ohne [Declass])
8. nichtransformierendes Typsystem für starke Sicherheit[in] mit Synchronisation und eine MLS-Politik (Regeln aus den Abbildungen 11, 12, 13)

[Const] $\dfrac{}{\Gamma \vdash Const : low}$ **[Low]** $\dfrac{\Gamma(l) = low}{\Gamma \vdash l : low}$ **[High]** $\dfrac{\Gamma(h) = high}{\Gamma \vdash h : high}$

[Op$_{low}$] $\dfrac{\Gamma \vdash Exp_1 : low \quad \Gamma \vdash Exp_2 : low}{\Gamma \vdash Exp_1\ op\ Exp_2 : low}$

[Op$_{high}$ 1] $\dfrac{\Gamma \vdash Exp_1 : high}{\Gamma \vdash Exp_1\ op\ Exp_2 : high}$ **[Op$_{high}$ 2]** $\dfrac{\Gamma \vdash Exp_2 : high}{\Gamma \vdash Exp_1\ op\ Exp_2 : high}$

[ArrLen] $\dfrac{}{\Gamma \vdash Arr\ .\mathsf{length} : low}$

[Arr$_{low}$] $\dfrac{\Gamma \vdash Exp : low \quad \Gamma \vdash Arr : low}{\Gamma \vdash Arr\ [\ Exp\ ] : low}$ **[Arr$_{high}$]** $\dfrac{}{\Gamma \vdash Arr\ [\ Exp\ ] : high}$

Abbildung 36: Sicherheitstypregeln für Ausdrücke unter der binären Flusspolitik. Γ ist eine Domänenzuweisung.

[SemDecl] $\dfrac{}{\Gamma \vdash Sem{:}Type{:}D}$

[Wait] $\dfrac{\Gamma(Sem) = low}{\Gamma \vdash \mathsf{wait}(Sem) \ \hookrightarrow \mathsf{wait}(Sem) \ : \mathsf{wait}(Sem)}$

[Signal] $\dfrac{\Gamma(Sem) = low}{\Gamma \vdash \mathsf{signal}(Sem) \ \hookrightarrow \mathsf{signal}(Sem) \ : \mathsf{signal}(Sem)}$

Abbildung 37: Transformierende Sicherheitstypregeln für Synchronisationsanweisungen unter der binären Flusspolitik. Diese Typregeln erweitern das Typsystem aus der Abbildung 39. Γ ist eine Domänenzuweisung.

$$\textsf{[VarDecl]} \qquad \frac{}{\Gamma \vdash Id{:}Type{:}D}$$

$$\textsf{[ArrDecl]} \qquad \frac{\Gamma \vdash Exp : low}{\Gamma \ \vdash \ Arr{:}Type[Exp]{:}D}$$

$$\textsf{[Skip]} \qquad \frac{}{\Gamma \vdash \textsf{skip}}$$

$$\textsf{[Assign}_{low}\textsf{]} \qquad \frac{\Gamma(l) = low \qquad \Gamma \vdash Exp : low}{\Gamma \vdash \ l{:=}Exp}$$

$$\textsf{[Assign}_{high}\textsf{]} \qquad \frac{\Gamma(h) = high}{\Gamma \vdash \ h{:=}Exp}$$

$$\textsf{[ArrAssign}_{low}\textsf{]} \qquad \frac{\Gamma(Arr) = low \quad \Gamma \vdash Exp_1 : low \quad \Gamma \vdash Exp_2 : low}{\Gamma \vdash \ Arr \ [\ Exp_1 \ ]{:=} \ Exp_2}$$

$$\textsf{[ArrAssign}_{high}\textsf{]} \qquad \frac{\Gamma(Arr) = high}{\Gamma \vdash \ Arr \ [\ Exp_1 \ ]{:=} \ Exp_2}$$

$$\textsf{[If}_{low}\textsf{]} \qquad \frac{\Gamma \vdash B : low \qquad \Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash \textsf{if}\, B\, \textsf{then}\, C_1\, \textsf{else}\, C_2}$$

$$\textsf{[While}_{low}\textsf{]} \qquad \frac{\Gamma \vdash B : low \qquad \Gamma \vdash C}{\Gamma \vdash \textsf{while}\, B\, \textsf{do}\, C}$$

$$\textsf{[Seq]} \qquad \frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1 \,; C_2}$$

$$\textsf{[Fork]} \qquad \frac{\Gamma \vdash C_1 \qquad \Gamma \vdash \vec{C_2}}{\Gamma \vdash \textsf{fork}(C_1 \vec{C_2})}$$

$$\textsf{[Par]} \qquad \frac{\Gamma \vdash C_0 \quad \ldots \quad \Gamma \vdash C_{n-1}}{\Gamma \vdash (C_0, \ldots, C_{n-1})}$$

Abbildung 38: Nicht-Transformierende Sicherheitstypregeln für MWL-Anweisungen unter der binären Flusspolitik. Die Typregeln für Ausdrücke sind in der Abbildung 36 definiert. Γ ist eine Domänenzuweisung.

$$\textbf{[VarDecl]}\quad \frac{}{\Gamma \vdash Id : Type : D \hookrightarrow Id : Type : D \ : \ Id : Type : D}$$

$$\textbf{[ArrDecl]}\quad \frac{\Gamma \vdash Exp : low}{\Gamma \ \vdash Arr : Type\ [\ Exp]{:}D \hookrightarrow Arr : Type\ [\ Exp\ ]{:}\ D \ : Arr : Type\ [\ Exp\ ]{:}\ D}$$

$$\textbf{[Skip]}\quad \frac{}{\Gamma \vdash \textsf{skip} \hookrightarrow \textsf{skip} \ : \ \textsf{skip}}$$

$$\textbf{[Assign}_{low}\textbf{]}\quad \frac{\Gamma(l) = low \qquad \Gamma \vdash Exp : low}{\Gamma \vdash l := Exp \hookrightarrow l := Exp \ : \ l := Exp}$$

$$\textbf{[Assign}_{high}\textbf{]}\quad \frac{\Gamma(h) = high}{\Gamma \vdash h := Exp \hookrightarrow h := Exp \ : \ \textsf{skip}}$$

$$\textbf{[ArrAssign}_{low}\textbf{]}\quad \frac{\Gamma(Arr) = low \quad \Gamma \vdash Exp_1 : low \quad \Gamma \vdash Exp_2 : low}{\Gamma \vdash Arr\ [\ Exp_1\ ]{:=}\ Exp_2 \ \hookrightarrow \ Arr\ [\ Exp_1\ ]{:=}\ Exp_2 \ : \ Arr\ [\ Exp_1\ ]{:=}\ Exp_2}$$

$$\textbf{[ArrAssign}_{high}\textbf{]}\quad \frac{\Gamma(Arr) = high}{\Gamma \vdash Arr\ [\ Exp_1\ ]{:=}\ Exp_2 \ \hookrightarrow \ Arr\ [\ Exp_1\ ]{:=}\ Exp_2 \ : \ \textsf{skip}}$$

$$\textbf{[While}_{low}\textbf{]}\quad \frac{\Gamma \vdash B : low \qquad \Gamma \vdash C \hookrightarrow C' \ : \ Sl}{\Gamma \vdash \textsf{while}\ B\ \textsf{do}\ C \ \hookrightarrow \textsf{while}\ B\ \textsf{do}\ C' \ : \ \textsf{while}\ B\ \textsf{do}\ Sl}$$

$$\textbf{[Seq]}\quad \frac{\Gamma \vdash C_1 \hookrightarrow C_1' \ : \ Sl_1 \qquad \Gamma \vdash C_2 \hookrightarrow C_2' \ : \ Sl_2}{\Gamma \ \vdash \ C_1;C_2 \hookrightarrow C_1';C_2' \ : \ Sl_1;Sl_2}$$

$$\textbf{[Fork]}\quad \frac{\Gamma \vdash C_1 \hookrightarrow C_1' \ : Sl_1 \qquad \Gamma \vdash \vec{C_2} \hookrightarrow \vec{C_2}' \ : \ \vec{Sl_2}}{\Gamma \vdash \textsf{fork}(C_1\vec{C_2}) \hookrightarrow \textsf{fork}(C_1'\vec{C_2}') \ : \ \textsf{fork}(Sl_1\vec{Sl_2})}$$

$$\textbf{[Par]}\quad \frac{\Gamma \vdash C_0 \hookrightarrow C_0' \ : Sl_0 \quad \ldots \quad \Gamma \vdash C_{n-1} \hookrightarrow C_{n-1}' \ : \ Sl_{n-1}}{\Gamma \vdash (C_0, \ldots, C_{n-1}) \hookrightarrow (C_0', \ldots, C_{n-1}') \ : \ (Sl_0, \ldots, Sl_{n-1})}$$

$$\textbf{[If}_{low}\textbf{]}\quad \frac{\Gamma \vdash B : low \qquad \Gamma \vdash C_1 \hookrightarrow C_1' \ : Sl_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash \textsf{if}\ B\ \textsf{then}\ C_1\ \textsf{else}\ C_2 \ \hookrightarrow \textsf{if}\ B\ \textsf{then}\ C_1'\ \textsf{else}\ C_2' \ : \ \textsf{if}\ B\ \textsf{then}\ Sl_1\ \textsf{else}\ Sl_2}$$

$$\textbf{[If}_{high}\textbf{]}\quad \frac{\Gamma \vdash B : high \quad \Gamma \vdash C_1 \hookrightarrow C_1' \ : Sl_1 \quad \Gamma \vdash C_2 \hookrightarrow C_2' \ : \ Sl_2 \quad al(Sl_1) = al(Sl_2) = False}{\Gamma \vdash \textsf{if}\ B\ \textsf{then}\ C_1\ \textsf{else}\ C_2 \ \hookrightarrow \textsf{if}\ B\ \textsf{then}\ C_1';Sl_2\ \textsf{else}\ Sl_1;C_2' \ : \textsf{skip};Sl_1;Sl_2}$$

Abbildung 39: Transformierende Sicherheitstypregeln für MWL-Anweisungen unter der binären Flusspolitik. Die Typregeln für Ausdrücke sind in der Abbildung 36 definiert. Γ ist eine Domänenzuweisung.

C. Zusätzliche Beweise

In diesem Abschnitt stehen die Beweise zu Satz 5 und Satz 8 für MWL mit allen Erweiterungen (Arrays, Deklarationen, Synchronisationsanweisungen und Deklassifikationsanweisungen) und für das Sicherheitstypsystem mit allen Sicherheitstypregeln aus Abschnitt 2.4.

Der Beweis des Satzes 5 entspricht zum größten Teil dem Beweis dieses Satzes aus [MS04] für MWL ohne Arrays, Deklarations- und Synchronisationsanweisungen. Außerdem ähnelt er dem Beweis des Satzes 17 im Abschnitt 4, der ebenfalls auf dem Beweis aus [MS04] basiert.

Zunächst folgen in den Beweisen verwendete Definition und Aussagen.

Definition und Aussagen Für die Beweise wird folgende Relation benötig:

Definition 27 (*D-bisimulation up to* $\approx_D^{in}$). Seien eine MLS-Politik $(\mathcal{D}, \leq, \rightsquigarrow)$ und eine Domänenzuweisung Γ gegeben. Sei $\mathcal{C}$ eine Menge von MWL-Anweisungen. Sei $D \in \mathcal{D}$.

Eine symmetrische Relation R ist eine *D-bisimulation up to* $\approx_D^{in}$, wenn

$$
\begin{array}{l}
\forall C, C', s, s', C_1, \ldots, C_n, t : C R C' \wedge s =_D s' \Rightarrow \\
\left[\begin{array}{c} \langle\!| C, s |\!\rangle \rightarrowtriangle_o \langle\!| C_1, \ldots, C_n, t |\!\rangle \\ \Rightarrow \exists C_1', \ldots, C_n', t' : \langle\!| C', s' |\!\rangle \rightarrowtriangle_o \langle\!| C_1', \ldots, C_n', t' |\!\rangle \\ \wedge \forall i \in \{1, \ldots, n\} : C_i (R \cup \approx_D^{in})^+ C_i' \wedge t =_D t' \end{array}\right] \wedge \\
\left[\begin{array}{c} \langle\!| C, s |\!\rangle \overset{\alpha}{\rightarrowtriangle} \langle\!| C_1, \ldots, C_n, t |\!\rangle \\ \Rightarrow \exists C_1', \ldots, C_n', t' : \langle\!| C', s' |\!\rangle \overset{\alpha}{\rightarrowtriangle} \langle\!| C_1', \ldots, C_n', t' |\!\rangle \\ \wedge \forall i \in \{1, \ldots, n\} : C_i (R \cup \approx_D^{in})^+ C_i' \wedge t =_D t' \end{array}\right] \wedge \\
\left[\begin{array}{c} \langle\!| C, s |\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!| C_1, \ldots, C_n, t |\!\rangle \\ \Rightarrow \exists C_1', \ldots, C_n', t' : \langle\!| C', s' |\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!| C_1', \ldots, C_n', t' |\!\rangle \\ \wedge \forall i \in \{1, \ldots, n\} : C_i (R \cup \approx_D^{in})^+ C_i' \wedge \\ ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t' \end{array}\right]
\end{array}
$$

Dabei ist $(\cdot)^+$ der transitive Abschluss, $\alpha \in \{\otimes Sem, \odot Sem\}$ und Sem ein Semaphorbezeichner.

Nützlich ist diese Bisimulation mit folgender Eigenschaft:

Proposition 22. *Sei eine starke D-Bisimulation[in]* $\approx_D^{in}$ *gegeben. Wenn* R *eine D-bisimulation up to* $\approx_D^{in}$ *ist, dann gilt* $R \subset \approx_D^{in}$.

Beweis: Seien eine starke D-Bisimulation[in] $\approx_D^{in}$ und eine *D-bisimulation up to* $\approx_D^{in}$ R gegeben. Es sei

$$
\begin{array}{l}
S = \{(\,(C_1, \ldots, C_k)\,, (C_1', \ldots, C_k')\,) | \\
\qquad\qquad k \in \mathbb{N} \wedge \forall i \in \{1, \ldots, k\} : C_i, C_i' \in \mathcal{C} \wedge C_i (R \cup \approx_D^{in})^+ C_i'\}
\end{array}
$$

Das ist genau die Relation, in der die Anweisungsvektoren in der Implikation der Definition 27 stehen.

Durch die Definition von S gilt $R \subseteq S$. Es ist noch zu zeigen $S \subseteq \approx_D^{in}$. Damit folgt $R \subseteq \approx_D^{in}$.

Seien $\vec{V}, \vec{V'} \in \vec{\mathcal{C}}$ mit $\vec{V} S \vec{V'}$ gegeben. Nach Definition ist S symmetrisch und $\vec{V} = (C_1, \ldots, C_k), \vec{V'} = (C'_1, \ldots, C'_k)$ sind gleichlang. Seien $i \in \{1, \ldots, k\}, s, s', t, \vec{W} = (C^*_1, \ldots, C^*_n)$ mit $s =_D s'$ und $\langle\!| C_i, s |\!\rangle \twoheadrightarrow \langle\!| \vec{W}, t |\!\rangle$ gegeben. Es lassen sich drei Fälle nach der Art der Transition unterscheiden: $\twoheadrightarrow_o$, $\twoheadrightarrow_d^{D_1 \to D_2}$ oder $\xrightarrow{\alpha}$ mit $\alpha \in \{\otimes Sem, \odot Sem\}$. Hier wird der Fall $\twoheadrightarrow_d^{D_1 \to D_2}$ gezeigt, die Argumentation für die anderen beiden Fälle folgt analog.

Es ist zu zeigen:

$$\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} S \vec{W'} \\ \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')$$

Die Aussage lässt sich durch Induktion über den $(R \cup \approx_D^{in})^+$-Abstand m von C_i und C'_i zeigen, wobei m die kleinste Länge einer Folge $(E_0, \ldots, E_m)$ mit $E_0 = C_i, E_m = C'_i$ und $\forall j \in \{0, \ldots, m-1\} : E_j (R \cup \approx_D^{in}) E_{j+1}$ ist. Diese existiert, da $C_i (R \cup \approx_D^{in})^+ C'_i$.

$m = 1, C_i R C'_i$: Die Aussage folgt direkt aus der Definition 27.

$m = 1, C_i \approx_D^{in} C'_i$: Nach Definition von $\approx_D^{in}$ gilt $\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W} \approx_D^{in} \vec{W'} \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')$. Da $\approx_D^{in} \subseteq S$, gilt auch $\vec{W} S \vec{W'}$.

$\exists t'', \vec{W''} : (\langle\!| E_{m-1}, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W''}, t'' |\!\rangle \wedge \vec{W} S \vec{W''} \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t'')), E_{m-1} R C'_i$: Da $s' =_D s'$ und $s' =_{D_1} s'$ gilt nach Definition 27: $\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W''} S \vec{W'} \wedge t'' =_D t'$. Da S und $=_D$ transitiv sind, gilt $\vec{W} S \vec{W'}$ und $((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')$.

$\exists t'', \vec{W''} : (\langle\!| E_{m-1}, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W''}, t'' |\!\rangle \wedge \vec{W} S \vec{W''} \wedge ((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t'')), E_{m-1} \approx_D^{in} C'_i$: Da $s' =_D s'$ und $s' =_{D_1} s'$ gilt nach Definition von $\approx_D^{in}$: $\exists \vec{W'}, t' : \langle\!| C'_i, s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| \vec{W'}, t' |\!\rangle \wedge \vec{W''} \approx_D^{in} \vec{W'} \wedge t'' =_D t'$. Da $\approx_D^{in} \subseteq S$, gilt auch $\vec{W''} S \vec{W'}$. Da S und $=_D$ transitiv sind, gilt $\vec{W} S \vec{W'}$ und $((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t')$.

□

Lemma 23. *Sei eine starke D-Bisimulation[in] $\approx_D^{in}$ und MWL-Programme bzw. Programmvektoren so gegeben, dass $C_1 \approx_D^{in} C'_1$, $C_2 \approx_D^{in} C'_2$ und $\vec{C} \approx_D^{in} \vec{C'}$. Dann gilt auch*

1. $C_1 ; C_2 \approx_D^{in} C'_1 ; C'_2$

2. $\mathsf{fork}(C_1\vec{C}) \approx^{in}_D \mathsf{fork}(C_1'\vec{C'})$
3. $[\forall s =_D s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle] \Rightarrow$
   a) $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approx^{in}_D \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$
   b) $\mathsf{while}\ B\ \mathsf{do}\ C_1 \approx^{in}_D \mathsf{while}\ B\ \mathsf{do}\ C_1'$

Beweis: Es lässt sich jeweils eine Relation R mit den genannten Bedingungen definieren und zeigen, dass diese eine *D-bisimulation up to* $\approx^{in}_D$ ist. Mit Proposition 22 folgt dann $R \subseteq \approx^{in}_D$.

Für 1.: Sei $R = \{(C_1;C_2, C_1';C_2') | C_1 \approx^{in}_D C_1' \wedge C_2 \approx^{in}_D C_2'\}$. Es gelte $C_1;C_2 R C_1';C_2'$, $s =_D s'$ und $\langle\!| C_1;C_2, s |\!\rangle \twoheadrightarrow \langle\!| \vec{V}, t |\!\rangle$.

Nach der Semantik gibt es vier Fälle. Es ist jeweils zu zeigen, dass es von $\langle\!| C_1';C_2', s' |\!\rangle$ eine Transition so gibt, dass die Bedingung aus Definition 27 erfüllt ist.

Fall 1 ($\langle\!| C_1, s |\!\rangle \twoheadrightarrow_o \langle\!| (), t |\!\rangle$): Hier gilt $\vec{V} = C_2$.
Da $C_1 \approx^{in}_D C_1'$ gibt es ein t' mit $\langle\!| C_1', s' |\!\rangle \twoheadrightarrow_o \langle\!| (), t' |\!\rangle$ und $t =_D t'$. Nach der Semantik gilt $\langle\!| C_1';C_2', s' |\!\rangle \twoheadrightarrow_o \langle\!| C_2', t' |\!\rangle$. Da $C_2 \approx^{in}_D C_2'$, erfüllen C_2' und t' die zu zeigende Bedingung.

Fall 2 ($\langle\!| C_1, s |\!\rangle \twoheadrightarrow_o \langle\!| C_3\vec{D}, t |\!\rangle$): Hier gilt $\vec{V} = (C_3;C_2)\vec{D}$.
Da $C_1 \approx^{in}_D C_1'$ gibt es $t', C_3', \vec{D'}$ mit $\langle\!| C_1', s' |\!\rangle \twoheadrightarrow_o \langle\!| C_3'\vec{D'}, t' |\!\rangle$, $C_3\vec{D} \approx^{in}_D C_3'\vec{D'}$ und $t =_D t'$. Nach der Semantik gilt $\langle\!| C_1';C_2', s' |\!\rangle \twoheadrightarrow_o \langle\!| (C_3';C_2')\vec{D'}, t' |\!\rangle$. Da $C_3 \approx^{in}_D C_3'$ und $C_2 \approx^{in}_D C_2'$, gilt $C_3;C_2 R C_3';C_2'$. Da auch $\vec{D} \approx^{in}_D \vec{D'}$, erfüllen $(C_3';C_2)\vec{D}$ und t' die zu zeigende Bedingung.

Fall 3 ($\langle\!| C_1, s |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| (), t |\!\rangle$): Hier gilt $\vec{V} = C_2$.
Da $C_1 \approx^{in}_D C_1'$ gibt es ein t' mit $\langle\!| C_1', s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| (), t' |\!\rangle$ und $((D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D \vee s =_{D_1} s') \Rightarrow t =_D t'')$. Nach der Semantik gilt $\langle\!| C_1';C_2', s' |\!\rangle \twoheadrightarrow_d^{D_1 \to D_2} \langle\!| C_2', t' |\!\rangle$. Da $C_2 \approx^{in}_D C_2'$, erfüllen C_2' und t' die zu zeigende Bedingung.

Fall 4 ($\langle\!| C_1, s |\!\rangle \overset{\alpha}{\twoheadrightarrow} \langle\!| (), t |\!\rangle$ für $\alpha \in \{\otimes Sem, \odot Sem\}$): Analog zu Fall 1.

Für 2.: Sei $R = \{(\mathsf{fork}(C_1\vec{C}), \mathsf{fork}(C_1'\vec{C'})) | C_1 \approx^{in}_D C_1' \wedge \vec{C} \approx^{in}_D \vec{C'}\}$. Es gelte $\mathsf{fork}(C_1\vec{C})\, R\, \mathsf{fork}(C_1'\vec{C'})$ und $s =_D s'$. Nach der Semantik gilt $\langle\!| \mathsf{fork}(C_1\vec{C}), s |\!\rangle \twoheadrightarrow_o \langle\!| C_1\vec{C}, s |\!\rangle$ und $\langle\!| \mathsf{fork}(C_1'\vec{C'}), s' |\!\rangle \twoheadrightarrow_o \langle\!| C_1'\vec{C'}, s' |\!\rangle$. Da $C_1 \approx^{in}_D C_1' \wedge \vec{C} \approx^{in}_D \vec{C'}$, ist R eine *D-bisimulation up to* $\approx^{in}_D$.

Für 3.a): Sei

$$R = \{(\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \mid C_1 \approx^{in}_D C_1' \wedge C_2 \approx^{in}_D C_2' \wedge \forall s =_D s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle\}.$$

Es gelte $(\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) R (\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2')$ und $s =_D s'$. Dann gilt $\langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle$. Sei $\langle\!| B, s |\!\rangle \downarrow true$, dann gilt auch $\langle\!| B, s' |\!\rangle \downarrow true$ (die Argumentation für $\langle\!| B, s |\!\rangle \downarrow false$ läuft analog). Dann gilt nach der Semantik

$$\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \twoheadrightarrow_o \langle\!| C_1, s |\!\rangle$$

und

$$\langle\!|\textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2', s'|\!\rangle \rightarrowtriangle_o \langle\!|C_1', s'|\!\rangle.$$

Da $C_1 \approx_D^{in} C_1'$ ist R eine *D-bisimulation up to* $\approx_D^{in}$

Für 3.b): Sei

$$\begin{aligned} R = \{&(C_1;\textsf{while } B \textsf{ do } C_2, C_1';\textsf{while } B \textsf{ do } C_2')| \\ &C_1 \approx_D^{in} C_1' \wedge C_2 \approx_D^{in} C_2' \wedge \forall s =_D s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle\} \\ \cup \{&(\textsf{while } B \textsf{ do } C, \textsf{while } B \textsf{ do } C')| \; C \approx_D^{in} C' \wedge \forall s =_D s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle.\} \end{aligned}$$

Für Elemente der ersten Menge lässt sich der Beweis analog zu 1. mit der Transition $\rightarrowtriangle_o$ führen, wobei für die zweite Anweisung der Sequenz die Relation R statt $\approx_D^{in}$ gilt. Für Elemente der zweiten Menge lässt sich der Beweis analog zu 3.a) führen. Dabei nutzen wir für den Fall $\langle\!|B, s|\!\rangle \downarrow true$ die erste Menge und für den Fall $\langle\!|B, s|\!\rangle \downarrow false$, dass $() \approx_D^{in} ()$.

□

Folgende Zusammensetzbarkeitseigenschaften sind ebenfalls aus [MS04] entlehnt, wobei für die If-Anweisung dort eine Aussage entsprechend der Aussage für die While-Anweisung gemacht wird.

Satz 24. *Wenn die Programme C_1, C_2 und der Programmvektor $\vec{C}$ stark sicher[in] sind, dann auch:*

1. $C_1;C_2$
2. ***fork***$(C_1\vec{C})$
3. ***while*** B ***do*** C_1, *falls es die Sicherheitsdomäne low gibt und falls gilt* $\forall s =_{low} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$
4. ***if*** B ***then*** C_1 ***else*** C_2, *falls* $\forall D \in \mathcal{D} : (\exists s =_D s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx_D^{in} C_2$

Beweis: 1. und 2. folgen direkt aus Lemma 23. Für 3. verwenden wir, dass

$$\forall D \in \mathcal{D} : s =_D s' \Rightarrow s =_{low} s'.$$

Daher gilt mit $\forall s =_{low} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$ auch

$$\forall D \in \mathcal{D} : \forall s =_D s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle.$$

Das erfüllt für alle $D \in \mathcal{D}$ die Bedingung von Lemma 23.3, also kann man dieses ebenfalls anwenden.

Für 4. zeigen wir, dass

$$\begin{aligned} R = \{&(\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, \textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2) \mid \\ &C_1, C_2 \text{ sind stark sicher[in] und} \\ &\forall D \in \mathcal{D} : (\exists s =_D s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx_D^{in} C_2\} \end{aligned}$$

für alle $D \in \mathcal{D}$ eine *bisimulation up to* $\approx_D^{in}$ ist. Dann folgt mit Lemma 22 $R \subseteq \approx_D^{in}$.

Es gelte (**if** B **then** C_1 **else** C_2)R(**if** B **then** C_1' **else** C_2'), das heißt $C_1' = C_1, C_2' = C_2$. Sei $D \in \mathcal{D}$ beliebig. Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($\forall s =_D s' : \langle\!| B, s |\!\rangle \approx \langle\!| B, s' |\!\rangle$): Damit ist die Bedingung von Lemma 23.3 erfüllt und man kann es mit $C_1 = C_1', C_2 = C_2'$ anwenden, um die Behauptung zu beweisen.

Fall 2 ($\exists s =_D s' : \langle\!| B, s |\!\rangle \not\approx \langle\!| B, s' |\!\rangle$): Dann gilt nach Definition $C_1 \approx_D^{in} C_2$. Da C_1 und C_2 stark sicher[in] sind, gilt auch $C_1 \approx_D^{in} C_1$ und $C_2 \approx_D^{in} C_2$.

Seien $s =_D s'$ beliebig. Nach der Semantik gilt

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s|\!\rangle \rightarrowtriangle_o \langle\!| C, s |\!\rangle$$

und

$$\langle\!|\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s'|\!\rangle \rightarrowtriangle_o \langle\!| C', s' |\!\rangle$$

mit $C \in \{C_1, C_2\}$ und $C' \in \{C_1, C_2\}$. Da $C_1 \approx_D^{in} C_2$, $C_1 \approx_D^{in} C_1$ und $C_2 \approx_D^{in} C_2$, gilt für alle $C, C' \in \{C_1, C_2\}$: $C \approx_D^{in} C'$. Da auch $s =_D s'$, ist R eine D-*bisimulation up to* $\approx_D^{in}$.

□

Beweis für Satz 5 *Beweis:* Satz 5 lässt sich durch eine Induktion über die Struktur von Anweisungen zeigen. Das heißt man zeigt, dass wenn in den Prämissen der Regeln für $\Gamma \vdash C$ „C ist stark sicher[in]“ eingesetzt wird (Induktionsvoraussetzung), jeweils aus den Prämissen die Konklusion folgt. Man zeigt also, dass jede einzelne Regel korrekt ist. Die ersten neun Fälle bilden dabei den Induktionsanfang.

[VarDecl]: Analog zu [Assign] mit einem konstanten Ausdruck.

[SemDecl]: Analog zu [Assign] mit einem konstantem Ausdruck.

[ArrDecl]: Analog zu [Assign] mit einem konstantem Ausdruck, wobei durch die Bedingung $\Gamma \vdash Exp : low$ für die Arraylänge auch der öffentlich sichtbare Ausdruck Arr.**length** in low-gleichen Zuständen gleich bleibt.

[Skip]: Es gilt $\forall s : \langle\!|\textbf{skip}, s|\!\rangle \rightarrowtriangle_o \langle\!| (), s |\!\rangle$. Da $\forall D : () \approx_D^{in} ()$, ist **skip** immer stark sicher[in].

[Assign]: Sei $C = Id{:=}Exp$ eine beliebige Zuweisung und $D \in \mathcal{D}$ mit $\Gamma \vdash Exp : D$ und $D \leq \Gamma(Id)$. Seien $s =_{D'} s'$ für ein beliebiges $D' \in \mathcal{D}$. Es gilt nach der Semantik

$$\langle\!| C, s |\!\rangle \rightarrowtriangle_o \langle\!| (), [Id = n]s |\!\rangle \text{ mit } \langle\!| Exp, s |\!\rangle \downarrow n \text{ und}$$
$$\langle\!| C, s' |\!\rangle \rightarrowtriangle_o \langle\!| (), [Id = n']s' |\!\rangle \text{ mit } \langle\!| Exp, s' |\!\rangle \downarrow n'.$$

Es gilt $() \approx_{D'}^{in} ()$. Es ist noch zu zeigen, dass $[Id = n]s =_{D'} [Id = n']s'$.

Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($D' \geq \Gamma(Id)$): Dann folgt auch $D' \geq D$. Nach Lemma 6 gilt $\langle\!|Exp, s|\!\rangle \approx \langle\!|Exp, s'|\!\rangle$. Damit gilt auch $[Id = n]s(Id) = n = n' = [Id = n']s'(Id)$. Für alle anderen Bezeichner Id' mit $\Gamma(Id') \leq D'$ gilt $[Id = n]s(Id') = s(Id') = s'(Id') = [Id = n']s'(Id')$. Damit gilt $[Id = n]s =_{D'} [Id = n]'s'$.

Fall 2 ($D' \not\geq \Gamma(Id)$): Es gilt $\forall Id' : \Gamma(Id') \leq D' \Rightarrow s(Id') = [Id = n]s(Id') \wedge s'(Id') = [Id = n']s'(Id')$. Daher folgt aus $s =_{D'} s'$ auch $[Id = n]s =_{D'} [Id = n]'s'$.

[ArrAssign]: Analog zu [Assign], nur dass im Fall 1 für Exp_1 und Exp_2 die Gleichheit der Auswertung in den beiden Zuständen gilt.

[Declass]: Sei $C = [Id{:=}Id']$ eine beliebige Deklassifikation mit $D_1 = \Gamma(Id'), D_2 = \Gamma(Id)$. Seien $s =_{D'} s'$ für ein beliebiges $D' \in \mathcal{D}$. Es gilt nach der Semantik

$$\langle\!|C, s|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|(), [Id = n]s|\!\rangle \text{ mit } s(Id') = n \text{ und}$$

$$\langle\!|C, s'|\!\rangle \rightarrowtriangle_d^{D_1 \to D_2} \langle\!|(), [Id = n']s'|\!\rangle \text{ mit } s'(Id') = n'.$$

Es gilt $() \approx_{D'}^{in} ()$. Es ist noch zu zeigen, dass

$$(D_1 \not\rightsquigarrow D_2 \vee D_2 \not\leq D' \vee s =_{D_1} s') \Rightarrow [Id = n]s =_{D'} [Id = n']s'.$$

$D_1 \not\rightsquigarrow D_2$ ist durch die Regelbedingung ausgeschlossen. Aus $D_2 \not\leq D'$ folgt, der Wert von Id ist für $=_{D'}$ irrelevant. Da sich nur dieser ändert, gilt $[Id = n]s =_{D'} [Id = n']s'$. Aus $s =_{D_1} s'$ folgt, $s(Id') = s'(Id')$. Also gilt auch $[Id = n]s(Id) = s(Id') = s'(Id') = [Id = n']s'(Id)$. Da sich nur der Wert von Id ändert, gilt $[Id = n]s =_{D'} [Id = n']s'$.

[Signal]: Sei $C = \mathsf{signal}(Sem)$ eine beliebige Signal-Anweisung. Seien $s =_D s'$ für ein beliebiges $D \in \mathcal{D}$. Es gilt nach der Semantik

$$\langle\!|\mathsf{signal}(Sem), s|\!\rangle \overset{\odot Sem}{\rightarrowtriangle} \langle\!|(), s|\!\rangle \text{ und}$$

$$\langle\!|\mathsf{signal}(Sem), s'|\!\rangle \overset{\odot Sem}{\rightarrowtriangle} \langle\!|(), s'|\!\rangle.$$

Mit $() \approx_D^{in} ()$ und $s =_D s'$ gilt $C \approx_D^{in} C$.

[Wait]: Sei $C = \mathsf{wait}(Sem)$ eine beliebige Wait-Anweisung. Seien $s =_D s'$ für ein beliebiges $D \in \mathcal{D}$. Da nach der Regelbedingung $\Gamma(Sem) = low$, gilt $s(Sem) = 0 \Leftrightarrow s'(Sem) = 0$. Für $s(Sem) = 0$ läuft die Argumentation analog zu der für [Signal]. Für $s(Sem) > 0$ gilt nach der Semantik:

$$\langle\!|\mathsf{wait}(Sem), s|\!\rangle \rightarrowtriangle_o \langle\!|(), [Sem = s(Sem) - 1]s|\!\rangle \text{ und}$$

$$\langle\!|\mathsf{wait}(Sem), s'|\!\rangle \rightarrowtriangle_o \langle\!|(), [Sem = s'(Sem) - 1]s'|\!\rangle.$$

Da $\Gamma(Sem) = low$ gilt $s(Sem) = s'(Sem)$, also $[Sem = s(Sem) - 1]s(Sem) = [Sem = s'(Sem) - 1]s'(Sem)$. Damit und mit $() \approx_D^{in} ()$ gilt $C \approx_D^{in} C$.

[While$_{low}$]: Da nach der Regelbedingung $\Gamma \vdash B : low$, folgt mit Lemma 6, dass $\forall s =_{low} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$. Damit und mit der Induktionsvoraussetzung für $\Gamma \vdash C$ ist die Voraussetzung für Satz 24.3 erfüllt, woraus die Korrektheit folgt.

[Seq]: Mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash C_2$ ist die Voraussetzung für Satz 24 erfüllt, woraus die Korrektheit folgt

[Fork]: Mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash \vec{C_2}$ ist die Voraussetzung für Satz 24 erfüllt, woraus die Korrektheit folgt

[Par]: Korrektheit folgt mit der Induktionsvoraussetzung für $\Gamma \vdash C_0, \ldots, \vdash C_{n-1}$ direkt aus Definition von $\approx^{in}_D$.

[If]: Nach der Regelbedingung gibt es ein $D \in \mathcal{D}$ mit $\Gamma \vdash B : D$ und $\forall D' \not\geq D : C_1 \approx^{in}_{D'} C_2$. Aus $\Gamma \vdash B : D$ folgt nach Lemma 6, dass $\forall s =_D s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$. Daraus folgt $\forall D' \geq D : \forall s =_{D'} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$. Daher gilt für ein beliebiges D' mit $\exists s =_{D'} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle$, dass $D' \not\geq D$. Daher gilt für dieses D': $C_1 \approx^{in}_{D'} C_2$.

Es gilt also $\forall D' : (\exists s =_{D'} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle) \Rightarrow C_1 \approx^{in}_{D'} C_2$. Damit und mit der Induktionsvoraussetzung für $\Gamma \vdash C_1$ und $\Gamma \vdash C_2$ ist die Voraussetzung für Satz 24.4 erfüllt, woraus die Korrektheit folgt

□

Beweis für Satz 8 *Beweis:* Sei $D \in \mathcal{D}$ und seien $C, C' \in \mathcal{C}$ so, dass C, C' stark sicher[in] sind und $C \sim_D C'$. Es werden explizit die Regeln aufgeführt, nach denen $\sim_D$ konstruiert ist und es wird durch Induktion über die kleinste Zahl der Regelanwendung zur Ableitungen von $C \sim_D C'$ gezeigt, dass $\forall D' \not\geq D : C \approx^{in}_{D'} C'$ (vgl. auch mit Beweis des Satzes 18, außer die ersten beiden Fälle sind die Beweise fast identisch.).

Sei $D' \not\geq D$ beliebig. Es lassen sich die Fälle nach der letzten Regel der Ableitung unterscheiden . Die ersten drei Fälle sind der Induktionsanfang.

Unsichtbare Zuweisung $\dfrac{\Gamma(Id) \geq D}{Id{:=}Exp \sim_D \mathsf{skip}}$ **:**

Seien $s =_{D'} s'$ beliebig. Seien $\vec{V} \in \vec{\mathcal{C}}$ und t ein Zustand mit $\langle\!|Id{:=}Exp, s|\!\rangle \rightarrow_o \langle\!|\vec{V}, t|\!\rangle$. Nach der Semantik ist $\vec{V} = ()$ und $t = [Id = n]s$ für ein vorgegebenes n. Es ist zu zeigen, dass $\exists t', \vec{V'} : \langle\!|\mathsf{skip}, s'|\!\rangle \rightarrow_o \langle\!|\vec{V'}, t'|\!\rangle \wedge \vec{V} \approx^{in}_{D'} \vec{V'} \wedge t =_{D'} t'$.

Da $\Gamma(Id) \geq D$ und $D' \not\geq D$, folgt $D' \not\geq \Gamma(Id)$. Also gilt $s =_{D'} t$, woraus mit $s =_{D'} s'$ auch $t =_{D'} s'$ folgt.

Nach der Semantik gilt $\langle\!|\mathsf{skip}, s'|\!\rangle \rightarrow_o \langle\!|(), s'|\!\rangle$. Da $() \approx^{in}_D ()$ und $t =_{D'} s'$, gilt $Id{:=}Exp \approx^{in}_D \mathsf{skip}$.

$$\frac{\Gamma(Arr) \geq D}{Arr[Exp_1]{:=}Exp_2 \sim_D \textsf{skip}}$$

Unsichtbare Arrayzuweisung :
Analog zur Zuweisung, da es irrelevant ist, wie sich der Wert von Arr bzw. Id verändert.

$$\frac{C = C'}{C \sim_D C'}$$

Reflexivität :
Da C stark sicher[in] ist, gilt nach Definition der starken Sicherheit[in], dass $C \approx^{in}_{D'} C$.

$$\frac{C' \sim_D C}{C \sim_D C'}$$

Symmetrie :
$C \approx^{in}_{D'} C'$ folgt mit der Induktionsvoraussetzung für $C' \sim_D C$ direkt aus der Symmetrie von $\approx^{in}_{D'}$.

$$\frac{C \sim_D C'' \quad C'' \sim_D C'}{C \sim_D C'}$$

Transitivität :
$C \approx^{in}_{D'} C'$ folgt mit der Induktionsvoraussetzung für $C \sim_D C''$ und $C'' \sim_D C'$ direkt aus der Transitivität von $\approx^{in}_{D'}$.

$$\frac{C_1 \sim_D C_1' \quad C_2 \sim_D C_2'}{C_1;C_2 \sim_D C_1';C_2'}$$

Sequentielle Komp. :
Aus der Induktionsvoraussetzung für $C_1 \sim_D C_1'$ und $C_2 \sim_D C_2'$ folgt mit Lemma 23.1, dass $C_1;C_2 \approx^{in}_{D'} C_1';C_2'$.

$$\frac{C_1 \sim_D C_1' \quad \vec{C_2} \sim_D \vec{C_2'}}{\textsf{fork}(C_1\vec{C_2}) \sim_D \textsf{fork}(C_1'\vec{C_2'})}$$

Parallele Komp. $\textsf{fork}(C_1\vec{C_2}) \sim_D \textsf{fork}(C_1'\vec{C_2'})$ (für $\vec{C_2} \sim_D \vec{C_2'}$ gilt $\sim_D$ punktweise)**:**
Aus der Induktionsvoraussetzung für $C_1 \sim_D C_1'$ und $\vec{C_2} \sim_D \vec{C_2'}$ folgt mit Lemma 23.2, dass $\textsf{fork}(C_1\vec{C_2}) \approx^{in}_{D'} \textsf{fork}(C_1'\vec{C_2'})$.

$$\frac{C_1 \sim_D C_1'}{\textsf{while } B \textsf{ do } C_1 \sim_D \textsf{while } B \textsf{ do } C_1'}$$

Schleifenkomp. :
Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($\forall s =_{D'} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$) **:**
Damit ist die Bedingung für Lemma 23.3 erfüllt und mit der Induktionsvoraussetzung für $C_1 \sim_D C_1'$ folgt $\textsf{while } B \textsf{ do } C_1 \approx^{in}_{D'} \textsf{while } B \textsf{ do } C_1'$.

Fall 2 ($\exists s =_{D'} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle$) **:**
Seien s, s' diese Zustände. Sei $\langle\!|B, s|\!\rangle \downarrow \mathit{true}$ (der Fall $\langle\!|B, s|\!\rangle \downarrow \mathit{false}$ ist symmetrisch). Dann gilt $\langle\!|B, s'|\!\rangle \downarrow \mathit{false}$. Nach der Semantik gilt

$$\langle\!|\textsf{while } B \textsf{ do } C_1, s|\!\rangle \rightarrow_o \langle\!|C_1;\textsf{while } B \textsf{ do } C_1, s|\!\rangle \text{ und}$$

$$\langle\!|\textsf{while } B \textsf{ do } C_1, s'|\!\rangle \rightarrow_o \langle\!|(), s'|\!\rangle.$$

Da $C_1;\textsf{while } B \textsf{ do } C_1$ und $()$ eine unterschiedliche Anzahl an Threads haben, sind sie nicht stark D-bisimular[in].
Da aber $C = \textsf{while } B \textsf{ do } C_1$ stark sicher[in] ist, ist das ein Widerspruch.

$$\frac{C_1 \sim_D C_1' \quad C_2 \sim_D C_2'}{\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2 \sim_D \textsf{if } B \textsf{ do } C_1' \textsf{ else } C_2'}$$

Verzweigungskomp. :

Es lassen sich zwei Fälle unterscheiden:

Fall 1 ($\forall s =_{D'} s' : \langle\!|B, s|\!\rangle \approx \langle\!|B, s'|\!\rangle$) :

Analog zu Fall 1 für die Schleifenkomposition.

Fall 2 ($\exists s =_{D'} s' : \langle\!|B, s|\!\rangle \not\approx \langle\!|B, s'|\!\rangle$) :

Seien s, s' diese Zustände. Sei $\langle\!|B, s|\!\rangle \downarrow \mathit{true}$ (der Fall $\langle\!|B, s|\!\rangle \downarrow \mathit{false}$ ist symmetrisch). Dann gilt $\langle\!|B, s'|\!\rangle \downarrow \mathit{false}$. Nach der Semantik gilt

$$\langle\!|\textsf{if } B \textsf{ then } C_1; \textsf{else } C_2, s|\!\rangle \rightarrow_o \langle\!|C_1, s|\!\rangle \text{ und}$$

$$\langle\!|\textsf{if } B \textsf{ then } C_1; \textsf{else } C_2, s'|\!\rangle \rightarrow_o \langle\!|C_2, s'|\!\rangle.$$

Da $C = \textsf{if } B \textsf{ then } C_1; \textsf{else } C_2$ stark sicher[in] ist, gilt $C \cong^{in}_{D} C$. Damit und da $\rightarrow_o$ deterministisch ist, gilt $C_1 \cong^{in}_{D'} C_2$. Analog folgt $C_1' \cong^{in}_{D'} C_2'$. Aus der Induktionsvoraussetzung für $C_1 \sim_D C_1'$ und $C_2 \sim_D C_2'$ und der Transitivität von $\cong^{in}_{D'}$ folgen $C_1 \cong^{in}_{D'} C_1'$, $C_2 \cong^{in}_{D'} C_2'$, $C_1 \cong^{in}_{D'} C_2'$ und $C_1' \cong^{in}_{D'} C_2$.

Seien nun $u =_{D'} u'$ beliebig. Nach der Semantik gilt

$$\langle\!|\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2, u|\!\rangle \rightarrow_o \langle\!|C^*, u|\!\rangle \text{ und}$$

$$\langle\!|\textsf{if } B \textsf{ then } C_1' \textsf{ else } C_2', u'|\!\rangle \rightarrow_o \langle\!|C'^*, u'|\!\rangle$$

mit $C^* \in \{C_1, C_2\}$ und $C'^* \in \{C_1', C_2'\}$. Da $C_1 \cong^{in}_{D'} C_1'$, $C_2 \cong^{in}_{D'} C_2'$, $C_1 \cong^{in}_{D'} C_2'$ und $C_2 \cong^{in}_{D'} C_1'$, gilt für alle $C^* \in \{C_1, C_2\}$ und $C'^* \in \{C_1', C_2'\}$: $C^* \cong^{in}_{D'} C'^*$. Da auch $u =_{D'} u'$, ist die Bedingung für $C \cong^{in}_{D'} C'$ erfüllt.

Der Beweis ist von dem Beweis für die Eignung einer beobachtbare Äquivalenz aus [KM05] inspiriert.

□

D. Benutzung des EIFA

D.1. Inhalt der CD-ROM

Dieser Arbeit liegt eine CD-ROM bei, die neben der Anwendung EIFA auch die Quelltexte des EIFA und diesen Text im PDF-Format enthält.

Das Wurzelverzeichnis der CD enthält folgende Dateien (ohne „/“ am Ende) und Verzeichnisse (mit „/“ am Ende):

README	-	enthält Text, der den Inhalt der CD-ROM beschreibt, entspricht diesem Anhang D
DiplomarbeitAReinhard.pdf	-	der Text der Diplomarbeit im PDF-Format (dieser Text)
EIFA.jar	-	das erstellte Programm (Experimental Information Flow Analyzer, EIFA) als ausführbares Java-Archiv
mwl/	-	enthält Beispielprogramme in der Sprache MWL, inklusive der Beispiele aus dieser Arbeit und der Beispiele aus [Pöp05] (s. Anhang D.3)
EIFA/	-	enthält Quelltexte usw. des EIFA, s.u.
doc/	-	enthält die Systemdokumentation des EIFA im html-Format Übersicht: doc/index.html

Das Verzeichnis EIFA/ enthält folgende Dateien und Verzeichnisse:

EIFA/build.xml	-	*ant*-build-Datei, spezifiziert den automatischen Erstellungsprozesse
EIFA/LICENSE	-	enthält Lizenzbedingungen des EIFA (GNU GPL Version 2)
EIFA/doc/	-	kurze Dokumentationsdateien als Ergänzung zur javadoc-Dokumentation (s. doc/)
EIFA/src/	-	Quelldateien, Unterverzeichnis jlexCup/ enthält Grammatikdateien, der Rest die Java-Quellen nach Paketen geordnet
EIFA/test/	-	Quelldateien für *junit*-Tests
EIFA/tools/	-	vom EIFA genutzte Bibliotheken

D.2. Ausführung des EIFA

Es ist nicht notwendig den EIFA zu installieren, um ihn auszuführen. Die Java-Archiv-Datei EIFA.jar ist direkt ausführbar.

Zur Ausführung wird mindestens die *Java Runtime Environment* in der Version 5.0 benötigt.

Mit folgender Eingabe auf der Kommandozeile im Wurzelverzeichnis der CD-ROM wird der EIFA ausgeführt:

```
java -jar EIFA.jar <PARAM> <MWLFILENAME>
```

Dabei legen die Kommandozeilenparameter <PARAM> fest, mit welchem Sicherheitstypsystem der EIFA die Sicherheit des Programms in der Datei <MWLFILENAME> überprüft.

Mit folgender Eingabe auf der Kommandozeile im Wurzelverzeichnis der CD-ROM wird der Hilfetext auf der Kommandozeile ausgegeben, der die Kommandozeilenparameter erläutert:

```
java -jar EIFA.jar --help
```

Der EIFA erfordert keine Interaktion mit dem Benutzer. Die Ergebnisse und die Fehlermeldungen gibt er auf der Kommandozeile aus.

D.3. Die Beispielprogramme

Hier wird aufgeführt, welche Beispielprogramme im Verzeichnis mwl/ der CD-ROM vorhanden sind und welche Kommandozeilenparameter (s.o. <PARAM>) jeweils zur Überprüfung mit dem EIFA sinnvoll sind.

- Beispiele aus [Pöp05] für den EIFAv1: Diese befinden sich im Unterverzeichnis mwl/eifav1mwl/ . Sie können ohne Parameter oder mit dem Parameter --transform geprüft werden.
- Beispiele aus dieser Arbeit:
  - zu Beispiel 1: mwl/webFormProcessing.smwl
  - zu Beispiel 2 und 6: mwl/voting.mmwl , mwl/false_voting.mmwl , mwl/voting_delrel.mmwl , mwl/thread_voting.smmwl
  - zu Beispiel 3 und 7: mwl/foureyes_sec.mmwl , mwl/foureyes_insec.mmwl
  - zu Beispiel 8: mwl/virfilter.mmwl

  Die Dateiendung zeigt jeweils die sinnvollen Kommandozeilenparameter an:

  - ein zusätzliches „s“ für den Parameter --synchronized
  - ein zusätzliches „m“ für den Parameter --multilevel oder die Parameter --multilevel --declassification
- Im Verzeichnis mwl/thesisexpl/ sind zusätzlich die zur Aufnahme in dieser Arbeit angepassten Versionen von Beispielprogrammen verfügbar (s. Abbildungen 8, 9, 10 und 28). Diese sind als MWL-Dateien weniger gut lesbar, als die im vorherigen Punkt genannten Versionen direkt im Verzeichnis mwl/ . Sie können aber genauso mit dem EIFA überprüft werden. Sie sind vorhanden, damit in dieser Arbeit gemachten Aussagen über die Beispiele exakt nachvollzogen werden können.

D.4. Neucompilierung des EIFA

Zur Neucompilierung wird mindestens die *Java Platform, Standard Edition* in der Version 5.0 benötigt. Außerdem benötigt man das Build-Werkzeug *ant* (getestet mit der Version 1.6.2). Für die automatischen Tests benötigt man *junit* (getestet mit der Version 3.8.1).

Zunächst muss das Verzeichnis EIFA auf einen beschreibbaren Datenträger kopiert werden.

Mit folgender Eingabe auf der Kommandozeile in dem (beschreibbaren) Verzeichnis EIFA wird eine neue jar-Datei in einem neuen Verzeichnis EIFA/dist/ erstellt, die genauso wie EIFA.jar ausgeführt werden kann:

```
ant
```

Mit folgender Eingabe auf der Kommandozeile in dem Verzeichnis EIFA werden die weiteren möglichen ant-Targets angezeigt:

```
ant -p
```

Diese können jeweils mit folgender Eingabe auf der Kommandozeile in dem (beschreibbaren) Verzeichnis EIFA gestartet werden:

```
ant TARGETNAME
```

Die wichtigsten Targets sind „doc“, welches die *javadoc*-Dokumentation neu im Verzeichnis EIFA/doc/javadoc/ erstellt und „junit“, welches die automatischen Tests ausführt.

E. Entwicklungsdokumentation

E.1. Funktionalität des EIFAv1

Hier wird die Funktionalität des EIFAv1 beschrieben, auf die sich die Änderungsanforderungen im Abschnitt 3.2 beziehen.

Der Benutzer startet den EIFAv1 über eine Kommandozeilenschnittstelle (*Command Line Interface*) und übergibt einen Dateinamen als Kommandozeilenargument. Es stehe hier FILENAME für diesen Dateinamen. Falls der Benutzer den EIFAv1 ohne Kommandozeilenargument startet, schreibt der EIFAv1 die Meldung „*Wrong number of arguments*“ und eine Anweisung, wie der EIFAv1 korrekt zu starten ist, auf die Fehlerausgabe der Kommandozeile.

Falls der Inhalt der Datei, die der Benutzer über das Kommandozeilenargument identifiziert, kein grammatikalisch korrektes MWL-Programm ist, schreibt der EIFAv1 „*Syntax error*“ auf die Fehlerausgabe der Kommandozeilenschnittstelle. Aus technischen Gründen wird in [Pöp05] die Syntax wie sie im Abschnitt 2.2 eingeführt ist, abgewandelt und konkretisiert (s. Abbildung 40). Der EIFAv1 akzeptiert MWL-Programme als grammatikalisch korrekt, falls diese der hier vorgestellten Syntax entsprechen.

C	::=	**skip** | Id : $Type$: $SecDom$ | Arr : $Type$ [Exp]: $SecDom$
		| Id := Exp | Arr [Exp]:= Exp
		| **if** B **then** C **end** | **if** B **then** C_1 **else** C_2 **end**
		| **while** B **do** C **end** | C_1 ; C_2 | **fork**(C < C_1 , ... , C_n >)
Exp	::=	$IntConst$ | Id | Exp Op Exp | Arr .**length** | Arr [Exp] | B
B	::=	**true** | **false** | Exp $boolOp$ Exp
op	::=	+ | − | ∗ | mod | div
$boolOp$	::=	&& | || | <= | < | > | >= | == | =!
$Type$	::=	**int** | **bool**
$SecDom$	::=	low | high

Abbildung 40: Grammatik der MWL-Syntax, wie sie im EIFAv1 implementiert ist. Bedingte Verzweigungen und Schleifen werden hier mit **end** abgeschlossen. Die Syntax für Anweisungsvektoren und Ausdrücke ist konkreter als in den Syntaxdefinitionen aus Unterabschnitt 2.2. Bezeichner Id bzw. Arr bestehen aus alphanummerischen Zeichen und beginnen mit einem Buchstaben.

Falls das MWL-Programm grammatikalisch korrekt ist, bestimmt der EIFAv1 eine Domänenzuweisung Γ wie folgt. Diese Domänenzuweisung enthält die Paare aus Bezeichner und Sicherheitsdomäne, die durch die Deklarationen des MWL-Programms folgendermaßen festgelegt sind: wenn das MWL-Programm die Deklaration „Id:$IdType$:$SecType$“ enthält, enthält Γ das Paar $(Id, SecType)$. Dabei ist es egal, ob $IdType$ ein einfacher Datentyp oder ein Datentyp mit Arraylängenspe-

zifikation ist. Falls ein Bezeichner mehrmals in dem MWL-Programm deklariert ist, nimmt der EIFAv1 nur das Paar der letzten Deklaration dieses Bezeichners auf. Vollkommen ignoriert werden Deklarationen, die in den komplexen Anweisungen **if**, **while** und **fork** auftreten.

Falls die Domänenzuweisung Γ bestimmt werden konnte (also das MWL-Programm grammatikalisch korrekt ist), überprüft der EIFAv1, ob das MWL-Programm zusammen mit Γ gemäß dem nichttransformierenden Typsystem, das in Abbildung 38 definiert ist, typisierbar ist. Dabei gelten alle Arrayausdrücke $Arr[Exp]$ bei denen $\Gamma(Ar) = low$ und $\Gamma \not\vdash Exp : low$ als nichttypisierbar.

Falls das Ergebnis dieser Überprüfung positiv ausfällt, gibt der EIFAv1 auf der Standardausgabe der Kommandozeilenschnittstelle „==> *The program FILENAME is secure.*“ aus.

Falls das Ergebnis der Überprüfung mit dem nichtransformierenden Typsystem negativ ausfällt, gibt der EIFAv1 auf der Standardausgabe der Kommandozeilenschnittstelle „==> *The program FILENAME is NOT secure.*“ aus. Anschließend überprüft der EIFAv1, ob das MWL-Programm zusammen mit Γ gemäß dem transformierenden Typsystem, das in Abbildung 39 definiert ist, typisierbar ist.

Falls das Ergebnis dieser Überprüfung positiv ausfällt, gibt der EIFAv1 auf der Standardausgabe der Kommandozeilenschnittstelle „==> *The program FILENAME can be transformed into a secure program.*“ aus und schreibt das transformiere Programm (der Typ, der über die Typregeln bestimmt wurde) in eine Datei mit dem Namen FILENAME_secure.

Falls das Ergebnis der Überprüfung mit dem transformierenden Typsystem negativ ausfällt, gibt der EIFAv1 auf der Standardausgabe der Kommandozeilenschnittstelle „==> *The program FILENAME can NOT be transformed into a secure program.*“ aus.

E.2. Fehlverhalten des EIFAv1

Unter Fehlverhalten wird hier nicht nur Verletzung der Spezifikation verstanden, da es keine explizite Spezifikation des EIFAv1 gibt. Es wird hier Verhalten aufgeführt, das man anders erwarten könnte. Zum Beispiel wird hier fehlerhafte Typisierung aufgeführt. Andererseits wird auch Verhalten aufgeführt, das explizit so in [Pöp05] beschrieben ist. Dort werden zum Beispiel einige Annahmen über die vom EIFAv1 zu prüfenden MWL-Programme gemacht, für die es Gründe gibt, dass der EIFA sie überprüfen sollte. Dies gilt für die Fehler ab dem vierten Punkt.

1. EIFAv1 meldet jede Arraydeklaration als sicher, auch wenn die Länge durch einen nicht *low*-typisierbaren Ausdruck spezifiziert ist und später die Länge des Arrays einem *low*-typisierbaren Variablenbezeichner zugewiesen wird. Dabei verletzt so ein Programm die Sicherheitsbedingung für starke Sicherheit und ist auch nicht typisierbar.

2. Die in Abbildung 40 definierte Syntax für Ausdrücke lässt keine Variablenbezeichner oder Arrayfelder als boolesche Ausdrücke zu, das heißt diese können

zum Beispiel nicht als Schleifenbedingung verwendet werden.

3. Der EIFAv1 liest Programme, welche die Fork-Anweisung enthalten unter Umständen nicht ein. Der EIFAv1 hält die Begrenzer < und > von Anweisungsvektoren nach Zuweisungsanweisungen für Operatoren und meldet das Programm als nicht grammatikalisch korrekt (wie oben beschrieben).

4. EIFAv1 unterscheidet keine Arraybezeichner von Variablenbezeichnern, das heißt zum Beispiel ein als Variable deklarierter Bezeichner kann in einem Arrayausdruck eines Programms verwendet werden, ohne dass EIFAv1 das Programm zurückweist. In realen Programmiersprachen führen solche Ausdrücke häufig zu für den Programmierer unerwarteten Verhalten und damit zu Programmierfehlern.

5. EIFAv1 unterscheidet Ausdrücke nicht nach Datentypen (**int** , **bool**). Dies ist eine Verallgemeinerung des vorherigen Punktes. Datentypfehler können praktisch zu Sicherheitsrelevanten Fehlern führen, insbesondere, wenn die Semantik eines Operators nicht für alle Datentypen definiert ist. Ohne Datentypüberprüfung kann man die Annahme der totalen Auswertung von Ausdrücken nicht aufrecht erhalten. Die Argumentation aus [Pöp05], dass Überprüfung auf Datentypsicherheit Aufgabe eines Compilers ist, ist schlüssig. Aber der EIFA (in beiden Versionen) ist experimentell und wird ohne Compiler benutzt.

6. Wie oben beschrieben, kann ein Bezeichner mehrmals in einem MWL-Programm deklariert werden, ohne dass der EIFAv1 dieses Programm als unsicher zurückweist. Dabei ist ein solches Programm nicht nach 38 typisierbar.

7. Wie ebenfalls oben beschrieben ignoriert der EIFAv1 Deklarationen innerhalb von Bedingungs-, Schleifen- und Threaderzeugungsanweisungen, ohne dass der EIFAv1 den Benutzer informiert.

E.3. Nebenanforderungen

Da sich diese Anforderungen nicht aus der Aufgabenstellung der Diplomarbeit ergeben, steht jeweils auch eine Begründung, warum der EIFA diese Anforderungen erfüllen soll.

1. Dateien, die der EIFA einliest können Kommentare enthalten: alle Zeichenketten beginnend mit // und endend mit dem ersten anschließenden Zeilenumbruch ignoriert der EIFA. Dies erleichtert dem Ersteller eines MWL-Programms für einen Leser verständliche Programme zu schreiben.

2. Bezeichner in MWL-Programmen dürfen auch das Zeichen „_“ enthalten, allerdings nicht als erstes Zeichen. Das erleichtert es ebenfalls dem Ersteller eines MWL-Programms für einen Leser verständliche Programme zu schreiben.

3. Die Syntax der Fork-Anweisung, wie sie der EIFA als grammatikalisch korrekt akzeptiert, weicht ab von der Syntax, wie sie der EIFAv1 als grammatikalisch korrekt akzeptiert. statt **fork**($C < D_1 , \ldots , D_n >$) nach der Syntax von EIFAv1 schreibt man nach der Syntax von EIFA **fork**($C , D_1 , \ldots , D_n$). Damit hat der EIFA nicht mehr das im dritten Punkt der Liste der Fehlverhalten (s. Anhang E.2) erläuterte Verhalten. Zusätzlich sind MWL-Programme besser lesbar, die nach der Syntax des EIFA für Fork-Anweisungen korrekt sind, als MWL-Programme, die nach der Syntax des EIFAv1 korrekt sind. Da es bisher noch kein MWL-Programm gab, das die Fork-Syntax des EIFAv1 benutzt, gibt es keine Kompatibilitätsprobleme.

4. Wie im Anhang E.1 beschrieben, gibt EIFAv1 „Syntax error“ aus, wenn das vom Benutzer spezifizierte MWL-Programm nicht grammatikalisch korrekt ist. Um die Fehlersuche für den Ersteller eines MWL-Programms zu vereinfachen, gibt EIFA zusätzlich die Zeile und die Stelle in der Zeile aus, an der EIFA den Syntaxfehler gefunden hat. Dabei werden beide Zahlen von 0 an gezählt.

5. Wie im Anhang E.1 beschrieben, weißt EIFAv1 mehr Arrayausdrücke als unsicher zurück, als nach den Typsystemen aus den Abbildungen 38 und 39 nötig. EIFA wendet für Arrayausdrücke genau die Typsysteme zur Sicherheitsüberprüfung der MWL-Programme an.

6. Die Syntax für Ausdrücke in MWL-Programmen, die der EIFA überprüft, ist gegenüber der Syntax von MWL-Programmen, wie sie der EIFAv1 überprüft (s. Abbildung 40) verändert (s. Abbildung 17). Dadurch lassen sich in MWL-Programmen auch Variablenbezeichner und Arrayfelder als Bedingungen verwenden. Die Unterscheidung von B und Exp in der Syntax für EIFAv1 verhinderte zwar die Verwendung von z.B. ganzzahligen Konstanten als Schleifenbedingung, stellte damit aber trotzdem keine richtige Datentypsicherheit her. Denn der Ausdruck **true** && 10 wäre trotzdem noch eine gültige Schleifenbedingung. Zur Datentypsicherheit s. den nächsten Punkt.

7. Falls das vom Benutzer spezifizierte MWL-Programm grammatikalisch korrekt aber nicht typsicher (*type safe*) ist, schreibt der EIFA eine Meldung auf die Fehlerausgabe der Kommandozeilenschnittstelle. Diese Meldung enthält die Art der Typsicherheitsverletzung und das verletzende Sprachelement. Anschließend beendet sich der EIFA ohne eine weitere Sicherheitsüberprüfung durchzuführen. Damit hat der EIFA nicht mehr das in den vierten und fünften Punkten der Liste der Fehlverhalten (s. Anhang E.2) erläuterte Verhalten.

   Aus Platzgründen wird hier Typsicherheit nur informell definiert. Bezeichner, die als Variablenbezeichner, Arraybezeichner bzw. Semaphorbezeichner verwendet werden, müssen auch als solcher in dem MWL-Programm deklariert sein. Ausdrücke haben in einem grammatikalisch korrekten Programm einen eindeutigen Datentyp (**bool** oder **int**). Variablenbezeichner und Arrayfelder haben den Datentyp, zu dem sie deklariert wurden. Ganzzahlige Konstanten

und Arraylängen haben den Datentyp **int** , die booleschen Konstanten haben den Datentyp **bool**. Zusammengesetze Ausdrücke mit den Arithmetikoperatoren (+ , − , ∗ , mod , div) haben den Datentyp **int**, die anderen zusammengesetzten Ausdrücke haben den Datentyp **bool**.

Ein grammatikalisch korrektes MWL-Programm ist datentypsicher, wenn

- jede Schleifen- oder Zweigbedingung den Datentyp **bool** hat,
- jede Arraylängenspezifikation und jeder Arrayfeldindex den Datentyp **int** hat,
- bei jeder Zuweisung und Deklassifikation die Datentypen beider Seiten übereinstimmen
- jeder Operand der zusammengesetzten Operationen mit dem booleschen Operatoren (&&, ||) vom Datentyp **bool** ist
- jeder Operand der anderen zusammengesetzten Operationen den Datentyp **int** hat.

F. Quelltext-Beispiele

Zur Veranschaulichung steht hier den Inhalt einiger der Quelltextdateien. Der Quelltext steht unter der GPL Version 2 (General Public License). Der vollständige Quelltext befindet sich auf der dieser Arbeit beiliegenden CD-ROM im Verzeichnis ElFA/src/ (s. Anhang D.1). Die aus den *javadoc*-Kommentaren generierte Dokumentation befindet sich im Verzeichnis doc/ .

F.1. Grammatikdateien für Scanner und Parser (Paket mwlParser)

Datei jlexCup/MWLScanner.lex:

```
package mwlParser;

import java.lang.System;
import java_cup.runtime.Symbol;

%%

%cup
%class MWLScanner
%public


/* line- and column-count for useful error reports
   implementation taken from http://www.jflex.de/manual.html*/
%line
%column

%{
  /* register line and column in generated symbol */
  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}


ALPHA=[A-Za-z]
DIGIT=[0-9]
NONNEWLINE_WHITE_SPACE_CHAR=[\ \t\b\012]
WHITE_SPACE_CHAR=[\n\ \t\b\012]
STRING_TEXT= {ALPHA}({ALPHA}|{DIGIT}|_)*

ANY_STRING=[^]*

/* //-type comment, be careful, always needs to be concluded with newline
   implementation taken from http://www.jflex.de/manual.html */
EOLCOMMENT= "//" ([^\r\n])* (\r|\n|\r\n)

%%

/* symbols for security policy specification */
<YYINITIAL> "{" { return (symbol(sym.LBRACE, yytext())); }
<YYINITIAL> "}" { return (symbol(sym.RBRACE, yytext())); }
<YYINITIAL> "=>" { return (symbol(sym.TFLOW, yytext())); }
<YYINITIAL> "->" { return (symbol(sym.IFLOW, yytext())); }
```

```
<YYINITIAL> ";" { return (symbol(sym.SEMI, yytext())); }
<YYINITIAL> "," { return (symbol(sym.COMMA, yytext())); }
<YYINITIAL> "(" { return (symbol(sym.LPAREN, yytext())); }
<YYINITIAL> ")" { return (symbol(sym.RPAREN, yytext())); }
<YYINITIAL> "[" { return (symbol(sym.LSQBRACK, yytext())); }
<YYINITIAL> "]" { return (symbol(sym.RSQBRACK, yytext())); }
<YYINITIAL> "+" { return (symbol(sym.PLUS, yytext())); }
<YYINITIAL> "-" { return (symbol(sym.MINUS, yytext())); }
<YYINITIAL> "*" { return (symbol(sym.TIMES, yytext())); }
<YYINITIAL> "mod" { return (symbol(sym.MOD, yytext())); }
<YYINITIAL> "div" { return (symbol(sym.DIV, yytext())); }
<YYINITIAL> "==" { return (symbol(sym.EQUAL_EQUAL, yytext())); }
<YYINITIAL> "!=" { return (symbol(sym.NOT_EQUAL, yytext())); }
<YYINITIAL> "<"  { return (symbol(sym.LESS, yytext())); }
<YYINITIAL> "<=" { return (symbol(sym.LESS_EQUAL, yytext())); }
<YYINITIAL> ">"  { return (symbol(sym.GREATER, yytext())); }
<YYINITIAL> ">=" { return (symbol(sym.GREATER_EQUAL, yytext())); }
<YYINITIAL> "&&"  { return (symbol(sym.AND, yytext())); }
<YYINITIAL> "||" { return (symbol(sym.OR, yytext())); }
<YYINITIAL> ":=" { return (symbol(sym.ASSIGN, yytext())); }
<YYINITIAL> ":" { return (symbol(sym.COLON, yytext())); }
<YYINITIAL> "." { return (symbol(sym.PERIOD, yytext())); }
<YYINITIAL> "skip" { return (symbol(sym.SKIP, yytext())); }
<YYINITIAL> "if" { return (symbol(sym.IF, yytext())); }
<YYINITIAL> "then" { return (symbol(sym.THEN, yytext())); }
<YYINITIAL> "else" { return (symbol(sym.ELSE, yytext())); }
<YYINITIAL> "end" { return (symbol(sym.END, yytext())); }
<YYINITIAL> "while" { return (symbol(sym.WHILE, yytext())); }
<YYINITIAL> "do" { return (symbol(sym.DO, yytext())); }
<YYINITIAL> "fork" { return (symbol(sym.FORK, yytext())); }
<YYINITIAL> "true" { return (symbol(sym.TRUE, yytext())); }
<YYINITIAL> "false" { return (symbol(sym.FALSE, yytext())); }
<YYINITIAL> "int" { return (symbol(sym.INT, yytext())); }
<YYINITIAL> "bool" { return (symbol(sym.BOOL, yytext())); }
/* <YYINITIAL> "high" { return (symbol(sym.HIGH, yytext())); }
   <YYINITIAL> "low" { return (symbol(sym.LOW, yytext())); } */
<YYINITIAL> "length" {return (symbol(sym.LENGTH, yytext())); }

/* symbols needed for sMWL (synchronized MWL) */
<YYINITIAL> "wait" { return (symbol(sym.WAIT, yytext())); }
<YYINITIAL> "signal" { return (symbol(sym.SIGNAL, yytext())); }
<YYINITIAL> "sem" { return (symbol(sym.SEM, yytext())); }

<YYINITIAL> {NONNEWLINE_WHITE_SPACE_CHAR}+ { break; }
<YYINITIAL> {EOLCOMMENT} { break; /*ignore*/}

<YYINITIAL> {DIGIT}+ { return (symbol(sym.NUMBER, new Integer(yytext())));}

<YYINITIAL> {STRING_TEXT} {
        return (symbol(sym.IDENTIFIER, yytext()));
}

/* undefined token */
<YYINITIAL> . {
        return (symbol(sym.UNDEFINED, "illegal character"));
}
```

Datei jlexCup/MWLParser.cup:

```
// CUP specification for MWL (no actions)

package mwlParser;

import ast.*;
import policy.*;
```

```
import java.util.*;
import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */


/* declare variable to store the parsed security policy */
action code {:
        public ExtMLSPolicy policy;
:}


/* provide method to access the parsed policy */
parser code {:
        /**
         * This gets the security policy as parsed or <code>null</code> if
         * there is no security policy declaration or no parsing is done until now.
         *
         * @return the security policy or null
         */
        public ExtMLSPolicy getPolicy() {
                return action_obj.policy;
        }


  /** Report a non fatal error (or warning).  This method takes a message
   *  string and an additional object (to be used by specializations
   *  implemented in subclasses).  In this imlementation the line and
   * column are reported.
   *
   * @param message an error message.
   * @param info    an extra object reserved for use by specialized subclasses.
   */
  public void report_error(String message, Object info)
    {
      System.err.print(message);
      if (info instanceof Symbol) {
        StringBuffer msg = new StringBuffer("");
                if (((Symbol)info).left != -1 && ((Symbol) info).right != -1)
                        msg.append (" at line " + ((Symbol)info).left +
                                " column " + ((Symbol) info).right +
                            " of input: ");
                if (((Symbol)info).value != null)
                        msg.append ("\"" + ((Symbol)info).value + "\"");
                else
                        msg.append (" symbol " + info);
                System.err.println(msg);
          }
      else System.err.println("");
    }

:};


/* Terminals (tokens returned by the scanner). */
terminal                LBRACE, RBRACE, TFLOW, IFLOW; // terminals for security
    policy specification
terminal            LPAREN, RPAREN, LSQBRACK, RSQBRACK;
terminal            SEMI, COMMA, ASSIGN, COLON, PERIOD;
terminal            PLUS, MINUS, TIMES, MOD, DIV;
terminal            LESS, LESS_EQUAL, GREATER, GREATER_EQUAL, EQUAL_EQUAL;
terminal            NOT_EQUAL, AND, OR;
terminal            SKIP, IF, THEN, ELSE, END, WHILE, DO, FORK, LENGTH, WAIT,
    SIGNAL;
terminal Boolean    TRUE, FALSE;
terminal String     INT, BOOL, SEM;                     // types of variables
```

```
//terminal String           HIGH, LOW;                  // security types of
    variables
terminal String     IDENTIFIER;
terminal Integer    NUMBER;

// for illegal characters
terminal                        UNDEFINED;

/* Non terminals */
non terminal    Program                 program;
non terminal    ProgramVector   progVector;
non terminal    Command                 command;
non terminal    Decl                    decl, varDecl, arrDecl, semDecl;
non terminal    VarIdentifier    varIdentifier;
non terminal    ArrField        arrField;
non terminal    Type                    type;  // , secDomain;
non terminal    Operator                arithOp, compOp, boolOp;
non terminal    Exp                             expression, atom, arithExpression,
    compExpression, boolExpression;

// non terminals for MLS policy
non terminal    ExtMLSPolicy    secDecl;
non terminal                                    domDecls;
non terminal                                    tFlowDecls, iFlowDecls;
non terminal                                    tFlowDecl, iFlowDecl;
non terminal    Program                 total;

/* Precedences */
precedence left OR;
precedence left AND;
precedence left LESS, LESS_EQUAL, GREATER, GREATER_EQUAL, EQUAL_EQUAL, NOT_EQUAL;
precedence left PLUS, MINUS;
precedence left TIMES;
precedence left MOD, DIV;




/* the total program file consists of an optional security policy declararation
        and the actual MWL-program. The MWL-Program is returned through by the
        standard <code>parse()</code>-call, the policy
        by an additional <code>getPolicy()</code>-call.
        */
total       ::= secDecl:sd program:p {:
                                policy = sd;
                                RESULT = p;
                        :}
                        | program:p {: RESULT =  p; :};



secDecl     ::= LBRACE domDecls:ds SEMI tFlowDecls:ts SEMI iFlowDecls:is RBRACE
                {: RESULT = new ExtMLSPolicy(   (Set<SecDomain>) ds,
                                                (PairSet<SecDomain, SecDomain>) ts,
                                                (PairSet<SecDomain, SecDomain>) is)
                                                    ;
                                :}
                        | LBRACE domDecls:ds SEMI tFlowDecls:ts RBRACE
                {: RESULT = new ExtMLSPolicy(   (Set<SecDomain>) ds,
                                                (PairSet<SecDomain, SecDomain>) ts,
                                                new PairHashSet<SecDomain,SecDomain
                                                    >());
                                :};

domDecls    ::= IDENTIFIER:d COMMA domDecls:ds {:
```

```
                                        ((Set<SecDomain>) ds).add(new
                                            IdentifiedSecDomain(d));
                                        RESULT = (Set<SecDomain>) ds;
                                    :}
                                | IDENTIFIER:d {:
                                        Set<SecDomain> ds = new HashSet<
                                            SecDomain>();
                                        ds.add(new IdentifiedSecDomain(d));
                                        RESULT = ds;
                                :};

tFlowDecls      ::= tFlowDecl:t COMMA tFlowDecls:ts{:
                                        ((PairSet<SecDomain, SecDomain>) ts).add( (
                                            Pair<SecDomain, SecDomain>) t);
                                        RESULT = (PairSet<SecDomain, SecDomain>) ts
                                            ;
                                :}
                                | tFlowDecl:t {:
                                        PairSet<SecDomain, SecDomain> ts = new
                                            PairHashSet<SecDomain, SecDomain>();
                                        ts.add((Pair<SecDomain, SecDomain>) t);
                                        RESULT = ts;
                                :};

iFlowDecls      ::= iFlowDecl:i COMMA iFlowDecls:is{:
                                        ((PairSet<SecDomain, SecDomain>) is).add( (
                                            Pair<SecDomain, SecDomain>) i);
                                        RESULT = (PairSet<SecDomain, SecDomain>) is
                                            ;
                                :}
                                | iFlowDecl:i {:
                                        PairSet<SecDomain, SecDomain> is = new
                                            PairHashSet<SecDomain, SecDomain>();
                                        is.add((Pair<SecDomain, SecDomain>) i);
                                        RESULT = is;
                                :};

tFlowDecl       ::= IDENTIFIER:from TFLOW IDENTIFIER:to {:
                                        RESULT = new Pair<SecDomain, SecDomain>(
                                                new IdentifiedSecDomain(from),
                                                new IdentifiedSecDomain(to)
                                        );
                                :};

iFlowDecl       ::= IDENTIFIER:from IFLOW IDENTIFIER:to {:
                                        RESULT = new Pair<SecDomain, SecDomain>(
                                                new IdentifiedSecDomain(from),
                                                new IdentifiedSecDomain(to)
                                        );
                                :};

/* The program grammar */
program     ::= command:c SEMI program:p
                {: RESULT = new Program(c, p); :}
              | command:c
                {: RESULT = new Program(c); :}
              ;

command     ::= SKIP {: RESULT = new Skip(); :}
              | IDENTIFIER:i ASSIGN expression:e
                {: RESULT = new VarAssign(new VarIdentifier(i), e); :}
              | arrField:af ASSIGN expression:e
                {: RESULT = new ArrAssign(af, e); :}
              | decl:d {: RESULT = d; :}
              | IF expression:b THEN program:p1 ELSE program:p2 END
                {: RESULT = new If(b, p1, p2); :}
```

```
            | IF expression:b THEN program:p END
              {: RESULT = new If(b, p); :}
            | WHILE expression:e DO program:p END
              {: RESULT = new While(e, p); :}
            | FORK LPAREN program:p COMMA progVector:pv RPAREN
              {: RESULT = new Fork(p, pv); :}
        | WAIT LPAREN IDENTIFIER:i RPAREN
                    {: RESULT = new Wait(new SemIdentifier(i)); :}
        | SIGNAL LPAREN IDENTIFIER:i RPAREN
                    {: RESULT = new Signal(new SemIdentifier(i)); :}
        | LSQBRACK IDENTIFIER:dest ASSIGN IDENTIFIER:src RSQBRACK
                    {: RESULT = new IntranDeclass( new VarIdentifier(dest),
                                                   new VarIdentifier(src)); :}
          ;


varIdentifier ::= IDENTIFIER:i {: RESULT = new VarIdentifier(i); :};


expression ::= atom:a {: RESULT = a; :}
             | compExpression:c {: RESULT = c; :}
             | arithExpression:c {: RESULT = c; :}
             | boolExpression:c {: RESULT = c; :}
             | LPAREN expression:e RPAREN {: RESULT = e; :}
             ;

compExpression ::=       expression:e1 compOp:o expression:e2
               {: RESULT = new CompareOperation(o, e1, e2); :}
             ;
arithExpression ::=      expression:e1 arithOp:o expression:e2
               {: RESULT = new ArithOperation(o, e1, e2); :}
             ;
boolExpression ::=       expression:e1 boolOp:o expression:e2
               {: RESULT = new BoolOperation(o, e1, e2); :}
             ;

atom       ::= NUMBER:n {: RESULT = new Int(n.intValue()); :}
             | TRUE {: RESULT = new True(); :}
             | FALSE {: RESULT = new False(); :}
             | IDENTIFIER:i PERIOD LENGTH {: RESULT = new ArrLength(new
                 ArrIdentifier(i)); :}
             | arrField:af {: RESULT = af; :}
             | varIdentifier:i {: RESULT = i; :}
             ;

arrField   ::= IDENTIFIER:i LSQBRACK expression:e RSQBRACK
               {: RESULT = new ArrField(new ArrIdentifier(i), e); :}
             ;

arithOp    ::= PLUS {: RESULT = new Plus(); :}
             | MINUS {: RESULT = new Minus(); :}
             | TIMES {: RESULT = new Times(); :}
             | MOD {: RESULT = new Modulus(); :}
             | DIV {: RESULT = new Div(); :}
             ;

compOp     ::= LESS {: RESULT = new Less(); :}
             | LESS_EQUAL {: RESULT = new LessEqual(); :}
        | GREATER {: RESULT = new Greater(); :}
        | GREATER_EQUAL {: RESULT = new GreaterEqual(); :}
        | EQUAL_EQUAL {: RESULT = new EqualEqual(); :}
             | NOT_EQUAL {: RESULT = new NotEqual(); :}
             ;

boolOp     ::= AND {: RESULT = new And(); :}
             |  OR {: RESULT = new Or(); :}
```

```
            ;

decl        ::= varDecl:v {: RESULT = v; :}
              | arrDecl:a {: RESULT = a; :}
              | semDecl:s {: RESULT = s; :}
              ;

semDecl     ::= IDENTIFIER:i COLON SEM COLON IDENTIFIER:s
                        {: RESULT = new SemDecl(new SemIdentifier(i), new
                            IdentifiedSecDomain(s)); :};

varDecl     ::= IDENTIFIER:i COLON type:t COLON IDENTIFIER:s
                {: RESULT = new VarDecl(new VarIdentifier(i),
                                        t,
                          new IdentifiedSecDomain(s) ); :};

arrDecl     ::= IDENTIFIER:i COLON type:t LSQBRACK NUMBER:
    | RSQBRACK COLON IDENTIFIER:s
                {: RESULT = new ArrDecl(new ArrIdentifier(i), I.intValue(),
                                        t, new IdentifiedSecDomain(s)); :}
              | IDENTIFIER:i COLON type:t LSQBRACK varIdentifier:
                  | RSQBRACK COLON IDENTIFIER:s
                {: RESULT = new ArrDecl(new ArrIdentifier(i), I,
                                        t, new IdentifiedSecDomain(s)); :}
              ;

type        ::= INT {: RESULT = new IntegerType(); :}
              | BOOL {: RESULT = new BooleanType(); :}
              ;


progVector  ::= program:p COMMA progVector:pv
                {: RESULT = new ProgramVector(p, pv); :}
              | program:p
                {: RESULT = new ProgramVector(p); :}
              ;
```

F.2. Paket ast

Wir zeigen als Beispiel die Quelldateien der Klassen ast.VarAssign und ast.Program.

Datei ast/VarAssign.java:

```
package ast;

import visitor.ASTVisitor;
import visitor.VisitorException;

/**
 * Representation of an assignment of an expression to an identifier.
 *
 * @author alex
 * @version 1.0
 */
public class VarAssign extends Assign {

        VarIdentifier var = null;

        /**
         * Constructs a VarAssign object.
         *
         * @param var the assigned to variable
         * @param exp the assigned expression
```

```
         */
        public VarAssign(VarIdentifier var, Exp exp) {
                super(exp);
                this.var = var;
        }

        /* (non-Javadoc)
         * @see ast.MwlElement#clone()
         */
        @Override
        public Object clone() {
                return new VarAssign((VarIdentifier) getVar().clone(), (Exp) getExp
                    ().clone());
        }

        /* (non-Javadoc)
         * @see ast.MwlElement#toString()
         */
        @Override
        public String toString() {
                return getVar().toString() + super.toString();
        }

        /* (non-Javadoc)
         * @see ast.MwlElement#accept(visitor.ASTVisitor)
         */
        public Object accept(ASTVisitor v) throws VisitorException {
                return v.visit(this);
        }

        /* (non-Javadoc)
         * @see ast.Assign#equalFeatures(java.lang.Object)
         */
        @Override
        protected boolean equalFeatures(Object obj) {
                VarAssign var = (VarAssign) obj; //save only if called from equals
                return super.equalFeatures(obj) && getVar().equals(var.getVar());
        }

        /* (non-Javadoc)
         * @see ast.Assign#hashCode()
         */
        @Override
        public int hashCode() {
                int code = 357793298;
                code = 37 * code + super.hashCode();
                return 37* code + getVar().hashCode();
        }

        /**
         * Returns the var value.
         *
         * @return Returns the var.
         */
        public VarIdentifier getVar() {
                return var;
        }

}
```

Datei ast/Program.java:

```
package ast;

import visitor.ASTVisitor;
```

```
import visitor.VisitorException;

/**
 * Representation of an MWL program which is composed of single commands.
 *
 * It is organized as a <code>null</code>-terminated linked list,
 * each element refering to a command
 *
 * @author         Christina P&ouml;pper
 * @author Alexander Reinhard
 * @version        2.0
 */
public class Program implements Cloneable, MwlElement {
    private Command comm;
    private Program prog;


    //-------------------------------------------------------------------------
    // Constructors
    //-------------------------------------------------------------------------
    /**
     * Constructs a Program object.
     *
     * @param   comm    the first command
     * @param   prog    the program after the first commmand
     */
    public Program (Command comm, Program prog) {
        this.comm = comm;
        this.prog = prog;
    }

    /**
     * Constructs a Program object.
     *
     * Constructed Program contains only one command
     *
     * @param   comm    the command
     */
    public Program (Command comm) {
        this.comm = comm;
        this.prog = null;
    }

    private Program () {}

    //-------------------------------------------------------------------------
    // Public Methods
    //-------------------------------------------------------------------------

    /**
     * Gets the command.
     *
     * @return  the command
     */
    public Command getCommand () {
        return comm;
    }

    /**
     * Gets the program.
     *
     * @return  the program
     */
    public Program getProgram () {
        return prog;
    }
```

```
/** Appends a program to this program.
 *
 * This method is needed by the
 * transform visitor in case of a high conditional.
 *
 * @param   pTail   the program to be appended
 */
public void append (Program pTail) {
            if (prog == null)
                prog = pTail;
            else
                prog.append(pTail);
}

/**
 * Clones this instance.
 *
 * @return  the clone, which is a new <code>Program</code>
 * instance
 */
public Object clone () {
    if (prog != null)
        return new Program((Command) comm.clone(),
                           (Program) prog.clone());
    else
        return new Program((Command) comm.clone());
}


/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
public String toString () {
    StringBuffer s = new StringBuffer();
    if (comm != null) s.append(comm.toString()+"\n");
    if (prog != null) s.append(prog.toString());
    return s.toString();
}


/* (non-Javadoc)
 * @see ast.MwlElement#accept(visitor.ASTVisitor)
 */
public Object accept (ASTVisitor v) throws VisitorException {
    return v.visit(this);
}


    /* (non-Javadoc)
     * @see MwlElement#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj) {
            if(obj == this) return true;
            if((obj == null) || (this.getClass() != obj.getClass())) return
                false;
            return equalFeatures(obj);
    }


    /**
     * If <code>obj</code> is of exactly this class
     * this returns <code>true</code> if and only if <code>obj</code>
     * is equal to the called object.
     *
```

```
         * It is only safe to call this, if <code>obj</code> is of this class
         *
         * @param obj object to compare to
         * @return true if <code>obj</code> is equal to this
         */
        protected boolean equalFeatures(Object obj) {
                Program p = (Program) obj; // save if only called from equals
                return ( ( getProgram() == p.getProgram() || getProgram().equals(p.
                    getProgram()))
                                && getCommand().equals(p.getCommand()));
        }


        /* (non-Javadoc)
         * @see java.lang.Object#hashCode()
         */
        @Override
        public int hashCode() {
                int code = 324234132;
                code = 37 * code + getCommand().hashCode();
                if(getProgram() != null) code = 37 * code + getProgram().hashCode()
                    ;
                return code;
        }

}
```

F.3. Paket checker

Wir zeigen als Beispiel die Quelldateien der Klassen checker.Checker, checker.CheckerNoTransform, checker.CheckerMLSNoTransform und checker.TypeDeclarations.

Datei checker/Checker.java:

```
package checker;

import visitor.ASTVisitor;
import visitor.VisitorException;
import java_cup.runtime.Scanner;
import java_cup.runtime.Symbol;
import mwlParser.MWLParser;
import mwlParser.MWLScanner;
import ast.Program;

/**
 * Controller organizing I/O, parsing and checking.
 *
 * This is the base class of a template pattern.
 * In the constructor it does the reading of the MWL program file
 * provided as filename to the constructor and then parses the
 * MWL program.
 * The <code>check()</code> method applies the visitor implementing
 * the security-type system.
 * Abstract methods are called for type-system specific details, which
 * have to be implemented in the sublasses. This covers:
 * <ul>
 *      <li>obtaining type-system specific results from the parser
 *      (<code>useAdditionalResults(MWLParser)</code>)</li>
 *      <li>creating the concrete visitor (<code>createVisitor()</code>)</li>
 *      <li>handling of exceptions thrown by the checking process
```

```
 *      (<code>handleException(VisitorException)</code>)</li>
 *      <li>handling of checking results exceeding success/failure
 *      (<code>handleResults(Object)</code>)</li>
 * </ul>
 *
 * @author         Christina P&ouml;pper
 * @author               Alexander Reinhard
 * @version        2.0
 */
public abstract class Checker {

        /**
         * Handles an exceptions of the checking process.
         *
         * This has to be overriden by subclasses to handle exceptions
         * thrown by the visitor implementing the type system.
         *
         * @param e     the exception raised by the visitor
         */
        protected abstract void handleException(VisitorException e);

        /**
         * Creates te visitor implementing the needed type system.
         *
         * Subclasses determine their concrete
         * visitor by implementing this method.
         *
         * @return the visitor to be used
         */
        protected abstract ASTVisitor createVisitor();

        /**
         * Obtains results of parser exceeding the MWL program itself.
         *
         * <p>
         * This can be overriden by subclasses to obtain additional parser results.
         * For example <code>CheckerMLSNoTransform</code> needs the security policy
         * declaration.
         * </p>
         *
         * <p>This implementation does nothing.</p>
         *
         * @param parser        the MWLParser whose <code>parse</code> already has
             to be
         *                                      called
         * @throws CheckerException a problem with specific parser results
         */
        protected void useAdditionalResults(MWLParser parser) throws
            CheckerException {
                //do nothing
        }


        /**
         * Handles result of the check.
         *
         * Subclasses may implement this to use results of the check
         * exceeding the report of success/failure.
         *
         * @param result the object return by the <code>accept</code>-call
         *                                  to the program
         */
        protected void handleResult(Object result) {
                // do nothing for visitors, that do not provide result
        }
```

```
        //-------------------------------------------------------------------------
        // Constants and Declarations
        //-------------------------------------------------------------------------
        TypeDeclarations decls; // contains the variable declarations

        // key: variable name, value: Decl
        String filename;

        Program root;



        //-------------------------------------------------------------------------
        // Public Methods
        //-------------------------------------------------------------------------
        /**
         * Constructs a Checker object.
         *
         * This calls the parser to parse the MWL program given as its filename.
         *
         * As last call it calls
         * the template method <code>useAdditionalResults</code>
         *
         * @param       file    the filename containing the MWL program
         * @throws      Exception  if the MWL program cannot be parsed
         */
        public Checker(String file) throws Exception {
                this.filename = file;
                Scanner scanner = new MWLScanner(new java.io.FileReader(file));
                MWLParser parser = new MWLParser(scanner);
                Symbol sym = parser.parse();
                root = (Program) sym.value;
                decls = new TypeDeclarations(root);
                useAdditionalResults(parser); //template call
        }

        /**
         * Returns whether the check has succeded or not.
         *
         * The program to check is configured in the constructor.
         *
         * It uses the template-methods <code>createVisitor(),
         * handleResult(Object), handleException(VisitorException)</code>.
         *
         * @throws      CheckerException  if the program is null
         * @return  the result of the check
         */
        public boolean check() throws CheckerException {
                if (this.root == null) {
                        throw new CheckerException("Abstract syntax tree is null");
                }
                ASTVisitor visitor = createVisitor(); //template call
                try {
                        Object vResult = this.root.accept(visitor);
                        handleResult(vResult);  //template-call
                        return true;
                } catch (VisitorException e) {
                        handleException(e); //template-call
                        return false;
                }
        }
}
```

Datei checker/CheckerNoTransform.java:

```
package checker;

import com.martiansoftware.jsap.JSAPResult;

import visitor.NoTransformVisitor;
import visitor.ASTVisitor;
import visitor.SyncNoTransformVisitor;
import visitor.VisitorException;

/**
 * Controller to check security of simple MWL programs without transforming them.
 *
 * This is a concrete checker. It only fills the templates
 * for visitor-creation and exception handling.
 *
 * This checks the security of MWL programs with binary security policy.
 * It checks for direct, indirect and timing information leaks
 * (for the concrete algorithms see the documentation
 * of the referenced visitors).
 * The programmer determines the exact features to check by passing the
 * <code>config</code> parameter to the constructor.
 * This parameter is evaluated in <code>createVisitor</code>.
 *
 * @author Christina P&ouml;pper
 * @author Alexander Reinhard
 * @version 2.0
 * @see checker.Checker
 * @see visitor.NoTransformVisitor
 * @see visitor.SyncNoTransformVisitor
 */
public class CheckerNoTransform extends Checker {

        private JSAPResult config;

        //
           ------------------------------------------------------------------

        // Constructor
        //
           ------------------------------------------------------------------

        /**
         * Constructs a CheckerNoTransform object.
         *
         * @param file          the string identifying the file
         *                                      containing the MWL program
         * @param config        the JSAPResult determining the features to check
         * @throws Exception if the MWL program cannot be parsed
         */
        public CheckerNoTransform(String file, JSAPResult config) throws Exception
            {
                super(file);
                this.config = config;
        }

        //
           ------------------------------------------------------------------

        // Public Methods
        //
           ------------------------------------------------------------------


        /**
         * Filling the template by putting out a message on standard out.
         * @param e     {@inheritDoc}
```

```
         */
        protected void handleException(VisitorException e) {
                System.out.println("Untypeable construct: "
                                + e.getMessage() + "\n"
                                + e.getElement() + "\n");
        }

        /**
         * Filling the template by creating
         * a {@link visitor.NoTransformVisitor} object.
         * The exact type depends on the <code>config</code>-parameter
         * passed to the constructor. By this the programmer determines the
         * features of the security check:
         * the check allows synchronization under some condtions
         * or rejects all synchronization primitives.
         *
         * @return an <code>visitor.NoTransformVisitor</code>
         */
        @Override
        protected ASTVisitor createVisitor() {
                if(config.getBoolean("synchronized")) {
                        return new SyncNoTransformVisitor(decls);
                } else {
                        return new NoTransformVisitor(decls);
                }
        }
}
```

Datei checker/CheckerMLSNoTransform.java:

```
package checker;

import com.martiansoftware.jsap.JSAPResult;

import policy.ProgramSecPolicy;
import mwlParser.MWLParser;
import visitor.*;


/**
 * Controller to check security of MWL programs using a MLS policy.
 *
 * This is a concrete checker. It additionally fills the templates
 * for visitor-creation and using of additional parser results
 * (to obtain the parsed security policy).
 *
 * This checks the security of MWL programs with
 * a multi level security policy.
 * It checks for direct, indirect and timing information leaks
 * (for the concrete algorithms see the documentation
 * of the referenced visitors).
 * The programmer determines the exact features to check by passing the
 * <code>config</code> parameter to the constructor.
 * This parameter is evaluated in <code>createVisitor</code>.
 *
 * @author Alexander Reinhard
 * @version 1.0
 * @see checker.Checker
 * @see visitor.MLSNoTransformVisitor
 * @see visitor.SyncMLSNoTransformVisitor
 * @see visitor.DeclassMLSNoTransformVisitor
 */
public class CheckerMLSNoTransform extends CheckerNoTransform {

        private ProgramSecPolicy policy;
```

```
private JSAPResult config;

/**
 * Constructs a CheckerNoTransform object.
 *
 * @param file          the string identifying the file
 *                                containing the MWL program
 * @param config        the JSAPResult determining the features to check
 * @throws Exception if the MWL program cannot be parsed
 */
public CheckerMLSNoTransform(String file, JSAPResult config) throws
    Exception {
        super(file, config);
        this.config = config;
}

/**
 * Filling the template by creating
 * a {@link visitor.MlsAstVisitor} object.
 * The exact type depends on the <code>config</code>-parameter
 * passed to the constructor. By this the programmer determines the
 * features of the security check:
 * <ul>
 *      <li>
 *              the check allows synchronization under some condtions
 *              or rejects all synchronization primitives.
 *      </li>
 *      <li>
 *              the check allows declassication under some conditions
 *              or rejects all declassification commands
 * </ul>
 *
 * @return an <code>visitor.MlsAstVisitor</code>
 */
@Override
protected MlsAstVisitor createVisitor() {
        MlsAstVisitor visitor = new MLSNoTransformVisitor(
                        getPolicy(),
                        new NonDVisibleEquality(getPolicy()));

        // decorate visitor accoring to configuration
        if (config.getBoolean("synchronized")) {
                visitor = new SyncMLSNoTransformVisitor(visitor);
        }
        if (config.getBoolean("declassification")) {
                visitor = new DeclassMLSNoTransformVisitor(visitor);
        }
        return visitor;
}

/**
 * Filling the template by obtaining the parsed security policy.
 * @param parser {@inheritDoc}
 * @throws CheckerException     if parser offers no security policy
 *                                                          (only <code
    >null</code>)
 */
@Override
protected void useAdditionalResults(MWLParser parser) throws
    CheckerException {
        super.useAdditionalResults(parser);
        if(parser.getPolicy() == null) throw new CheckerException("no␣
            security␣policy␣declared");
        policy = new ProgramSecPolicy(decls, parser.getPolicy());
}
```

```
        /**
         * Gets the policy.
         *
         * @return Returns the policy.
         */
        protected ProgramSecPolicy getPolicy() {
                return policy;
        }


}
```

Datei checker/TypeDeclarations.java:

```
package checker;

import java.util.HashMap;
import java.util.Map;

import ast.*;

/**
 * Stores declarations indexed by the declared identifiers.
 *
 * This is used to determine data types and security types
 * of te identifiers.
 *
 * @author Alexander Reinhard
 * @version 1.0
 * @see ast.Decl
 */
public class TypeDeclarations {

        private Map<Identifier, Decl> typeMap = new HashMap<Identifier, Decl>();


        /**
         * Representing the low security domain of the binary security policy.
         *
         * This is used by the visitors implementing security type systems
         * using a binary security policy,
         * @deprecated          This has been used, as EIFA didn't support MLS
            policies.
         *                                      A replacement might be
         *                                      <code>policy.IdentifiedSecDomain("
            low")</code>, maybe
         *                                      defined as constant somewhere in
            the package
         *                                      <code>policy</code>. Another
            replacement would be an
         *                                      enum located in <code>policy</code
            >.
         */
        @Deprecated
        public static final int LOW = 0;

        /**
         * Representing the high security domain of the binary security policy.
         *
         * This is used by the visitors implementing security type systems
         * using a binary security policy,
         * @deprecated          This has been used, as EIFA didn't support MLS
            policies.
         *                                      A replacement might be
```

```
     *                                          <code>policy.IdentifiedSecDomain("
        high")</code>, maybe
     *                                          defined as constant somewhere in
        the package
     *                                          <code>policy</code>. Another
        replacement would be an
     *                                          enum located in <code>policy</code
        >.
     */


    @Deprecated
    public static final int HIGH = 1;

    /**
     * Constructs a TypeDeclarations object.
     *
     * It gathers all declarations at top level of program <code>p</code>
     * @param p     the program containing the declarations to gather
     * @throws CheckerException     if there are more than one
     *
        declarations of the same type for
     *                                                                  an
        identifier
     */
    public TypeDeclarations(Program p) throws CheckerException {
            Command c;

            while (p != null) {
                    c = p.getCommand();
                    if ((c != null) && (c instanceof Decl)) {
                            // insert this Decl into the Map hMap
                            Decl d = (Decl) c;
                            Identifier id = d.getIdent();
                            if(typeMap.containsKey(id)) //identfier already
                                declared
                                    throw new CheckerException("redeclaration␣
                                        of␣identifier␣<" + d
                                                + ">␣previously␣declared␣<"
                                                    + typeMap.get(id) + ">
                                                ");
                            typeMap.put(id, d);
                    }
                    p = p.getProgram();
            }
    }

    /**
     * Gets the declaration of the identifier <code>id</code>
     *
     * Returns <code>null</code> if there is not declaration for it.
     *
     * @param id     the identifier whose declaration will be returned
     * @return       the declaration of <code>id</code> or <code>null</code>
     */
    public Decl get(Identifier id) {
            return typeMap.get(id);
    }

}
```

F.4. Paket policy

Wir zeigen als Beispiel die Quelldateien der Klassen policy.SecDomain, policy.ExtMLSPolicy und policy.ProgramSecPolicy.

Datei policy/SecDomain.java:

```
package policy;

/**
 * Implementing this interface states, that the implementing class may be used as
     security domain.
 *
 * @author Alexander Reinhard
 */
public interface SecDomain {

}
```

Datei policy/ExtMLSPolicy.java:

```
package policy;

import java.util.*;

/**
 * This implements a "MLS policy with exception".
 *
 * This policy is defined in
 * "Heiko Mantel and David Sands, Controlled Declassification Based on Intransitive
     Noninterference,
 *  Lecture Notes in Computer Science
 *  Volume 3302 / 2004 (doi: 10.1007/b102225), p. 129"
 *
 *  It is defined by a triple of a set and two relations (implemented as
 *  <code>Set&lt;SecDomain&gt;</code>
 *  and <code>PairSet&lt;SecDomain, SecDomain&gt;</code> provided as
 *  parameters of the constructor.
 *
 *  Unlike in the article, the first relation doesn't have to be cycle-free.
 *  But it is required to have an element related to all other elements.
 *
 *  The class only provides protected access to the set and the relations.
 *  It instead provides methods to ask, if security domains are related:
 *  <code>transFlow</code>, <code>instransFlow</code>. Further it provides
 *  a getter for the element, which is related to all others according to
 *  the first relation: <code>getLow()</code>.
 *
 * @author Alexander Reinhard
 * @version 1.0
 */
public class ExtMLSPolicy {

        /**
         * Set of security domains.
         *
         * These are all security domains available.
         */
        protected Set<SecDomain> secDomains;

        /**
         * Transtive flow policy.
         *
```

```
         * These states, between which security domains
         * information may flow.
         * It is transitive and reflexive.
         * The element accessible by {@link #getLow()}
         * is related to all elements of {@link #secDomains}
         */
        protected PairSet<SecDomain, SecDomain> transPolicy;

        /**
         * Intransitve flow policy.
         *
         * These indicates between which security domains
         * information may be declassified.
         */
        protected PairSet<SecDomain, SecDomain> intransPolicy;

        //protected SecDomain high;

        private SecDomain low;

        /**
         * Constructs an ExtMLSPolicy object.
         *
         * @param domains the set of security domains
         * @param policy This is the traditional policy.
         * It needs to be transitive, reflexive and have a low element.
         * @param policy2 This can be any relation
         * @throws WrongFlowPolicy indicates that a flow policy doesn't have all
             required properties
         */
        public ExtMLSPolicy(Set<SecDomain> domains,
                        PairSet<SecDomain, SecDomain> policy,
                        PairSet<SecDomain, SecDomain> policy2) throws
                            WrongFlowPolicy
        {
                transPolicy = policy;
                secDomains = domains;
                intransPolicy = policy2;

                low = findLow();
                //high = findHigh();
                consistencyCheck();
        }

        /**
         * Checks for consistency of <code> transPolicy </code>.
         *
         * This is called by the constructer and checks if
         * <code> transPolicy </code>
         * <ul>
         *      <li> is transitive </li>
         *      <li> is reflexive </li>
         *      <li> relates an element to all other elements of <code>secDomains</
             code></li>
         * </ul>
         * @throws WrongFlowPolicy if check-result is negative
         */
        protected void consistencyCheck() throws WrongFlowPolicy {
                if(low == null)
                        throw new WrongFlowPolicy("the transitive policy has no '
                            low' element," +
                                        "i.e. no element, from which information
                                            flow is allowed to every other element"
                                            );

//              if(high == null)
```

```
//                      throw new WrongFlowPolicy("the transitive policy has no '
    high' element, " +
//                                      "i.e. no element, to which information flow
     is allowed from every other element");

                if(!isComplete())
                        throw new WrongFlowPolicy("the␣flow␣relations␣relate␣
                            elements" +
                                        "␣that␣are␣not␣part␣of␣the␣specified␣set␣of
                                            ␣security␣domains");

                if(!isTransitive())
                        throw new WrongFlowPolicy("the␣'transitive'␣flow␣relation␣
                            '=>'␣is␣not␣transitive");

                if(!isReflexive())
                        throw new WrongFlowPolicy("the␣'transitive'␣flow␣relation␣
                            '=>'␣is␣not␣reflexive");
        }

//      /**
//       *
//       *
//       * @return Returns the high.
//       */
//      public SecDomain getHigh() {
//              return high;
//      }

        /**
         * Gets the low element of the security policy.
         *
         * This element is related to all other elements according to
         * <code>transPolicy</code>. This includes
         * <code>this.transFlow(this.getLow(), sd)</code> to return
         * <code>true</code> for all elements in <code>secDomain</code>
         *
         * @return Returns the low.
         */
        public SecDomain getLow() {
                return low;
        }



        /**
         * Checks if information flow is allowed from security domain <code>from</
            code>
         * to security domain <code>to</code> according to the standard transitive
            policy.
         *
         * @param from  a security domain as source of information flow
         * @param to            a security domain as destination of information
            flow
         * @return the check result
         */
        public boolean transFlow(SecDomain from, SecDomain to){
                return transPolicy.contains(new Pair<SecDomain, SecDomain>(from, to
                    ));
        }

        /**
         * Checks if information flow is allowed from all security domains in <code
            >fromSet</code>
         * to security domain <code>to</code> according to the standard transitive
            policy.
```

```
         *
         * @param fromSet               a set of security domains, where each is a
             source of
         *                                      information flow
         * @param to            a security domain as destination of information
             flow
         * @return the check result
         */
        public boolean transFlow(Set<SecDomain> fromSet, SecDomain to){
                for (SecDomain d: fromSet) {
                        if(!transPolicy.contains(new Pair<SecDomain, SecDomain>(d,
                            to)))
                                return false;
                }
                return true;
        }


        /**
         * Checks if information flow is allowed from security domain <code>from</
             code>
         * to security domain <code>to</code> according to the intransitive policy.
         *
         * @param from  a security domain as source of information flow
         * @param to            a security domain as destination of information
             flow
         * @return the check result
         */
        public boolean intransFlow(SecDomain from, SecDomain to){
                return intransPolicy.contains(new Pair<SecDomain, SecDomain>(from,
                    to));
        }

        /**
         * Determines the low element.
         *
         * It determines an element which is related to all other elements
         * in <code>secDomains</code> according to <code>transPolicy</code>.
         *
         * It return <code>null</code> if no such element exists.
         *
         * @return the determined element or <code>null</code>
         */
        private SecDomain findLow() {
                for (SecDomain d: secDomains) {
                        if(transPolicy.getSeconds(d).equals(secDomains)) return d;
                }
                return null;
        }

//      private SecDomain findHigh() {
//              for (SecDomain d: secDomains) {
//                      if(transPolicy.getFirsts(d).equals(secDomains)) return d;
//              }
//              return null;
//      }


        /**
         * Checks if <code>transPolicy</code> is transitive.
         *
         * @return the check result
         */
        private boolean isTransitive() {
                for (Pair<SecDomain,SecDomain> p1: transPolicy) {
                        for (Pair<SecDomain,SecDomain> p2: transPolicy) {
```

```
                                if( p1.getSecond().equals(p2.getFirst()) //
                                    transitive condition
                                        && !this.transFlow(p1.getFirst(), p2.
                                            getSecond()))
                                        return false;

                        }
                }
                return true;
        }

        /**
         * Checks if  <code>transPolicy</code> is reflexive.
         *
         * @return the check result
         */
        private boolean isReflexive() {
                for (SecDomain d: secDomains)
                        if(!transFlow(d,d)) return false;
                return true;
        }

        /**
         * Checks if <code>secDomains</code> contains all security domains
         * related by <code>transPolicy</code> und <code>intransPolicy</code>.
         *
         * @return the check result
         */
        private boolean isComplete() {
                for (Pair<SecDomain,SecDomain> p: transPolicy) {
                        if (    !secDomains.contains(p.getFirst())
                                        || !secDomains.contains(p.getSecond()))
                                return false;
                }
                for (Pair<SecDomain,SecDomain> p: intransPolicy) {
                        if (    !secDomains.contains(p.getFirst())
                                        || !secDomains.contains(p.getSecond()))
                                return false;
                }
                return true;
        }
}
```

Datei policy/ProgramSecPolicy.java:

```
package policy;

import java.util.Set;

import visitor.SecDomCollect;
import visitor.VisitorException;
import ast.MwlElement;
import ast.Exp;
import ast.Identifier;
import checker.TypeDeclarations;

/**
 * Security policy of a program on the level of actual variables.
 *
 * This integrates the two parts of the actual security policy of a program:
 * The flow relations between security domains and the assignment of security
 * domains to identifiers and expressions.
 * For example it is possible to check directly if transitive information
 * flow from an expression to an identifier is allowed according to the security
 * domain assigment and the flow relation between security domains.
```

```
 *
 * @author Alexander Reinhard
 * @version 1.0
 */
public class ProgramSecPolicy {

        private TypeDeclarations decls;
        private ExtMLSPolicy policy;


        /**
         * Constructs a ProgramSecPolicy object.
         *
         * @param decls the type declarations of the program for which this policy
         *                                      is meant
         * @param policy        the security policy on the level of security
           domains
         */
        public ProgramSecPolicy(TypeDeclarations decls, ExtMLSPolicy policy) {
                this.decls = decls;
                this.policy = policy;
        }

        /**
         * Determines the security typ of <code>id</code>.
         *
         * @param id    the identifier whose security domain is to be determined
         * @return              the security domain assigned to <code>id</code>
         * @throws VisitorException     if the security domain could not be
           determined,
         *                                                                  for example
           , if <code>id</code> is not declared
         */
        @SuppressWarnings("unchecked")
        public SecDomain secDomainOf(Identifier id) throws VisitorException {
                Set<SecDomain> s = (Set<SecDomain>) id.accept(new SecDomCollect(
                    decls, policy.getLow()));
                assert(s.size() == 1);
                return s.iterator().next();
        }

        /**
         * Determines the set of security types in <code>element</code>.
         *
         * @param element the MWL element in which occurence of security domains
         *                                      will be collected
         * @return              the set of security domains occuring in <code>
           element</code>
         * @throws VisitorException     if the security domain could not be
           determined,
         *                                                                  for example
           , if identifiers are not declared
         */
        @SuppressWarnings("unchecked")
        public Set<SecDomain> secDomainsOf(MwlElement element) throws
            VisitorException {
                return (Set<SecDomain>) element.accept(new SecDomCollect(decls,
                    policy.getLow()));
        }

        /**
         * Checks if information flow from <code>exp</code> is allowed to low.
         *
         * Checks if transitive flow is allowed from
         * the security domain of the expression <code>exp</code>
         * to the security domain low.
```

```
     *
     * @param exp   the expression for which to property is checked
     * @return result of check
     * @throws VisitorException if flow could not be determined
     */
    public boolean transToLowVar(Exp exp) throws VisitorException {
            return getPolicy().transFlow(secDomainsOf(exp), getPolicy().getLow
                ());
    }

    /**
     * Checks if information flow is allowed from  <code>id</code> to low.
     *
     * Checks if transitive flow is allowed from
     * the security domain of the identifier <code>id</code>
     * to the security domain low.
     *
     * @param id the identifier for which to property is checked
     * @return result of check
     * @throws VisitorException if flow could not be determined
     */
    public boolean transToLowVar(Identifier id) throws VisitorException {
            return getPolicy().transFlow(secDomainsOf(id), getPolicy().getLow()
                );
    }

    /**
     * Checks if information flow is allowed from <code>exp</code> to <code>id
         </code>
     *
     * Checks if transitive flow is allowed from
     * the security domain of the expression <code>exp</code>
     * to the security domain of the identifier <code>id</code>.
     *
     * @param exp the expression which is the source of the flow
     * @param id    the identifier which is the destination of the flow
     * @return result of check
     * @throws VisitorException if flow could not be determined
     */
    public boolean transFlow(Exp exp, Identifier id) throws VisitorException {
            return getPolicy().transFlow(secDomainsOf(exp), secDomainOf(id));
    }

    /**
     * Checks if information flow is allowed from <code>fromSet</code> to <code
         >id</code>.
     *
     * Checks if transitive flow is allowed from
     * all security domains in <code>fromSet</code>
     * to the security domain of the identifier <code>id</code>.
     *
     * @param fromSet the set of security domains which is the source of the
         flow
     * @param id the identifier which is the destination of the flow
     * @return result of check
     * @throws VisitorException if flow could not be determined
     */
    public boolean transFlow(Set<SecDomain> fromSet, Identifier id) throws
        VisitorException {
            return getPolicy().transFlow(fromSet, secDomainOf(id));
    }

    /**
     * Checks if declassification is allowed from <code>src</code> to <code>
         dest</code>.
     *
```

```
         * Checks if intransitive flow is allowed from
         * the security domain of the identifier <code>src</code>
         * to the security domain of the identifier <code>dest</code>.
         *
         * @param src the identifier which is the source of the declassification
         * @param dest the identifier which is the destination of the
           declassification
         * @return result of check
         * @throws VisitorException
         */
        public boolean intransFlow(Identifier src, Identifier dest) throws
            VisitorException {
                return getPolicy().intransFlow(secDomainOf(src), secDomainOf(dest))
                    ;
        }

        /**
         * Gets the policy value.
         *
         * @return Returns the policy.
         */
        protected ExtMLSPolicy getPolicy() {
                return policy;
        }

}
```

F.5. Paket visitor

Wir zeigen als Beispiel die Quelldateien der Klassen visitor.SecDomCollect, visitor.MlsAstVisitor, visitor.MlsAstVisitorDecorator, visitor.MLSNoTransformVisitor, visitor.DeclassMLSNoTransformVisitor und visitor.SyncMLSNoTransformVisitor.

Datei visitor/SecDomCollect.java:

```
package visitor;

import java.util.HashSet;
import java.util.Set;

import policy.SecDomain;

import ast.*;
import checker.TypeDeclarations;

/**
 * Collects security domains occuring in expressions and identifiers.
 *
 * This visitor visits all kinds of expressions or anything that can have a
 * security type (identifier).
 * Each implemented call returns a <code>Set&lt;SecDomain&gt;</code>.
 * The meaning is, each element of the set occured in the expression.
 * So any security domain that may be assigned to
 * the visited expression according to the security typing rules
 * has to be &ge; than each of the elements of the set.
 *
 * @author Alexander Reinhard
 * @version 1.0
 */
public class SecDomCollect extends DefaultASTVisitor {
```

```
private TypeDeclarations decls;
private SecDomain lowDomain;



//-------------------------------------------------------------------------
// Constructors
//-------------------------------------------------------------------------
/**
 * Constructs a SecDomCollect object.
 *
 * @param       decls   a Map containing the declarations for the
 *                      program to be visited
 * @param low the security domain considered to be "low" (visible to all
     security domains)
 */
public SecDomCollect(TypeDeclarations decls, SecDomain low) {
        this.decls = decls;
        lowDomain = low;
}

/**
 * Returns security domains occuring in <code>ce</code>.
 *
 * @param       ce  an <code>ComposedExpression</code>
 * @return      set of security domains occuring in argument
 * @throws VisitorException if method couldn't determine security domains
 */
@SuppressWarnings("unchecked")
protected Object visitComposedExpression(ComposedExpression ce) throws
    VisitorException {
        Set s = (Set) ce.getExp1().accept(this);
        s.addAll((Set) ce.getExp2().accept(this));
        return s;
}

/**
 * Generates a <code>Set&lt;SecDomain&gt;</code> containing only
 * the "low"-element of the security policy
 *
 * @return "low"-element
 */
private Set<SecDomain> genLowSet() {
        Set<SecDomain> s = new HashSet<SecDomain>();
        s.add(lowDomain);
        return s;
}

/**
 * @see visitor.SecDomCollect#visitComposedExpression(ComposedExpression)
 */
@Override
public Object visit(ArithOperation ao) throws VisitorException {
        return visitComposedExpression(ao);
}

/**
 * Returns security domains occuring in <code>af</code>.
 *
 * @param       af  array field to visit
 * @return      set of security domains occuring in argument
 * @throws VisitorException if method couldn't determine security domains
 */
@SuppressWarnings("unchecked")
@Override
```

```
public Object visit(ArrField af) throws VisitorException {
        Set s = (Set) af.getIndex().accept(this);
        s.addAll((Set) af.getId().accept(this));
        return s;
}

/**
 * Returns security domains low.
 *
 * @param       al  array-length expression to visit
 * @return      low security domain
 * @throws VisitorException if method couldn't determine security domains
 */
@Override
public Object visit(ArrLength al) throws VisitorException {
        return genLowSet();
}



/**
 * @see visitor.SecDomCollect#visitComposedExpression(ComposedExpression)
 */
public Object visit(BoolOperation bo) throws VisitorException {
        return visitComposedExpression(bo);
}

/**
 * @see visitor.SecDomCollect#visitComposedExpression(ComposedExpression)
 */
@Override
public Object visit(CompareOperation co) throws VisitorException {
        return visitComposedExpression(co);
}

/**
 * Returns security domain low.
 *
 * @param       f       "false" constant to visit
 * @return      low security domain
 * @throws VisitorException if method couldn't determine security domains
 */
@Override
public Object visit(False f) throws VisitorException {
        return genLowSet();
}

/**
 * Returns security domain of <code>id</code>.
 *
 * @param       id  identifier to visit
 * @return      set of security domains occuring in argument
 * @throws VisitorException if method couldn't determine security domains
 */
@Override
public Object visit(Identifier id) throws VisitorException {
        Decl decl = decls.get(id);

        // Check whether the identifier has actually been declared
        if (decl == null) {
                throw new VisitorException("identifier not declared", id);
        }

        Set<SecDomain> s = new HashSet<SecDomain>();
        s.add(decl.getSecDom());
        return s;
```

```
        }

        /**
         * Returns security domain low.
         *
         * @param       in      integer constant to visit
         * @return      low security domain
         * @throws VisitorException if method couldn't determine security domains
         */
        @Override
        public Object visit(Int in) throws VisitorException {
                return genLowSet();
        }

        /**
         * Returns security domain low.
         *
         * @param       t       "true" constant to visit
         * @return      low security domain
         * @throws VisitorException if method couldn't determine security domains
         */
        @Override
        public Object visit(True t) throws VisitorException {
                return genLowSet();
        }


}
```

Datei visitor/MlsAstVisitor.java:

```
package visitor;

import policy.ProgramSecPolicy;
import ast.MwIElement;

/**
 * Defines what is required from a visitor that uses a MLS policy and that
 * is able to be decorated.
 *
 * The visitors implementing different features are implemented according
 * to the decorator pattern. But this combination of patterns has some
 * peculiarities. If a visitor does a recursive
 * <code>accept</code>/<code>visit</code>-call. It passes an visitor
 * to the <code>accept</code>-method, normally itsself. Then by polymorphism
 * the visit-method is chosen. But if the visitor is decorated,
 * the expected behaviour is to call <code>visit</code> of the decorating
 * object. So the decorated object has to know the decorating class.
 * This interface defines the methods for this mechanism.
 *
 * @author Alexander Reinhard
 * @version 1.0
 * @see MlsAstVisitorDecorator
 */
public interface MlsAstVisitor extends ASTVisitor {

        /**
         * This method is to be called by subclasses, if at the current position in
              the AST
         * no typing rule matches, i.e. the element is not typable.
         *
         * @param msg message explaining, why no rule matches
         * @param element AST-element, for which no rule matches
         * @throws VisitorException
```

```
         */
        public abstract void handleNoRuleMatch(String msg, MwlElement element)
            throws VisitorException;

        /**
         * Gets the security policy which determines the flow relations
         * between variables.
         *
         * @return the security policy
         */
        public abstract ProgramSecPolicy getPolicy();

        /**
         * If visit-method want to do recursive visit-calls, they need to
         * know the right object (<code>this</code> is not the right one).
         *
         * This method should be called by decorating classes in the constructor,
         * that means it should only be called in <code>MlsAstVisitorDecorator</
           code>
         *
         * @param recCallTarget the target for recursive calls
         * @see #recursiveVisit(MwlElement)
         */
        public abstract void setRecCallTarget(MlsAstVisitor recCallTarget);

        /**
         * Calls <code>element.accept(recCallTarget)</code>,
         * where <code>recCallTarget</code> is set by <code>setRecCallTarget</code
           >.
         *
         * This should be used to do recursive <code>visit</code> calls
         * instead of using <code>element.accept(this)</code>,
         * because it allows decorated visitors to "call" <code>visit</code>
         * of the decorating visitor.
         *
         * @param element the MWL element to call <code>accept</code> on
         * @return the result of the <code>accept</code>-call
         * @throws VisitorException thrown by <code>accept</code>-call
         * @see #setRecCallTarget(MlsAstVisitor)
         */
        public abstract Object recursiveVisit(MwlElement element) throws
            VisitorException;

}
```

Datei visitor/MlsAstVisitorDecorator.java:

```
package visitor;

import policy.ProgramSecPolicy;
import ast.*;

/**
 * This is a decorator class of a decorator pattern for MlsAstVisitor.
 *
 * It provides more flexibility in adding type rules to type systems.
 *
 * For example <br />
 * <code>
 *      <pre>
 *              MlsAstVisitor visitor           = new MLSNoTransformVisitor(decls,
     policy, approxRel);
 *              MlsAstVisitor declassVisitor    = new DeclassMLSNoTransformVisitor(
     visitor);
```

```
 *              MIsAstVisitor syncDeclassVisitor= new SyncMLSNoTransformVisitor(
    declassVisitor);
 *      </pre>
 * </code>
 *
 * where <code> DeclassMLSNoTransformVisitor </code>
 * and <code> SyncMLSNoTransformVisitor </code> are decorators, constructs
 * a visitor implementing the multi-level security type system with
    declassification
 * and syncronization.
 *
 * With only two extensions this is not much gain compared to a design using
    inheritence.
 * But with inheritence one needs a class for each combination of features,
 * where by using the decorator pattern one only needs a class for each feature.
 *
 *
 * @author Alexander Reinhard
 * @version 1.0
 *
 */
public abstract class MIsAstVisitorDecorator implements MIsAstVisitor {

        private MIsAstVisitor visitor;


        /**
         * Constructs a MIsAstVisitorDecorator object.
         *
         * @param visitor to which features should be added
         */
        public MIsAstVisitorDecorator(MIsAstVisitor visitor) {
                this.visitor = visitor;
                // register decorator as target for recursive calls
                setRecCallTarget(this);
        }

        /**
         * Delegates the method call to the same method of
         * the decorated object <code>visitor</code>.
         *
         */
        public void handleNoRuleMatch(String msg, MwIElement element) throws
            VisitorException {
                visitor.handleNoRuleMatch(msg, element);
        }

        /**
         * Delegates the method call to the same method of
         * the decorated object <code>visitor</code>.
         *
         */
        public Object visit(ArithOperation ao) throws VisitorException {
                return visitor.visit(ao);
        }

        /**
         * Delegates the method call to the same method of
         * the decorated object <code>visitor</code>.
         *
         */
        public Object visit(ArrDecl ad) throws VisitorException {
                return visitor.visit(ad);
        }

        /**
```

```
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(ArrField af) throws VisitorException {
            return visitor.visit(af);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(ArrLength al) throws VisitorException {
            return visitor.visit(al);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(VarAssign a) throws VisitorException {
            return visitor.visit(a);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(ArrAssign a) throws VisitorException {
            return visitor.visit(a);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(BoolOperation bo) throws VisitorException {
            return visitor.visit(bo);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(CompareOperation co) throws VisitorException {
            return visitor.visit(co);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
     */
    public Object visit(False f) throws VisitorException {
            return visitor.visit(f);
    }

    /**
     * Delegates the method call to the same method of
     * the decorated object <code>visitor</code>.
     *
```

```
 */
public Object visit(Fork fk) throws VisitorException {
        return visitor.visit(fk);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(Identifier id) throws VisitorException {
        return visitor.visit(id);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(If i) throws VisitorException {
        return visitor.visit(i);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(Int in) throws VisitorException {
        return visitor.visit(in);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(Program p) throws VisitorException {
        return visitor.visit(p);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(ProgramVector pv) throws VisitorException {
        return visitor.visit(pv);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(Skip s) throws VisitorException {
        return visitor.visit(s);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(True t) throws VisitorException {
        return visitor.visit(t);
```

```
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(VarDecl vd) throws VisitorException {
        return visitor.visit(vd);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(While w) throws VisitorException {
        return visitor.visit(w);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object getResult() {
        return visitor.getResult();
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(SemDecl sd) throws VisitorException {
        return visitor.visit(sd);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(SemCommand sc) throws VisitorException {
        return visitor.visit(sc);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object visit(IntranDeclass declass) throws VisitorException {
        return visitor.visit(declass);
}

/**
 * Delegates the method call to the same method of
 * the decorated object <code>visitor</code>.
 *
 */
public Object recursiveVisit(MwlElement element) throws VisitorException {
        return visitor.recursiveVisit(element);
}

/**
```

```
         * Delegates the method call to the same method of
         * the decorated object <code>visitor</code>.
         *
         */
        public void setRecCallTarget(MlsAstVisitor recCallTarget) {
                visitor.setRecCallTarget(recCallTarget);
        }

        /**
         * Delegates the method call to the same method of
         * the decorated object <code>visitor</code>.
         *
         */
        public ProgramSecPolicy getPolicy() {
                return visitor.getPolicy();
        }
}
```

Datei visitor/MLSNoTransformVisitor.java:

```
package visitor;

import java.util.Set;

import policy.ProgramSecPolicy;
import policy.SecDomain;
import ast.*;


/**
 * MLSNoTransformVisitor implements the check logic for security checks
 * of programs without transformation using a MLS policy.
 *
 * It realizes the following
 * rules:
 * <ul>
 *   <li>[Skip]</li>
 *   <li>[Assign]</li>
 *   <li>[If]</li>
 *   <li>[While_low]</li>
 *   <li>[Fork]</li>
 *   <li>[Par]</li>
 *   <li>[Seq]</li>
 * </ul>
 * <p>
 * The [If] rule has a semantic side condition, that has to be
 * approximated be a Safe Approximation Relation. This provided in form of
 * an object implementing the interface <code>SafeApproxRel</code>
 * to the constructor. So this is an instantiation of the strategy patter,
 * where <code>SafeApproxRel</code> is the strategy interface and
 * this class is the context.
 * For the security type system to be sound, the implementation of this
 * interface must have the properties of a Safe Approximation Relation.
 * </p>
 * <p>
 * If an <code>VisitorException</code> is thrown, the checked program
 * is not typable. If the call to <code>visit</code> returns, the program
 * is typable.
 * </p>
 *
 * <p>
 * Each <code>visit</code> method returns <code>null</code> because
 * the return value is of no interest.
 * </p>
 * <p>
```

```
 *      This class can be decorated by derivations of <code>MIsAstVisitorDecorator
    </code>.
 * </p>
 * @author       Alexander Reinhard
 * @version      1.0
 */
public class MLSNoTransformVisitor extends DefaultASTVisitor implements
    MIsAstVisitor {

        private SafeApproxRel safeApprox;
        private ProgramSecPolicy policy;

        private MIsAstVisitor recCallTarget;


        /**
         * Constructs a MLSNoTransformVisitor object.
         *
         * @param       policy the security policy for the program
         * @param       safeApprox the approximation relation to be used by [If]
             rule
         */
        public MLSNoTransformVisitor(ProgramSecPolicy policy,
                        SafeApproxRel safeApprox) {
                this.policy = policy;
                this.safeApprox = safeApprox;

                this.recCallTarget = this;
        }

        /**
         * @deprecated always true
         */
        @Deprecated
        public Object getResult() {
                return new Boolean(true);
        }

        /**
         * Implements the [Skip] rule.
         *
         * @param       s       a <code>Skip</code> command
         * @return      null
         * @throws VisitorException never thrown
         */
        public Object visit(Skip s)  throws VisitorException  {
                return null;
        }

        /**
         * Implements the [Var] rule.
         *
         * @param       vd      a <code>VarDecl</code> command
         * @return      null
         * @throws VisitorException never thrown
         */
        public Object visit(VarDecl vd)  throws VisitorException  {
                return null;
        }

        /**
         * Implements the [ArrDecl] rule.
         *
         * @param   ad      a <code>VarDecl</code> command
         * @return  null
         * @throws VisitorException if <code>ad</code> is not typable
```

```
 */
public Object visit(ArrDecl ad) throws VisitorException {
        Exp exp = ad.getLength();
        if(! getPolicy().transToLowVar(exp)) {
                handleNoRuleMatch("array␣declaration␣with␣non␣low␣
                    expression␣as␣length", ad);
        }
        return null;
}

/**
 * Implements the [VarAssign] rule.
 *
 * @param       as      an <code>VarAssign</code> command
 * @return      null
 * @throws VisitorException if <code>as</code> is not typable
 */
public Object visit(VarAssign as) throws VisitorException {
        if (!getPolicy().transFlow(as.getExp(), as.getVar()))
                handleNoRuleMatch("assignment␣of␣expression␣" +
                                  "containing␣a␣security␣domain␣not␣lower␣and
                                      ␣not␣equal␣to␣" +
                                  "that␣of␣variable", as);

        return null;
}

/**
 * Implements the [ArrAssign] rules .
 *
 * @param       as      an <code>ArrAssign</code> command
 * @return      null
 * @throws VisitorException if <code>as</code> is not typable
 */
public Object visit(ArrAssign as) throws VisitorException {
        ArrField field = as.getField();
        //test if rule applies
        if(             getPolicy().transFlow(field.getIndex(), field.getId
            ())
                &&      getPolicy().transFlow(as.getExp(), field.getId()) )
                     {
                return null;
        } else {
                handleNoRuleMatch("assignment␣to␣a␣low␣array␣" +
                                  "with␣high␣expression␣or␣high␣index", as);
        }
        return null;
}

/**
 * Implements the [If] rule.
 *
 * The check for the sematic side condition is passed
 * as <code>SafeApproxRel</code> to the constructor
 * (strategy of strategy pattern).
 *
 * @param       i       an <code>If</code> command
 * @return      null
 * @throws VisitorException if <code>i</code> is not typable
 */
public Object visit(If i) throws VisitorException {
        Set<SecDomain> sdCond = getPolicy().secDomainsOf(i.getCond());
        recursiveVisit(i.getIfProgram());
        Program pElse = i.getElseProgram();
        if (pElse != null) recursiveVisit(pElse);
```

```
        if (!getSafeApprox().equiv(sdCond, i.getIfProgram(), pElse) ) {
                handleNoRuleMatch("branches␣of␣condition␣too␣different␣" +
                                  "(exactly:␣not␣in␣safe␣approximation␣
                                      relation),␣" +
                                  "possible␣implicit␣information␣flow", i);
        }
        return null;
}

/**
 * Implements the [While_low] rule.
 *
 * @param       wh      a <code>While</code> command
 * @return      null
 * @throws VisitorException if <code>as</code> is not typable
 */
public Object visit(While wh) throws VisitorException {
        if(! getPolicy().transToLowVar(wh.getCond()) ) // [While_low] not
            applicable
                handleNoRuleMatch("while␣loop␣with␣non␣low␣condition:\n",
                    wh);

        recursiveVisit(wh.getProgram());
        return null;
}

/**
 * Implements the [Fork] rule.
 *
 * @param       fk      a <code>Fork</code> command
 * @return      null
 * @throws VisitorException if <code>as</code> is not typable
 */
public Object visit(Fork fk) throws VisitorException { // [Fork]
        Program p = fk.getProgram();
        ProgramVector pv = fk.getProgramVector();
        recursiveVisit(p);
        if (pv != null)
                recursiveVisit(pv);
        return null;
}

/**
 * Implements the [Seq] rule.
 *
 * Splits up the check for a Program into the checks of its Commands.
 *
 * @param       p       a <code>Program</code>
 * @return      null
 * @throws VisitorException if <code>p</code> is not typable
 */
public Object visit(Program p) throws VisitorException {
        Command c = p.getCommand();
        Program nextP = p.getProgram();
        recursiveVisit(c);
        if (nextP != null)
                recursiveVisit(nextP);
        return null;
}

/**
 *  Implementes the [Par] rule,
 *
 * Splits up the check for a ProgramVector into the checksof its
 * Programs.
 *
```

```
         * @param       pv      a <code>ProgramVector</code>
         * @return      null
         * @throws VisitorException if <code>p</code> is not typable
         */
        public Object visit(ProgramVector pv) throws VisitorException {
                Program p = pv.getProgram();
                ProgramVector nextPv = pv.getProgramVector();
                recursiveVisit(p);
                if (nextPv != null)
                        nextPv.accept(this);
                return null;
        }


        /* (non-Javadoc)
         * @see visitor.MlsAstVisitor#recursiveVisit(ast.MwlElement)
         */
        public Object recursiveVisit(MwlElement element) throws VisitorException {
                return element.accept(getRecCallTarget());
        }


        /**
         * Gets the recCallTarget value.
         *
         * @return recCallTarget value
         */
        private MlsAstVisitor getRecCallTarget() {
                return recCallTarget;
        }

        /* (non-Javadoc)
         * @see visitor.MlsAstVisitor#setRecCallTarget(visitor.MlsAstVisitor)
         */
        public void setRecCallTarget(MlsAstVisitor recCallTarget) {
                this.recCallTarget = recCallTarget;
        }

        /**
         * Returns the safe approx value.
         *
         * @return the safe approx value
         */
        protected SafeApproxRel getSafeApprox() {
                return safeApprox;
        }

        /* (non-Javadoc)
         * @see visitor.MlsAstVisitor#getPolicy()
         */
        public ProgramSecPolicy getPolicy() {
                return policy;
        }
}
```

Datei visitor/DeclassMLSNoTransformVisitor.java:

```
package visitor;

import ast.IntranDeclass;

/**
 * This extends the check logic of visitors
 * implementing <code>MlsAstVisitor</code> with declassification.
 *
```

```
 * It is a concrete decorator class. It only handles declassification
 * by itself, all other <code>visit</code> calls are delegated.
 *
 * @author Alexander Reinhard
 * @version 1.0
 * @see MLSNoTransformVisitor
 */
public class DeclassMLSNoTransformVisitor extends MIsAstVisitorDecorator {

        /**
         * Constructs a DeclassMLSNoTransformVisitor object.
         *
         * @param visitor a visitor to which features should be added
         */
        public DeclassMLSNoTransformVisitor(MIsAstVisitor visitor) {
                super(visitor);
        }

        /**
         * Checks if the intransitive flow corresponding to this
         * declassification is allowed.
         *
         */
        @Override
        public Object visit(IntranDeclass declass) throws VisitorException {
                if(! getPolicy().intransFlow(declass.getSrc(), declass.getDest()))
                    {
                        this.handleNoRuleMatch("declassification command, " +
                                        "where no intransitve flow is allowed",
                                            declass);
                }
                return null;
        }

}
```

Datei visitor/SyncMLSNoTransformVisitor.java:

```
package visitor;

import ast.SemCommand;
import ast.SemDecl;

/**
 * This extends the check logic of visitors
 * implementing <code>MIsAstVisitor</code> with synchronization.
 *
 * This is a concrete decorator class, decorating the
 * visitor given to the constructor with the ability
 * to handle synchronization primitives. It implements
 * the typing rules [Wait] and [Signal].
 *
 * @author Alexander Reinhard
 * @version 1.0
 *
 */
public class SyncMLSNoTransformVisitor extends MIsAstVisitorDecorator {

        /**
         * Constructs a SyncMLSNoTransformVisitor object.
         *
         * @param visitor to be decorated
         */
        public SyncMLSNoTransformVisitor(MIsAstVisitor visitor) {
                super(visitor);
```

```
        }

        /**
         * Implementing rules [Signal] and [Wait].
         */
        @Override
        public Object visit(SemCommand sc) throws VisitorException {
                if(! getPolicy().transToLowVar(sc.getSi())) {
                        handleNoRuleMatch("synchronization command " +
                                        "refering to non low semaphore variable",
                                            sc);
                }
                return null;
        }

        /**
         * Accepts the declararation.
         */
        @Override
        public Object visit(SemDecl sd) throws VisitorException {
                return null;
        }



}
```