Translation Validation for Optimizing Compilers

Master's Thesis

submitted by

Franziska Ebert

supervised by

Prof. Bernd Finkbeiner, PhD

and

Dipl.-Inform. Anne Proetzsch, Dipl.-Inform. Lars Kuhtz

reviewed by

Prof. Bernd Finkbeiner, PhD

Prof. Dr.-Ing. Holger Hermanns

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Saarbrücken, August 7, 2007

Eidesstattliche Erklärung:
Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Saarbrücken, den 7. August 2007

Einverständniserklärung:
Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.

Saarbrücken, den 7. August 2007

Abstract

The task of verifying a compiler formally becomes harder and harder due to the size of the compiler, its ongoing evolution and modifications. Whereas generic compiler verification has to be adapted after each compiler modification, translation validation is a novel approach that is fully independent of the compiler. Instead of verifying the compiler itself, the source program and the target program are compared after each translation. If the target program is proven to be a correct translation of the source program, the compiler was proven to be correct for this special input. In order to define a correct translation, refinement mappings are introduced which map states and variables of the target program to states and variables, respectively, of the source program.

While compiling a program, an optimizing compiler might apply several program transformations which affect the structure of the original program. Thus, establishing the refinement mapping is often not obvious. This thesis work shows how it can be proven automatically that the target program is a correct translation of the source program without instrumenting the optimizing compiler. Thereby, the focus of this thesis is set on structure-modifying transformations.

The algorithm for finding a refinement mapping searches for a transformation chain that transforms the source system into the target system. If such a transformation chain can be found, the trivial refinement mapping between the target system and the transformed source system can be established. The transformation chain together with the refinement mapping are a proof that the target program is a correct translation of the source program.

This algorithm is based on an exhaustive search using Breadth First Search (BFS), i.e. all possible transformation chains are computed and it is checked whether the target system refines one of the resulting systems. This algorithm is shown to be correct and terminating for a given (restricted) set of transformations. Furthermore, we show which properties a compiler transformation has to fulfill such that the algorithm can be extended with this transformation without loosing termination or correctness.

To reduce the search space, some optimizations for the algorithm are introduced that remove duplicated subtrees and infinite transformation chains. Also we show how metrics can be used to extend the class of transformations such that the algorithm still is terminating and correct.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 3

2 Translation Validation in Theory **5**
2.1 Transition Systems . . . 5
2.2 Representation of Transition Systems . . . 6
2.3 Refinement Mappings . . . 7
2.4 Invariants and Proof Conditions . . . 9

3 Compiler Transformations **13**
3.1 Preliminaries . . . 13
3.1.1 Loops - Definitions and Notation . . . 13
3.1.2 Dependences between Statements . . . 16
3.2 The considered Compiler Transformations . . . 21
3.2.1 Structure-Preserving Transformations . . . 21
3.2.2 Structure-Modifying Transformations . . . 22
3.3 Parameters for Transformations . . . 30
3.4 Classification of Compiler Transformations . . . 31
3.4.1 Definitions and Notation . . . 31
3.4.2 Enabling Transformations . . . 31
3.4.3 Inverse Transformations . . . 33
3.4.4 Commutative Transformations . . . 33

4 Finding a Refinement Mapping **35**
4.1 Naive Algorithm . . . 35
4.1.1 Correctness . . . 40
4.1.2 Termination . . . 41
4.2 Optimizations . . . 42
4.2.1 Encoding Transformations in Transition Systems . . . 42
4.2.2 Removing duplicated Subtrees . . . 43
4.2.3 Removing Circles . . . 44
4.2.4 Correctness of the Rules . . . 45

4.3 Extending the Algorithm with more Transformations 55
4.3.1 Loop unrolling 56

5 Implementation **59**
5.1 The Components 59
5.1.1 Abstract Transition Systems 59
5.1.2 Dependency Analysis 60
5.1.3 Transformations 62
5.2 Putting it All Together 63
5.2.1 The Input 63
5.2.2 Normalform of Transition Systems 64
5.2.3 Implementation of the Compiler 64
5.2.4 Implementation of the Algorithm 66

6 Experiments **67**
6.1 A First Example 67
6.2 Transformation Tests 67
6.3 Termination 71
6.4 Optimizations 71

7 Conclusion and Future Work **75**
7.1 Conclusion 75
7.2 Future Work 76

A Implementation: Class Hierarchy **77**

Chapter 1

Introduction

1.1 Motivation

The task of verifying a compiler formally becomes harder and harder due to the size of the compiler, its ongoing evolution and modifications. Whereas generic compiler verification has to be adapted after each compiler modification, translation validation is a novel approach that is fully independent of the compiler. Instead of verifying the compiler itself, the input program of the compiler (the source program) and its translated version (the target program) are compared and it is checked whether they fulfill the same properties. If the target program is proven to be a correct translation of the source program, the compilation process was proven to be correct for this special input. The advantages of translation validation over generic compiler verification are that the verification process is independent of the compiler version and it is fully automatic.

Figure 1.1 illustrates the concept of translation validation. To determine

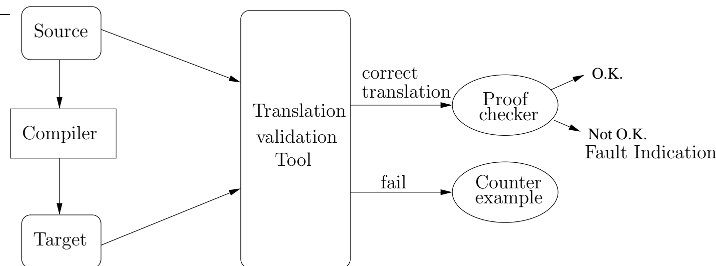


Figure 1.1: The concept of translation validation.

whether the target program is a correct translation of the source program, the translation validation tool has to find a simulation relation (refinement mapping) between the source program and the target program, i.e. a control mapping between the states of the source and the target control flow

graph and a data mapping between variables of source and target program. Afterwards, proof conditions are derived that can be checked by a generic theorem prover. If all proof conditions are proven to be valid the target program is a correct translation of the source program.

While translating the source program the compiler can apply optimization methods, like loop unrolling or loop fusion. It is differentiated between structure preserving and structure modifying optimizations. Structure preserving optimizations do not change the structure of the source program and therefore can be handled easier by the translation validation tool than structure modifying optimizations that reorder the instructions in the source program or insert and delete loops. Hence, establishing a refinement mapping is complicated.

The major problem in finding a refinement mapping for a source program and a target program is that it is not known which compiler optimizations were applied and in which order. In general, the source program and the target program have nothing in common so that it is impossible to establish a control mapping directly. Moreover, it cannot be seen which program part of the source program was transformed into which program part of the target program.

This thesis considers reactive systems as source programs of the optimizing compiler, i.e. systems that interact with an environment. We observe the level of the optimizing compiler where code optimizing transformations are applied, i.e. we do not care about code translation and generation. Thus, the compiler output is also a reactive system and the task is to find a refinement mapping between the input system and the output system if there exists one. To solve this task the most common compiler transformations like for example sequential transition refinement, loop unrolling, loop fusion, and loop distribution, are analyzed and formalized. These considerations will be based on the notion of transition systems. Afterwards, an algorithm is presented that determines whether the output system is a correct translation of the input system, i.e. how a refinement mapping between these two reactive systems, where the described compiler optimizations have been applied arbitrarily, can be found. This algorithm works by searching a transformation chain that transforms the source transition system into the target transition system. If such a transformation chain can be found, the trivial refinement mapping can be established, i.e. the states and variables can be mapped one-to-one. The algorithm is based on an exhaustive search and uses breadth first search (BFS). It terminates if the target transition system is found or if it can be excluded that there exists a refinement mapping. This algorithm is shown to be correct and terminating. To reduce the search space, some optimizations for the algorithm are introduced that delete infinite paths in the BFS tree or subtrees that occur twice.

The rest of this thesis is organized as follows. In Chapter 2 an overview of

the basics and the theory behind translation validation is given. Chapter 3 deals with the various compiler optimizations like reordering of statements, loop unrolling, loop fusion, loop interchange, etc.. These transformations are formalized and classified such that the search space of the algorithm can be restricted, and thus, the algorithm can be optimized. Chapter 4 presents the main result of this thesis - the algorithm for finding a refinement mapping between two systems, where one is the translated version of the other. The algorithm is proven to be correct and terminating. Furthermore, it is defined which properties a transformation has to fulfill such that the algorithm can be extended with this transformation without loosing correctness or termination. An overview of the implementation of this algorithm and an example compiler can be found in Chapter 5. Chapter 6 gives some examples and experiments for executing and testing the algorithm.

1.2 Related Work

Translation validation was first introduced by Pnueli, Siegel, and Singerman in [PSS98b]. They use the result of [AL91] which shows the existence of refinement mappings. However, their tool critically depends on some simplifying assumptions that restrict the source and target to programs with a single external loop. Furthermore, only a very limited set of optimizations is supported. The solution for the translation validation task for optimizing compilers presented by [GZB05] and [Fan05] extends [PSS98b]. The reordering of the statements and loops during the compilation process is analyzed. Each transformation has its own characteristic reordering function by which the source program can be mapped to the target program. However, this approach uses some simplifying assumptions and needs to instrument the compiler.

Necula shows in [Nec00] his approach in translation validation for the GNU C compiler. Arbitrary loop nests are supported and no compiler instrumentation is required. Although Necula presents various heuristics to recover and identify the the optimizations the compiler has applied, some optimizations such as loop invariant code motion can not be handled.

The basics of temporal verification of reactive systems, i.e. finding invariants and proof conditions, etc. is presented in [MP95] by Pnueli and Manna. The whole verification process is based on this work.

For the compiler related tasks such as compiler construction, compiler transformations, etc., [AK02], [ASU86], [WM], and [GE99] were used.

Chapter 2

Translation Validation in Theory

2.1 Transition Systems

In this thesis reactive systems are considered. A reactive system is a system that maintains an ongoing interaction with an environment. The formal semantics of a reactive system is represented by the means of transition systems [MP95]. A and C denote the transition system of the source program and the transition system of the target program, respectively.

Definition 2.1. *A transition system T is a state machine defined by a 4 tuple $T = (V, O, \Theta, \rho)$ with*

- V*: The set of state variables $V = D \cup \{l\}$ consists of the finite set of data variables D and a distinguished variable $l \in V \setminus D$ called control variable.*
- O*: The set of observable variables with $O \subseteq D$.*
- Θ*: The initial condition that describes the initial states of the transition system.*
- ρ*: A boolean expression that relates a state to its possible successors.*

All state variables $x \in V$ are typed. By $dom(x)$ we denote the **domain** of state variable $x \in V$. The domain of the control variable l is finite. The set of **data variables** D is the set of variables that are manipulated during execution of a program, i.e. they can be read, and values of their domain can be assigned to them.

The **states** of a transition system have to be type-consistent interpretations over V. The value of variable $x \in V$ in state s can be referred to with $s[x]$.

States are related by the **transition relation** ρ which is a boolean expression over the set of variables V', with $V' = V \cup \{x' \mid x \in V\}$. Note that $dom(x) = dom(x')$. We say that s' is a successor state of s, denoted by $(s, s') \in \rho$, iff ρ evaluates to `true` for the variable interpretations of s and s'. For example if $\rho = (l = l_0 \wedge l' = l_1 \wedge x' = x + 1)$, where l is the control variable and x' and l' refer to the values of x and l in the successor state, then $(s, s') \in \rho$ iff $s[l] = l_0$ and $s'[l] = l_1$ and $s'[x] = s[x] + 1$. The transformation of the value of data variable x is called **data transformation** of x, i.e. the data transformation describes the update of the variables in each transition. Data transformations are induced by **statements**, e.g. data transformation $x' = x + 1$ is induced by statement $x = x + 1$.

ρ has always the form

$$\rho = \bigvee_{l_i,l_j} \rho_{l_i l_j}$$

where $\rho_{l_i l_j} = (l = l_i \wedge l' = l_j \wedge g \wedge a)$. By a, the data transformation is denoted which refers to the primed and unprimed version of the data variables. g is called the *guard* that is defined as:

$$g ::= \texttt{true} \mid \texttt{false} \mid x \leq c \mid x < c \mid x \geq c \mid x > c \mid g \vee g \mid g \wedge g$$

where c is a constant and $x \in D$. Note, if $g = \texttt{false}$ in $\rho_{s[l]s'[l]} = (l = s[l] \wedge l' = s'[l] \wedge g \wedge a)$ then $(s, s') \notin \rho$.

If a state s fulfills the initial condition Θ, denoted by $s \models \Theta$, s is called an **initial state**. The **observable variables** form the external behavior of a reactive system, i.e. they describe the behavior of the program in an environment. Namely, the observables are the input and output variables of a program. Note that the control variable l is never an observable variable, i.e. $l \notin O$.

2.2 Representation of Transition Systems

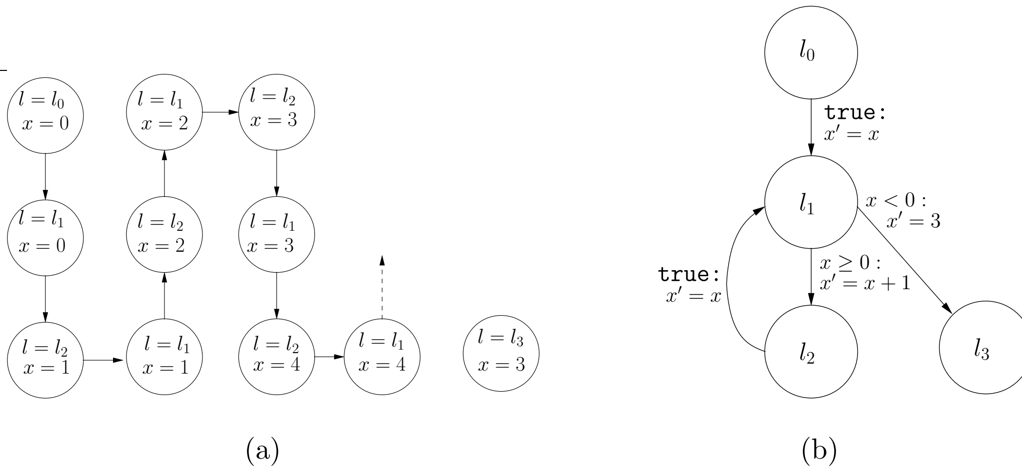
For a finite representation of transition systems the following notations are introduced. Since states are type consistent interpretations over the set of variables $V = D \cup \{l\}$, the number of states could be infinite. However, the domain of the control variable $dom(l)$ is finite. Thus, in order to obtain a finite representation, all states with the same value in l are combined to a so-called (control) location *loc*. By $loc[l]$ we denote the value of the control variable l in location *loc* and with loc_{l_i} we denote the location where $loc[l] = l_i$, i.e. $loc_{l_i}[l] = l_i$.

For a better readability we annotate the data transformations induced by ρ on the transitions in the graph. There is a transition from location loc_i to location loc_j labeled with guard g and data transformation a, denoted by $loc_i \rightarrow_{g:a} loc_j$, if $\rho = \ldots \vee \rho_{l_i,l_j} \vee \ldots$ with $\rho_{l_i,l_j} = (l = l_i \wedge l' = l_j \wedge g \wedge a) =$ `true`.

Example 2.1. *Consider the following transition system with control variable l:*

- $V = \{l\} \cup D$ *with* $D = \{x\}$*, and* $O = \emptyset$
- $\Theta = (l = l_0 \wedge x = 0)$
- $\rho = \rho_{l_0 l_1} \vee \rho_{l_1 l_2} \vee \rho_{l_2 l_1} \vee \rho_{l_1 l_3}$ *with:*
  $\rho_{l_0 l_1} = (l = l_0 \wedge l' = l_1 \wedge x' = x)$
  $\rho_{l_1 l_2} = (l = l_1 \wedge l' = l_2 \wedge x \geq 0 \wedge x' = x + 1)$
  $\rho_{l_2 l_1} = (l = l_2 \wedge l' = l_1 \wedge x' = x)$
  $\rho_{l_1 l_3} = (l = l_1 \wedge l' = l_3 \wedge x < 0 \wedge x' = 3)$

The respective infinite transition system is depicted in Figure 2.1 a) and the finite representation of the transition system as described in this section is depicted in Figure 2.1 b).



(a) (b)

Figure 2.1: a) Example transition system, b) transition system in finite representation

2.3 Refinement Mappings

In translation validation the main task is to compare two transition systems, namely the target transition system and the source transition system. If the target transition system can be proven to be a *correct translation* of the source program, the compiler was proven to be correct for this special input. This section presents a definition of "correct translation" based on the notion of *refinement* ([PSS98a], [AL91]).

Definition 2.2. *A* **path fragment** *in a transition system is a finite sequence of states* $s_1, s_2, \ldots, s_n$ *such that every two consecutive states are related by the transition relation* ρ*, i.e.* $(s_i, s_{i+1}) \in \rho$.

Definition 2.3. *A* **path** *of a transition system is an infinite sequence of states* $s_0, s_1, \ldots$*, such that every two consecutive states* s_i *and* s_{i+1} *are related by the transition relation* ρ*, i.e.* $(s_i, s_{i+1}) \in \rho$*. A path* $s_0, s_1, \ldots$ *is* **initial** *if* s_0 *satisfies the initial condition* Θ*, i.e.* $s_0 \models \Theta$.

From now on we consider transition systems with only infinite paths, i.e. every state has a successor (and thus every location in the finite representation). Also we assume that exactly one transition can be always taken. That means, all guards of the outgoing transitions of a location are disjoint and always one guard evaluates to true.

Definition 2.4. *A* **computation** $s_0, s_1, s_2, \ldots$ *of a transition system is an initial path.*

Each state can be split into its observable part $s_i[O]$ and its non-observable part $s_i[V \setminus O]$, i.e. a state can also be written as $s_i = \langle s_i[O], s_i[V \setminus O]\rangle$.

Definition 2.5. *Let* $\sigma : s_0, s_1, s_2, \ldots$ *be a computation. Then the* **observable computation** *of* σ *consists of only these states of* σ *that have a different observable part as their predecessor state.*

In other words, the observable computation describes the external behavior of a computation. As an example consider the computation
$\langle s_0[O], s_0[V \setminus O]\rangle$,
$\langle s_1[O] = s_0[O], s_1[V \setminus O]\rangle$,
$\langle s_2[O] \neq s_1[O], s_2[V \setminus O]\rangle$,
$\langle s_3[O] \neq s_2[O], s_3[V \setminus O]\rangle$,
$\langle s_4[O] = s_3[O], s_4[V \setminus O]\rangle$,
$\langle s_5[O] = s_3[O], s_5[V \setminus O]\rangle$,
$\langle s_6[O] \neq s_5[O], s_6[V \setminus O]\rangle, \ldots$.
Then the observable computation is $s_0, s_2, s_3, s_6, \ldots$.

Definition 2.6. *The* **traces** *of a transition system* $T(V, O, \Theta, \rho)$ *are defined as:*
$traces(T) = \{s_0[O]s_1[O] \ldots \mid s_0, s_1, \ldots$ *is an observable computation of* $T\,\}$

Definition 2.7. *Let* $A = (V^a, O^a, \Theta^a, \rho^a)$ *and* $C = (V^c, O^c, \Theta^c, \rho^c)$ *be two transition systems with* $O^a = O^c$.
C **refines** A*, denoted* C **ref** A*, if for all observable computations* $\tau : t_0, t_1, ...$ *of* C *there exists a observable computation* $\sigma : s_0, s_1, ...$ *of* A *such that*

$$\forall x \in O^a . \forall s_i \in \sigma . s_i[x] = t_i[x]$$

This definition implies that C refines A if all traces of C can also be found in A.

Lemma 2.1. *A transition system* $C = (V^c, O^c, \Theta^c, \rho^c)$ *is a* **correct translation** *of a transition system* $A = (V^a, O^a, \Theta^a, \rho^a)$ *iff* C *ref* A.

Hence, in order to check whether the compiler has translated the source program correctly, one has to check whether C ref A.

In order to establish this notion of refinement, for each observable computation of C the corresponding observable computation of A that fulfills the requirements from definition 2.7 has to be constructed. This is usually done by means of a *refinement mapping*.

A refinement mapping f from C to A maps states and variables of C to states and variables, respectively, of A. It has to fulfill the following properties:

1. *Initiation*: $s \models \Theta^c$ implies $f(s) \models \Theta^a$, forall states $s \in C$
2. *Propagation*: $(s, s') \in \rho^c$ implies $(f(s), f(s')) \in \rho^a$, forall states s, $s' \in C$
3. *Preservation of Observables*: $\forall x \in O^a . \forall s \in C . f(s)[x] = s[y]$ with $f(y) = x$

Since the set of states is infinite, this notion would yield an infinite mapping. In order to avoid this, *control mappings* are introduced. A control mapping maps locations from C to A while respecting the above properties "Initiation" and "Propagation". To preserve the third property "Preservation of Observables" a so-called *data mapping* is established that maps variables, i.e. the observable variables, from C to A.

2.4 Invariants and Proof Conditions

After establishing the refinement mapping between the target transition system and the source transition system the actual verification task is started. The task is to establish a simulation relation induced by the refinement mapping. In order to prove the correctness of the simulation relation, proof conditions which are quantifier-free first-order formulae, are derived. These proof conditions can be proven by a generic theorem prover. If all proof conditions are valid, the target system is a correct translation of the source system. In the following it is shown, how these proof conditions are derived ([MP95]).

Theorem 2.1. *(Simulation Theorem)*
Let $A = (V^a, O^a, \Theta^a, \rho^a)$ *and* $C = (V^c, O^c, \Theta^c, \rho^c)$ *be transition systems and let* $\sim$ *be a relation that relates states of the source system and states of the target system.* $\sim$ *is a simulation relation if*

$$\forall (s, s') \in \rho^c . \forall t, s \sim t . \exists t', (t, t') \in \rho^a . s' \sim t'$$

Intuitively this means that each step C can perform can also be performed by A.

The refinement mapping is provided by means of a labeling of the locations of the source transition system.

Definition 2.8. *Let $T = (V, O, \Theta, \rho)$ be a transition system with control variable l. A state formula ϕ is an invariant if it holds in all states. ϕ is called inductive, if:*

(i) $\Theta \rightarrow \phi$ and

(ii) forall transitions $\tau \in \{(loc[l_1], loc[l_2]) | l_1, l_2 \in dom(l)\}$ it holds:

$$\rho_\tau \wedge \phi \rightarrow \phi'$$

where ϕ' is obtained by replacing each free occurrence of a variable $y \in V$ by y'.

Note that an inductive state formula is always an invariant but the converse is not always true. The established simulation relation has to be inductive. Note that the control mapping is inductive by nature.

The data mapping as well as the control mapping are written as invariants into the locations of the source transition system. The control mapping describes which location of the target transition system is related to which location of the source transition system. The data mapping describes in which correlation the variables of the source transition system are with the variables of the target transition system. Hence, the invariants are state formulas over the union of the set of variables of the source transition system and the set of variables of the target transition system. To distinguish the variables of the target system they are labeled with an ":", e.g. variable x of the target transition system is denoted by $:x$ in the invariant. Since the purpose of this thesis is to establish the trivial refinement mapping, and thus $V^c = V^a$, the invariant of location loc_a has the form:

$$inv(loc_a) = p^c@loc_c \wedge (\textstyle\bigwedge_{x \in V^a} x = :x)$$

where $p^c@loc_c$ denotes that transition system C is in location loc_c. Note that $p^c@loc_c$ describes the refinement mapping, i.e. if f is the established refinement mapping then $f(loc_c) = loc_a$.

After annotating each location of the source transition system with its invariant the proof conditions can be derived. Recall definition 2.8 and the simulation theorem.

First, the initial verification condition is established.

$$(\forall s, s \models \Theta^c . p^c@s) \Longrightarrow \textstyle\bigvee_{\forall t. t \models \Theta^a} inv(t)$$

The initial verification condition guarantees that if the target system is in an initial state, then there has to be a initial state in the source system such that its invariant holds.

Afterwards, the inductive verification conditions are derived.

$$\begin{gathered}\forall (s, s') \in \rho^c. \\ \forall t^a.\exists (t, t'), t = t^a \wedge (t, t') \in \rho^a. \\ p^c@s \wedge p^c@\text{'}s' \wedge \rho^c_{s[l_c], s'[l_c]} \wedge inv(t) \\ \implies (\rho^a_{t[l_a], t'[l_a]} \implies inv'(t'))\end{gathered}$$

where l_a and l_c are the control variables of the source system and the target system, respectively. t^a is a state of the source system and with $p^c@\text{'}s'$ we denote that after taking transition $\rho^c_{s[l_c], s'[l_c]}$ transition system C is in state s'. These proof conditions say that if the target system is in state s and can take a transition that leads to s' and the invariant of t holds, i.e. state s is mapped by the refinement mapping to t, then there has to be a transition in the source system that can be taken. This transition ends up in state t' of the source system and the invariant of t' has to hold. Note that the form of these proof conditions are analog to the definition of the simulation relation.

These conditions can be proven by a generic theorem prover. If they are valid, we have a proof that the established refinement mapping is correct, and thus it was proven that C is a correct translation of A.

Chapter 3

Compiler Transformations

When a compiler translates a source program, it can apply several transformations to obtain an optimized target program. These transformations can change the whole program structure and therefore, also the execution order of the statements in the program. Thus, it has to be defined under which circumstances a transformation is valid.

One can distinguish between structure-modifying and structure-preserving transformations. The focus of this thesis is set on the structure-modifying transformations. This chapter describes the considered compiler transformations by means of transition systems and under which preconditions they can be applied.

After presenting the compiler transformations, they are classified, i.e. the transformations are analyzed by their effect on a transition system and their effect on each other.

3.1 Preliminaries

In preparation for the presentation of the various compiler transformations, we introduce some preliminary notion.

3.1.1 Loops - Definitions and Notation

In general, n nested loops in a transition system have the form depicted in Figure 3.1. $S(i_1, i_2, \ldots, i_n)$ is the loop body that is dependent on the index variables of the loops $(i_1, \ldots, i_n)$. $S(i_1, i_2, \ldots, i_n)$ is an abbreviation for a block of data transformations $a_1(i_1, i_2, \ldots, i_n)$ to $a_m(i_1, i_2, \ldots, i_n)$ that are performed during execution of the loop body and may be also dependent on the index variables. The guards $g_1, \ldots, g_n$ are dependent on $i_1, \ldots, i_n$, respectively. They have the form

$$g(i, H) ::= \texttt{true} \mid \texttt{false} \mid i \leq H \mid g \vee g \mid g \wedge g$$

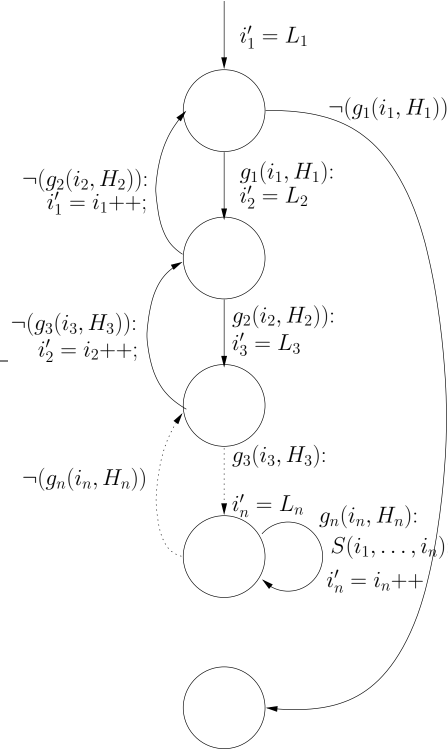


Figure 3.1: transition system with n nested loops

where H is a constant and i is the index variable of the loop.

The vector $\vec{i} = (i_1, \ldots, i_n)$ of index variables changes in the different iterations of the loops in the domain $\mathcal{I} = \{(L_1, \ldots, L_n), \ldots, (H_1, \ldots, H_n)\}$, where $H_1, \ldots, H_n$ are the constants of the guards and $L_1, \ldots, L_n$ are the initial values of the index variables. If the loop guard always evaluates to true, set $\mathcal{I}$ is infinite. On set $\mathcal{I}$ we define a total ordering $\prec_{\mathcal{I}}$.

Definition 3.1. *$\prec_{\mathcal{I}}$ is a total ordering over all $\vec{i} \in \mathcal{I}$, i.e. $\vec{i}_o \prec_{\mathcal{I}} \vec{i}_1 \prec_{\mathcal{I}} \ldots$. On this order the functions $++ : \mathcal{I} \mapsto \mathcal{I}$ and $-- : \mathcal{I} \mapsto \mathcal{I}$ are defined in postfix notation as:*

$$\vec{i}' = \vec{i} + + \textit{ iff } \vec{i} \prec_{\mathcal{I}} \vec{i}' \wedge \nexists \vec{i}'' \in \mathcal{I}. \vec{i} \prec_{\mathcal{I}} \vec{i}'' \prec_{\mathcal{I}} \vec{i}'.$$

$$\vec{i}' = \vec{i} - - \textit{ iff } \vec{i}' \prec_{\mathcal{I}} \vec{i} \wedge \nexists \vec{i}'' \in \mathcal{I}. \vec{i}' \prec_{\mathcal{I}} \vec{i}'' \prec_{\mathcal{I}} \vec{i}.$$

For n nested loops $\prec_{\mathcal{I}}$ is an ascending, lexicographic total ordering, i.e. $\prec_{\mathcal{I}} = <_{lex}$.

This description of n nested loops can be combined to a definition for a general loop:

Definition 3.2. *A* **general loop** $L(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ *in a transition system* $T(V, O, \Theta, \rho)$ *that starts in state* s_0 *is defined as follows.*
Let l_0 *be the value of control variable* l *in state* s_0*, i.e.* $s_0[l] = l_0$*. It holds the following:*

1. $s_0 \models \vec{i} = \vec{i}_0$
2. $\exists s_1, \ldots, s_n, s_{n+1}. s_0, s_1, \ldots, s_n$ *is a path fragment in* T *with* $s_n[l] = l_0$ *and*

   (i) $loc_{l_0} \rightarrow_{g_1:a_1(\vec{i})} loc_{s_1[l]} \wedge$

   (ii) $loc_{s_{n-1}[l]} \rightarrow_{g_n:a_n(\vec{i});\vec{i}'=\vec{i}++} loc_{l_0} \wedge$

   (iii) $loc_{l_0}, \rightarrow_{\neg g_1:a_{n+1}} loc_{s_{n+1}[l]}$

with $\vec{i}_0 \prec_\mathcal{I} \vec{i}_1 \prec_\mathcal{I} \ldots \prec_\mathcal{I} \vec{i}_N$ *and* $g_1 := \vec{i} \preceq_\mathcal{I} \vec{i}_N$
Location loc_{l_0} *is also called the* **starting location** *of the loop.*
Statement $\vec{i} = \vec{i}_0$ *is called* **initialization statement***.*
The dimension of $\vec{i}$ *(*$= dim(\vec{i})$*) denotes the number of nested loops in* L*.*

The first item says that the index vector has to be initialized right before the loop is entered, i.e. that state s_0 satisfies $\vec{i} = \vec{i}_0$. Item 2 guarantees that there is a circle starting in loc_{l_0} and that there have to be two transitions leaving loc_{l_0} where the guard and the negated guard (the abort condition of the loop) are checked. Item 2 also states that the index vector is incremented at the end of each loop cycle. Note that if the guard always evaluates to `true` the transition that is labeled with the negated guard (which is `false`) can just be omitted. The structure of a general loop is depicted in Figure 3.2.

Definition 3.3. *The* **execution** *of loop* L *with* $L = (\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ *is:*

$$\pi(L) : S(\vec{i}_0), S(\vec{i}_1), \ldots, S(\vec{i}_N)$$

For observing the execution of a loop, the loop body is instantiated with each $\vec{i} \in I$. Hence, a loop can be serialized by observing its execution.

Definition 3.4. *A loop* L_1 *with starting location* loc_{l_p} *is* **directly followed** *by a loop* L_2 *with starting location* loc_{l_q} *if there exist the following transitions in the transition system:*

$$loc_{l_k} \rightarrow_{g:i'=i_0} loc_{l_p} \rightarrow_{\neg g_n:j'=j_0} loc_{l_q}$$

where g_n *is the abort condition, i.e. the loop guard of* L_1*, and* g *is an arbitrary guard.*

From the definition of general loops follows the general definition for compiler loop transformations.

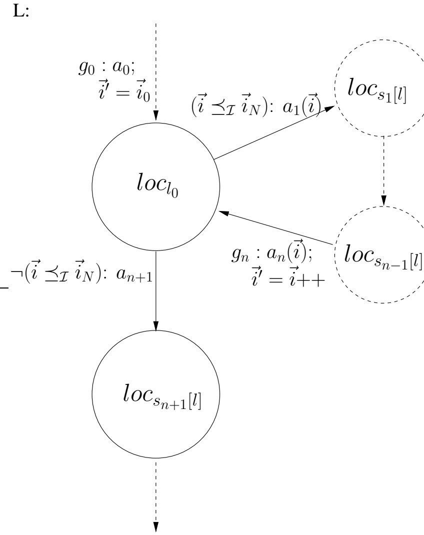


Figure 3.2: A general loop as defined in Definition 3.2.

Definition 3.5. *In general, a* **compiler loop transformation** *has the form:*
$L(\mathcal{I}, \prec_{\mathcal{I}}, S(\vec{i})) \quad \Longrightarrow_{\mathcal{T}} \quad L_{\mathcal{T}}(\mathcal{J}, \prec_{\mathcal{J}}, S'(\vec{j}))$
where $S'(\vec{j})$ is the transformed loop body that depends on loop index $\vec{j}$.

After applying a loop transformation, one can obtain the following changes:

1. Change of $\pi(L)$: After applying the transformation, the statements are executed in a different order. This is caused by
   - Change of total ordering $\prec_{\mathcal{I}}$
   - Change of set $\mathcal{I}$
   - Change of loop body $S(\vec{i})$
2. Change of index vector $\vec{i} = (i_1, \ldots, i_n)$. This is caused by:
   - Change of loop order, i.e. if nested loops are switched
   - Loop insertion/removal

3.1.2 Dependences between Statements

The statements of a transition system can be pairwise related. For example, there is a relation between a statement a_1 that writes to a variable x and a statement a_2 that later reads x, i.e. a_2 is dependent on a_1. The possibilities for dependences between statements a_1 and a_2 are the following (confer [AK02]):

- Data dependence: There is a data dependence from a_2 to a_1 iff both statements access the same variable and at least one of them writes to it and if there is a path fragment in the transition system that connects a_1 and a_2. The following data dependences can be distinguished:
  - a_2 is *truely dependent* on a_1: a_1 writes to a variable x and a_2 reads afterwards the value of x.
  - a_2 is *antidependent* on a_1: a_1 reads the value of a variable x before a_2 writes to x.
  - a_2 is *output dependent* on a_1: First a_1 writes to a variable x and afterwards a_2 writes to x.
  - a_2 is *input dependent* on a_1: First a_1 reads the value of a variable x and afterwards a_2 reads x. This is not a true dependence.
  - a_1 is *observable dependent* on a_2: a_1 and a_2 both write to an observable variable.
- Control dependence: These are all dependences that arise because of the control flow of the program. a_2 is *control dependent* on a_1 if the execution of a_2 depends on the result of a_1, i.e. a_2 is conditionally guarded by a_1. Confer example 3.1.

For loops we can distinguish the following two dependences:

- a_2 has a *loop-carried dependence* on a_1: There is a data dependence between statement a_2 of the loop body in iteration i and statement a_1 of the loop body in iteration i-1. That is $a_2(\vec{i})$ is dependent on $a_1(\vec{i}--)$. A loop-carried dependence can be
  - *forward dependent*: a_2 appears after a_1 in the loop body.
  - *backward dependent*: a_2 appears before a_1 in the loop body or a_1 and a_2 are the same statement. Confer example 3.2.
- *loop-independent dependence*: Data dependences between statements of the loop body that are not carried by the loop.

Example 3.1. *Figure 3.3 shows the control dependence of statement* $a_2 := (r' = r/x)$ *and statement* $a_1 := \neg(x == 0)$*.* a_2 *is control dependent on* a_1 *because if* x *is equal to 0 statement* a_2 *would not be executed.*

Example 3.2. *Figure 3.4 shows a loop-carried backward dependence of statement* $a_1 := (b(i+1) = b(i) + e)$ *and itself since* $b(i+1)$ *in iteration* i *depends on the result of iteration* $i-1$ *(where* $b(i)$ *was computed).*

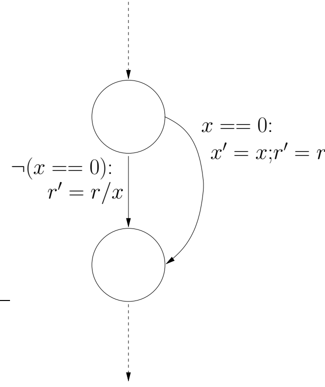


Figure 3.3: Control dependence between a_2 and a_1.

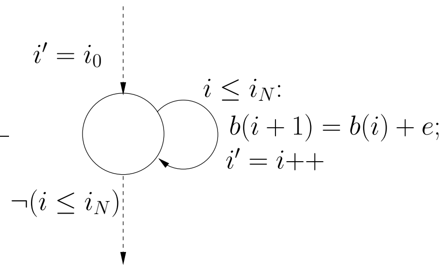


Figure 3.4: Loop-carried backward dependence between a_1 and itself.

Applying a transformation on a transition system can change the order of the execution of the statements. That means, before applying a transformation, the dependences between all pairs of statements have to be checked and it has to be taken care that these dependences are preserved by the transformation. Otherwise, the observable behavior of the transition system is changed. This is stated in the following theorem, a slightly changed version of the theorem in [AK02].

Theorem 3.1. *Any transformation $\mathcal{T}$ that reorders statements while preserving every dependence in a transition system T, preserves the traces of that transition system.*

Proof of Theorem 3.1. This proof is based on transition systems with a linear structure. Since transition systems can contain loops, we have to consider the execution of each loop, i.e. the loop body is instantiated with each $\vec{i} \in \mathcal{I}$. This yields a linear sequence of statements.
Proof by contradiction: After applying a transformation that reorders statements, all dependences between statements are preserved but the traces have changed.

Let $a_1; a_2; ...$ be a sequence of statements in the order in which they are executed in the transition system. This sequence induces the observable computation $s_0, s_1, s_2, ...$ in the transition system with its respective trace $s_0[O]s_1[O]s_2[O]....$ After applying a transformation that reorders statements we obtain a new sequence of statements $b_1; b_2; \ldots$ which is a permutation of $a_1; a_2; \ldots$ and induces the observable computation $t_0, t_1, ...$ with trace $t_0[O]t_1[O]t_2[O] \ldots$. Since by assumption the trace has changed, there is a smallest k such that $t_k[O]$ and $s_k[O]$ are different. Let a_m be the statement that results in the observable variable assignment of $s_k[O]$, and let b_l be the statement that results in the observable variable assignment of $t_k[O]$. Since s_k and t_k are states in the observable computation, a_m and b_l write both to an observable variable. There are the following possibilities why $t_k[O]$ and $s_k[O]$ can be different:

- b_l was originally executed after a_m but was inserted by the transformation before a_m: That means b_l and a_m were switched by the transformation. But this is a contradiction to the assumption that the dependences are preserved, since b_l and a_m both write to an observable and thus are observable dependent.

- b_l was originally executed before a_m but was moved by the transformation behind a_m: That means b_l and a_m were switched by the transformation. But this is a contradiction to the assumption that the dependences are preserved, since b_l and a_m both contain an observable and thus are observable dependent.

If a_m and b_l are the same statement, nevertheless, $t_k[O]$ and $s_k[O]$ can be different:

- A statement b_i that originally writes to a non-observable variable y before b_l reads it, now writes to y after b_l reads it. But this is a contradiction to the assumption that the dependences are preserved, since true dependence between b_i and b_l is not preserved.

- A statement b_i that originally writes to a non-observable variable y after b_l reads it, now writes to y before b_l reads it. But this is a contradiction to the assumption that the dependences are preserved, since antidependence between b_l and b_i is not preserved.

- b_l and a_m read a non-observable variable y that has different values when read by b_l and a_m, respectively. That b_l reads a "wrong" value of y can happen because:

  - Two statements b_i and b_j that both write to y before b_l reads it, were switched by the transformation. But this is a contradiction to the assumption that the dependences are preserved, since output dependence between b_i and b_j is not preserved.

- A statement b_i that writes to y before b_l reads it, reads originally a non-observable variable x before a statement b_j writes to x. If b_i and b_j are switched, b_i reads x after b_j writes to it, which produces a different result for y. But this is a contradiction to the assumption that the dependences are preserved, since anti dependence between b_i and b_j is not preserved.
- A statement b_i that writes to y before b_l reads it, reads originally a non-observable variable x after statement b_j writes to x. If b_i and b_j are switched, b_i reads x before b_j writes to it, which produces a different result for y. But this is a contradiction to the assumption that the dependences are preserved, since true dependence between b_i and b_j is not preserved.

Note, that a "wrong" result after switching two dependent statements b_i and b_j can also be propagated until it is read by b_l, i.e. b_l has not to read directly the wrong result produced by b_i and b_j. But these cases are analog to the last three cases.

These cases exhaust the way the traces can differ. Since all cases lead to a contradiction the theorem was proven. □

Definition 3.6. *A transformation is called* **valid** *for a transition system T if it preserves all dependences of T.*

We use theorem 3.1 to show under which conditions two statements are permutable.

Definition 3.7. *Two statements a_1 and a_2 are* **permutable***, denoted $a_1; a_2 \sim a_2; a_1$, if a_1 is executed before a_2 results in the same traces as executing them vice versa.*
$\mathcal{R}$ is the set of permutable statements, i.e.

$$(a_1, a_2) \in \mathcal{R} \Leftrightarrow a_1; a_2 \sim a_2; a_1$$

This definition can also expanded for a block of statements.

Definition 3.8. *Let $S_1(\vec{i}_1)$, $S_2(\vec{i}_2)$ be two loop bodies, i.e. two blocks of statements.*

$$(S_1(\vec{i}_1), S_2(\vec{i}_2)) \in \mathcal{R} \Leftrightarrow \forall a_1 \in S_1, a_2 \in S_2 : (a_1, a_2) \in \mathcal{R}$$

Theorem 3.2. *Two statements a_1 and a_2 are permutable if*

(i) they are neither antidependent, true dependent nor output dependent on each other

(ii) they contain no observables, i.e. they are not observable dependent on each other.

Proof of Theorem 3.2. This follows directly from theorem 3.1. □

3.2 The considered Compiler Transformations

In this section the considered compiler transformations are presented and formally defined. It is distinguished between structure-preserving and structure-modifying transformations. Since structure-modifying transformations are of more interest for solving the task of finding a refinement mapping than structure-preserving transformations, the focus of this thesis is set on the former.

3.2.1 Structure-Preserving Transformations

The following compiler transformations do not change the linear structure of the transition system, i.e. they do not change the branching structure or loop structure. Thus, they are called structure-preserving transformations.

Sequential Transition Refinement The transformation $\mathcal{T}_{str}$ splits a transition into several new transitions.

Definition 3.9. *Let a be a statement. Thus, there is the following transition in the transition system: $loc_{l_i} \rightarrow_{g_1:a} loc_{l_j}$, and the data transformation of a can be also computed by execution of several other statements $a_1 \dots a_n$. After applying $\mathcal{T}_{str}$ we obtain the following transitions:*
$loc_{l_i} \underbrace{\rightarrow_{g_1:a_1} loc_{l_i'} \rightarrow_{true:a_2} loc_{l_i''} \dots \rightarrow_{true:a_k}}_{\rightarrow_{g_1:a_1}} loc_{l_j}$.

Confer Figure 3.5 for an example.

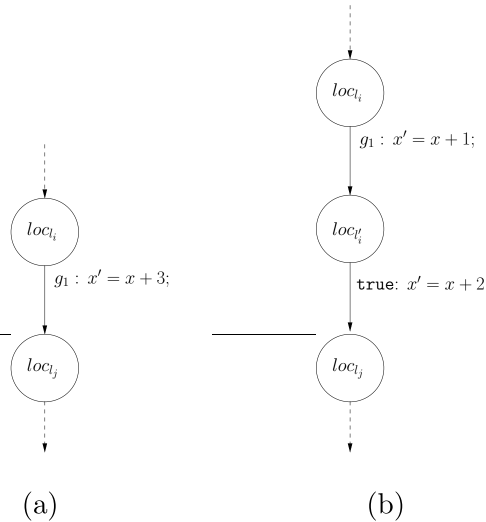


Figure 3.5: Application of transformation $\mathcal{T}_{str}$ to statement $x' = x + 3$.

Reordering of Statements The transformation $\mathcal{T}_{reordS}$ changes the order of statements in the transition system without inserting or deleting any transition.

Definition 3.10. *Let a_1 and a_2 be two statements that lie next to each other. Thus, there are the following transitions in the transition system: $loc_{l_i} \rightarrow_{g_1:a_1} loc_{l_j} \rightarrow_{\texttt{true}:a_2} loc_{l_k}$, where loc_{l_j} and loc_{l_k} must not be a starting location of a loop. After applying $\mathcal{T}_{reordS}$ we obtain the following transitions: $loc_{l_i} \rightarrow_{g_1:a_2} loc_{l_k} \rightarrow_{\texttt{true}:a_1} loc_{l_j}$.*

From this definition follows that not only the transitions itself are changed, but also its target locations. Confer Figure 3.6 for an example of $\mathcal{T}_{reordS}$.

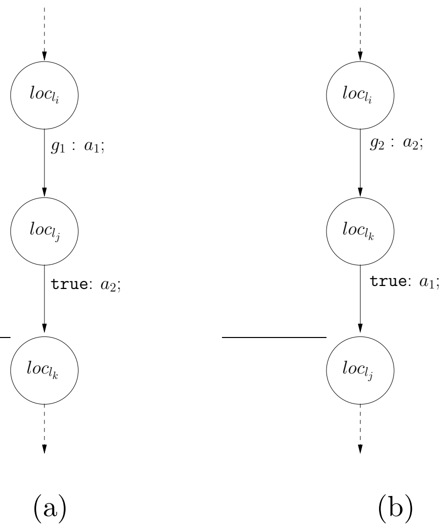


Figure 3.6: Application of transformation $\mathcal{T}_{reordS}$ to a_1 and a_2.

Since the dependences between the statements that are switched have to be preserved, the statements must not have a dependence, except input dependence. This is stated in the following lemma.

Lemma 3.1. *For two statements a_1 and a_2 $\mathcal{T}_{reordS}$ is a valid transformation if*

$$(a_1, a_2) \in \mathcal{R}$$

Note that the basic condition - $\mathcal{T}_{reordS}$ can be applied if a_1 and a_2 lie next to each other - has to be fulfilled in addition to this lemma.

3.2.2 Structure-Modifying Transformations

The transformations presented so far are structure preserving transformations. In the following the various compiler transformations for loops, which

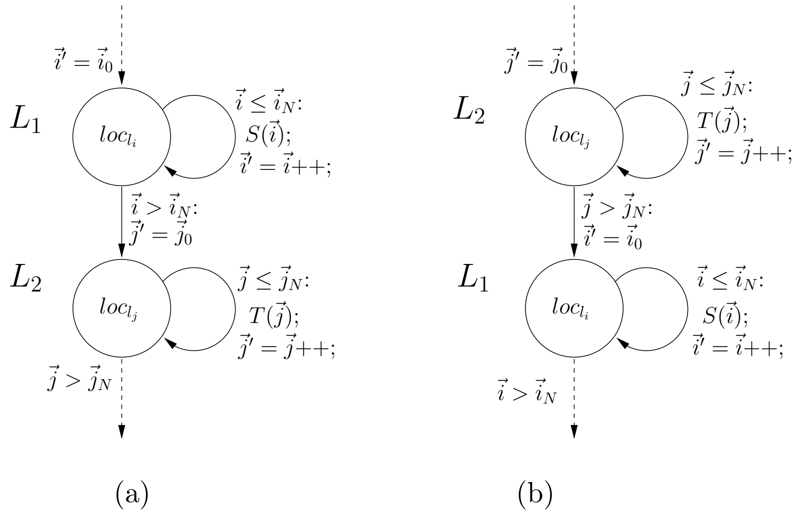


Figure 3.7: Application of transformation $\mathcal{T}_{reordl}$ to L_1 and L_2.

are structure-modifying transformations, are explained. The illustrations of the transition systems in this section show the change of the loop structure and statement structure, respectively.

Loop reordering The transformation $\mathcal{T}_{reordL}$ on loops is analog to transformation $\mathcal{T}_{reordS}$ for statements. But instead of switching two statements, $\mathcal{T}_{reordL}$ switches two loops.

Definition 3.11. *Let $L_1(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ and $L_2(\mathcal{J}, \prec_\mathcal{J}, T(\vec{j}))$ be two loops with starting locations loc_{l_i} and loc_{l_j}, respectively, and L_1 is directly followed by L_2. After applying transformation $\mathcal{T}_{reordL}$, loops L_1 and L_2 are switched such that now L_2 is directly followed by L_1.*

Note that the starting locations of the loops are also switched. Confer Figure 3.7 for an illustration.

Since after application of $\mathcal{T}_{reordL}$ the loop body of L_2 is executed before the loop body of L_1, there has to be a check for dependences between all statements in the loop body of L_1 with all statements in the loop body of L_2. This is stated in the following lemma.

Lemma 3.2. *For two loops $L_1(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ and $L_2(\mathcal{J}, \prec_\mathcal{J}, S_2(\vec{j}))$ $\mathcal{T}_{reordl}$ is a valid transformation if*

(i) the basic condition holds, i.e. L_1 is directly followed by L_2,

(ii) $\forall i_l \in \mathcal{I} : \ (a, S(i_l)) \in \mathcal{R}$, where a is the initialization statement of L_2, i.e. $a := j' = j_0$, and

(iii) $\forall \vec{i}_l \in \mathcal{I}, j_m \in \mathcal{J} : (S(\vec{i}_l), T(\vec{j}_m)) \in \mathcal{R}$.

Note that loop-carried dependences are preserved by this transformation since the execution of both loops is not influenced by this transformation.

Loop Fusion The transformation $\mathcal{T}_{fus}$ is the converse of transformation $\mathcal{T}_{dst}$. Two loops that range over the same index domain are combined to one loop. Figure 3.8 shows the transformation loop fusion.

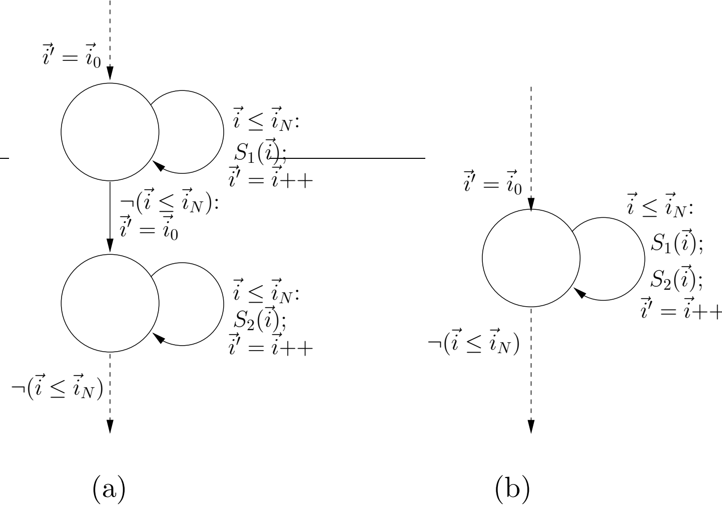


Figure 3.8: Loop fusion: a) source transition system, b) target transition system,
Loop Distribution: a) target transition system, b) source transition system

Definition 3.12. *Let* $L_1(\mathcal{I}_1, \prec_{\mathcal{I}_1}, S_1(\vec{i}))$ *and* $L_2(\mathcal{I}_2, \prec_{\mathcal{I}_2}, S_2(\vec{i}'))$ *be two loops with* $\mathcal{I}_1 = \mathcal{I}_2 = \{\vec{i}_0, \ldots, \vec{i}_N\}$ *and index* $\vec{i} \in \mathcal{I}_1$ *and* $\vec{i}' \in \mathcal{I}_2$. *The total orderings are defined as* $\prec_{\mathcal{I}_1} = \prec_{\mathcal{I}_2}$ *with* $\vec{i}_0 \prec_{\mathcal{I}_1} \vec{i}_1 \prec_{\mathcal{I}_1} \ldots \prec_{\mathcal{I}_1} \vec{i}_N$. *After applying* $\mathcal{T}_{fus}$ *we obtain* $\boldsymbol{L_{fus}(\mathcal{J}, \prec_{\mathcal{J}}, S(\vec{j}))}$ *with index* $\vec{j} \in \mathcal{J} := \mathcal{I}_1$, *a total ordering* $\prec_{\mathcal{J}} := \prec_{\mathcal{I}_1}$ *with* $\vec{j}_0 \prec_{\mathcal{J}} \vec{j}_1 \prec_{\mathcal{J}} \ldots \prec_{\mathcal{J}} \vec{j}_N$, *and the loop body* $S(\vec{j}) := S_1(\vec{j}); S_2(\vec{j})$.

In order to apply $\mathcal{T}_{fus}$, L_1 has to be directly followed by L_2 and the index range and the total ordering of L_1 and L_2 have to be the same (basic condition). After applying $\mathcal{T}_{fus}$, the index range $\mathcal{J}$ with the ordering $\prec_{\mathcal{J}}$ of L_{fus} is: $\mathcal{J} = \mathcal{I}_1 = \mathcal{I}_2$ and $\prec_{\mathcal{J}} = \prec_{\mathcal{I}_1} = \prec_{\mathcal{I}_2}$. The loop body of L_{fus} is the concatenation of the loop body of L_1 and the loop body of L_2, i.e. $S(\vec{j}) = S_1(\vec{j}); S_2(\vec{j})$.

If L_1 is directly followed by L_2 we obtain the following execution: $i' = i_0, S_1(\vec{i}_0)$, …, $S_1(\vec{i}_N)$, $i' = i_0$, $S_2(\vec{i}_0)$, …, $S_2(\vec{i}_N)$. After applying $\mathcal{T}_{fus}$ we obtain $\pi(L_{fus}) : S_1(\vec{i}_0), S_2(\vec{i}_0), \ldots, S_1(\vec{i}_N), S_2(\vec{i}_N)$. Since the statement blocks

are reordered, the dependences have to be checked. In order to apply $\mathcal{T}_{fus}$ the following lemma has to hold.

Lemma 3.3. *For two loops* $L_1(\mathcal{I}, \prec_\mathcal{I}, S_1(\vec{i}))$ *and* $L_2(\mathcal{I}, \prec_\mathcal{I}, S_2(\vec{i}))$ $\mathcal{T}_{fus}$ *is a valid transformation if* L_1 *is directly followed by* L_2 *and*

$$\forall \vec{i_l}, \vec{i_m} \in \mathcal{I} : \vec{i_0} \preceq_\mathcal{I} \vec{i_l} \prec_\mathcal{I} \vec{i_m} \prec_\mathcal{I} \vec{i_N} \Longrightarrow (S_1(\vec{i_m}), S_2(\vec{i_l})) \in \mathcal{R}$$

This lemma states that we only have to check for dependences between statements of block $S_1(\vec{i_m})$ with statements of block $S_2(\vec{i_l})$ where $\vec{i_l} \prec_\mathcal{I} \vec{i_m}$. All other pairs of statement blocks preserve their dependences, e.g. loop-carried dependences are preserved. This is shown in Figure 3.9, where $S_1(\vec{i})$ is the loop body of L_1 and $S_2(\vec{i})$ is the loop body of L_2 with $\vec{i} \in \mathcal{I} = \{0, 1, 2\}$. If two lines intersect, the statement pairs have to be permutable.

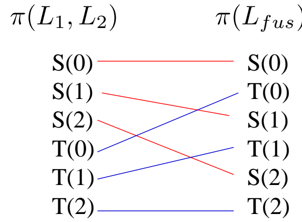


Figure 3.9: Reordering of statement blocks for $\mathcal{T}_{fus}$.

Proof of Lemma 3.3. In order to verify lemma 3.3, one can distinguish between four cases:

Case 1 Dependences between $S_1(\vec{i_m})$ and $S_1(\vec{i_l})$ $\forall i_m, i_l \in \mathcal{I}$ are preserved (the red lines do not intersect with each other): Since the total ordering $\prec_\mathcal{I}$ has not changed after applying $\mathcal{T}_{fus}$, $S_1(\vec{i_m})$ and $S_1(\vec{i_l})$ are not swapped in the execution. Hence, the dependences don't have to be checked.

Case 2 Dependences between $S_2(\vec{i_m})$ and $S_2(\vec{i_l})$ $\forall i_m, i_l \in \mathcal{I}$ are preserved (the blue lines do not intersect with each other): This case is analog to case 1.

Case 3 Dependences between $S_1(\vec{i_m})$ and $S_2(\vec{i_l})$ $\forall i_m, i_l \in \mathcal{I}$ with $i_m \preceq_\mathcal{I} i_l$ are preserved: In the original loops (L_1 and L_2), S is executed for all $\vec{i} \in \mathcal{I}$ before S_2 is executed, i.e. this holds for $S_1(i_m)$ and $S_2(i_l)$. Obviously, in L_{fus} $S_1(i_m)$ is also executed before $S_2(i_l)$ $\forall i_m \preceq_\mathcal{I} i_l$.

Case 4 Dependences between $S_1(\vec{i_m})$ and $S_2(\vec{i_l})$ $\forall i_m, i_l \in \mathcal{I}$ with $i_l \prec_\mathcal{I} i_m$ are **not** preserved (the intersecting lines): In the original loops (L_1 and L_2), S is executed for all $\vec{i} \in \mathcal{I}$ before S_2 is executed, i.e. this

holds for $S_1(i_m)$ and $S_2(i_l)$. However, in L_{fus} $S_1(i_m)$ is executed **after** $S_2(i_l)$. Hence, $S_1(i_m)$ and $S_2(i_l)$ are reordered and dependences have to be checked.

□

Loop Distribution The transformation $\mathcal{T}_{dst}$ is the converse of transformation $\mathcal{T}_{fus}$. $\mathcal{T}_{dst}$ splits the loop body into two parts and introduces two new loops, one for each loop body part, with the same index range. This is depicted in Figure 3.8, where b) is the source transition system and a) is the target transition system.

Definition 3.13. *Let $L(\mathcal{I}, \prec_{\mathcal{I}}, S(\vec{i}))$ be a loop with index $\vec{i} \in \mathcal{I} = \{\vec{i}_0, \ldots, \vec{i}_N\}$, a total ordering $\prec_{\mathcal{I}}$ with $\vec{i}_0 \prec_{\mathcal{I}} \vec{i}_1 \prec_{\mathcal{I}} \ldots \prec_{\mathcal{I}} \vec{i}_N$, and the loop body $S(\vec{i}) = S_1(\vec{i}); S_2(\vec{i})$. After applying $\mathcal{T}_{dst}$ we obtain two loops $\boldsymbol{L_{dst,1}(\mathcal{J}_1, \prec_{\mathcal{J}_1}, S_1(\vec{j}))}$ and $\boldsymbol{L_{dst,2}(\mathcal{J}_2, \prec_{\mathcal{J}_2}, S_2(\vec{j}'))}$ with index $\vec{j}, \vec{j}' \in \mathcal{J}_1 = \mathcal{J}_2 = \mathcal{I} = \{\vec{j}_0, \ldots, \vec{j}_N\}$. The total orderings are defined as $\prec_{\mathcal{J}_1} = \prec_{\mathcal{J}_2} = \prec_{\mathcal{I}}$ with $\vec{j}_0 \prec_{\mathcal{J}_1} \vec{j}_1 \prec_{\mathcal{J}_1} \ldots \prec_{\mathcal{J}_1} \vec{j}_N$.*

The basic condition that has to hold in order to apply $\mathcal{T}_{dst}$ is that the loop body S can be split in two parts, i.e. in S_1 and S_2. Since $\mathcal{T}_{dst}$ is the same transformation as $\mathcal{T}_{fus}$ only in the reverse direction, lemma 3.3 has also to hold for $\mathcal{T}_{dst}$ with the same explanations as above.

Lemma 3.4. *For a loop $L(\mathcal{I}, \prec_{\mathcal{I}}, S(\vec{i}) := S_1(\vec{i}); S_2(\vec{i}))$ $\mathcal{T}_{dst}$ is a valid transformation if*

$$\forall \vec{i}_l, \vec{i}_m \in \mathcal{I} : \vec{i}_0 \preceq_{\mathcal{I}} \vec{i}_l \prec_{\mathcal{I}} \vec{i}_m \prec_{\mathcal{I}} \vec{i}_N \Longrightarrow (S_1(\vec{i}_m), S_2(\vec{i}_l)) \in \mathcal{R}$$

From this lemma follows that if there is no loop-carried backward dependence of S_1 on S_2 the transformation can be applied safely.

Loop Unrolling The transformation $\mathcal{T}_{unr}(k)$ reduces the number of loop iterations by replicating the loop body. The parameter k is a natural number that denotes how often the loop body is replicated. Since the loop body is replicated, the number of iterations gets less. Figure 3.10 shows source transition system and target transition system where loop unrolling with parameter k has been applied.

Definition 3.14. *Let $L(\mathcal{I}, \prec_{\mathcal{I}}, S(\vec{i}))$ be a loop with index $\vec{i} \in \mathcal{I}$, a total ordering $\prec_{\mathcal{I}}$, and the loop body $S(\vec{i})$. Let k be a natural number. After applying $\mathcal{T}_{unr}(k)$ we obtain $\boldsymbol{L_{unr}(\mathcal{J}, \prec_{\mathcal{J}}, S'(\vec{j}))}$ with index $\vec{j} \in \mathcal{J} = \mathcal{I}$, a total ordering $\prec_{\mathcal{J}} := \prec_{\mathcal{I}}$, and the loop body*
$S'(\vec{j}) := S(\vec{j}); S(\vec{j}{+}{+}); \ldots; S(((\vec{j}\underbrace{{+}{+}) \ldots){+}{+}}_{k-times}); ((\vec{j}\underbrace{{+}{+}) \ldots){+}{+}}_{k-times}.$

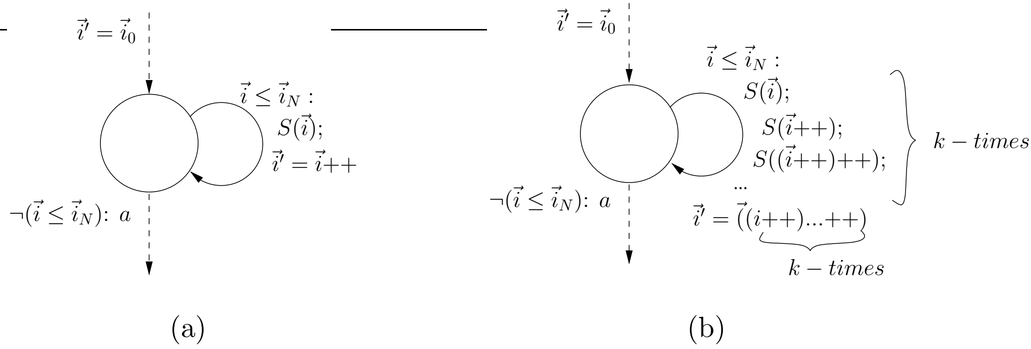


Figure 3.10: Loop Unrolling: a) source transition system, b) target transition system

Hence, loop L_{unr} has k times less iterations than loop L since the loop body is executed in each iteration of L_{unr} exactly k times. If k is chosen as 1, L_{unr} is the same loop as the original loop L. Note that if k does not fulfill $|\mathcal{I}|$ `mod` $k = 0$, the loop body is executed too often and a backtrack has to be done, i.e. the last execution of the loop body is revoked and the loop body $S(\vec{j})$ is executed one by one until $\vec{j}$ reaches $\vec{j}_N$.

When applying $\mathcal{T}_{unr}(k)$, the order of the execution of the statements in L_{unr} and L is the same, i.e. $\pi(L_{unr}) = \pi(L)$. Therefore, all dependences are preserved by this transformation and $\mathcal{T}_{unr}(k)$ can always be applied.

Loop Reversal The transformation $\mathcal{T}_{rev}$ reverses the range of the index variable while not changing the loop body. Transformation $\mathcal{T}_{rev}$ is depicted in Figure 3.11.

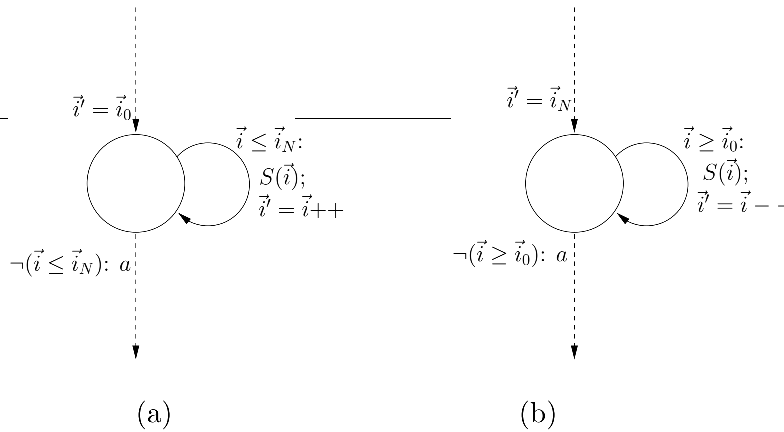


Figure 3.11: Loop Reversal: a) source transition system, b) target transition system

Definition 3.15. *Let $L(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ be a loop with index $\vec{i} \in \mathcal{I} = \{\vec{i}_0, \ldots, \vec{i}_N\}$, a total ordering $\prec_\mathcal{I}$ with $\vec{i}_0 \prec_\mathcal{I} \vec{i}_1 \prec_\mathcal{I} \ldots \prec_\mathcal{I} \vec{i}_N$, and the loop body $S(\vec{i})$. After applying $\mathcal{T}_{rev}$ we obtain $\boldsymbol{L_{rev}(\mathcal{J}, \prec_\mathcal{J}, S(\vec{j}))}$ with index $\vec{j} \in \mathcal{J} = \{\vec{j}_0, \ldots, \vec{j}_N\} = \mathcal{I}$, a total ordering $\prec_\mathcal{J} := \prec_\mathcal{I}^{-1}$ with $\vec{j}_N \prec_\mathcal{J} \vec{j}_{N-1} \prec_\mathcal{J} \ldots \prec_\mathcal{J} \vec{j}_0$, and the loop body $S(\vec{j})$.*

It can be seen easily that for loop reversal the index domain does not change. Since the index is reversed, the new total ordering $\prec_\mathcal{J}$ has to be reversed also. Hence, $\prec_\mathcal{J} := \prec_\mathcal{I}^{-1}$. In L the statements are executed in the order $S(\vec{i}_1), \ldots, S(\vec{i}_N)$ whereas in L_{rev} the order is reversed: $S(\vec{j}_N), \ldots, S(\vec{j}_1)$. Hence, if $S(\vec{i}_l)$ is executed before $S(\vec{i}_m)$ in L then $S(\vec{i}_m)$ is executed before $S(\vec{i}_l)$ in L_{rev} and the dependences between $S(\vec{i}_m)$ and $S(\vec{i}_l)$ need to be preserved under reordering. Loop-independent dependences are always preserved under $\mathcal{T}_{rev}$. Hence, in order to apply $\mathcal{T}_{rev}$ the following lemma has to hold.

Lemma 3.5. *For a loop $L(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ $\mathcal{T}_{rev}$ is a valid transformation if*

$$\forall \vec{i}_l, \vec{i}_m \in \mathcal{I} : \vec{i}_l \prec_\mathcal{I} \vec{i}_m \Longrightarrow (S(\vec{i}_l), S(\vec{i}_m)) \in \mathcal{R}$$

This lemma states that $\mathcal{T}_{rev}$ is a valid transformation if there are no loop-carried dependences.

Loop Interchange The transformation $\mathcal{T}_{icg}$ changes the nesting order of multiple loops. Transformation $\mathcal{T}_{icg}$ is depicted in Figure 3.12.

Definition 3.16. *Let $L(\mathcal{I}, \prec_\mathcal{I}, S(\vec{i}))$ be a loop with index $\vec{i} = (\vec{i}_1, \vec{i}_2) \in \mathcal{I}_1 \times \mathcal{I}_2 =: \mathcal{I}$. Forall $k \in \{1, 2\}$, let $\mathcal{I}_k = \{\vec{i}_{k,0}, \ldots, \vec{i}_{k,N}\}$ and $\prec_{\mathcal{I}_k}$ a total ordering on $\mathcal{I}_k$ with $\vec{i}_{k,0} \prec_{\mathcal{I}_k} \vec{i}_{k,1} \prec_{\mathcal{I}_k} \ldots \prec_{\mathcal{I}_k} \vec{i}_{k,N}$. The total ordering $\prec_\mathcal{I}$ is then defined as $(\vec{i}_1, \vec{i}_2) \prec_\mathcal{I} (\vec{i}'_1, \vec{i}'_2)$ iff $\vec{i}_1 \prec_{\mathcal{I}_1} \vec{i}'_1 \vee (\vec{i}_1 =_\mathcal{I} \vec{i}'_1 \wedge \vec{i}_2 \prec_{\mathcal{I}_2} \vec{i}'_2)$. The loop body is $S(\vec{i}) = S(\vec{i}_1, \vec{i}_2)$.*

After applying $\mathcal{T}_{icg}$ we obtain $\boldsymbol{L_{icg}(\mathcal{J}, \prec_\mathcal{J}, S(\vec{j}_2, \vec{j}_1))}$ with loop index $\vec{j} = (\vec{j}_1, \vec{j}_2) \in \mathcal{I}_2 \times \mathcal{I}_1 := \mathcal{J}$, total ordering $\prec_\mathcal{J}$ with $(\vec{j}_1, \vec{j}_2) \prec_\mathcal{J} (\vec{j}'_1, \vec{j}'_2)$ iff $(\vec{j}_2, \vec{j}_1) \prec_\mathcal{I} (\vec{j}'_2, \vec{j}'_1)$, and loop body $S(\vec{j}_2, \vec{j}_1)$.

The index vector $\vec{i}$ is split into two parts $(\vec{i}_1, \vec{i}_2)$. The total ordering $\prec_\mathcal{I}$ orders these index pairs lexicographically with $\prec_{\mathcal{I}_1}$ is a total ordering on the first component of the pair and $\prec_{\mathcal{I}_2}$ is a total ordering on the second component. Hence, L can be regarded as two nested loops L_1 and L_2 with $L_1(\mathcal{I}_1, \prec_{\mathcal{I}_1}, L_2)$ and $L_2(\mathcal{I}_2, \prec_{\mathcal{I}_2}, S(\vec{i}_1, \vec{i}_2))$. We obtain the following execution: $S(\vec{i}_{1,0}, \vec{i}_{2,0})$, …, $S(\vec{i}_{1,0}, \vec{i}_{2,N})$, $S(\vec{i}_{1,1}, \vec{i}_{2,0})$, …, $S(\vec{i}_{1,1}, \vec{i}_{2,N})$, …, $S(\vec{i}_{1,N}, \vec{i}_{2,0})$, …, $S(\vec{i}_{1,N}, \vec{i}_{2,N})$.

$\mathcal{T}_{icg}$ interchanges the nesting order of L_1 and L_2, i.e. L_2 is now the outer loop and L_1 is the inner loop. Since the loop body remains the same we obtain the execution: $S(\vec{i}_{1,0}, \vec{i}_{2,0})$, …, $S(\vec{i}_{1,N}, \vec{i}_{2,0})$, $S(\vec{i}_{1,0}, \vec{i}_{2,1})$, …, $S(\vec{i}_{1,N}, \vec{i}_{2,1})$, …, $S(\vec{i}_{1,0}, \vec{i}_{2,N})$, …, $S(\vec{i}_{1,N}, \vec{i}_{2,N})$.

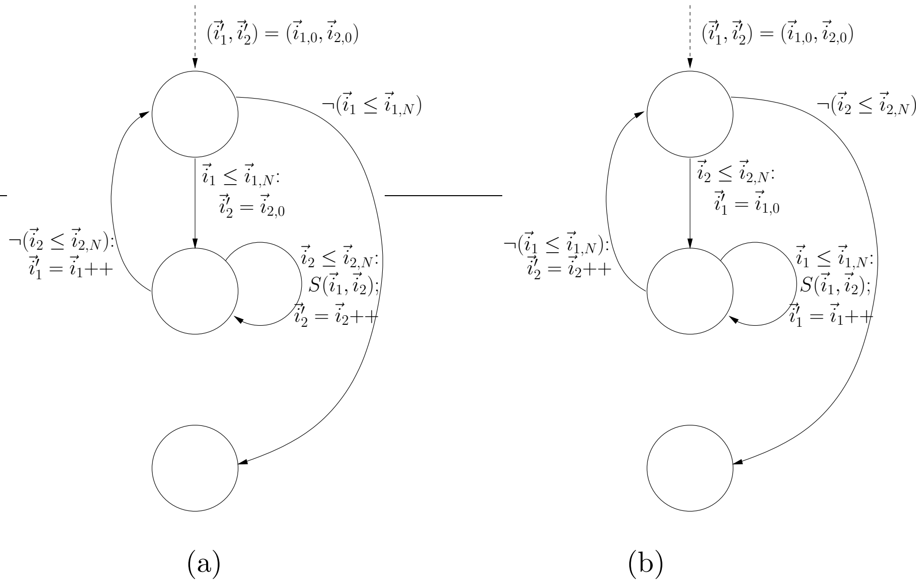


Figure 3.12: The transformation loop interchange: a) source transition system, b) target transition system

Since some statements are permuted, in order to apply $\mathcal{T}_{icg}$ the following lemma has to hold.

Lemma 3.6. *For a loop* $L(\mathcal{I}_1 \times \mathcal{I}_2, \prec_{\mathcal{I}}, S(\vec{i}_1, \vec{i}_2))$ $\mathcal{T}_{icg}$ *is a valid transformation if*

$$\forall (\vec{i}_1, \vec{i}_2), (\vec{k}_1, \vec{k}_2) \in \mathcal{I} : \vec{i}_1 \prec_{\mathcal{I}_1} \vec{k}_1 \wedge \vec{k}_2 \prec_{\mathcal{I}_2} \vec{i}_2 \Longrightarrow (S(\vec{i}_1, \vec{i}_2), S(\vec{k}_1, \vec{k}_2)) \in \mathcal{R}$$

Figure 3.13 shows this lemma graphically for $\mathcal{I}_1 = \mathcal{I}_2 = \{0, 1, 2\}$. If two lines intersect, the order of the respective statements has changed and thus they have to be permutable.

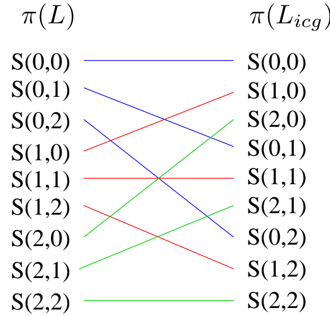


Figure 3.13: Reordering of statement blocks for $\mathcal{T}_{icg}$.

Proof of lemma 3.6. In order to verify lemma 3.6, four cases have to be checked. Figure 3.13 illustrates them.

Case 1 Dependences between $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2)$ are preserved if $\vec{i}_1 =_{\mathcal{I}_1} \vec{k}_1$ (lines of the same color do not intersect): trivial

Case 2 Dependences between $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2)$ are preserved if $\vec{i}_2 =_{\mathcal{I}_2} \vec{k}_2$: trivial

Case 3 Dependences between $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2)$ are preserved if w.l.o.g. $\vec{i}_1 \prec_{\mathcal{I}_1} \vec{k}_1 \wedge \vec{i}_2 \prec_{\mathcal{I}_2} \vec{k}_2$: From the assumption follows with definition of $\prec_{\mathcal{I}}$ that $(\vec{i}_1, \vec{i}_2) \prec_{\mathcal{I}} (\vec{k}_1, \vec{k}_2)$. Thus we have to show that $(\vec{i}_1, \vec{i}_2) \prec_{\mathcal{J}} (\vec{k}_1, \vec{k}_2)$. By definition of $\prec_{\mathcal{J}}$ we obtain $(\vec{i}_2, \vec{i}_1) \prec_{\mathcal{I}} (\vec{k}_2, \vec{k}_1)$. Since by assumption $\vec{i}_2 \prec_{\mathcal{I}_2} \vec{k}_2$, $(\vec{i}_1, \vec{i}_2) \prec_{\mathcal{J}} (\vec{k}_1, \vec{k}_2)$ holds. Thus, $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2))$ are not permuted and thus dependences are preserved.

Case 4 Dependences between $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2)$ are **not** preserved if w.l.o.g. $\vec{i}_1 \prec_{\mathcal{I}_1} \vec{k}_1 \wedge \vec{k}_2 \prec_{\mathcal{I}_2} \vec{i}_2$: From the assumption follows with definition of $\prec_{\mathcal{I}}$ that $(\vec{i}_1, \vec{i}_2) \prec_{\mathcal{I}} (\vec{k}_1, \vec{k}_2)$. Thus we have to show that $(\vec{i}_1, \vec{i}_2) \prec_{\mathcal{J}} (\vec{k}_1, \vec{k}_2)$. By definition of $\prec_{\mathcal{J}}$ we obtain $(\vec{i}_2, \vec{i}_1) \prec_{\mathcal{I}} (\vec{k}_2, \vec{k}_1)$. Since by assumption $\vec{k}_2 \prec_{\mathcal{I}_2} \vec{i}_2$, $(\vec{i}_1, \vec{i}_2) \not\prec_{\mathcal{J}} (\vec{k}_1, \vec{k}_2)$. Thus, $S(\vec{i}_1, \vec{i}_2)$ and $S(\vec{k}_1, \vec{k}_2))$ are permuted and thus dependences have to be checked.

□

3.3 Parameters for Transformations

For each transformation, parameters are chosen such that the transformation for a transition system is described uniquely. The loops of the transition system are identified by a unique loop name. How to choose these loop names is shown in the next chapter. For now, it suffices to choose the loop names as natural numbers.

- Loop Fusion $(\mathcal{T}_{fus}, loop1, loop2)$: Loop with name *loop*1 is fused with loop with name *loop*2.
- Loop Reversal $(\mathcal{T}_{rev}, loop1)$: Loop with name *loop*1 is reversed.
- Loop Distribution $(\mathcal{T}_{dst}, loop1, loc1)$: *loop*1 is split behind location *loc*1 (*loc*1 is a location of the loop body of *loop*1). *loc*1 is also called **split location**.
- Loop Reordering $(\mathcal{T}_{reordL}, loop1, loop2)$: *loop*1 and *loop*2 are switched.
- Reordering of Statements $(\mathcal{T}_{reordS}, loc1, loc2)$: Analog to the definition of $\mathcal{T}_{reordS}$, *loc*1 and *loc*2 (together with their incoming edges) are switched by this transformation.

- Loop unrolling $(\mathcal{T}_{unr}, loop1, k)$: $loop1$ is unrolled $k - times$.

The first parameter of each transformation is called the **type** of the transformation. In this chapter we consider transformations of type $\mathcal{T}_{fus}$, $\mathcal{T}_{dst}$, $\mathcal{T}_{reordL}$, $\mathcal{T}_{reordS}$, $\mathcal{T}_{unr}$, and $\mathcal{T}_{rev}$. If the parameters are of less importance, we just write the type of the transformation as an abbreviation for the whole transformation with arbitrary parameters.

3.4 Classification of Compiler Transformations

In practice, a compiler applies not only one, but several transformations - one after another - to a transition system. The application of a compiler transformation affects the set of applicable transformations, e.g. if a loop L is reversed, its index range is reversed and thus loop fusion may be enabled. An effect that can also occur when applying more than one transformation, is that the transformations may erase each other, i.e. transition system T is obtained again after applying two transformations that erase each other on T. An example for this is the application of loop fusion and afterwards loop distribution on the same loop. In this section the compiler transformations that were presented above are classified by the effects they have on each other. Note, that this leads not to disjunctive sets of transformations.

3.4.1 Definitions and Notation

Definition 3.17. *A* **transformation chain** B *is a finite sequence of transformations* $\mathcal{T}_1, \ldots, \mathcal{T}_n$*, i.e.* $B = \mathcal{T}_1; \mathcal{T}_2; \ldots; \mathcal{T}_n$.

To denote that a transformation $\mathcal{T}$ is applied to transition system T, we write $T[\mathcal{T}]$. According to this definition we can define $T[B]$, where $B = \mathcal{T}_1; B'$ is a transformation chain: First $\mathcal{T}_1$ is applied to T yielding transition system T', and afterwards B' is applied recursively to T' (until B' is empty).

To denote that a transformation $(\mathcal{T}, p_1, \ldots, p_n)$ produces new loops $(l_1, \ldots, l_n)$, we write $(\mathcal{T}, p_1, \ldots, p_n) \to (l_1, \ldots, l_n)$, where $l_1, \ldots, l_n$ are the loop names of the new loops. For example $(\mathcal{T}_{fus}, L1, L2) \to L3$.

3.4.2 Enabling Transformations

The transformations of this type may enable other transformations that were not possible before. That means the order in which they are applied to a transition system is important.

Definition 3.18. *A transformation* $\mathcal{T}_1$ *is called a* **candidate transformation** *for a transition system* T *iff all basic conditions of* $\mathcal{T}_1$ *hold.*

That means, a candidate transformation is a valid transformation if all dependences are preserved.

Definition 3.19. *Let T be a transition system. Transformation $\mathcal{T}_1$* **enables** *$\mathcal{T}_2$ if $\mathcal{T}_2$ is not a candidate transformation for T, but for $T[\mathcal{T}_1]$.*

The following transformations enable others:

- $\mathcal{T}_{fus}$ enables $\mathcal{T}_{dst}$: After fusion of two loops, the resulting loop can be again distributed.
- $\mathcal{T}_{fus}$ enables $\mathcal{T}_{reordS}$: After fusion of two loops, statements that were not in the same loop body before can be reordered.
- $\mathcal{T}_{rev}$ enables $\mathcal{T}_{rev}$: A loop that was just reversed can be reversed again.
- $\mathcal{T}_{rev}$ enables $\mathcal{T}_{fus}$: After reversing a loop, its index range has changed. Therefore fusion transformations could be enabled.
- $\mathcal{T}_{dst}$ enables $\mathcal{T}_{fus}$: After a loop is distributed, the resulting two loops can be fused again.
- $\mathcal{T}_{dst}$ enables $\mathcal{T}_{rev}$: The loop body of a loop can consist of a part that is reversible and a part that is not reversible. That means, if the loop is distributed, loop reversal can be applied to one of the resulting loops.
- $\mathcal{T}_{dst}$ enables $\mathcal{T}_{reordL}$: Here the same argument holds as for $\mathcal{T}_{dst}$ enables $\mathcal{T}_{rev}$.
- $\mathcal{T}_{reordS}$ enables $\mathcal{T}_{dst}$: The statements in the loop body can be reordered such that distribution is possible.
- $\mathcal{T}_{reordS}$ enables $\mathcal{T}_{reordS}$: If two statements are switched, exactly these two statements can be reordered again or two statements that are now new neighbors can be also switched.
- $\mathcal{T}_{reordL}$ enables $\mathcal{T}_{fus}$: After reordering there are new neighboring loops which can be fused.
- $\mathcal{T}_{reordL}$ enables $\mathcal{T}_{reordL}$: Here holds the same argument as for $\mathcal{T}_{reordS}$ enables $\mathcal{T}_{reordS}$, only for loops instead of statements.
- $\mathcal{T}_{unr}$ enables $\mathcal{T}_{fus}$: Since transformation $\mathcal{T}_{unr}$ changes the number of loop iterations, $\mathcal{T}_{fus}$ is enabled.
- $\mathcal{T}_{unr}$ enables $\mathcal{T}_{dst}$: Since transformation $\mathcal{T}_{unr}$ changes the loop body by replicating the original one, $\mathcal{T}_{dst}$ is enabled.
- $\mathcal{T}_{unr}$ enables $\mathcal{T}_{reordS}$: Since transformation $\mathcal{T}_{unr}$ changes the loop body by replicating the original one, $\mathcal{T}_{reordS}$ is enabled.

3.4.3 Inverse Transformations

This class of transformations contains all pairs of transformations that erase each other, i.e. the same transition system is obtained as applying none of these two transformations.

Definition 3.20. *Let T be a transition system. Transformation $\mathcal{T}_2$* **erases** *$\mathcal{T}_1$ iff $T[\mathcal{T}_1;\mathcal{T}_2] = T$.*
We also say that $\mathcal{T}_2$ is the **inverse** *of $\mathcal{T}_1$.*

The following transformations erase each other:

- ($\mathcal{T}_{fus}$,$\mathcal{T}_{dst}$): Fusing two loops and again distributing the resulting loop yields the same transition system as applying none of them. The reverse also holds: If a loop is distributed and the resulting loops are again fused, the original transition system is obtained.
- ($\mathcal{T}_{rev}$,$\mathcal{T}_{rev}$): Reversing the same loop two times yields the original loop.
- ($\mathcal{T}_{reordS}$,$\mathcal{T}_{reordS}$): If the same two statements that were switched before are switched again, this results in the same transition system as applying none of these transformations.
- ($\mathcal{T}_{reordL}$,$\mathcal{T}_{reordL}$): Here the same argument holds as for $\mathcal{T}_{reordS}$.

Note that $\mathcal{T}_{unr}$ has no inverse transformation.

3.4.4 Commutative Transformations

The pairs of transformations presented here are commutative. That means, the order of applying them to a transition system is not essential, i.e. if $\mathcal{T}_1$ is applied before $\mathcal{T}_2$ the same transition system T can be found as applying them vice versa.

Definition 3.21. *Let T be a transition system and $\mathcal{T}_1$ and $\mathcal{T}_2$ are valid transformations for T. Transformation $\mathcal{T}_1$ is* **weak commutative** *to $\mathcal{T}_2$ iff there exists a transformation $\mathcal{T}_3 \in \{\mathcal{T}_1, \mathcal{T}_2\}$ such that*

$$T[\mathcal{T}_1;\mathcal{T}_2] = T[\mathcal{T}_2;\mathcal{T}_3;\mathcal{T}_1].$$

Transformation $\mathcal{T}_1$ is **commutative** *to $\mathcal{T}_2$ iff*

$$T[\mathcal{T}_1;\mathcal{T}_2] = T[\mathcal{T}_2;\mathcal{T}_1].$$

Note that commutativity implies weak commutativity.

In the following, pairs of transformations $(\mathcal{T}_1, \mathcal{T}_2)$ that are commutative or weak commutative are presented. Confer also table 3.1 for an overview.

- $(\mathcal{T}_{fus}, \mathcal{T}_{fus})$: It does not matter in which order loops are fused.

$\mathcal{T}_1$	$\mathcal{T}_2$					
	$\mathcal{T}_{fus}$	$\mathcal{T}_{rev}$	$\mathcal{T}_{dst}$	$\mathcal{T}_{reordS}$	$\mathcal{T}_{reordL}$	$\mathcal{T}_{unr}$
$\mathcal{T}_{fus}$	C	W	X	E,C	W	-
$\mathcal{T}_{rev}$	E,W	X	W	C	C	C
$\mathcal{T}_{dst}$	X	E,W	C	C	E	-
$\mathcal{T}_{reordS}$	C	C	E,C	X	C	-
$\mathcal{T}_{reordL}$	E,W	C	W	C	X	C
$\mathcal{T}_{unr}$	E	C	E	E	C	-

Table 3.1: Commutative (C), Weak commutative (W), Erasing (X), and Enabling (E) transformations.

- $(\mathcal{T}_{fus}, \mathcal{T}_{rev})$: Fusing two loops L_1 and L_2 and afterwards reversing the resulting loop yields the same transition system as reversing L_1 and L_2 an afterwards fusing them.
- $(\mathcal{T}_{fus}, \mathcal{T}_{reordL})$: Fusing two loops L_1 and L_2 and afterwards reorder the resulting loop with L_3 yields the same transition system as reordering L_2 with L_3 and afterwards L_1 with L_3 and afterwards fusing L_1 and L_2.
- $(\mathcal{T}_{rev}, \mathcal{T}_{dst})$: Reversing a loop L and distributing the resulting loop yields the same as distributing L and afterwards reversing each of the resulting loops.
- $(\mathcal{T}_{dst}, \mathcal{T}_{dst})$: Distributing a loop two times yields 3 loops. The order in which these loops are produced is not important.
- $(\mathcal{T}_{unr}, \mathcal{T}_{reordL})$ and $(\mathcal{T}_{reordL}, \mathcal{T}_{unr})$: Since $\mathcal{T}_{reordL}$ does not influence the loop body, $\mathcal{T}_{unr}$ can be done before or after $\mathcal{T}_{reordL}$.

Note that also all pairs of transformations are commutative, that do not effect the same loops (or part) of the transition system, e.g. $(\mathcal{T}_{dst}, L_1, loc_1)$ and $(\mathcal{T}_{fus}, L_2, L_3)$ are commutative.

The described relations between transformations (due to the classification presented in the last sections) are presented in table 3.1.

Chapter 4

Finding a Refinement Mapping

Based on the results of chapter 2 and chapter 3 an algorithm is presented that checks whether one system is a correct translation of another. This algorithm is shown to be correct and terminating.

4.1 Naive Algorithm

The task of the algorithm presented here is to find a refinement mapping from a given target transition system to the given source transition system if there exists one. Recall that a refinement mapping maps locations and variables of the target program to locations and variables, respectively, of the source program. If such a refinement mapping (with the properties given in chapter 2) exists, the target transition system is proven to be a correct translation of the source transition system, and thus, the compiler was proven to be correct for this special input.

As we have seen, a compiler applies optimizing transformations during the compilation process of the input transition system that can change the whole structure of this system. Hence, the refinement mapping between the source and target transition system can not be found directly. The goal of the algorithm is to find a transformation chain $B = \mathcal{T}_1; \ldots; \mathcal{T}_n$ that transforms the source transition system A into the target transition system C, i.e. we have $A \rightarrow_{\mathcal{T}_1} A_1 \ldots \rightarrow_{\mathcal{T}_n} A_n$ where $A_n = C$. Since we have already shown that all transformations the compiler can apply preserve the traces of the transition system (confer chapter 3), it holds that in order to prove C ref A it suffices to show that C ref A_n. The following theorem follows directly from theorem 3.1.

Theorem 4.1. *Let A and C be two transition systems and C is the translated version of A. Let $B = \mathcal{T}_1; \ldots; \mathcal{T}_n$ be a transformation chain that transforms A into C with $A \rightarrow_{\mathcal{T}_1} A_1 \ldots \rightarrow_{\mathcal{T}_n} A_n$. Then it holds*

$$A \approx A_1 \approx \ldots \approx A_{n-1} \approx A_n \wedge C \textit{ ref } A_n \Rightarrow C \textit{ ref } A$$

where $A_i \approx A_j$ denotes that transition system A_i and transition system A_j have the same traces, i.e. transformation $\mathcal{T}_j$ preserves all dependences.

Since the transformation chain transforms A to C, the refinement mapping between C and A_n is just the trivial refinement mapping that maps locations and variables one-to-one. By finding this mapping and the transformation chain, it is proven that C is a correct translation of A.

The naive algorithm for finding a transformation chain is based on Breadth First Search (BFS). For now, we assume that the compiler has only applied $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{dst}$, $\mathcal{T}_{reordS}$, and $\mathcal{T}_{reordL}$. All possible transformation chains that can be applied to the input transition system A are established. First, all transformation chains of length 1 are computed. If none of these transformation chains produces C, the length is incremented by 1 and all transformation chains of the new length are established. This is done until C is found or until it can be excluded that C is a correct translation. If a refinement mapping between C and some A_n could be found, the algorithm terminates and outputs the computed transformation chain and the refinement mapping between C and A_n. It is proven in the next section that the algorithm is correct. In the case that C is not a correct translation of A, C does not refine any A_n in the BFS tree and the algorithm outputs false. Later it is shown that the algorithm terminates, i.e. that the number of different A_n's that can be produced by all transformation chains is finite. Compare listings 4.1 and 4.2 where *TS* is an abbreviation for *Transition System* and *depth* is a counter for the currently checked length of the transformation chain. How *maxdepth* is computed is shown in the next paragraph.

Since the BFS tree grows exponentially (depending on the number of allowed transformations), some optimizations to reduce the size of the BFS tree are introduced later.

Computation of maxdepth

In this section it is shown how *maxdepth* can be computed. Recall that *maxdepth* is a bound for the length of the transformation chains. Unfortunately, a transformation chain can be infinitely long. Consider for example the path consisting of an alternating sequence of loop fusion and loop distribution transformations on the same loops, or the path consisting just of reversal transformations on the same loop. However, note that on such an infinite path the same transition system occurs over and over again, i.e. at some point a circle is reached.

Thus, instead of choosing *maxdepth* as the maximal length of a transformation chain (which is infinite), we choose *maxdepth* as the maximal length of a transformation chain where all produced transition system are pairwise distinct. In other words, if transformation chain B which consists

```
Input: Source transition system A, Target transition system C
Output: if C is a correct translation of A:
                Transformation chain and refinement mapping.
        Otherwise: output false.

BFSTree tree;      // empty at the beginning
                   // the nodes are TSs and
                   // the edges are labeled with the
                   // applied transformation
tree.root = A;     // the root of tree is set to A
int depth = 0;     // current tree depth
int maxdepth = compute_maxdepth();
set N = emptyset;   // N is the set of applicable
                    // transformations
vector<TS> currentTS; // the produced TSs

if (C ref A)
  return (refinement mapping + empty transformation chain);
           // compiler has applied no transformation

while (depth <= maxdepth)
{
  add all leaves of tree to currentTS;
  while (!(currentTS.empty()))
  {
    fill_N(currentTS[0]); //computes all valid
                          // transformations applicable to
                          // currentTS[0]
    forall transformations T in N do
    {
      pick T of N;
      remove T from N;
      TS nextA = apply T on currentTS[0];
      add nextA to tree; // nextA becomes child of currentTS[0]
      if (C ref nextA)
        return (refinement mapping + transformation chain);
                        // a transformation chain, namely the
                        // path from the root node
                        // of tree to nextA, was found
    }
    erase first element of currentTS;
  }
  depth++;
}
return false;        // no transformation chain was found
                     // such that C refines some A_n
```

Listing 4.1: Naive Algorithm

```
fill_N (TS T)
{
   add all valid transformations for T to N;
}
```

Listing 4.2: Procedure `fill_N`

of *maxdepth* transformations, is applied to A, applying one more transformation to $A[B]$ would erase another transformation in B and thus, the same transition system A_i as somewhere else in the BFS tree is obtained. That means, A_i can be also obtained by a transformation chain shorter than *maxdepth*. That means, we choose *maxdepth* in a way such that all possible A_n's can be produced by transformation chains with length smaller than *maxdepth*. Thus, we show that the number of all possible A_n's is finite.

Therefore, the case that all transformations are applied sequentially to every loop and every location of transition system A, is analyzed. Since each of the considered transformation is the inverse of another transformation, applying one more transformation would erase the other one and a shorter transformation chain leading to the same transition system can be found. Thus, applying a transformation chain that consists of more than *maxdepth* transformations to A yields a transition system that already occurs somewhere else in the BFS tree and is also reachable by a transformation chain with less than *maxdepth* transformations.

In the following we assume that at each transition there is only one statement. *maxdepth* is dependent on the number of transformation types the compiler applies, the number of loops, and the number of locations in the transition system A. In the case that the compiler only applies $\mathcal{T}_{dst}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordL}$, $\mathcal{T}_{reordS}$, and $\mathcal{T}_{rev}$, we obtain the following bound for *maxdepth*:

$$\begin{array}{lcll} maxdepth & \leq & \frac{n*(n-1)}{2} & \text{Bound for } \mathcal{T}_{reordS} \\ & + & \frac{m*b*(m*b-1)}{2} & \text{Bound for } \mathcal{T}_{reordL} \\ & + & m*b-1 & \text{Bound for } \mathcal{T}_{fus} \\ & + & m*b & \text{Bound for } \mathcal{T}_{dst} \\ & + & m*b & \text{Bound for } \mathcal{T}_{rev} \end{array}$$

where m is the number of loops, n is the number of locations that are not contained in a loop body, and b is the number of locations of the loop body with the highest number of locations in the transition system.

Bound for $\mathcal{T}_{dst}$ Since each distribution transformation reduces the number of locations in the loop body, the number of $\mathcal{T}_{dst}$ is bounded. Let m be the number of loops and b be the number of locations of the loop body of the loop with the highest number of locations. In the worst case all loops are distributed at all locations such that each resulting loop has just one location in the loop body. Thus, we obtain

$$\text{maximal number of } \mathcal{T}_{dst} \leq m * b.$$

Bound for $\mathcal{T}_{reordL}$ Let m be the number of loops and b be the number of locations of the loop body of the loop with the highest number of locations. In the worst case all loops were distributed at each location and afterwards they are all reordered. Since the number of loops produced by loop distribution is $m * b$ we obtain

$$\text{maximal number of } \mathcal{T}_{reordL} \leq \frac{(m*b)*(m*b-1)}{2}.$$

Bound for $\mathcal{T}_{rev}$ Let m be the number of loops and b be the number of locations of the loop body of the loop with the highest number of locations. In the worst case all loops were distributed at each location and afterwards they are all reversed. Since the number of loops produced by loop distribution is $m * b$ we obtain

$$\text{maximal number of } \mathcal{T}_{rev} \leq m * b.$$

Bound for $\mathcal{T}_{fus}$ Let m be the number of loops and b be the number of locations of the loop body of the loop with the highest number of locations. In the worst case all loops were distributed at each location and reordered. Afterwards, they are all fused again. Since the number of loops produced by loop distribution is $m * b$ we obtain

$$\text{maximal number of } \mathcal{T}_{fus} \leq m * b - 1.$$

Bound for $\mathcal{T}_{reordS}$ Let n be the number of locations that are not contained in a loop body. We have to check for all possibilities they can be reordered. In the case that the locations are arranged sequentially, i.e. there are no branches, and that the order of the statements is reversed, the maximal number of reordering transformations is obtained. Thus, each location has to be switched with all other locations. Hence, we obtain

$$\text{maximal number of } \mathcal{T}_{reordS} \leq \frac{n*(n-1)}{2}.$$

Reordering of statements in a loop body can be simulated by loop reordering of loops that have just one location. Thus, this case is already handled in the bound for $\mathcal{T}_{reordL}$.

This approximation is very coarse and in general the maximal length over all transformation chains is much smaller. But this does not influence the correctness of the algorithm: If the real maximal length of all transformation chains is reached, only transition systems that have already been checked are produced. Thus, the output result of the algorithm remains the same.

4.1.1 Correctness

In this section it is shown that the presented algorithm is correct.

Theorem 4.2. *(Correctness) If a compiler which uses only the transformations $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, $\mathcal{T}_{reordL}$, and $\mathcal{T}_{dst}$ gets as input the source transition system A and outputs target transition system C, then the algorithm finds a transformation chain if and only if C is a correct translation of A.*

Note that the transformation chain found by the algorithm is not necessarily the same as the transformation chain the compiler has applied, i.e. the algorithm can find a shorter transformation chain that leads also to C. This is due to the existence of erasing and weak commutative transformations. For example if the compiler has applied the transformation chain $B = (\mathcal{T}_{dst}, Lx, loc1);(\mathcal{T}_{rev}, Lx1);(\mathcal{T}_{rev}, Lx2)$ the algorithm finds the shorter transformation chain $B' = (\mathcal{T}_{rev}, Lx);(\mathcal{T}_{dst}, Lx, loc1)$ which leads to the same transition system as $A[B]$.

Lemma 4.1. *If the algorithm is restricted to transformations $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, $\mathcal{T}_{reordL}$, and $\mathcal{T}_{dst}$, then the set N of valid transformations is always finite.*

Since all domains of the parameters of the transformations are finite, the transformations can only be instantiated finitely often. Thus, the set of all valid transformations is also finite.

Correctness Proof. "$\Rightarrow$" to show: If the algorithm outputs a transformation chain, then C is a correct translation of A.

All transformations the algorithm applies were shown to be correct (confer chapter 3). Hence, if there is a transformation chain that transforms A to A_n and C ref A_n, then C is a correct translation by theorem 4.1 and lemma 2.1.

"$\Leftarrow$" to show: If C is a correct translation of A then the algorithm finds a transformation chain.

This is shown by induction over the number n of transformations the compiler has applied.

Assumption: The dependency analysis of the algorithm is at least as powerful as the dependency analysis of the compiler.

Induction Hypothesis:

$n = 0$: In this case the compiler has not applied a transformation, hence $A = C$ and the algorithm terminates because A ref A by returning the empty transformation chain.

$n = 1$: The compiler has applied exactly one transformation $\mathcal{T}$. Since by assumption the dependency analysis of the algorithm is at least as powerful as the dependency analysis of the compiler, $\mathcal{T}$ is also added to N (in function `fill_N`). With lemma 4.1, $\mathcal{T}$ is applied to A and we obtain C. Thus

the algorithm has found a transformation chain, namely the transformation chain only consisting of transformation $\mathcal{T}$, that transforms A to C.
Induction Step:
$n-1 \rightarrow n$: There has to exist a transition system A_{n-1} that is reached after the compiler has applied $n-1$ transformations to A. If the compiler applies one more transformation, i.e. transformation $\mathcal{T}$, on T, $A_n = C$ is obtained. By the I.H. it holds that the algorithm finds a transformation chain TC from A to A_{n-1}. Hence, we can find A_{n-1} also in the BFS tree. Since the dependency analysis of the algorithm is at least as powerful as the dependency analysis of the compiler, $\mathcal{T}$ is also added to N (in function `fill_N`).
Case 1: if $n < maxdepth$, $\mathcal{T}$ is applied to A_{n-1} (by lemma 4.1) and we obtain C. Thus, the algorithm has found a transformation chain, namely the transformation chain $TC;\mathcal{T}$ that transforms A to C.
Case 2: If $n \geq maxdepth$, $\mathcal{T}$ is not applied to A_{n-1}. However, by the definition of *maxdepth* and the fact that each transformation is the inverse of another transformation, we know that applying one more transformation to TC erases another transformation. Thus, there has to exist a transformation chain TC' which is shorter than $TC;\mathcal{T}$ that also produces transition system C. Hence, the algorithm finds transformation chain TC'. □

4.1.2 Termination

The termination proof of the algorithm is split into two parts. First, it is proven that the algorithm terminates if C is a correct translation of A. Afterwards it is shown that the algorithm also terminates if C is not a correct translation of A and thus, C refines no A_n in the BFS tree.

The following theorem states that the presented algorithm terminates if C is a correct translation of A.

Theorem 4.3. *(Termination (Part I)) If a compiler which uses only the transformations $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, $\mathcal{T}_{reordL}$, and $\mathcal{T}_{dst}$ gets as input the source transition system A and returns transition system C which is a correct translation of A, then the algorithm terminates by finding a transformation chain that transforms A to C.*

Termination Proof (Part I). Since we have already shown that the algorithm is correct, i.e. that a transformation chain is found if C is a correct translation of A, the algorithm trivially terminates by returning this transformation chain. □

The following theorem states, that the presented algorithm also terminates if C is not a correct translation of A.

Theorem 4.4. *(Termination (Part II)) If a compiler which uses only the transformations $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, $\mathcal{T}_{reordL}$, and $\mathcal{T}_{dst}$ gets as input the source*

```
fill_N (TS T)
{
  add all valid transformations for T to N;

  check for rules;
                // remove transformations from N that would
                // yield the same TS that occurs already
                // somewhere in the BFS tree
}
```

Listing 4.3: Improved procedure `fill_N`

transition system A and returns transition system C which is not a correct translation of A, then the algorithm terminates by reaching maxdepth.

Termination Proof (Part II). As shown above, the length of all transformation chains is bounded by *maxdepth*. Since C is not a correct translation of A, C does not refine any A_n in the BFS tree. Hence, *depth* exceeds *maxdepth* at some point and the algorithm terminates by returning false. □

4.2 Optimizations

In this section optimizations are presented that reduce the size of the BFS tree and thus lead to a better performance of the algorithm.

In order to reduce the size of the BFS tree, procedure `fill_N` has to be adapted. By filling N in a convenient way, it is prevented that the same transition system occurs over and over again in the BFS tree. This is done by defining some rules that restrict the set of transformations that are added to N. Confer listing 4.3.

In the following these rules are presented and it is proven that the algorithm is still correct.

4.2.1 Encoding Transformations in Transition Systems

Each loop and location in the transition system should be identified uniquely, also after applying some transformations to it. Furthermore, we want to keep track of the transformations applied so far. This is done by encoding the applied transformations in the loop names.

Before applying a transformation, the locations of the given transition system are enumerated from 1 to n, where n is the number of locations. Afterwards, it is searched for all loops in the transition system. To each loop, a loop name is assigned. A loop name has the form $Lx|w$, where x is called a block and w stands for a sequence of blocks, i.e. $w = y|u|v| \dots$. Each non-empty block consists of a sequence of digits. The first digit is in

$\{1,...,m\}$, where m is the number of loops in the transition system, and the latter ones are in $\{1,2\}$. Additionally, each non-empty block can be annotated with r. An empty sequence of blocks is denoted by an underline '_'. With $w_{(r)}$ we denote a sequence of blocks, where some blocks may be annotated with r. Initially, the loops are enumerated with $L1|_$ to $Lm|_$. Note that locations, as well as loops, are enumerated in an ascending order (in the order they occur in the transition system). That means, if loop Lx is followed by loop Ly then $x < y$. In the following w and z denote sequences of blocks, x and y denote blocks, and i and j represent digits.

After applying a transformation, the loop name changes in the following way:

- $(\mathcal{T}_{rev}, Lw_{(r)})$, with $w \neq _$, produces loop $Lw_{(r^{-1})}$: each block that was not annotated with r, is now annotated with r, and vice versa.
- $(\mathcal{T}_{fus}, Lw_{(r)}, Lz_{(r)})$, with $w \neq _$ and $z \neq _$, produces loop $Lw_{(r)}|z_{(r)}$: the blocks that were annotated with r remain annotated with r.
- $(\mathcal{T}_{dst}, Lz_{(r)}|x|w_{(r)}, loc)$ produces loops $Lz_{(r)}|x1, Lx2|w_{(r)}$, if loc is a location of the loop body of loop $Lx|_$. Sequences z and w can also be empty.
- $(\mathcal{T}_{dst}, Lz_{(r)}|x_r|w_{(r)}, loc)$ produces loops $Lz_{(r)}|x1_r, Lx2_r|w_{(r)}$, if loc is location of loop body of loop $Lx|_$. z and w can also be empty.

The reordering transformations leave the loop names unchanged. It can be checked by the order of the loop names and the order of the location names, respectively, whether two loops (or statements) were switched. That means, loc_1 and loc_2 were not switched, if $loc_1 < loc_2$ and $Lx|_$ and $Ly|_$ were not reordered, if $x <_{lex} y$. Figure 4.1 shows an illustration of all transformations. The circles are an abstract representation of loops in a transition system and the colored circles are locations of the loop body.

In the following, the loop names are chosen as described in this section.

4.2.2 Removing duplicated Subtrees

As we have seen in the last chapter, there are many transformation pairs that are commutative or weak commutative. Hence, there are subtrees in the BFS tree that are exactly the same. Consider for example the path that starts with $(\mathcal{T}_{fus}, L1, L2)$;$(\mathcal{T}_{reordL}, L1|2, L3)$ and the path that starts with $(\mathcal{T}_{reordL}, L2, L3)$;$(\mathcal{T}_{reordL}, L1, L3)$; $(\mathcal{T}_{fus}, L1, L2)$. The produced transition systems T_1 and T_2 are the same. Thus, the subtree with T_1 as root and the subtree with T_2 as root are exactly the same. and hence, one of these subtrees can be removed. Some rules are brought up which prevent that such duplicated subtrees are inserted in the BFS tree.

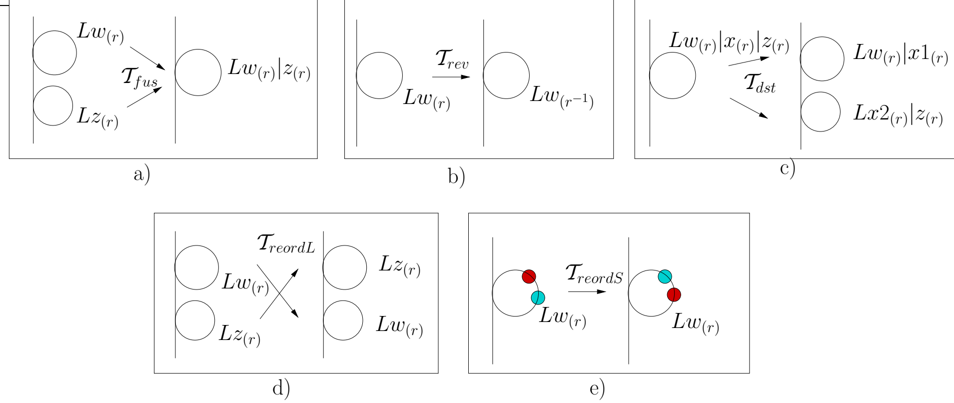


Figure 4.1: Illustration of Transformations: a) Loop fusion b) Loop reversal c) Loop distribution d) Loop reordering e) Reordering of statements.

In the following w,u,v, and z denote sequences of blocks, x and y denote blocks, and i and j stand for digits. $B[Lw_{(r)} \mapsto Lz_{(r)}]$, where B is a transformation chain, denotes that all parameters in B named $Lw_{(r)}$ are replaced by name $Lz_{(r)}$. We write $w \subseteq z$ for two non-empty sequences of blocks w and z, if all blocks of w also occur in z. The order of the blocks does not play a role.

N is already filled with the currently valid transformations for transition system T.

Rule 1: if $((p = (\mathcal{T}_{reordL}, Lu_{(r)}|w_{(r)}, Lz_{(r)})$ and $z \neq _ \wedge w \neq _ \wedge u \neq _)$: remove p from N
A loop that was fused already should not be reordered with other loops.

Rule 2: if $(p = (\mathcal{T}_{rev}, Lx_{(r)}|w_{(r)})$ and $w \neq _)$: remove p from N
A loop that was fused already should not be reversed.

Rule 3: if $(p = (\mathcal{T}_{dst}, Lx_r|_, loc))$: remove p from N
A loop that was reversed so far should not be distributed.

That these rules preserve the correctness criterion of the algorithm is shown later.

4.2.3 Removing Circles

In order to avoid infinite paths, and thus circles, in the BFS tree, some more rules are established. They have the task to remove those transformations from N that would yield a transition system that already occurs somewhere in the path from the root of the BFS tree to $currentTS[0]$.

In the following w and z denote sequences of blocks, x and y denote blocks, and i and j stand for digits. N is already filled with the currently valid transformations for transition system T.

Rule 4: if $(p = (\mathcal{T}_{dst}, Lx_{(r)}|w_{(r)}, loc)$ and $w \neq _)$: remove p from N
This rule avoids that a loop that was already fused, is distributed again.

Rule 5: if $(p = (\mathcal{T}_{fus}, Lw_{(r)}, Lz_{(r)}))$ and $w \neq _ \wedge z \neq _$: if $\exists (xi, yj) \in w \times z.x == y$: remove p from N
All blocks in w and z are compared pairwise without their last digit. If two blocks are equal (and non-empty), the loops were derived by loop distribution of the same loop and they are not allowed to be fused again.

Rule 6: if $(p = (\mathcal{T}_{rev}, Lx_r|_)$: remove p from N
Block x was already reversed. Hence, reversing once again is avoided by this rule.

Rule 7: if $(p = (\mathcal{T}_{reordL}, Lx_{(r)}|_, Ly_{(r)}|_))$ and $y <_{lex} x$: remove p from N
Since the loop names are in an lexicographic, ascending order if no reordering of loops was applied before, $Lx_{(r)}|_$ and $Ly_{(r)}|_$ were already reordered if $y <_{lex} x$. Thus, reordering them again is avoided by this rule.

Rule 8: if $(p = (\mathcal{T}_{reordS}, loc_1, loc_2))$ and $(loc_1 > loc_2)$: remove p from N
Locations loc_1 and loc_2 were already reordered.

Example 4.1. *This example shows the BFS tree that is produced by the algorithm for the shown part of the input transition system A (Figure 4.2). For the sake of simplicity we assume that the compiler was only allowed to use loop fusion, loop distribution and loop reversal. Confer Figure 4.3 for the respective BFS tree. If C is a correct translation of A then there has to exist a node A_i in the BFS tree with C ref A_i. Note that because of the rules, the BFS tree is finite.*

4.2.4 Correctness of the Rules

In order to prove that these rules preserve the correctness criterion of the algorithm, consider the following lemmas. Note that termination of the algorithm is not influenced by introducing these rules.

Lemma 4.2. *(Rule 1) Let T be a transition system with loops Lu, Lw, and Lz, where u, w, and z are non-empty sequences of blocks. Let B be a transformation chain. B can be split into B' and B'' and it holds:*

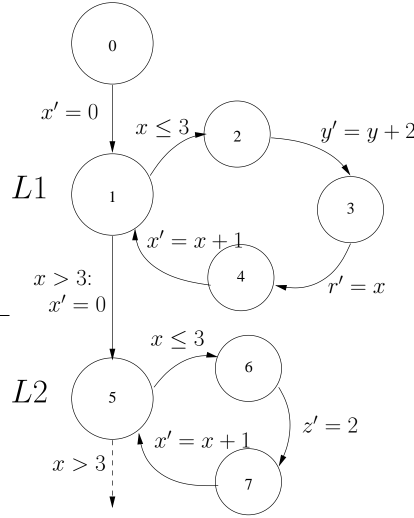


Figure 4.2: Transition system A. Confer example 4.1.

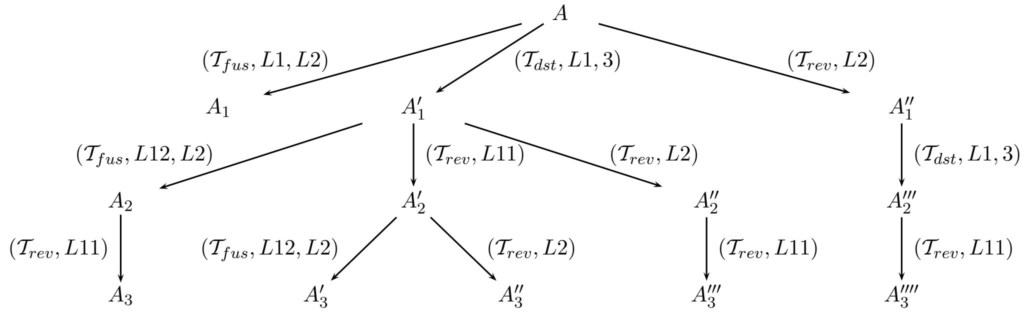


Figure 4.3: BFS Tree for transition system A of example 4.1.

$$T[(\mathcal{T}_{fus}, Lu_{(r)}, Lw_{(r)}); B; (\mathcal{T}_{reordL}, Lu_{(r)}|w_{(r)}, Lz_{(r)})] ==$$
$$T[B''; (\mathcal{T}_{reordL}, Lw_{(r)}, Lz_{(r)}); (\mathcal{T}_{reordL}, Lu_{(r)}, Lz_{(r)}); (\mathcal{T}_{fus}, Lu_{(r)}, Lw_{(r)}); B']$$

This lemma states that if $\mathcal{T}_1 = (\mathcal{T}_{reordL}, Lu_{(r)}|w_{(r)}, Lz_{(r)})$ is removed from N by rule 1, nevertheless a transformation chain can be found that produces the same transition system as applying $\mathcal{T}_1$. Confer Figure 4.4 for an illustration.

Proof of lemma 4.2. Let $T' = T[(\mathcal{T}_{fus}, Lu_{(r)}, Lw_{(r)}); B;$ $(\mathcal{T}_{reordL}, Lu_{(r)}|w_{(r)}), Lz_{(r)}]$. B can be split into B' that only contains transformations on loop $Lu_{(r)}|w_{(r)}$, and B'' that contains all other transformations on other loops. With rule 2 (no $\mathcal{T}_{rev}$ after $\mathcal{T}_{fus}$) and rule 4 (no $\mathcal{T}_{dst}$ after $\mathcal{T}_{fus}$), B' can only contain reordering of statements transformations on $Lu_{(r)}|w_{(r)}$. Thus, B' can also be applied after $\mathcal{T}_{reordL}$. B'' can also be applied before $\mathcal{T}_{fus}$, since it contains no transformation that affects $Lu_{(r)}|w_{(r)}$.

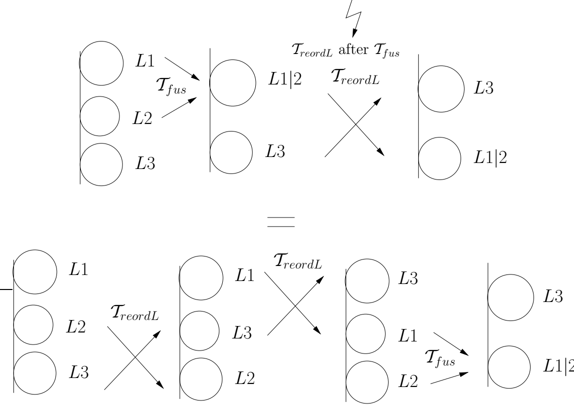


Figure 4.4: Loop reordering can always be applied before loop fusion (Lemma 4.2).

We obtain the transformation chain $T[B''; (\mathcal{T}_{reordL}, Lw_{(r)}, Lz_{(r)}); (\mathcal{T}_{reordL}, Lu_{(r)}, Lz_{(r)}); (\mathcal{T}_{fus}, Lu_{(r)}, Lw_{(r)}); B'] == T'$ which is a transformation chain that can be found in the BFS tree that leads to T'. □

Note that if sequences w, z, or u have more than one block, lemma 4.2 can be applied recursively until only loops with one block are reordered.

Lemma 4.3. *(Rule 2) Let T be a transition system with loops Lz and Lw, where z and w are non-empty sequences of blocks, and let B be a transformation chain. Then it holds:*

$$T[(\mathcal{T}_{fus}, Lz_{(r)}, Lw_{(r)}); B; (\mathcal{T}_{rev}, Lz_{(r)}|w_{(r)})] == \\ T[(\mathcal{T}_{rev}, Lz_{(r)}); (\mathcal{T}_{rev}, Lw_{(r)}); (\mathcal{T}_{fus}, Lz_{(r^{-1})}, Lw_{(r^{-1})}); \\ B[Lz_{(r)}|w_{(r)} \mapsto Lz_{(r^{-1})}|w_{(r^{-1})}]]$$

This lemma states that if by rule 2 transformation $\mathcal{T}_1 = (\mathcal{T}_{rev}, Lz_{(r)}|w_{(r)})$ is removed from N, nevertheless, a transformation chain can be found, that produces the same transition system as applying $\mathcal{T}_1$.

Proof of lemma 4.3. Let $T' = T[(\mathcal{T}_{fus}, Lz_{(r)}, Lw_{(r)}); B; (\mathcal{T}_{rev}, Lz_{(r)}|w_{(r)})]$. This transformation chain can not be found in the BFS tree, since $(\mathcal{T}_{rev}, Lz_{(r)}|w_{(r)})$ is removed from N by rule 2. Thus, we have to find another transformation chain that also leads to T'.

B can only contain reordering of statements transformations on the loop body of loop $Lz_{(r)}|w_{(r)}$ by rule 1 (no $\mathcal{T}_{reordL}$ after $\mathcal{T}_{fus}$) and rule 4 (no $\mathcal{T}_{dst}$ after $\mathcal{T}_{fus}$), and transformations that do not influence loop $Lz_{(r)}|w_{(r)}$. Since loop-independent dependences are preserved by $\mathcal{T}_{rev}$, the reordering of statements transformations can also be applied after $\mathcal{T}_{rev}$ on the reverse of loop $Lz_{(r)}|w_{(r)}$ which is $Lz_{(r^{-1})}|w_{(r^{-1})}$. Since all other transformations in B do not affect loop $Lz_{(r)}|w_{(r)}$, B can be applied after $\mathcal{T}_{rev}$.

We obtain the transformation chain
$(\mathcal{T}_{rev}, Lz_{(r)})$; $(\mathcal{T}_{rev}, Lw_{(r)})$; $(\mathcal{T}_{fus}, Lz_{(r^{-1})}, Lw_{(r^{-1})})$;
$B[Lz_{(r)}|w_{(r)} \mapsto Lz_{(r^{-1})}|w_{(r^{-1})}]$ that leads also to T' and can be found in the BFS tree. □

Note that if sequences w or z have more than one block, lemma 4.3 can be applied recursively until only loops with one block are reversed.

Lemma 4.4. *(Rule 3) Let T be a transition system with loop $Lx|_$. Let B be a transformation chain. Then it holds:*

$$T[(\mathcal{T}_{rev}, Lx|_); B; (\mathcal{T}_{dst}, Lx_r|_, loc)] ==$$
$$T[B[Lx_r|_ \mapsto Lx|_]; (\mathcal{T}_{dst}, Lx|_, loc); (\mathcal{T}_{rev}, Lx1|_); (\mathcal{T}_{rev}, Lx2|_)]$$

From this lemma follows that if $(\mathcal{T}_{dst}, Lx_r|_, loc)$ is removed from N by rule 3, nevertheless a transformation chain can be found, that yields the same transition system as applying $(\mathcal{T}_{dst}, Lx_r|_, loc)$.

Proof. Let $T' = T[(\mathcal{T}_{rev}, Lx|_); B; (\mathcal{T}_{dst}, Lx_r|_, loc)]$. B can neither contain loop fusion transformations (otherwise $Lx_r|_$ could not be distributed again by rule 4), nor loop reversal (by rule 6). Thus, B can only contain reordering transformations. Hence, B can also be applied to loop $Lx|_$ before $\mathcal{T}_{rev}$.

Since, $\mathcal{T}_{rev}$ is a valid transformation for T, $(\mathcal{T}_{dst}, Lx|_, loc)$ is also a valid transformation for T. This is implied by the fact, that loop reversal checks for more dependences than loop distribution (confer lemma 3.4 and lemma 3.5). We obtain the following transformation chain that applied to T also yields T': $T[B[Lx_r|_ \mapsto Lx|_]; (\mathcal{T}_{dst}, Lx|_, loc); (\mathcal{T}_{rev}, Lx1|_); (\mathcal{T}_{rev}, Lx2|_)] == T'$. □

To prove the correctness of rules 4 - 8, the following lemma is needed.

Lemma 4.5. *Let T be a transition system with loops Lw, Lx, and Lz, where w and z are a non-empty sequences of blocks, and let B be a transformation chain. Then B can be split into B' and B'' and it holds:*

$$T[(\mathcal{T}_{fus}, Lw_{(r)}, Lx_{(r)}); B; (\mathcal{T}_{fus}, Lw_{(r)}|x_{(r)}, Lz_{(r)})] ==$$
$$T[B''; (\mathcal{T}_{fus}, Lx_{(r)}, Lz_{(r)}); (\mathcal{T}_{fus}, Lw_{(r)}, Lx_{(r)}|z_{(r)});$$
$$B'[Lw_{(r)}|x_{(r)} \mapsto Lw_{(r)}|x_{(r)}|z_{(r)}]]$$

This lemma states that the order of fusion transformations does not play a role.

Proof. (Proof of lemma 4.5)
Let $T' = T[(\mathcal{T}_{fus}, Lw_{(r)}, Lx_{(r)}); B; (\mathcal{T}_{fus}, Lw_{(r)}|x_{(r)}, Lz_{(r)})]$. B can be split into B' that only contains transformations on loop $Lw_{(r)}|x_{(r)}$, and B'' that contains all other transformations on other loops. With rule 1, rule 2, and

rule 4 of the algorithm, B' can only contain reordering of statements transformations on $Lw_{(r)}|x_{(r)}$. Thus, B' can also be applied on $Lw_{(r)}|x_{(r)}|z_{(r)}$ after the second fusion transformation.

B'' can also be applied before $(\mathcal{T}_{fus}, Lx_{(r)}, Lz_{(r)})$, since it contains no transformation that affects $Lw_{(r)}|x_{(r)}$. By definition of loop fusion we obtain the same loop if we fuse $Lw_{(r)}|x_{(r)}$ with $Lz_{(r)}$ or $Lw_{(r)}$ with $Lx_{(r)}|z_{(r)}$. Thus, $T[B''; (\mathcal{T}_{fus}, Lx_{(r)}, Lz_{(r)}); (\mathcal{T}_{fus}, Lw_{(r)}, Lx_{(r)}|z_{(r)});$
$B'[Lw_{(r)}|x_{(r)} \mapsto Lw_{(r)}|x_{(r)}|z_{(r)}]] == T'$. □

Proof. (Correctness is preserved by rules 4 - 8)
Let TC be the transformation chain from the root of the BFS tree (which is A) to $currentTS[0]$ and p be the transformation that is removed by one of the above rules in procedure `fill_N`. We have to show that the algorithm nevertheless can find a transformation chain from A to $A_i = (currentTS[0])[p]$.

Rule 4: Let $p = (\mathcal{T}_{dst}, Lz_{(r)}|x_{(r)}|w_{(r)}, loc_k)$ and z or w (or both) is non-empty.
w.l.o.g. Let loc_k be a location of loop $Lx_{(r)}$.
$p = (\mathcal{T}_{dst}, Lz_{(r)}|x_{(r)}|w_{(r)}, loc_k) \rightarrow (Lz_{(r)}|x1_{(r)}, Lx2_{(r)}|w_{(r)})$. By construction of the names of the loops and lemma 4.5, $Lz_{(r)}|x_{(r)}|w_{(r)}$ is obtained by loop fusion of $Lz_{(r)}$, $Lx_{(r)}|_$, and $Lw_{(r)}$. Thus, $p_1 = (\mathcal{T}_{fus}, Lx_{(r)}|_, Lw_{(r)})$ and $p_2 = (\mathcal{T}_{fus}, Lz_{(r)}, Lx|w_{(r)})$ has to be in TC. We can split TC into $TC = B_1; p_1; p_2; B_2$. Let T' be the transition system before p_1 is applied (i.e. $T'[p_1; p_2; B_2] = currentTS[0]$). Note that if z or w is empty, p_1 and p_2, respectively, are not needed and can just be omitted.

There are the following cases for transformations contained in B_2:

(a) B_2 contains no $\mathcal{T}_{reordS}$
Note that $TC; p$ can not be detected by the algorithm, since p can not be applied to $currentTS[0]$ (because rule 4 is violated).
Since p is a valid transformation for $currentTS[0]$, $(\mathcal{T}_{dst}, Lx_{(r)}|_, loc_k) \rightarrow (Lx1_{(r)}|_, Lx2_{(r)}|_)$ is also a valid transformation for T'. Note that here no rule is violated and distribution can be applied by the algorithm.
The transformation chain B_2 only contains fusion- transformations to loop $Lz_{(r)}|x_{(r)}|w_{(r)}$, since loop reversal is not possible by rule 2. B_2 can also not contain any loop reordering transformation by rule 1. We can split B_2 into B_2' that has only $\mathcal{T}_{fus}$ with $Lz_{(r)}|x_{(r)}|w_{(r)}$ as second argument, and B_2'' that has only $\mathcal{T}_{fus}$ with $Lz_{(r)}|x_{(r)}|w_{(r)}$ as first argument.
With lemma 4.5, the algorithm finds the following transformation chain from A to A_i:

$B_1; (\mathcal{T}_{dst}, Lx_{(r)}|_, loc_k); (\mathcal{T}_{fus}, Lz_{(r)}|, Lx1_{(r)}|_);$
$(\mathcal{T}_{fus}, Lx2_{(r)}|_, Lw_{(r)}); B_2'[Lz_{(r)}|x_{(r)}|w_{(r)} \mapsto Lz_{(r)}|x1_{(r)}];$
$B_2''[Lz_{(r)}|x_{(r)}|w_{(r)} \mapsto Lx2_{(r)}|w_{(r)}].$
Figure 4.5 illustrates this case, where
$p_3 = (\mathcal{T}_{dst}, Lx_{(r)}|_, loc_k)$ and $p_4 = (\mathcal{T}_{fus}, Lz_{(r)}|, Lx1_{(r)}|_)$ and $p_5 = (\mathcal{T}_{fus}, Lx2_{(r)}|_, Lw_{(r)})$.

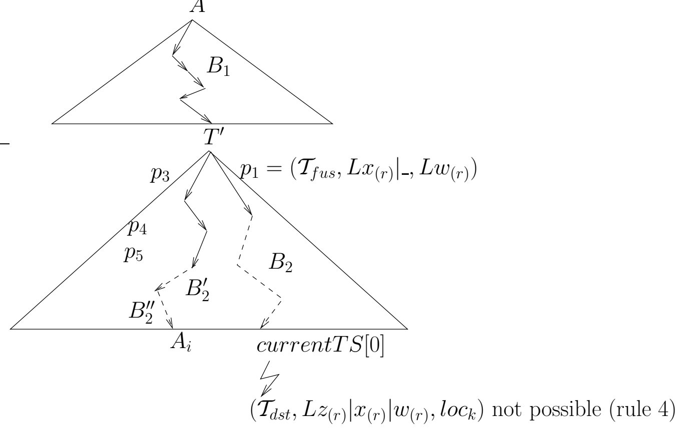


Figure 4.5: Whole BFS tree: Loop distribution should be applied to an already fused loop.

(b) B_2 contains reordering of statements transformations
Let $loc_1, \ldots, loc_k$ be locations of $Lz_{(r)}|x1_{(r)}$ and $loc_{k+1}, \ldots, loc_n$ be locations of $Lx2_{(r)}|w_{(r)}$.

(b - i) Statements are reordered only within $\{loc_1, \ldots, loc_k\}$ or within $\{loc_{k+1}, \ldots, loc_n\}$
That means, that the reordering of statements transformations can still be applied after $(\mathcal{T}_{dst}, Lx_{(r)}|_, loc_k)$ was applied. Thus, this case is analog to case (a).

(b - ii) Statements are reordered arbitrarily in $Lz_{(r)}|x_{(r)}|w_{(r)}$.
That means, there is a $loc_i \in \{loc_1, \ldots, loc_k\}$ and a $loc_j \in \{loc_{k+1}, \ldots, loc_n\}$ such that
$p_3 = (\mathcal{T}_{reordS}, loc_i, loc_j)$ is in B_2. But this reordering of statements can be also simulated by loop reordering on T'. This is depicted in Figure 4.6. Thus the following transformation chain is obtained: B_1;loop internal reordering of statements; loop distribution; simulation of reordering of statements by reordering of loops; loop fusion; loop internal reordering of statements; B_2 without reordering of statements.

Note that the loop internal reordering of statements is always possible, since it is also possible in $Lz_{(r)}|x_{(r)}|w_{(r)}$. Loop distribution is possible, since the statements that are distributed would also be distributed in $Lz_{(r)}|x_{(r)}|w_{(r)}$. Transformation loop reordering is allowed because reordering of exactly these statements of the loop body is also allowed in $Lz_{(r)}|x_{(r)}|w_{(r)}$.

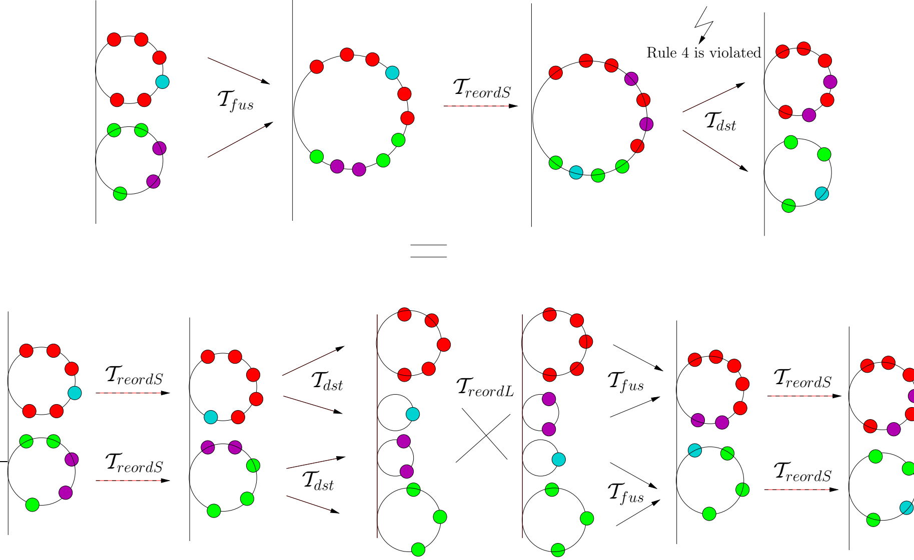


Figure 4.6: Reordering of statements was applied in B_2.

Rule 5: $p = (\mathcal{T}_{fus}, Lw_{(r)}, Lz_{(r)})$ where exists $(xi, yj) \in w \times z$ with $x == y$ and w.l.o.g. $i = 1$ and $j = 2$.
Thus, $TC; p$ can not be found by the algorithm.
Since $x == y$, $x1$ and $y2$ were derived by loop distribution of the same loop. Thus, the transformation
$p' = (\mathcal{T}_{dst}, Lx_{(r)}|_, loc) \rightarrow (Lx1_{(r)}|_, Lx2_{(r)}|_)$ has to be in TC. Let T' be the transition system before p' is applied. TC can be split into $TC = B_1; p'; B_2$. B_2 can be split into B_2'' that contains only contains reordering transformations on $Lx1_{(r)}|_$ and $Lx2_{(r)}|_$ and B_2' that contains the remaining transformations.

B_2' can not contain a distribution transformation on $Lx1_{(r)}|_$ or $Lx2_{(r)}|_$. Otherwise $x \neq y$.

If B_2' contains $\mathcal{T}_{rev}$ on $Lx1_{(r)}|_$ then it contains also $(\mathcal{T}_{rev}, Lx2_{(r)}|_)$ because otherwise p (= loop fusion) would not be a valid transformation. Since both parts of $Lx_{(r)}$ could be reversed, also $Lx_{(r)}$ could be reversed before distributing it.

Since p is a valid transformation for $currentTS[0]$ and with lemma 4.5, loop fusion transformations in B_2' can also be applied to $Lx_{(r)}$.

B_2'' contains loop reordering transformations on $Lu_{(r)}|x1_{(r)}|u'_{(r)}$ and $Lv_{(r)}|x2_{(r)}|v'_{(r)}$, where u, v, u', and v' are sequences of blocks (note that loop fusion on $Lx1_{(r)}|_$ and $Lx2_{(r)}|_$ is possible) with $u, v \subseteq w$ and $u', v' \subseteq z$. Since p is a valid transformation for $currenTS[0]$, $Lu_{(r)}|x1_{(r)}|u'_{(r)}$ and $Lv_{(r)}|x2_{(r)}|v'_{(r)}$ have to lie directly next to each other. Thus, reordering of loops can be simulated by reordering of statements on loop $Lu_{(r)}|u'_{(r)}|x_{(r)}|v_{(r)}|v'_{(r)}$. Thus, this can also be done without distribution of $Lx_{(r)}|_$. Figure 4.7 shows the case when B_2'' contains loop reordering transformations, i.e. how loop reordering can be simulated by reordering of statements.

With the above arguments we have shown that all transformations B_2' can contain can be also applied before distribution of $Lx_{(r)}|_$. Afterwards B_2'' can be applied. The algorithm finds the following transformation chain that yields the same transition system as applying $TC; p$ on A: B_1; $B_2'[Lx1_{(r)}|_ \mapsto Lx_{(r)}|_, Lx2_{(r)}|_ \mapsto Lx_{(r)}|_,]$; $B_2''[Lx1_{(r)}|_ \mapsto Lw_{(r)}|z_{(r)}, Lx2_{(r)}|_ \mapsto Lw_{(r)}|z_{(r)}]$;

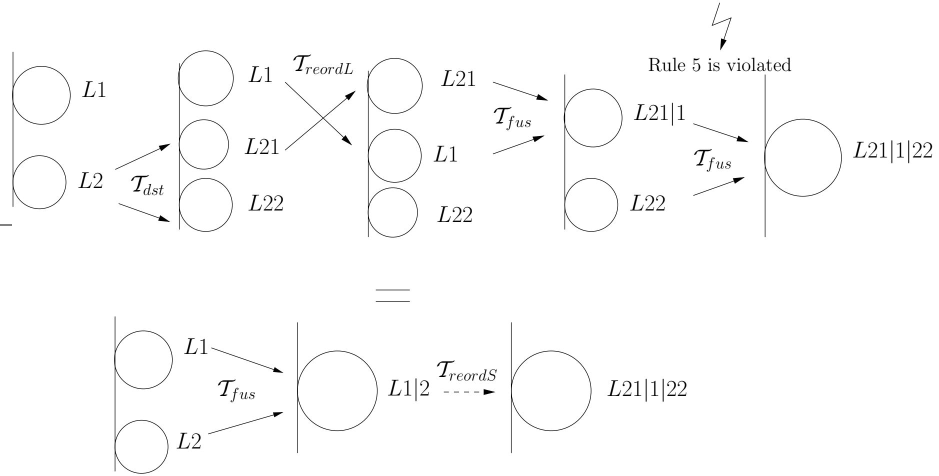


Figure 4.7: Loop fusion should be applied and B_2 contains loop reordering transformations.

Figure 4.8 illustrates this case.

Rule 6: $p = (\mathcal{T}_{rev}, Lx_r|_) \rightarrow Lx|_$
Since x is annotated with r, p is removed from N by rule 6. Hence, the path $TC; p$ does not exist in the BFS tree. Therefore, we have to find another transformation chain. With rule 3 and rule 2, we

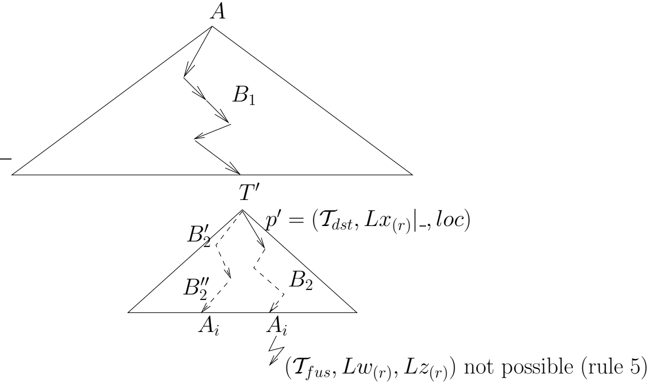


Figure 4.8: Whole BFS tree: Loop fusion should be applied to an already distributed loop.

know that there has to be a transformation $p' = (\mathcal{T}_{rev}, Lx|_{_})$ in TC. We can split TC into $TC = B_1; p'; B_2$. Let T' be the transition system before p' is applied. B_2 cannot contain loop fusion transformations since $Lx_r|_{_}$ consists of just one block, and by rule 3, B_2 can not contain loop distribution on $Lx_r|_{_}$. Thus, B_2 consists only of reordering transformations.

The algorithm finds the following transformation chain: B_1; $B_2[Lx_r|_{_} \mapsto Lx|_{_}]$. This transformation chain is valid, since reordering transformations on a reversed loop are also valid for the original loop. Figure 4.9 illustrates this case, where $B' = B_2[Lx_r|_{_} \mapsto Lx|_{_}]$.

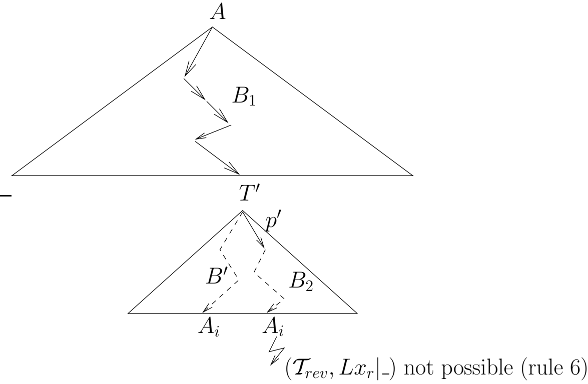


Figure 4.9: Whole BFS tree: Loop reversal should be applied to $Lx_r|_{_}$.

Rule 7: $p = (\mathcal{T}_{reordL}, Lux_{(r)}|_{_}, Lvy_{(r)}|_{_})$ and $vy <_{lex} ux$. x and y could be empty, u and v are sequences of digits that contain at least one digit.

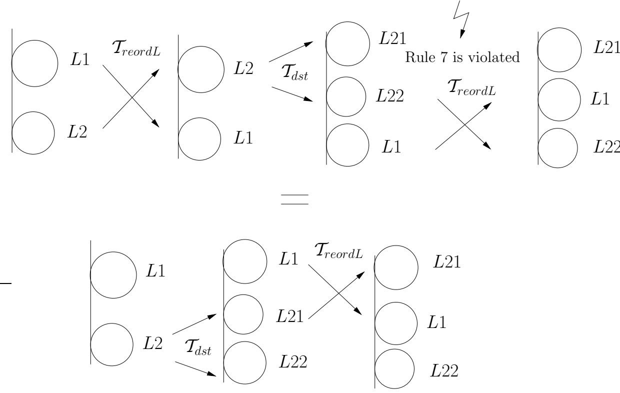


Figure 4.10: Loop reordering should be applied.

In this case, rule 7 of the algorithm is violated. Since $vy <_{lex} ux$, there exist two loops $Lu_{(r)}|_{_}$ and $Lv_{(r)}|_{_}$ that were reordered before. Thus, there is a transformation
$p' = (\mathcal{T}_{reordL}, Lu_{(r)}|_{_}, Lv_{(r)}|_{_})$ with $v >_{lex} u$ in TC. Let T' be the transition system before p' is applied. TC can be split into $TC = B_1; p'; B_2$.

If x and y are non-empty, B_2 contains loop distribution operations, by construction of the loop names. Thus, B_2 consists of distribution transformations, reordering transformations and reversal transformations. Since B_2 cannot contain loop fusion transformations (because the second block of $Lux_{(r)}|_{_}$ and $Lvy_{(r)}|_{_}$) is empty), B_2 can also be applied on T' (before p' was applied). Afterwards $\mathcal{T}_{reordL}$ is applied several times, i.e. until $Lux|_{_}$ and $Lvy|_{_}$ are reordered and A_i is reached. Figure 4.10 illustrates this case by an example.

Rule 8: $p = (\mathcal{T}_{reordS}, loc_1, loc_2)$ and $loc_1 > loc_2$
This case occurs if loc_1 and loc_2 were reordered yet, i.e. there is a $p' = (\mathcal{T}_{reordS}, loc_2, loc_1)$ or a $p'' = (\mathcal{T}_{reordL}, Lx_{(r)}|_{_}, Ly_{(r)}|_{_})$ in TC, where loc_1 is a location of loop $Ly_{(r)}|_{_}$ and loc_2 is a location of loop $Lx_{(r)}|_{_}$. In the case that p' was applied by the compiler, we just remove p' from TC. In the case that p'' was applied, the compiler applied the transformation chain: $p''; (\mathcal{T}_{fus}, Lx|_{_}, Ly|_{_}); p$. In the derivation tree we can find the following transformation chain:
$(\mathcal{T}_{fus}, Lx|_{_}, Ly|_{_});(\mathcal{T}^*_{reordS}, loc_1, loc_2)$ where $(\mathcal{T}^*_{reordS}, loc_1, loc_2)$ denotes, that $\mathcal{T}_{reordS}$ is applied several times, until loc_1 and loc_2 are switched.

□

4.3 Extending the Algorithm with more Transformations

So far, it was shown that the algorithm is correct and terminating if only $\mathcal{T}_{rev}$, $\mathcal{T}_{dst}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, and $\mathcal{T}_{reordL}$ were applied by the compiler. Now it is analyzed which properties a transformation has to fulfill such that the algorithm can be extended by this transformation without violating correctness or termination.

In order to guarantee correctness of the algorithm, the added transformation has to be proven correct, i.e. it is proven to preserve all dependences. Furthermore, the BFS tree has to be finitely branching. That means that to each transition system in the BFS tree only finitely many transformations can be applied. If this is not the case, we would never go into depth and although C is a correct translation of A we would never find a transformation chain.

Lemma 4.6. *(Property 1)*
Let $\mathcal{T}$ be a transformation that should extend the algorithm. If $\mathcal{T}$ preserves all dependences of the transition system and if the domain of all parameters of $\mathcal{T}$ is finite, then correctness of the algorithm is preserved when adding this transformation.

If the parameters have a finite domain, the number of instantiations of the transformations is finite. Thus, it is guaranteed that transformations can be applied only finitely often to a transition system. Note, that transformations $\mathcal{T}_{rev}$, $\mathcal{T}_{dst}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{reordS}$, and $\mathcal{T}_{reordL}$ all have this property. The domains of their parameters which are the set of loops and the set of locations, are all finite.

$\mathcal{T}_{unr}$ violates this property, since parameter k is a natural number and thus infinite. Hence, the BFS tree would not be finitely branching, and correctness is violated. However, if some metrics are used, nevertheless, the algorithm can be extended by $\mathcal{T}_{unr}$ as it is shown later.

Termination is given, if a transformation $\mathcal{T}$ can only be applied finitely often in a transformation chain or if it can be applied infinitely often but in this case produces the same transition system over and over again. $\mathcal{T}_{rev}$, $\mathcal{T}_{reordS}$, and $\mathcal{T}_{reordL}$ are only applicable finitely often, as it was shown in section 4.1.2. At some point, the transformation was applied to all loops and locations, respectively. Hence, applying it once more, erases another transformation and the same transition system as somewhere before is obtained. $\mathcal{T}_{dst}$ and $\mathcal{T}_{fus}$ change the number of locations in a loop and the number of loops, respectively. Thus, at some point each loop consists of one location or the transition system contains just one loop and $\mathcal{T}_{dst}$ and $\mathcal{T}_{fus}$, respectively, are no longer applicable. Hence, we can conclude:

Lemma 4.7. *(Property 2) Let $\mathcal{T}$ be a transformation that should extend the*

algorithm. Termination of the algorithm is preserved, if $\mathcal{T}$ is applicable only finitely often in each transformation chain. This is the case if

- *$\mathcal{T}$ erases $\mathcal{T}$ or*
- *$\mathcal{T}$ changes the transition system in such a way (e.g. the number of loops), that after finitely many applications of $\mathcal{T}$, $\mathcal{T}$ is not applicable any longer.*

$\mathcal{T}_{unr}$ also violates this property, since it can be applied infinitely often in a transformation chain, without producing the same transition system as somewhere before.

$\mathcal{T}_{icg}$ fulfills all properties, since it erases itself and because the instantiation of its parameters, i.e. the set of loops, is finite. Hence, the algorithm can be extended by $\mathcal{T}_{icg}$ (analogously to transformation $\mathcal{T}_{reordL}$).

These two properties suffice to guarantee termination and correctness since they guarantee that the BFS tree is only finitely branching and each path has only a finite depth when adding a new transformation. Thus, the BFS tree remains finite.

4.3.1 Loop unrolling

As we have seen in the last section, $\mathcal{T}_{unr}$ violates both properties that guarantee correctness and termination of the algorithm. Thus, the algorithm can not be extended by $\mathcal{T}_{unr}$ without doing some further work. In the following we show a possibility, how the domain of parameter k can be made finite and thus, $\mathcal{T}_{unr}$ fulfills the above properties.

By definition, $\mathcal{T}_{unr}$ changes the update of the index variable of the loop. For example, if $k = 5$ and index variable i is changed in the original loop L by 1, i.e. $i' = i+1$, then i is changed in L_{unr} by $i' = (((((i+1)+1)+1)+1)+1) = i + 5$. Because the target transition system C is given, the largest change of the index variable in the whole transition system can be found. Since there exists no transformation that erases $\mathcal{T}_{unr}$, i.e. the incrementation of the index variable can only increase and never can be decreased, the domain of parameter k can be made finite by checking at which $l \in dom(k)$ the largest change of the index variable in C is exceeded. Hence, for all $k > l$ no loop in C can be found to which L_1 can be mapped after applying $(\mathcal{T}_{unr}, L_1, k)$. Thus, all parameters greater than l can be omitted. With this restriction of parameter k the first property is fulfilled.

Now lets care about the second property. We have to show, that by knowing the largest change of the index variable in C, $\mathcal{T}_{unr}$ can only be applied finitely often in a transformation chain and thus property 2 would be fulfilled. But this is easy to show:

Since each application of $\mathcal{T}_{unr}$ increases the change of the index variable, at some point the largest change of the index variable in C is exceeded, and

thus, this transformation chain can be discarded. Hence, $\mathcal{T}_{unr}$ has not to be applied further more and the transformation chain is finite. Thus, property 2 is also fulfilled.

Since by using this metric both properties are fulfilled for $\mathcal{T}_{unr}$, the algorithm can be extended by $\mathcal{T}_{unr}$ as just presented.

Chapter 5

Implementation

The presented algorithm as well as an example compiler were implemented in the TVopt library (Translation Validation for Optimizing Compilers) which is embedded in the VRP (Verisoft Refinement Prover). The chosen language for the implementation is C++. The respective class diagrams and the hierarchy of them can be found in the appendix. All transition systems of this chapter were generated by the TVopt tool.

5.1 The Components

The following components form the basis for the implementation of the compiler as well as for the implementation of the algorithm.

5.1.1 Abstract Transition Systems

The class `AbstractTransitionSystem` (ATS) represents transition systems in a very abstract way, i.e. transition systems are reduced to their loops. It saves all information about loops which are the starting location, locations that belong to the loop body, the name of the index variable, the initialization and update of the index variable during execution of the loop, and the abort condition of the loop.

Furthermore, this class has the task to search all loops in a given transition systems. Formally a loop in a transition system is a strongly connected component that can be defined as followed (confer [ASU86]).

Definition 5.1. *A location s in a transition system T* **dominates** *location b if every path from the initial location to b goes through s.*

There is a **loop** *with starting location s in the transition system if there exists a b such that there is a transition from b to s and s dominates b.*

The best way to find loops in a transition system is by using Depth First Search (DFS), i.e. the algorithm starts at the initial location of the

transition system and moves through all other locations. If there is an edge that goes back to a location we have seen already (a so-called backward edge), a loop was found.

The class `AbstractTransitionSystem` also keeps track of the loop names, i.e. if a transformation is applied to a transition system also its ATS is updated. Compare the definition of the change of the loop names when applying a transformation (chapter 4). ATSs are important for checking the rules of the algorithm (chapter 4.2).

5.1.2 Dependency Analysis

To determine which transformations are valid for a given transition system, the dependency analysis class is invoked. It consists of two components: the control flow analysis and the data flow analysis. The control flow analysis checks for control dependences by building the control flow graph (CFG) and the data flow analysis checks for data dependences by constructing the data dependency graph. Confer chapter 3 for the definition of control dependence and data dependence.

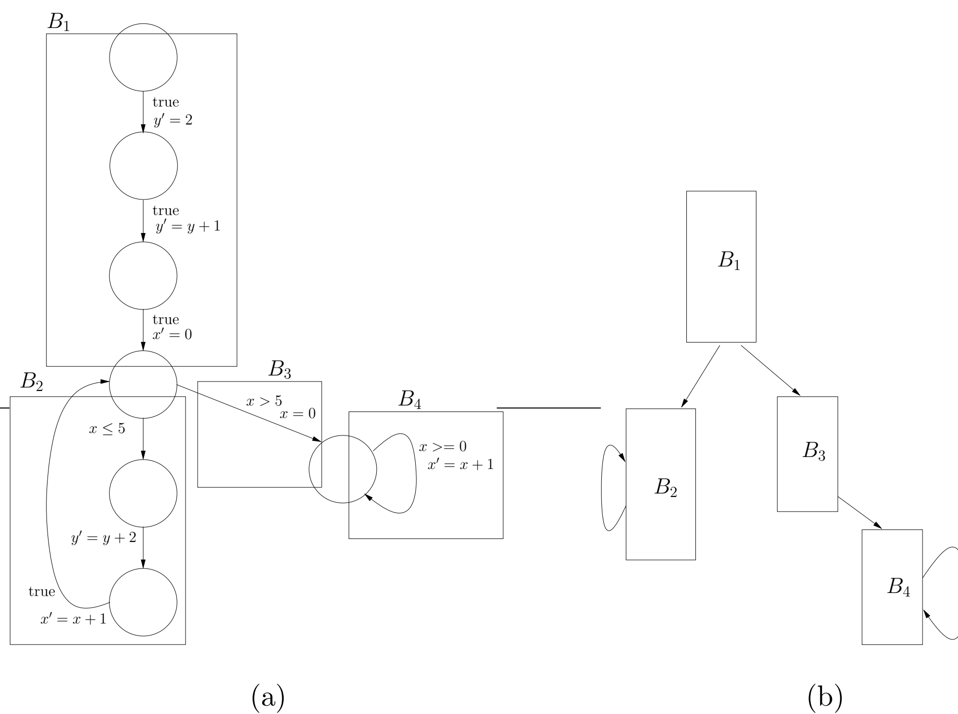
Control Flow Graph The nodes of a control flow graph (CFG) are basic blocks, i.e. blocks, that consist of a sequence of consecutive statements with no branching statements inside it. Basic Blocks have exactly one entry point and one exit point. In the case of transition systems a basic block is a path fragment where each location has exactly one incoming and one outgoing transition.

An edge of a CFG connects two basic blocks. There is an edge if the exit point of the source block is the entry point of the target block. That means, that the last location of the source block is also the first location of the target block. Confer Figure 5.1 for an example transition system and its CFG. The transition system consists of four basic blocks (B1,B2, B3, and B4).

Data Dependency Graph and Definition-Use-Chains The data dependency graph and the definition-use-chains describe the data flow in a program or in our case in a transition system.

The nodes of a dependency graph are statements. There is an edge between two nodes if there is a true dependence, an anti dependence, an output dependence, or an observable dependence between these two statements. Together with the CFG, the dependency graph is used to check whether two statements can be reordered and if $\mathcal{T}_{reordS}$ is a valid transformation.

For checking dependences between statements that are not direct neighbors, definition-use-chains (du-chains) are used. Confer also [ASU86]. du-chains are used to determine, whether $\mathcal{T}_{rev}$, $\mathcal{T}_{fus}$, $\mathcal{T}_{dst}$, and $\mathcal{T}_{reordL}$ are valid transformations.



(a) (b)

Figure 5.1: (a) Example transition system (b) CFG of example transition system.

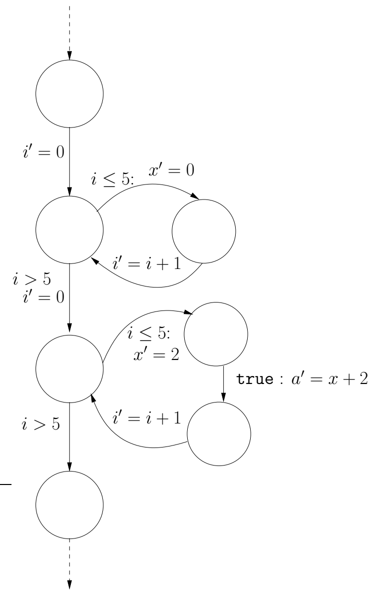


Figure 5.2: Example: Definition-Use-Chain

Definition 5.2. *A variable is* **used** *at a transition τ if its value is read by the statement present at τ.*

A variable is **defined** *at a transition τ if it is assigned a value by the statement present at τ.*

For example, $a = b + c$ uses variables b and c and defines variable a.

A definition of a variable x can be redefined (i.e. x is defined more than one time) on a path in a transition system. A statement that uses x should be only dependent on its last definition, i.e. the very strict definition of reordable statements (chapter 3) is loosen a bit for the purposes of the implementation. Consider for an example the transition system of Figure 5.2. Statement $a' = x + 2$ is only dependent on statement $x' = 2$ since the definition of x in statement $x' = 0$ is redefined. Thus, for example, loop fusion would be a valid transformation, although $x' = 0$ and $a' = x + 2$ are dependent.

5.1.3 Transformations

The compiler transformations given by their parameters are implemented in class `Transformation`. This class does not check if the transformation which should be applied is valid, i.e. this has to be checked before calling this class.

When applying a transformation, the transition system is transformed but also the respective ATS has to be adjusted. Both tasks are handled by this class. The implementation of the application of a transformation on a

transition system is one-to-one to the definitions given in chapter 3. After applying the transformation, the transformed transition system as well as the transformed ATS are returned.

5.2 Putting it All Together

The just presented components are composed for the implementation of the compiler as well as for the implementation of the algorithm. Thus, we are now able to show the framework of both, the compiler and the implementation of the algorithm. But at first, valid input transition systems are specified.

5.2.1 The Input

The input for the compiler as well as the input for the algorithm are transition systems that represent a reactive system. Valid inputs for both are specified in the following. These restrictions on input transition systems are only made because of keeping the dependency analysis of the example compiler and the algorithm simple. Note that the correctness of the algorithm is independent of the dependency analysis and therefore the dependency analysis can be replaced easily by a more complex one.

- Only one process per reactive system is permitted.
- Only integer variables are permitted.
- There is only one initial location.
- At most there are two outgoing edges per location.
- Branches are exclusive and one transition can always be taken.
- Loop guards have the form: $x \leq e | x < e | x > e | x \geq e | x == e | x \neq e$, where e is an expression and x is the loop index variable.
- If a start loc of a loop has several input edges, they all initialize x with the same value. x has to be initialized with a constant.
- The update of index variable x is always done on the last transition that goes back to the start loc of the loop. During the loop body execution x is only used.
- The update of x has the form: $x = e$, where e is an expression that contains no other variables than x.

5.2.2 Normalform of Transition Systems

The goal of the algorithm is to establish a trivial refinement mapping, i.e. a mapping that maps states and variables one-to-one. Therefore the input transition systems have to be transformed into a normalform:

- Each transitions updates only one variable: Transitions that update more than one variable are split, i.e. new locations and transitions are introduced. The names of the introduced locations start with "F". Confer Figure 5.3.
- There is no data transformation if the guard is not `true` or `false`: Transitions where the guard is not `true` or `false` and there is a data transformation are split, i.e. new locations and transitions are introduced. The names of the introduced locations start with "G". Confer Figure 5.3.
- All guards are in normal form: A guard g is in normal form if $g ::= x \leq e \mid x \geq e \mid x == e \mid x! = e$, where x is the loop index variable and e is an expression.
- The loop index variable is always initialized directly before the loop is entered
- The loop index variable is always updated on the last transition that leads to the starting location of the loop

In Figure 5.3 is presented an example transition system with its normal form.

5.2.3 Implementation of the Compiler

The compiler gets as input a transition system (the so-called source transition system) that should be compiled, i.e. arbitrary many transformations (of the specified compiler transformations) should be applied in arbitrary order. Afterwards, the compiled version of the source transition system (the so-called target transition system) is returned.

The compiler works as follows. First a random number d is computed, that gives the number of transformations that should be applied to the input transition system T, i.e. d-times the main loop is executed which performs the following steps.

1. Compute all valid transformations for T: This is done by invoking the Dependency Analysis class.
2. Pick randomly one transformation of the set of valid transformations
3. Apply this transformation to T: Class `Transformation` is called.

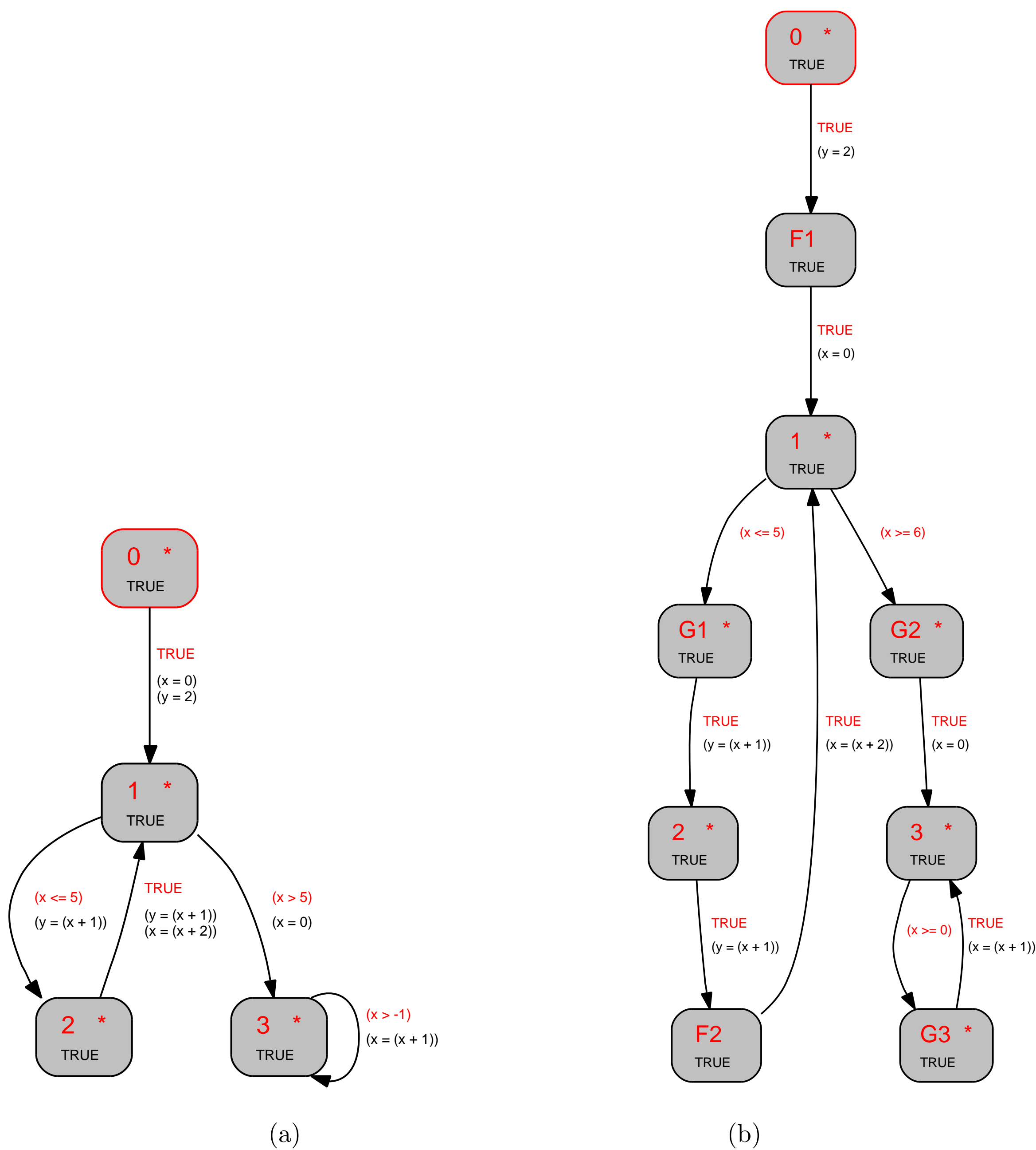


Figure 5.3: (a) Example transition system (b) Example transition system in Normalform.

The compiler can be found in the VRP. It can be executed with the following options

```
./compiler   -a sourceTS.xml
             [ -c targetTS.xml ]
             [ -d number_of_transformations ]
             [ -h ]
```

The option `-a` is essential and has to be set, since this is the transition system (in xml-format) that should be compiled. The other options are optional. `-c` specifies the outputfile, which is overwritten if it already exists. If `-c` is not specified the target transition system is written by default to `out.xml`. If option `-d` with an integer number is set, the number of transformations the compiler applies is not chosen randomly. Instead d is set to *number_of_transformations*. Option `-h` returns the help message.

5.2.4 Implementation of the Algorithm

The algorithm with all its optimizations is also implemented in the VRP. The implementation is analog to the pseudo-code of chapter 4. To keep the memory usage small, the BFS tree does not store all transition systems. Instead, only the transformation chains are stored from which the A_i's can be reconstructed. Thus, as data structure for the BFS tree a two-dimensional vector is chosen that contains all computed transformation chains.

Furthermore, the computation of *maxdepth* is not needed by the implementation, since rules 4-8 preserve that no infinite transformation chains can occur. Thus, all produced transformation chains are finite and at some point they were all checked and the algorithm terminates.

The algorithm can be executed with the following options

```
./refinementopt   -a sourceTS.xml
                  -c targetTS.xml
                  --prover provername
                  [--opt ]
                  [ -h ]
```

As for the compiler `-a` specifies the source transition system in xml-format and `-c` (which is essential for the TVopt tool) specifies the target transition system, also in xml-format. The option `--prover` specifies the prover that should be used to prove the conditions established by the refinement task. Option `--opt` enables the optimizations, i.e. rules 1-3, and option `-h` returns the help message.

If the target transition system is a correct translation of the source transition system, the tool outputs the computed transformation chain, the number of produced transition systems (the number of the A_n's), and a proof for C ref A_n. The refinement mapping, i.e. the transformed source transition system annotated with all invariants needed for the refinement proof, can be found in file `_invariants.xml`.

Chapter 6

Experiments

In this chapter some running examples and experiments that show termination and performance of the algorithm are presented. The source files of the examples and their respective graphical representation can be found in the testsuite of the VRP.

6.1 A First Example

The first example system "Multiplication" is depicted in Figure 6.1 (a). It computes the product of two integers by using addition. While translating this system, three optimizations were applied: $\mathcal{T}_{rev}$ and two times $\mathcal{T}_{reordS}$. The target system is depicted in Figure 6.1 (b).

The TVopt tool is called with

```
./refinementopt  -a multiplication.xml
                 -c multiplication_concrete_rev_reord_reord.xml
                 --prover yices
```

It outputs the transformation chain $(\mathcal{T}_{rev}, L4|_{_})$; $(\mathcal{T}_{reordS}, F1, F3)$; $(\mathcal{T}_{reordS}, F3, F4)$. The transformed source system annotated with all invariants is shown in Figure 6.2. It can easily be seen that this system simulates the target system.

6.2 Transformation Tests

For each transformation there is a correctness test. Here only the test for $\mathcal{T}_{fus}$ is presented. The other test files can be found in the testsuite as well.

For this test, the compiler has applied only one transformation, namely the transformation that should be tested, to the source program. By these tests it can be checked that the TVopt tool detects all transformations and outputs exactly the transformation chain consisting of one transformation.

The input systems for the test of $\mathcal{T}_{fus}$ are presented in figure 6.3.

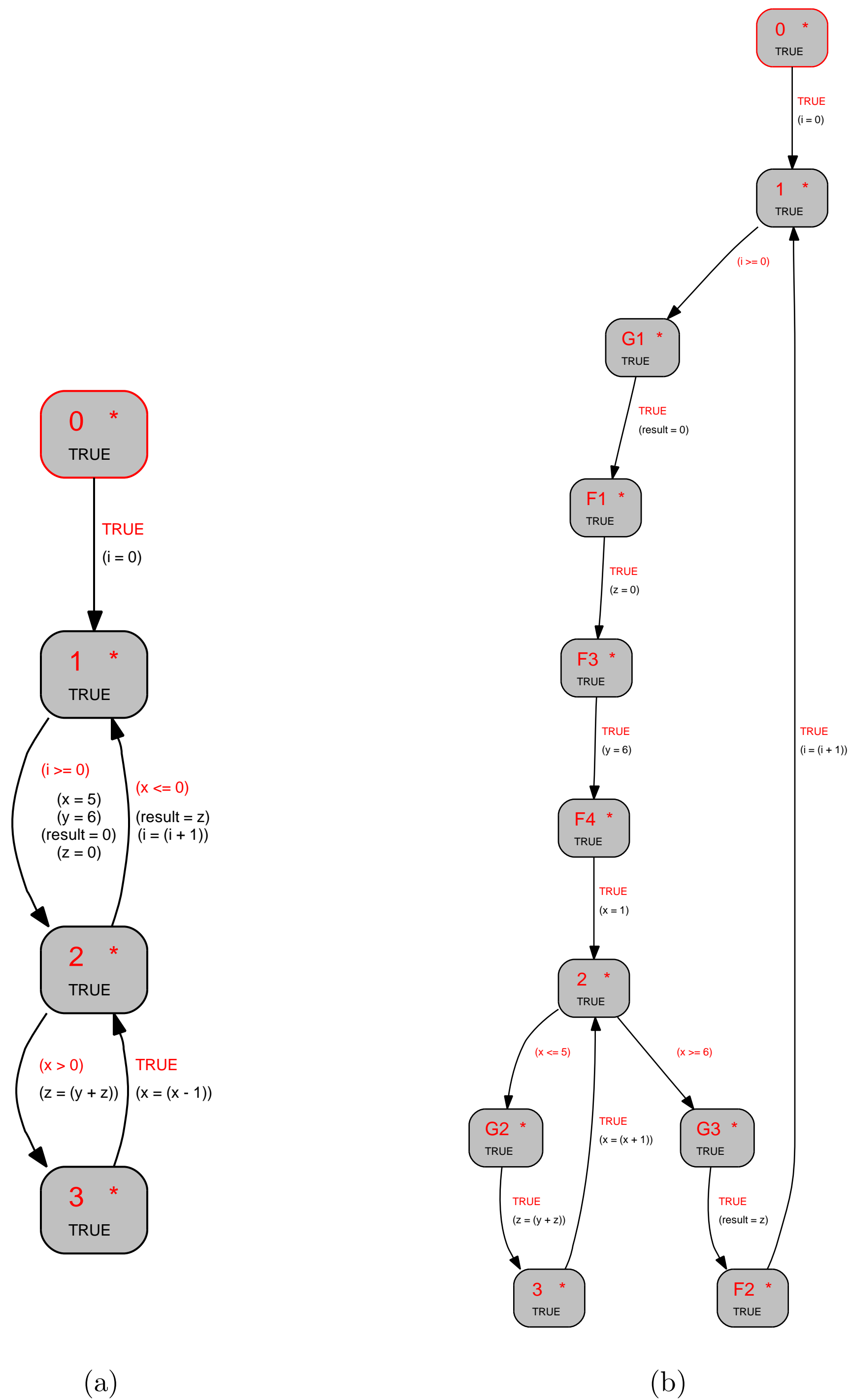


Figure 6.1: (a) Example system "Multiplication" (b) Loop reversal and two times reordering of statements were applied to the source system.

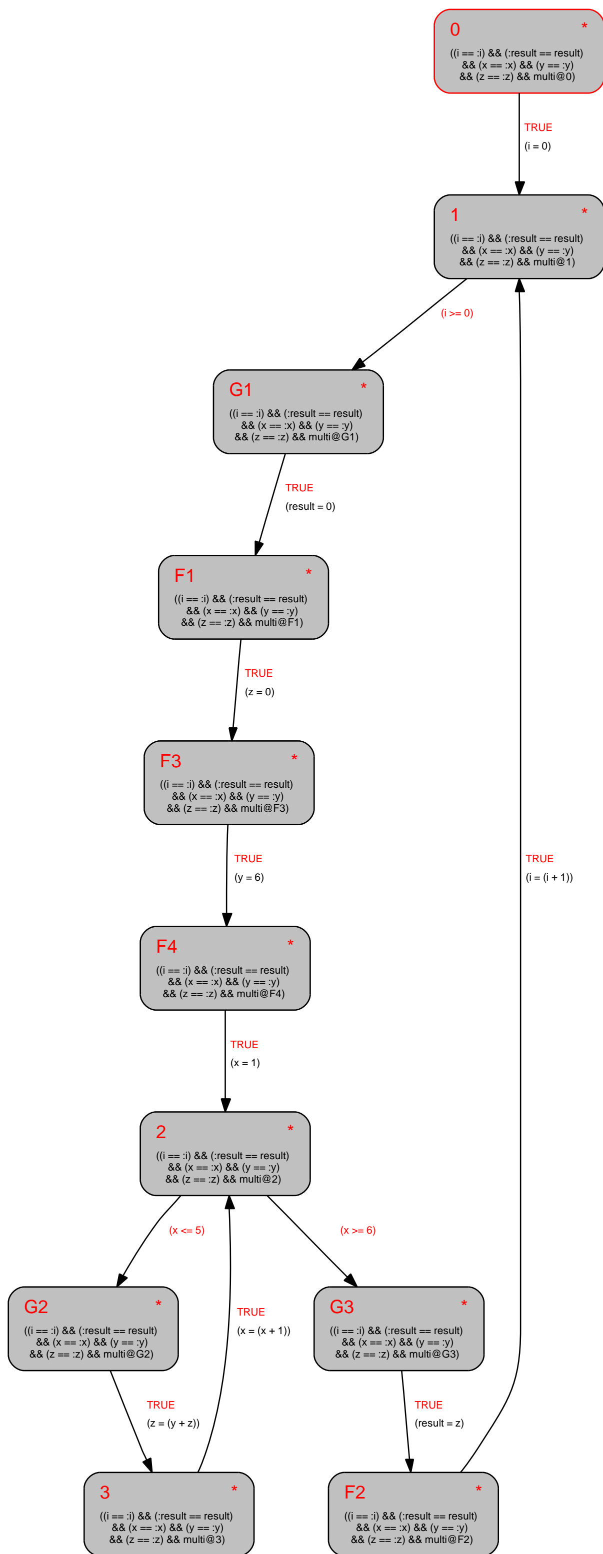


Figure 6.2: Transformed system “Multiplication” with annotated invariants.

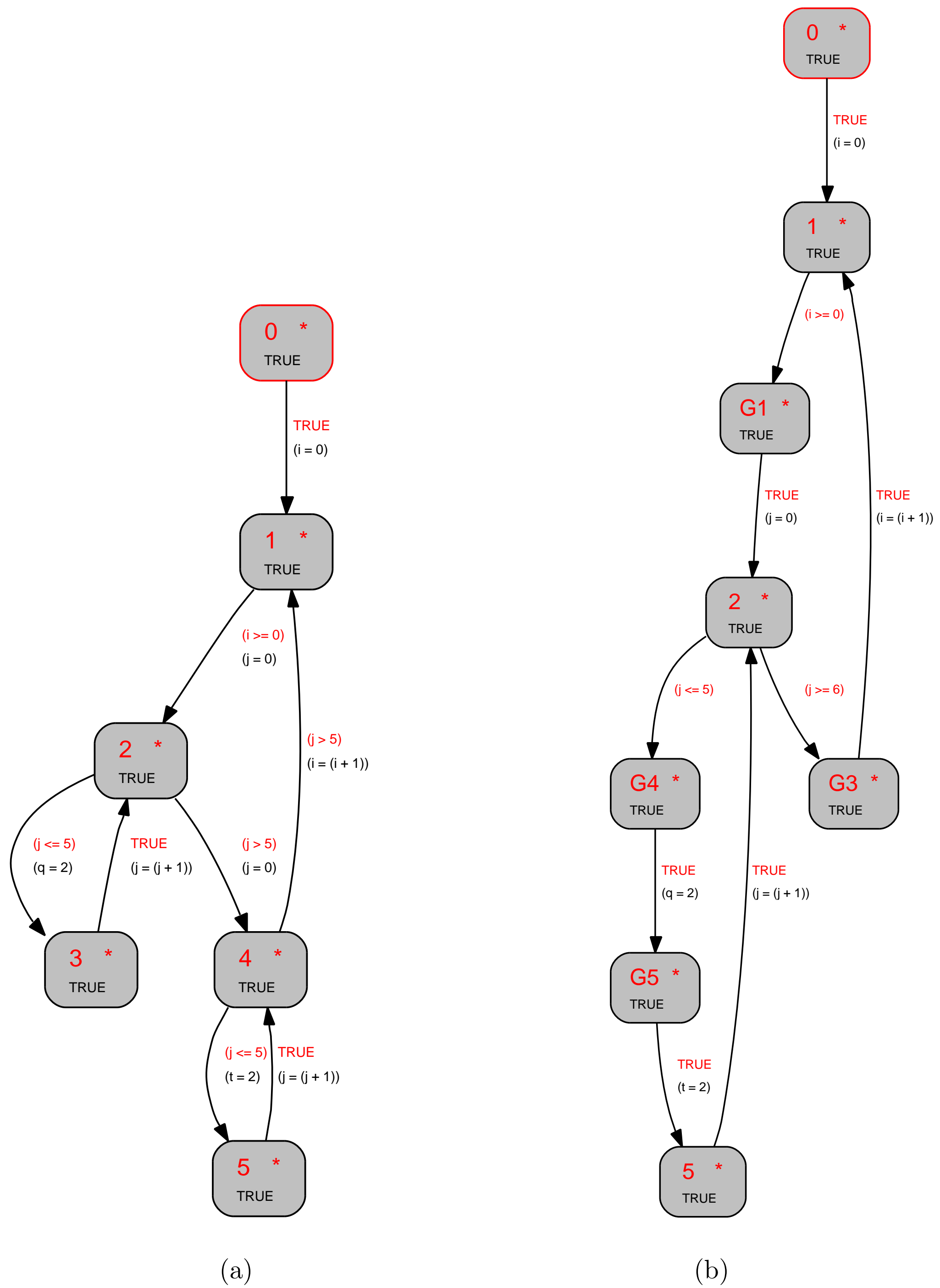


Figure 6.3: (a) Source system for loop fusion test (b) Target system.

The TVopt tool is called with

`./refinementopt` `-a` *twoloops.xml*
`-c` *twoloops_concrete_fus.xml*
`--prover` *yices*

Indeed the algorithm outputs the correct transformation chain $(\mathcal{T}_{fus}, L3|_{_}, L4|_{_})$ consisting of just one transformation.

6.3 Termination

For testing termination of the algorithm in the case that the target system is not a correct translation of the source system, consider the input systems in Figure 6.4. The source transition system can not be transformed to the target transition system by any transformation chain, since statement $t = 3$ is lost somewhere in the compilation process. The TVopt tool is called with

`./refinementopt` `-a` *testfile1.xml*
`-c` *testfile1_concrete_terminationfail.xml*
`--prover` *yices*

The algorithm does not find a transformation chain and outputs `false`.

6.4 Optimizations

The goal of the optimizations was to keep the size of the BFS tree small. The here presented tests will show what can be gained from the implemented optimizations.

For this test the TVopt tool is executed twice for each pair of input systems. The second time the `--opt` option is enabled. For all executions the prover `yices` is chosen.

Recall that the transformation chains produced by the tool without optimizations and the tool with enabled optimizations can be of different length. This is due to the fact that the optimizations remove paths in the BFS tree. Thus, in order to compare the sizes of the BFS trees, the produced transformation chains of both executions have to be the same.

Tabular 6.1 shows the results of this test. The first two columns contain the name of the input transition systems (TS). Their source files can be found also in the testsuite of the VRP. The third column shows the number of transition systems produced (i.e. the number of transformation chains that were checked) until the target system was found when the optimizations were not enabled. The fourth column shows the same for enabled optimizations. The fifth and the sixth column show the number of transformation chains that were produced so far but that were not yet checked.

As expected, the number of checked transformation chains until C is found gets less if optimizations are enabled. Furthermore, the size of the BFS tree is reduced. Thus, in the case that C is not a correct translation of

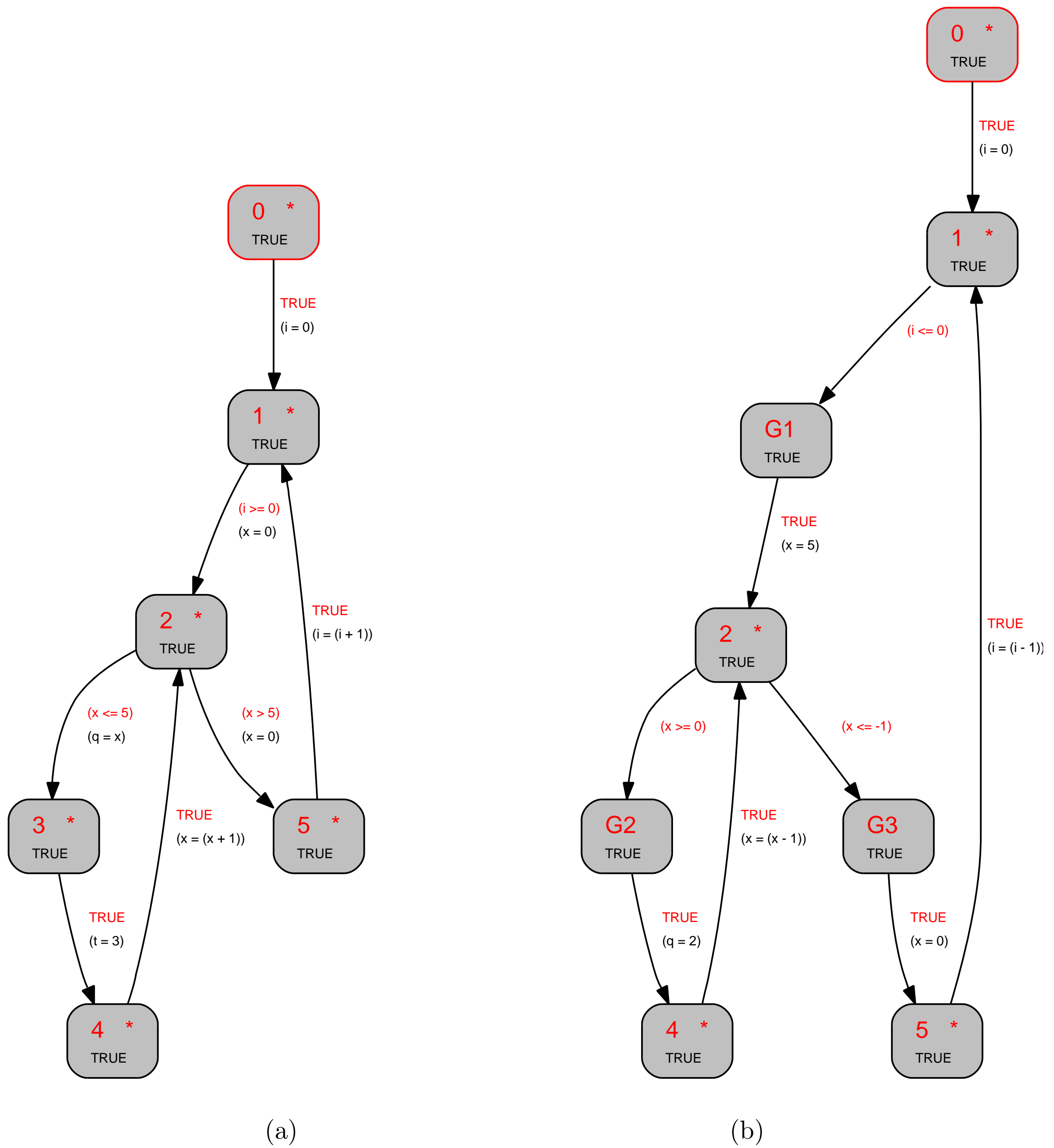


Figure 6.4: (a) Source system for termination test (b) Incorrect translation of source system.

Source TS	Target TS	checked TC	`--opt`	unprocessed TC	`--opt`
twoloops	twoloops_concrete_3	15	14	16	11
opttest	opttest_concrete_3a	62	59	295	253
opttest	opttest_concrete_3b	88	82	388	323
opttest	opttest_concrete_4	752	592	2478	1684

Table 6.1: Results of optimization test.

A, less transformation chains have to be checked. Note, the larger the BFS tree gets, the more optimizations are applied.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have presented an algorithm that takes as input two transition systems and returns a proof if the target transition system is a correct translation of the source transition system. Otherwise it outputs `false`. To define "correct translation", the notion of refinement was introduced, i.e. the target transition system C is a correct translation of the source transition system A if a refinement mapping between C and A can be established.

As we have seen, finding such a refinement mapping is often not obvious, since the target transition system and the source transition system often have completely different structures and it can not be determined which part of the target transition system is mapped to which part of the source transition system. This is due to the fact that a compiler applies several optimization methods while translating the source transition system. The focus of this thesis was set on structure-modifying transformations such as loop fusion, loop distribution, loop interchange, loop reversal, etc. These transformations were defined formally and afterwards they were classified. This classification was made by analyzing the effect of the transformations on a transition system and the effect the transformations have on each other. For example, two transformations can erase each other, or they are commutative.

The presented algorithm was based on breadth first search (BFS). It searches exhaustively for a transformation chain that transforms the source transition system into the target transition system. If such a transformation chain exists, the trivial refinement mapping can be established, i.e. the locations and variables of the target system can be mapped one-to-one to the locations and variables, respectively, of the source system. The algorithm terminates if the target transition system is found or if it can be excluded that there exists a refinement mapping. This algorithm was shown to be correct and terminating for a restricted set of transformations. To deter-

mine by which transformations this set can be extended without violating correctness or termination, some properties the transformation has to fulfill were given. We have seen, that for example loop unrolling violates these properties. Thus, the algorithm can only be extended by loop unrolling if a metric was used. For loop unrolling such a metric was presented.

To reduce the search space, some optimizations were introduced that delete infinite paths in the BFS tree or subtrees that occur twice. These optimizations were based on the classification of the compiler transformations. It was shown that the presented optimizations preserve the correctness and termination criteria of the algorithm.

7.2 Future Work

There are many directions for future work. We list a few as follows:

- The set of the transformations can be extended:
  There exist many other transformations a compiler applies while translating a source program. The algorithm can be extended by these transformations. Furthermore, the assumed compiler can be a parallelizing compiler.

- More optimizations can be established:
  The optimizations that remove duplicated subtrees can be extended.

- Metrics can be developed:
  Some metrics can be developed such that it can be determined earlier in a transformation chain, that this transformation chain can not lead to the target transition system. This would reduce the search space.

- The set of valid input transition systems can be extended:
  In this thesis we have only considered systems with one process. The approach can be extended to systems with more than one process. Furthermore, the data structures for variables can be extended to arrays, pointers, etc.

Appendix A

Implementation: Class Hierarchy

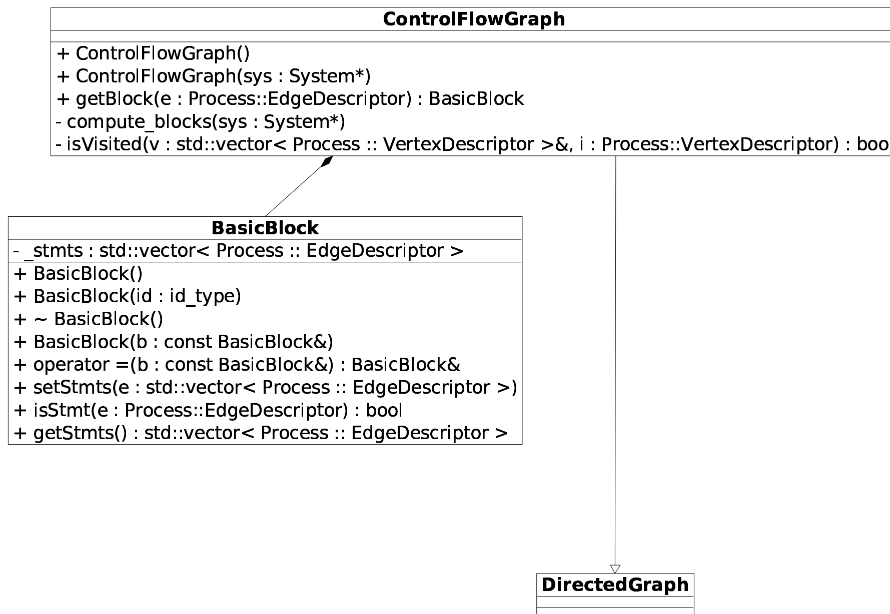


Figure A.1: Class hierarchy of class `ControlFlowGraph`

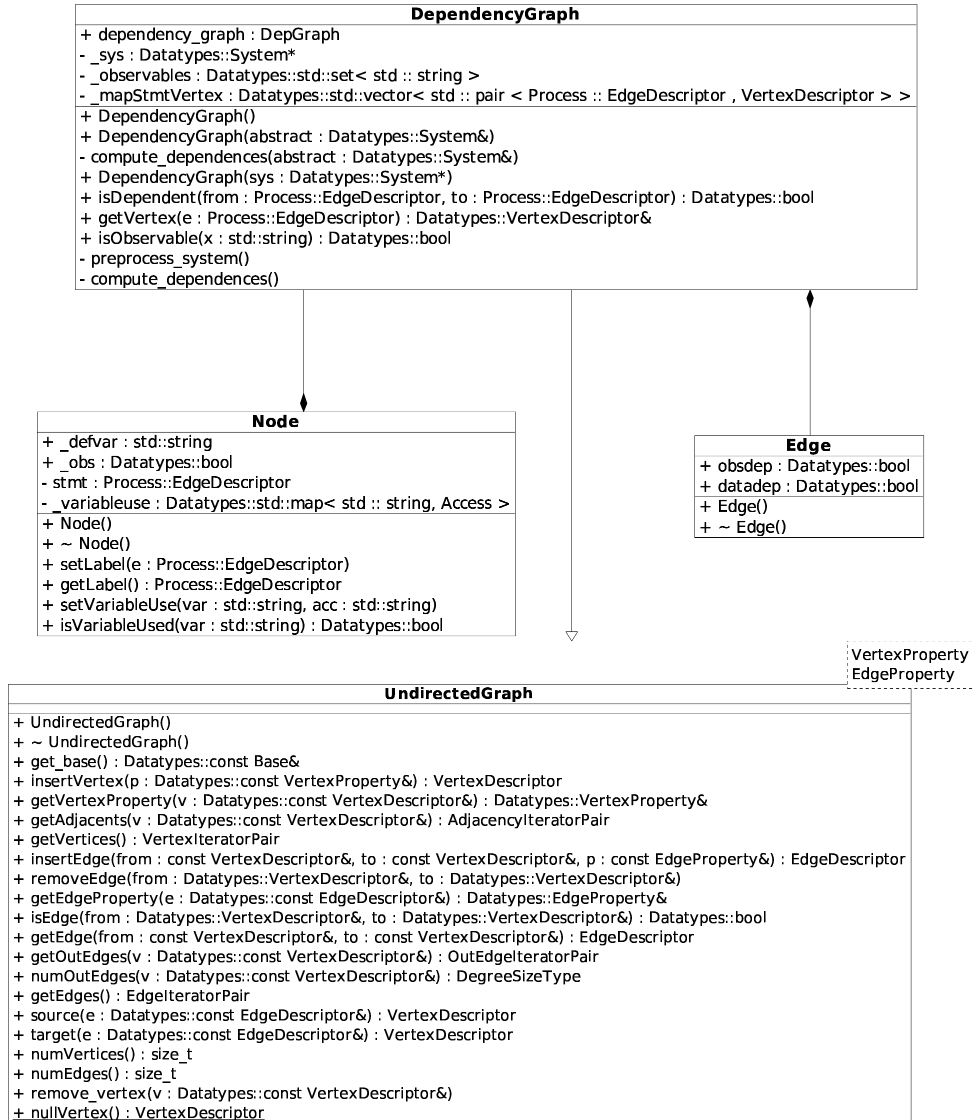


Figure A.2: Class hierarchy of class `DependencyGraph`

Figure A.3: Class hierarchy of class `Transformation`

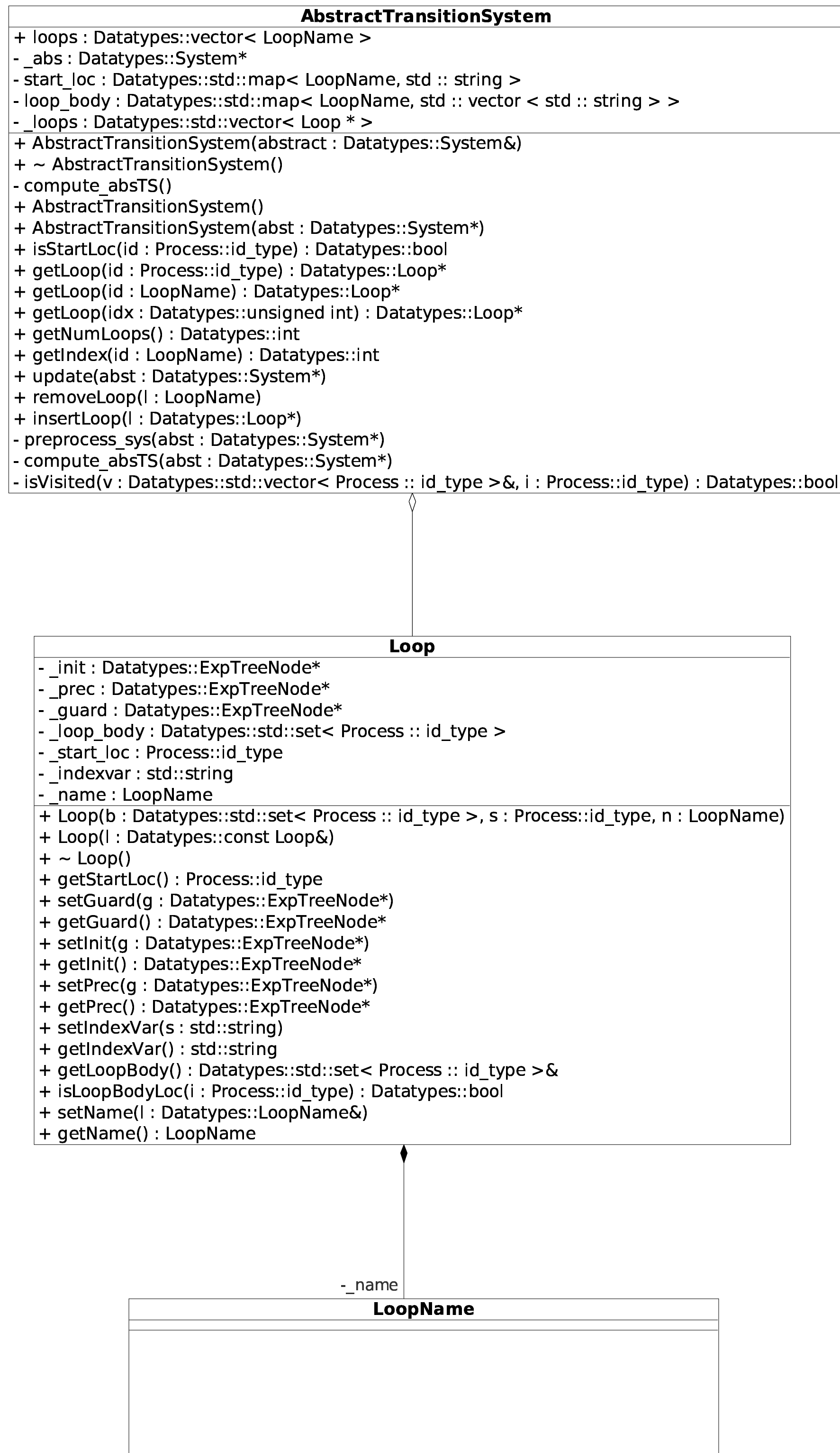


Figure A.4: Class hierarchy of class `AbstractTransitionSystem`

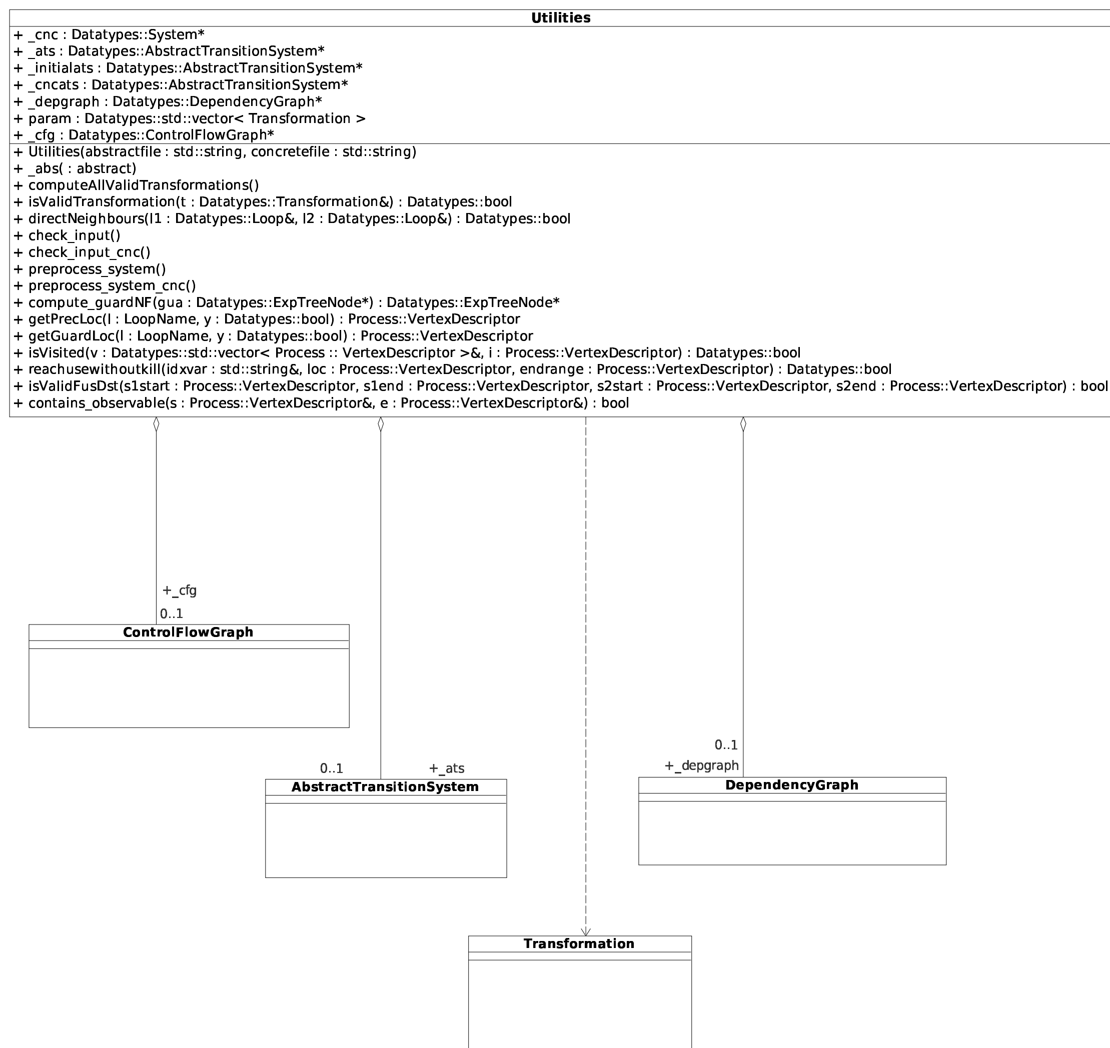


Figure A.5: Class hierarchy of class `Utilities`

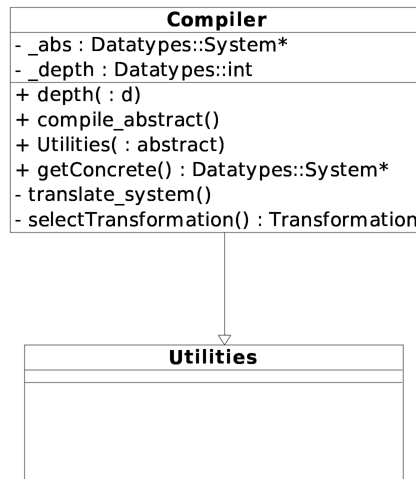


Figure A.6: Class hierarchy of class `Compiler`

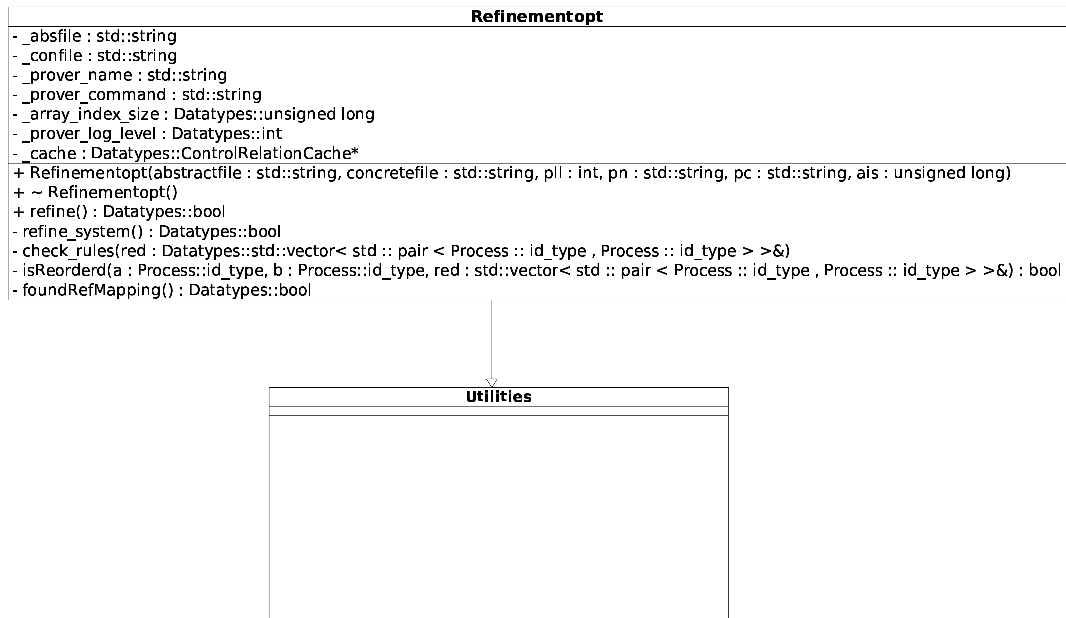


Figure A.7: Class hierarchy of class `Refinementopt` (Implementation of the algorithm)

Bibliography

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[BN00] Franz Baader and Tobias Nipkow. Term rewriting and all that. *SIGACT News*, 31(3):24–26, 2000. Reviewer-Paliath Narendran.

[Fan05] Yi Fang. *Translation Validation of Optimizing Compilers*. PhD thesis, New York University, 2005.

[GE99] Ralf Hartmut Güting and Martin Erwig. *Übersetzerbau - Techniken, Werkzeuge, Anwendungen*. Springer, 1999.

[GZB05] Benjamin Goldberg, Lenore D. Zuck, and Clark W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electr. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, USA, 1995.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[OW96] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Lehrbuch. Spektrum Akad. Verl., Heidelberg - Berlin - Oxford, 3., überarb. aufl. edition, 1996.

[PSS98a] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation for synchronous languages. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages*

and Programming, pages 235–246, London, UK, 1998. Springer-Verlag.

[PSS98b] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS* [PSS98b], pages 151–166.

[PZ06] Amir Pnueli and Ganna Zaks. Translation validation of interprocedural optimizations. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006)*. Computing Research Repository (CoRR), August 2006.

[Wir95] Niklaus Wirth. *Algorithmen und Datenstrukturen*. Leitfäden und Monographien der Informatik. Teubner, Stuttgart, 4., durchges. aufl. edition, 1995.

[WM] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau*.

[ZPG+05] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, 2005.