Bachelorarbeit
im Studiengang "Angewandte Informatik"

Intraprocedural Program Analysis via Model Checking

Anna-Lena Lamprecht

am Lehrstuhl für
Service Engineering für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen
20. September 2005
Georg-August-Universität Göttingen
Zentrum für Informatik
Lotzestraße 16-18
37083 Göttingen
Germany

Tel.   +49 (551) 39-144 14
Fax   +49 (551) 39-144 15
Email office@informatik.uni-goettingen.de
WWW   www.informatik.uni-goettingen.de
Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 20. September 2005
Bachelorarbeit

Intraprocedural Program Analysis
via Model Checking

Anna-Lena Lamprecht

20. September 2005

Betreut durch Prof. Dr. Margaria-Steffen
Lehrstuhl für Service Engineering für Verteilte Systeme
Georg-August-Universität Göttingen
Contents

1 Introduction 4

2 Data-Flow Analysis 6
   2.1 Fundamental Principles .................................. 6
   2.2 Four Basic Data Flow Analysis Problems ................. 8
   2.3 Solving Data Flow Equations .............................. 11
   2.4 Derived Data Flow Information and Further Problems ... 13

3 Model Checking 19
   3.1 Modeling .................................................. 19
   3.2 Specification ............................................. 22
   3.3 Verification .............................................. 26

4 Data-Flow Analysis via Model Checking 30
   4.1 From Programs to Program Models ......................... 31
   4.2 From DFA Equations to Modal Specifications .............. 32
   4.3 From DFA Algorithms to Model Checking .................. 35

5 The jABC Framework 36
   5.1 Principles ................................................. 36
   5.2 The Model-Checking Plugin ................................ 36
   5.3 The UnitGraph2SibGraph Plugin ........................... 40

6 Example 43
   6.1 Program: Uglified Fibonacci Number Computing .......... 43
   6.2 Program Model and Basic Properties ....................... 47
   6.3 Analyses and Results ..................................... 49
   6.4 Derived Optimizations ..................................... 56

7 Further Results 58
   7.1 Router Simulation .......................................... 58
   7.2 Strassen Algorithm: Fast Matrix Multiplication .......... 59
   7.3 Some Algorithms on Strings ............................... 59

8 Conclusion 61
Contents

Bibliography 64
List of Figures 68
1 Introduction

Program analysis, or more precisely data-flow analysis [1, 20, 39], concerns static compile-time techniques for computing reliable approximate information about the dynamic behaviour of programs.

A main application is in compilers (figure 1.1). The frontend translates the source program into an intermediate representation, and the backend translates the intermediate representation into the actual target program. While in the intermediate representation, code optimizations can be applied — data-flow analysis is a fundamental prerequisite for improvements possibly made in the optimization phase. Code may be generated that avoids redundant computations, e.g. by reusing available results or by moving loop invariant computations out of loops, or avoids superfluous computations, e.g. of results known to be not needed or of results known already at compile-time.

![Compilation steps](image)

Figure 1.1: Compilation steps

Among the more recent applications are the validation of software (to reduce the likelihood of malicious or unintended behaviour), transformations between data representations (for solving problems such as the Y2K problem), and also the documentation, debugging, testing and annotation of programs [35, 39].

In general, data-flow analysis is preceded by control-flow analysis. Control-flow analysis is the encoding of pertinent, possible program control-flow structure or flow of control, usually in the form of one or more graphs. A common representation for the calling relationships among the methods, i.e. the interprocedural flow of a program, is a directed graph named a call graph, which shows "what may directly call what". The potential flow of control of each procedure, i.e. the intraprocedural flow, is typically represented by a directed graph called a control-flow graph (CFG) or simply a flow graph, which depicts all
possible execution paths. The construction, representation, structure, and properties of such graphs are part of control-flow analysis.

Data-flow analysis is then the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or various attributes of variables) in a computer program. It is customary to think of a program as a graph (that may have been created by control-flow analysis).

*Model Checking* [5, 7, 8, 37] is a method for the formal verification of systems. This is achieved by checking if a model (usually a graph structure deriving from a software or hardware design) satisfies a formal specification (usually some modal logic formula\(^1\)).

Model checking is not, however, restricted to verification. As [41, 42, 43, 45] have revealed, model checkers can be used for data-flow analysis by creating the model from a program and encoding a data-flow analysis problem into a modal specification. The striking advantage of this approach is, that it is not necessary to implement a new algorithm for each problem, as it is done in classical data-flow analysis, but only to provide an appropriate modal specification for the model checker.

Thus, the objective of this thesis was to perform intraprocedural data-flow analyses via model checking (*DFA-MC*) using the jABC Framework [38] and following the purpose of receiving information useful for code improvement. In order to achieve this, the first step was to acquire the principles of data-flow analysis, model checking and after all *DFA-MC*, which are revealed in chapters 2, 3, and 4, respectively. While exploring the framework further, it turned out that some functionalities needed for the process of *DFA-MC* were not yet provided. These features were implemented soon, however, by the jABC-developers at the University of Dortmund. The complete framework is described in chapter 5. Finally, after building up a collection of Java programs, several analyses could be performed. The results for a selected program are given in detail in chapter 6, further programs and results are discussed in chapter 7. The last chapter (chapter 8) of the thesis draws a conclusion and presents ideas for future work.

---

\(^1\)In modal logics [15, 29, 47], the truth of a formula is relative to a state, e.g., current time or current store contents.
2 Data-Flow Analysis

The aim of data-flow analysis (DFA) is the transmission of useful relationships from all parts of the program to places where the information can be of use. Based on [1, 2, 20, 39], this chapter deals with the information-gathering techniques needed to implement safely code optimizations. The important point is that flow analysis collects information useful for or prerequisite to code improvement. It does not accomplish the complete code improvement.

The techniques that are introduced here are developed for performing a global data-flow analysis [24] in a single method of a program that is given in three-address-code. ¹ After introducing some basic notions of data-flow analysis, this chapter discusses four basic DFA problems and the solving of DFA equations. Finally, further problems and derived information are described.

2.1 Fundamental Principles

This section discusses basic concepts needed for intraprocedural data-flow analysis, i.e. the representation of a procedure by a control-flow graph and basic properties of program statements. As data-flow analysis assumes that a control-flow graph with some properties is already available, the process of creating such a graph is not discussed.

Program Representations

It is useful to represent the program by a directed graph called a control-flow graph, or CFG. Having a program in three-address-representation, every statement $s$ of the program is regarded as a separate unit and thus forms a node for the graph.

Note that, although this is done in many DFA applications, it is not necessary to group the nodes into basic blocks. Basic blocks are sequences of consecutive statements which may be entered only at the beginning, and when entered are executed in sequence without halt or possibility of branch (except at the end of the basic block).

Basic blocks are useful when the number of nodes in a CFG is supposed to be small. Single-statement nodes, on the other hand, have the advantage that several program properties of interest are easier to define and to be computed for single statements than for a sequence of statements in a basic block.

¹Three-address code, or three-address representation, is often also referred to as quadruples. It describes statements consisting of one operator, at most one result, and at most two arguments.
A CFG represents all alternatives of control flow by portraying the nodes and their successor relationships. One node is distinguished as initial; it is the node representing the first statement of the whole program. There is a directed edge from node $n_1$ to block $n_2$ if $n_2$ could immediately follow $n_1$ during execution, that is, if

1. there is a conditional or unconditional jump from $n_1$ to $n_2$, or

2. $n_2$ immediately follows $n_1$ in the order of the program, and $n_1$ does not end in an unconditional jump.

Thus, $n_1$ is called a predecessor of $n_2$, and $n_2$ a successor of $n_1$.

Figure 2.1 shows a Java-like piece of code (computing the sum of the numbers 1 through 10) and its representation by a control flow graph. Note that an edge of the CFG from node $n$ to node $n'$ does not tell whether the conditional jump at the end of $n$ (if there is a conditional jump) goes to $n'$ if the condition is satisfied or if the condition is not satisfied.

![CFG example](image)

**Basic Properties**

The terms point and node are used to refer to single intermediate-language statements, and will be used synonymously. Whether the moment before or after the execution of the statement is meant, depends on the context.

Three basic properties can be defined on the structure of statements:

- A use of a variable $A$ or an expression $XopY$ is any occurrence of $A$ or $XopY$ as an operand. For each node $n$, the set $USE(n)$ contains those variables and expressions
that are used at \( n \).

- A **definition** of a variable \( A \) is a statement that can (potentially) modify the value of \( A \), such as an assignment statement. For each node \( n \), the set \( \text{DEF}(n) \) contains those variables that are defined at \( n \).

- A **modification** of an expression \( X \text{op} Y \) is a definition of either \( X \) or \( Y \). For each node \( n \), the set \( \text{MOD}(n) \) contains those expressions that are modified at \( n \).

\( \text{DEF} \), \( \text{USE} \) and \( \text{MOD} \) are very basic properties that can be computed by simply parsing the statement, or by determining whether operands are contained in the \( \text{DEF} \)-set, so it is assumed here that they are already available for each node in the CFG. Having these three properties available for each node in the CFG, several basic flow analyses are possible.

### 2.2 Four Basic Data Flow Analysis Problems

Four basic data flow analysis problems, which are standard in literature on program analysis, are introduced in this section: **Live Variables**, **Very Busy Expressions**, **Available Expressions**, and **Reaching Definitions**. They form the foundation for further investigations about the program behaviour (to be discussed in section 2.4).

#### Live Variables

**Live variable analysis** determines for a variable \( A \) and point \( p \) whether the value of \( A \) at \( p \) could be used along some path in the flow graph starting at \( p \). If so, \( A \) is live at \( p \); otherwise \( A \) is dead at \( p \). Consider figure 2.2 for an example. The set of live variables at line 2 is \( \{ b, c \} \), but the set of live variables at line 1 is only \( \{ b \} \) since variable \( c \) is assigned in line 2.

```
1: b = 3;
2: c = 5;
3: a = b + c;
```

*Figure 2.2: Live variables example*

The set of all variables that are live at program point \( p \) can be determined by the follow-

\(^2\text{Note that expressions don't contain more than two operands, as the CFGs are generated from three-address-representations.}\)


\[ LV(p) = \bigcup_{p' \in \text{succ}(p)} (LV(p') \setminus \text{DEF}(p') \cup \text{USE}(p')) \]

\[ = \bigcup_{p' \in \text{succ}(p)} (LV(p') \cap \neg \text{DEF}(p') \cup \text{USE}(p')) \quad (2.1) \]

where \( \text{USE}(p) \subseteq \text{USE}(p) \) denotes the set of all variables that are used in the statement at \( p \), and \( \text{succ}(p) \) is the set of all successors of \( p \). Note that, if \( \text{succ}(p) = \emptyset \), then \( LV(p) = \emptyset \).

This analysis might be used as the basis for Dead Assignment Elimination: if a variable \( A \) is dead immediately after an assignment to \( A \), this assignment can be eliminated. Another important use for live variable information comes when object code is generated: after a value is computed in a register, and presumably used within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if all registers are full and another register is needed, using a register with a dead value should be favored since that value does not have to be stored.

**Very Busy Expressions**

An expression \( X \circ Y \) is *very busy* at point \( p \) if along every path from \( p \) control comes to a computation of \( X \circ Y \) before any definition of \( X \) or \( Y \). Consider figure 2.3 for an example. The expression \( a + b \) is very busy at line 4, the expression \( a \ast b \) is very busy at lines 4 and 5.

\[
\begin{align*}
1: & \quad a = 3; \\
2: & \quad b = 2; \\
3: & \quad a = 1; \\
4: & \quad c = a + b; \\
5: & \quad d = a \ast b; \\
6: & \quad a = 1;
\end{align*}
\]

*Figure 2.3: Very busy expressions example*

The set \( VBE \) of all expressions that are very busy at point \( p \) can be determined by the following equation \([1, 20]\):

\[ VBE(p) = \bigcup_{p' \in \text{succ}(p)} (VB(p') \cap \neg \text{MOD}(p') \cup \text{USE}(p')) \quad (2.2) \]

where \( \text{USE}(p) \subseteq \text{USE}(p) \) denotes the set of all expressions that are used in the statement at \( p \). Note that, if \( \text{succ}(p) = \emptyset \), then \( VBE(p) = \emptyset \).

\[ ^3 \text{Recall set theory: Let } A, B \text{ be subsets of some universe } U: A, B \subseteq U. \text{ Then: } A \setminus B = A \cap (U \setminus B) = A \cap \neg B \] (\( \neg B \) denoting the complement of \( B \)).
A possible optimization based on this information is to evaluate expressions at nodes where they are very busy and store their values for later use; this optimization is sometimes called hoisting an expression.

**Available Expressions**

An expression $XopY$ is available at a point $p$ if every path (not necessarily cycle-free) from the initial node to $p$ contains $XopY$, and after the last such occurrence prior to reaching $p$, there are no subsequent assignments to $X$ or $Y$ (i.e. the value of $XopY$ has not changed).

Consider figure 2.4. The set of available expressions for line 1 is empty, because no expression has been evaluated previously. At line 2, however, the expression $b + c$ is available, and the set of available expressions for line 3 is $\{b + c, e \ast f\}$.

\[
\begin{align*}
1: & \quad a = b + c; \\
2: & \quad d = e \ast f; \\
3: & \quad g = a + d;
\end{align*}
\]

*Figure 2.4: Available expressions example*

Let $s$ be the initial node. Then

\[
AE(s) = \emptyset
\]  \hspace{1cm} (2.3)

as no expressions are available at the beginning of the program. The set of expressions that are available at any point $p$, $p \neq s$, can be determined by the following equation [1, 20]:

\[
AE(p) = \bigcap_{p' \in \text{pred}(p)} ((AE(p') \setminus \text{MOD}(p')) \cup \text{GEN}(p'))
\]

\[
= \bigcap_{p' \in \text{pred}(p)} ((AE(p') \cap \text{notMOD}(p')) \cup \text{GEN}(p'))
\]  \hspace{1cm} (2.4)

where $\text{GEN}(p)$ is the set of expressions that is generated at $p$, i.e. those expressions $XopY$ that are used at $p$, and neither $X$ nor $Y$ is assigned at $p$. $\text{pred}(p)$ denotes the set of all predecessors of $p$.

This information can be used to avoid the re-computation of an expression (e.g. in Common Subexpression Elimination).

**Reaching Definitions**

A definition of a variable $A$ reaches a point $p$ if there is a path in the flow graph from that definition to $p$, such that no other definitions of $A$ appear on the path. Consider figure 2.5 for an example. The reaching definitions of variable $\hat{a}$ at line 7 are the elements of the set $\{2: \hat{a} = 5; 4: \hat{a} = 3;\}$.
2 Data-Flow Analysis

1: if b==4 then
2:     a = 5;
3: else
4:     a = 3;
5: endif
6:
7: if a < 4 then
8: ...

Figure 2.5: Reaching definitions example

The set of all definitions that reach a program point $p$ can be determined by the following equation [1, 20]:

$$RD(p) = \bigcup_{p' \in \text{pred}(p)} (RD(p') \cap \text{PRESERVED}(p') \cup SDEF(p'))$$

(2.5)

where $SDEF(p)$ is the set of definition statements at $p$, and $\text{PRESERVED}(p)$ is the set of definitions that are not overridden by the statement at $p$. Note that, if $\text{pred}(p) = \emptyset$, then $p$ is the initial node and $RD(p) = \emptyset$.

A main application of reaching definitions analysis is in the construction of direct links between blocks that produce values and blocks that use them.

2.3 Solving Data Flow Equations

Figure 2.6 summarizes the four types of basic data flow analysis problems that were introduced in the previous section. Available Expressions (AE) and Very Busy Expressions (VBE) use the set intersection operation, whereas Reaching Definitions (RD) and Live Variables (LV) use the set union operation. For AE and RD the operation is over predecessors, whereas for VBE and LV it is over successors. AE and RD are called top-down (or forward) problems because information is propagated in the same direction as control flow to solve these problems. Conversely, VBE and LV are called bottom-up (or backward) problems because information must be propagated in the opposite direction of control flow to solve these problems.

The union-type problems have a great deal in common. In each case, the solution to the data-flow equations is not unique, and the smallest solution is wanted. The equations are solved by starting with the approximation in which all unknowns are $\emptyset$ and iterating until the desired solution is reached from below. In contrast, the intersection-type problems have equations whose largest solution is wanted. Solving the equations is started with the unknowns all equal to $U$, the universal set, and iterate until the solution is reached from above.
The forward problems have also something in common. They are all efficiently solved by iterating over nodes in depth-first order. The backward problems, on the other hand, are efficiently solved by iterating in the reverse of depth-first order.

**Space requirements** Note that is does not necessarily require much space to iterate over graph structures and sets. Classical analyses operate over elements of $\mathcal{P}(D)$, where $D$ is a finite set of variables, expressions, definitions or something similar. But the elements can be represented as bit vectors: Each element of $D$ can be assigned a unique bit position $i$ ($1 \leq i \leq n$, $n = |D|$). A subset $S$ of $D$ is then represented by a vector of $n$ bits:

- if the $i^{th}$ element of $D$ is in $S$ then the $i^{th}$ bit is 1.
- if the $i^{th}$ element of $D$ is not in $S$ then the $i^{th}$ bit is 0.

Thus, not much space is required to hold the sets the algorithm works with. Furthermore, there are efficient implementations for set union (logical or) and set intersection (logical and), so the required operations don’t take much time. Usually data-flow equations are solved using bit vectors.

**Complexity** If the size of the bit vector is $n$, and $r$ is the number of nodes in the control flow graph, there are $O(r \times n)$ properties. Thus, in the worst case $O(r \times n)$ iterations are needed. Each iteration involves the computation of the properties for $r$ nodes. If all the $n$ bits can be processed in one step, the complexity becomes $O(r^2 \times n)$. As the nodes can be visited in postorder or reversted postorder, $d + 2$ iterations are sufficient, where $d$ is the depth of the flow graph [1]. Hence the complexity is

$$O((d + 2) \times r \times n)$$

There are several techniques to increase the speed of DFA algorithms [20, 39], but it would be beyond the scope of this work to discuss them. For further DFA problems, the procedure for solving the equations remains generally the same, i.e. iterating until a least or largest fixed point is reached. As this thesis deals with the gathering of DFA information
using a model checker, a detailed discussion of the classical DFA algorithms is omitted (model checking algorithms are discussed in section 3.3).

2.4 Derived Data Flow Information and Further Problems

Beyond the four basic analyses problems that were introduced in section 2.2, there are many more problems, often direct preparations for code optimization transformations. Many of them are based on AE, RD, LV and VBE, but there are different approaches, either.

This section discusses a selection of further data-flow problems and lays no claim to completeness. As the problems are later solved using a model checker, the data-flow equations are only given if they can directly be translated into model checking formulas. Otherwise, i.e. if the model checking formulas can be derived, e.g., from the definition of the properties, the data-flow equations are omitted and it is only sketched how the information can be computed.

Live Definitions

A definition \(d\) is live at point \(p\) iff \(d \in RD(p) \cap LV(p)\) \(^4\). That is, not only does the definition reach \(p\), but it can potentially be used. Note that the live definitions problem is different from the live variables problem and that the former gives "stronger" information. The sets of live definitions are useful when assigning index registers: registers holding dead definitions can be reused immediately.

Common Subexpressions

The term common subexpression refers to redundant expression evaluation. Figure 2.7 shows a simple example, where \(b \ast c\) is a common subexpression that is evaluated three times, although it would be sufficient to evaluate it once an re-use the result. This is what common subexpression elimination does.

\[
1: \ a = b \ast c + g; \\
2: \ d = b \ast c \ast d; \\
3: \ e = b \ast c;
\]

\(\text{Figure 2.7: Common subexpressions example}\)

Common subexpressions are determined in two steps. First, states have to be found where expressions are re-evaluated. For an expression \(XopY\) this is the case for those states \(s\) on which \(XopY\) is used and available, i.e.

\[
XopY \in EUSE(s) \quad \land \quad XopY \in AE(s)
\]

\(^4\) It is assumed here that definitions and uses are suitably encoded so that they can be meaningfully intersected.
Second, it has to be found out where the first evaluation of that expression takes place. This can be done by examining the reaching definitions information for all states \( s \) gathered in the previous step: those definitions in \( RD(s) \) that have \( XopY \) on the right, i.e. that use \( XopY \)

Having determined all occurrences of the common subexpression, a temporary variable is introduced before the first. The result of the evaluation of the expression is assigned to this temporary variable, and the occurrences of the expression that follow are replaced by this variable.

Figure 2.8 shows how the code from the example in figure 2.7 (containing four multiplications and one addition) can be replaced by code containing only two multiplications and one addition by common subexpression elimination.

\begin{verbatim}
1: tmp = b * c;
2: a = tmp + g;
3: d = tmp * d;
4: e = tmp;
\end{verbatim}

*Figure 2.8: Common subexpression elimination example*

Thus, common subexpression elimination can help to avoid unnecessary recomputations of values and therewith possibly decrease the program's execution time. But the compiler needs to be judicious about the number of subexpressions it saves in temporaries: an excessive number of temporary values creates register pressure possibly resulting in register allocation saving and loading temporaries to/from memory, which may take longer than simply recomputing an arithmetic result when it is needed.

**Use-Definition-Chaining**

An often discussed DFA-problem is called *use-definition-chaining (ud-chaining)*. Roughly stated, ud-chaining answers the question: Given that variable \( A \) is used at point \( p \), at what points could the value of \( A \) used at \( p \) have been identified? For the example in figure 2.9, the ud-chain for the use of \( a \) at line 5 is the set \( \{2: a = 1; 4: a = 2;\} \).

\begin{verbatim}
1: if (a <= 0)
  2:   a = 1;
  3: else
    4:   a = 2;
  5: b = 2 * a;
\end{verbatim}

*Figure 2.9: Use-Definition Chains example*

Ud-chains can be computed from the reaching definitions information: Consider a use
of $A$ at point $p$. The ud-chain for this use consists of all definitions of $A$ in $RD(p)$.

There is a variety of applications for ud-chaining information. Some examples:

- If there is only one definition of name $A$ which reaches a point $p$, and that definition is $A := 3$, then $A$ has the value 3 at $p$, and $A$ can be substituted by 3 if there is a use of $A$ at $p$.

- It can be determined whether a particular definition reaches anywhere at all. Taking the logical or of all the RDs gives this information.

- It can be determined whether for a particular use of a variable $A$, if it is possible that $A$ is undefined at that point. Dummy definitions of all variables are introduced prior to the initial block. If the dummy definition of $A$ can reach a point which uses $A$, the compiler could print a warning that $A$ might be undefined.

- Ud-chaining can be used for constant folding, i.e. replacing expressions by their value if the value can be computed at compile time. The algorithm works as follows.

As long as changes occur, the operands $O$ of all statements $s$ of the program are checked:

- If there is a unique definition of $O$ that reaches $s$ and that definition is of the form $O := c$ for a constant $c$, then replace $O$ by $c$ in $s$.

- If all operands of $s$ are now constants, then evaluate the right side of $s$ and replace $s$ by $A := e$, where $A$ is the name assigned to by $s$ and $e$ is the value of the right side of $s$.

**Definition-Use-Chaining**

The definition-use-chaining (du-chaining) problem is to compute for a point $p$ that defines $A$ the sets of uses $s$ of $A$, such that there is a path from $p$ to $s$ that does not redefine $A$. Figure 2.10 shows an example where the du-chain for the definition of $a$ at line 1 is the set $\{3: \ c = a * b; , 5: \ c = a / b; \}$.

```plaintext
1: a = 1;
2: if (b > 0)
3:   c = a * b;
4: else
5:   c = a / b;
```

*Figure 2.10: Definition-Use Chains example*

Supposed the DEF, USE, RD, and LV information is known for each point $p$. By considering both $RD(p)$ and $USE(p)$ together, a pointer from each use in $USE(p)$ to the location of
zero or more definitions in RD(p) can be established. A similar association can be estab-
lished between LV(p) and DEF(p). This double linking forms the definition-use chains.

Such information is useful for dead code elimination, constant propagation, and error
detection. For example, if a given definition affects no uses, that definition can be elimi-
nated. If all definitions reaching a particular use are the same constant, this fact can be
used to perform constant propagation.

**Copy Propagation**

Various code optimization algorithms introduce copy steps *(copies)* of the form \( A = X \).
Copies may also be directly generated by the intermediate code generator. It is possible
to eliminate statement \( s : A = X \) if all places where this definition of \( A \) is used are deter-
mined. \( X \) may then be substituted for \( A \) in all these places, provided these conditions are
met by every such use \( u \) of \( A \):

1. Statement \( s \) must be the only definition of \( A \) reaching \( u \) (that is, the ud-chain for use
   \( u \) consists only of \( s \)).
2. On every path from \( s \) to \( u \), including paths that go through \( u \) several times (but do
   not go through \( s \) a second time), there are no assignments to \( X \).

Consider figure 2.11. Statement 1: \( a = x \); can not be eliminated, as the second prop-
erty does not hold (due to the assignment to \( x \) at line 3). Statement 2: \( b = y \) can be
eliminated, as both conditions are satisfied; hence, line 4 becomes \( c = a + y \) and line 2 is
removed.

\[
\begin{align*}
1: & \quad a = x; \\
2: & \quad b = y; \\
3: & \quad x = z; \\
4: & \quad c = a + b;
\end{align*}
\]

*Figure 2.11: Copy Propagation example*

Obviously, condition (1) can be checked using ud-chaining information. For condition
(2) a new data-flow analysis problem must be set up, in which CP(n) is the set of copies
\( A = X \) such that every path from the initial node to \( n \) contains the statement \( A = X \),
and subsequent to the last occurrence of \( A = X \), there are no assignments to \( X \). A copy
statement \( s : A = X \) is *generated* at node \( n \) if \( s \) occurs in \( n \). \( s : A = X \) is *killed* at \( n \) if \( A \) or \( X \)
is assigned there and \( s \) is not in \( n \).

Let \( \mathcal{U} \) be the “universal” set of all copy statements in the program. Define \( CGEN(n) \) to
be the set of all copies generated in \( n \) and \( CKILL(n) \) to be the set of copies in \( \mathcal{U} \) which are
killed in \( n \). Then the following equation relates the quantities defined:

\[
CP(p) = \bigcap_{p' \in pred(p)} ((CP(p') \cap notCKILL(p')) \cup CGEN(p)) \tag{2.6}
\]
Thus, the second condition is satisfied, if \( u \in CP(s) \)

**Optimal Computation Points**

Given a fixed term \( XopY \), an algorithm for the placement of computations proceeds along the following lines [44, 45]:

- Initialize a distinct auxiliary variable (or a register) \( h \) at some program points with the value of \( XopY \), and
- Replace all original computations of \( XopY \) by \( h \).

Consider figure 2.12. The computation of \( x + y \) can be placed between lines 2 and 3, and the computations of \( x + y \) at lines 4 and 6 replaced by the auxiliary variable.

```
1: a = x;
2: b = y;
3: if (c > 0)
4:   x = x + y;
5:   else
6:   y = x + y;
```

*Figure 2.12: Optimal computation points example*

A placement algorithm is fully specified by means of the set of initialization points, its *placements* [26]. The notion of correctness is defined by requiring that the auxiliary variable contains the required value at each of its usages, and that the initializations of the auxiliary variable do not introduce any new values on any paths. Whereas the first requirement is immediately clear, the second requirement is necessary, because the introduction of new computations may lead to new run-time errors, e.g. "division by 0" or "overflow".

A placement is

- **safe**, iff for every computation point \( n \), a computation of \( XopY \) at \( n \) does not introduce a new value on a path through \( n \). This property is necessary in order to guarantee that the program semantics is preserved by the placement process.

- **earliest**, iff for every computation point \( n \), there is a path leading from the initial node to \( n \) where no node \( n' \) prior to \( n \) is safe and delivers the same value as \( n \) when computing \( XopY \).

A placement is computationally optimal if it is safe and earliest. Thus, the set of all expressions whose optimal computation point (OCP) is at node \( n \) is

\[
OCP(n) = SAFE(n) \cap EAR(n)
\]

(2.7)
where

\[ SAFE(n) = \bigcap_{n' \in \text{succ}(n)} (SAFE(n') \cap EUSE(n') \cup \text{notMOD}(n')) \quad (2.8) \]

specifies safety and

\[ EAR(n) = \bigcup_{n' \in \text{pred}(n)} (EAR(n') \cap \text{notSAFE}(n') \cup \text{MOD}(n')) \quad (2.9) \]

specifies earliestness (with EAR for the initial node equal to the set of all states). The least solution is required for SAFE, whereas EAR requires a largest solution.

A safe placement is computationally optimal if the results of any other safe placement always require at least as many computations at run-time. In fact, safety and earliestness are sufficient to characterize a computationally optimal placement [26].

Note that in completely arbitrary graph structures, the placement process may deliver unsatisfactory results, because specific patterns may cause the code motion process to get blocked [40, 46]. This problem can be solved by means of the following transformation: insert an artificial node into each original edge that ends at a node with more than one predecessor; afterwards the artificial nodes have unique ingoing and outgoing edges [45].
3 Model Checking

Model Checking is an automatic technique for the formal verification of finite state systems. This is achieved by verifying if the model, often deriving from a hardware or software design, satisfies a logical specification. The model is usually expressed as a state transition system, i.e. directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node. The nodes represent states of a system, the edges represent possible executions which alter the state, while the atomic propositions represent the basic properties that hold at a point of execution. The specification is often written as temporal logic formulas.

The model checking process can be divided into three major tasks, each of which will be discussed in this chapter, following [8, 22, 37]:

- **Modeling** Convert a design (software or hardware) into a formalism accepted by a model checking tool.

- **Specification** State the properties that the design must satisfy (some logical formalism, common is modal logic).

- **Verification** Check if the model satisfies the specification. Ideally the verification is completely automatic. However, in practice it often involves human assistance in the analysis of the results, e.g. in interpreting error traces.

The main challenge in model checking is dealing with the state explosion problem. This problem occurs in systems with many components that can interact with each other or systems that have data structures that can assume many different values. In such cases the number of global states can be enormous [6, 27, 34]. Considerable progress has been made on the state explosion problem. The most successful techniques for dealing with this problem are based on the partial order reduction. Other approaches are, for example, compositional reasoning, abstraction, symmetry, symbolic model checking and induction [4, 8, 9, 33]. Exploring these approaches further is one of the current major research topics. As the data-flow analysis in this work is not concerned with systems of the kind described above, however, the state explosion problem is not discussed further.

3.1 Modeling

Model Checking works on a discrete model of the system in question. Typically, the system’s behavior is (abstractly) represented by a graph structure. Two approaches common
in use are Kripke structures (the graph’s nodes are annotated with so-called atomic propositions), and labeled transition systems (the edges are annotated with so-called actions). Furthermore, Kripke structures and labeled transition systems can be combined to Kripke transition systems, which are often more convenient for modeling purposes.

The generation of models from first order representations, digital circuits, programs, etc., it is often simply a compilation task. In other cases, the modeling of a design may require the use of abstraction to eliminate irrelevant or unimportant details. The state transition systems used for the data-flow analysis in this thesis, are built from CFGs. Thus, the subject of model generation will be discussed when describing the used framework (section 5.3), while this section deals with different kinds of models.

Kripke structures

A Kripke structure $M$ over a finite set $AP$ of atomic propositions is a triple $M = (S, R, I)$ where

1. $S$ is a finite set of states.
2. $R \subseteq S \times S$ is a transition relation that must be total, i.e. for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in R$.
3. $I : S \rightarrow 2^{AP}$ is an interpretation function that labels each state with the set of atomic propositions valid in that state. AP always contains the propositions true and false and, for any state $s$, $true \in I(s)$ and $false \notin I(s)$.

An example Kripke structure $M = (S, R, I)$ with

- $S = \{0, 1, 2, 3, 4\}$,
- $R = \{(0, 2), (1, 1), (2, 1), (2, 3), (3, 2), (3, 4), (4, 4)\}$ and
- $I = \{I(2) = I(3) = I(4) = \text{"black"}, I(0) = I(1) = \text{"white"}\}$

is given in figure 3.1.

![Figure 3.1: Kripke structure](image-url)
Labeled Transition Systems

A labeled transition system (LTS) is a triple $M = (S, Act, \rightarrow)$, where

1. $S$ is a finite set of states.
2. $Act$ is a finite set of actions.
3. $\rightarrow \subseteq S \times Act \times S$ is a total transition relation.

A transition $(s, a, s') \in \rightarrow$, also written as $s \xrightarrow{a} s'$, states that the system can evolve from a state $s$ to a state $s'$ and thereby exchanging action $a$ with its environment. Then, $s'$ is called an $a$-successor of $s$, whereas $s$ is an $a$-predecessor of $s'$.

An example transition system $M = (S, Act, \rightarrow)$ with

$$
S &= \{0,1,2,3,4\}, \\
Act &= \{a,b\} \text{ and } \\
\rightarrow &= \{(0,b,1),(1,a,2),(1,b,4),(2,b,1),(2,a,3),(3,b,3),(4,a,4)\}
$$

is given in figure 3.2.

![Figure 3.2: Labeled Transition System](image)

In an LTS, the focus lies on the actions, as the edges are labeled (with single transitions), while in a Kripke structure, the focus lies on the states themselves, as they are labeled (with sets of atomic propositions). Logical considerations can usually be translated between these two representations. For modeling purposes, however, it is often natural to have both kinds of labeling available, and therefore, LTS and KS are combined.

Kripke Transition Systems

A Kripke transition system (KTS) over a finite set $AP$ of atomic propositions is a structure $M = (S, Act, \rightarrow, I)$, where

1. $S$ is a finite set of states.
2. $Act$ is a finite set of actions.
3. $\rightarrow \subseteq S \times Act \times S$ is a total transition relation.
4. $I : S \rightarrow 2^{AP}$ is an interpretation function.
Kripke transitions systems generalize both Kripke structures and labeled transition systems: a KS is a KTS with an empty set of actions, an a LTS is a KTS with a trivial interpretation $I$.

An example Kripke transition system $M = (S, \text{Act}, \rightarrow, I)$ with
\[
S &= \{0, 1, 2, 3, 4\}, \\
\text{Act} &= \{a, b\}, \\
\rightarrow &= \{(0, b, 2), (1, a, 1), (2, a, 3), (2, b, 1), (3, b, 2), (3, a, 4), (4, b, 4)\} \text{ and} \\
I &= \{I(2) = I(3) = I(4) = "black", I(0) = I(1) = "white"\}
\]
is given in figure 3.3.

![Kripke Transition System](image)

Figure 3.3: Kripke Transition System

A path in a structure (KS, LTS, or KTS) from a state $s$ is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. Often it is necessary to designate a certain state $s_0 \in S$ in a structure as the initial state. Intuitively, the execution of the system starts in this state. A structure having an initial state is called a rooted structure.

### 3.2 Specification

Temporal logic is a formalism for describing sequences of transitions between states [21]. Temporal operators can be combined with boolean connectives and nested arbitrarily. There are several temporal logics, differing in the operators they provide and the semantics of the operators.

The modal $\mu$-calculus [22, 28, 31, 49], which is introduced in this section, is a powerful language for expressing properties of transition systems by using least and greatest fixed point operators. Many temporal and program logics can be encoded into the $\mu$-calculus, among others the Computation Tree Logic (CTL) which is also discussed later in this section.

**The Modal $\mu$-Calculus**

A formula of the modal $\mu$-calculus can be generated according to the following Backus-Naur form (BNF):

\[
\Phi ::= \text{true} \mid \text{false} \mid p \mid \neg \Phi \mid \Phi \lor \Phi \mid \Phi \land \Phi \mid \Phi \Rightarrow \Phi \\
\mid [a]\Phi \mid \langle a \rangle \Phi \mid [\overline{a}]\Phi \mid \langle \overline{a} \rangle \Phi \mid X \mid \mu X.\Phi \mid \nu X.\Phi
\]

22
\( [\text{true}]_T(e) = S \)
\( [\text{false}]_T(e) = \text{not} S \)
\( [p]_T(e) = \{ s \in S \mid p \in I(s) \} \)
\( [\neg \Phi]_T(e) = S \setminus [\Phi]_T(e) \)
\( [\Phi_1 \lor \Phi_2]_T(e) = [\Phi_1]_T(e) \cup [\Phi_2]_T(e) \)
\( [\Phi_1 \land \Phi_2]_T(e) = [\Phi_1]_T(e) \cap [\Phi_2]_T(e) \)
\( [\Phi_1 \Rightarrow \Phi_2]_T(e) = (S \setminus [\Phi_1]_T(e)) \cup [\Phi_2]_T(e) \)
\( [\langle a \rangle \Phi]_T(e) = \{ s \in S \mid \exists s' \text{ with } s \xrightarrow{a} s' : s' \in [\Phi]_T(e) \} \)
\( [\langle a \rangle \Phi]_T(e) = \{ s \in S \mid \exists s' \text{ with } s \xrightarrow{a} s' : s' \in [\Phi]_T(e) \} \)
\( [\langle a \rangle \Phi]_T(e) = \{ s \in S \mid \exists s' \text{ with } s \xrightarrow{a} s' : s' \in [\Phi]_T(e) \} \)
\( [\mu X. \Phi]_T(e) = \text{ the least fixed point of the predicate transformer } \tau : 2^S \rightarrow 2^S \)
\( \text{ defined by: } \tau(W) = [\Phi]_T(e)[W/X] \)
\( [\nu X. \Phi]_T(e) = \text{ the greatest fixed point of the predicate transformer } \tau : 2^S \rightarrow 2^S \)
\( \text{ defined by: } \tau(W) = [\Phi]_T(e)[W/X] \)

Figure 3.4: Semantics of the \( \mu \)-calculus

Here, \( p \) is an atomic proposition, \( a \) ranges over the set of all actions \( Act \), \( X \) is a variable from \( Var \). Variables in the \( \mu \)-calculus can be either free or bound by a fixed point operator. Closed formulas are the formulas without free variables. The \( \mu \) and \( \nu \) operators are used to express least and greatest fixed points, respectively. The empty set of states is denoted by \( false \), and the set of all states \( S \) is denoted by \( true \).

The intuitive meaning of the formula \( \langle a \rangle \Phi \) (pronounced as diamond \( a \)) is "it is possible to make an \( a \)-transition to a state where \( \Phi \) holds". Similarly, \( [a] \Phi \) (box \( a \)) means that "\( \Phi \) holds in all states reachable (in one step) by making an \( a \)-transition".

Note that the set of operators that is given here corresponds to those available within the later used framework. It is not necessary to give this whole set, however, as some formulas can be derived from others.

Formally, the semantics of a formula \( \Phi \) is a set of states in which \( \Phi \) is true. It is denoted as \( [\Phi]_T(e) \), where \( T = (S, Act, \rightarrow, I) \) is a Kripke transition system and \( e : Var \rightarrow 2^S \) is an environment. \( e[W/X] \) denotes a new environment that is the same as \( e \) except that \( e[W/X](X) = W \). The set \( [\Phi]_T(e) \), i.e. the semantics of a \( \mu \)-calculus formula \( \Phi \), is defined recursively as given in figure 3.4 [8].

23
Within formulas, the negation is restricted in use. Thus, monotonicity is guaranteed and the fixed points are well defined. Formally, every logical connective except negation is monotonic, and all the negations can be pushed down to the atomic propositions using De Morgan’s laws and the dualities

\[
\neg[a] \Phi \equiv \langle a \rangle \neg \Phi \\
\neg \langle a \rangle \Phi \equiv [a] \neg \Phi \\
\neg[a] \Phi \equiv \langle a \rangle \neg \Phi \\
\neg \langle a \rangle \Phi \equiv [a] \neg \Phi \\
\neg \mu X. \Phi(X) \equiv \nu X. \neg \Phi(\neg X) \\
\neg \nu X. \Phi(X) \equiv \mu X. \neg \Phi(\neg X)
\]

Because bound variables are under an even number of negations, they will be negation-free after this process. Thus, each possible formula in a fixed point operator is monotonic and hence each possible \( \tau \) is also monotonic (\( S \subseteq S' \) implies \( \tau(S) \subseteq (S') \)). This is enough to ensure the existence of the fixed points [13, 52]. Furthermore, because the formulas are evaluated over finite transition systems, monotonicity of \( \tau \) implies that \( \tau \) is also \( U \)-continuous and \( \cap \)-continuous [8]. Hence the least and greatest fixed points can be computed by iterative evaluation:

\[
[[\mu X. \Phi]]_{\tau}(e) = \bigcup_i \tau^i(\text{false})
\]

\[
[[\nu X. \Phi]]_{\tau}(e) = \bigcap_i \tau^i(S)
\]

where \( \tau^0(X) \) is defined recursively by \( \tau^0(X) = X \) and \( \tau^{i+1}(X) = \tau(\tau^i(X)) \).

Because the domain \( S \) is finite, the iteration stops after a finite number of steps. More precisely, for some \( i, j \leq |S| \), the least fixed point is equal to \( \tau^i(\text{false}) \) and the greatest fixed point is equal to \( \tau^i(\text{true}) \). To find these fixed points, \( \tau \) is applied repeatedly starting from either \text{false} or \text{true} until the result does not change.

In the following, \([\ ]\), \(\prod\), \(\langle \rangle\), and \(\overline{\langle \rangle}\) will be written instead of \([\text{Act}]\), \([\text{Act}]\), \(\langle \text{Act} \rangle\), and \(\overline{\langle \text{Act} \rangle}\), respectively.

**CTL**

While the \( \mu \)-calculus forms a low-level specification language, the Computation Tree Logic (CTL) is referred to as a high-level specification language. Regarding expressiveness, CTL is a subset of the \( \mu \)-calculus [14], i.e. all CTL formulas can be translated into \( \mu \)-calculus formulas. Using CTL may often be more convenient, as its operators are more intuitively to understand.

Conceptually, CTL formulas describe properties of computation trees. The tree is formed by designating a state in the model as the initial state and then unwinding the structure
into an infinite tree with the initial state at the root. The computation tree shows all of the possible executions starting from the initial state.

A CTL formula can be generated according to the following BNF:

\[
\Phi ::= \ p | \neg \Phi | \Phi \lor \Phi | \Phi \land \Phi | \Phi \Rightarrow \Phi |
\]

\[
EX(\Phi) | EF(\Phi) | EG(\Phi) | ESU(\Phi, \Phi) | EWU(\Phi, \Phi) |
\]

\[
AX(\Phi) | AF(\Phi) | AG(\Phi) | ASU(\Phi, \Phi) | AWU(\Phi, \Phi) |
\]

\[
\overline{EX}(\Phi) | \overline{EF}(\Phi) | \overline{EG}(\Phi) | \overline{ESU}(\Phi, \Phi) | \overline{EWU}(\Phi, \Phi) |
\]

\[
\overline{AX}(\Phi) | \overline{AF}(\Phi) | \overline{AG}(\Phi) | \overline{ASU}(\Phi, \Phi) | \overline{AWU}(\Phi, \Phi)
\]

Like for the \(\mu\)-calculus, \(p\) is an atomic proposition. Again, this is the set of operators that is available within the later used jABC-Framework, and it is not necessary to give this whole set, as some formulas can be derived from others.

CTL operators consists of two parts: A ("for all") and E ("exists") are the so-called path quantifiers, as they state on which paths the formula must hold, therewith describing the branching structure. X ("next"), G ("globally"), F ("finally"), SU ("strong until"), and WU ("weak until") are the state quantifiers, they express when, on certain paths, the formula must hold. Like the \(\mu\)-calculus, CTL also provides backward operators, denoted by overlined operators.

The semantics for CTL expressions is identical with the semantics of the \(\mu\)-calculus, or can be given by translations into \(\mu\)-calculus equivalents (figures 3.5 and 3.6).

\[
AX(\Phi) = [X]\Phi
\]

\[
EX(\Phi) = \langle X \rangle \Phi
\]

\[
AG(\Phi) = n X.(\Phi \land [X] X)
\]

\[
EG(\Phi) = \nu X.X.(\Phi \land (X \lor [false]))
\]

\[
AF(\Phi) = \mu X.(\Phi \lor [X \land (true)])
\]

\[
EF(\Phi) = \mu X.(\Phi \lor (\langle X \rangle X))
\]

\[
ASU(\Phi, \Psi) = \mu X.(\Psi \lor \Phi \land [X \land (true)])
\]

\[
ESU(\Phi, \Psi) = \mu X.(\Psi \lor \Phi \land (\langle X \rangle X))
\]

\[
AWU(\Phi, \Psi) = \nu X.(\Psi \lor \Phi \land [X])
\]

\[
EWU(\Phi, \Psi) = \nu X.(\Psi \lor \Phi \land (\langle X \rangle X))
\]

Figure 3.5: CTL forward operators and \(\mu\)-calculus equivalents
3 Model Checking

\[\begin{align*}
\forall X (\Phi) & = \bigcap \Phi \\
\exists X (\Phi) & = \bigcup \Phi \\
\forall G (\Phi) & = \nu X. (\Phi \land [\bigcap] X) \\
\exists G (\Phi) & = \nu X. (\Phi \lor (\exists X \lor \bigcap \text{false})) \\
\forall F (\Phi) & = \mu X. (\Phi \land [\bigcap] X \land [\bigcup] \text{true}) \\
\exists F (\Phi) & = \mu X. (\Phi \lor (\exists X \lor \bigcup X) \\
\forall SU (\Phi, \Psi) & = \mu X. (\Psi \lor \Phi \land [\bigcup] X \land [\bigcap] \text{true}) \\
\exists SU (\Phi, \Psi) & = \mu X. (\Psi \lor \Phi \land [\bigcup] X) \\
\forall WU (\Phi, \Psi) & = \nu X. (\Psi \lor \Phi \land [\bigcup] X) \\
\exists WU (\Phi, \Psi) & = \nu X. (\Psi \lor \Phi \land [\bigcup] X)
\end{align*}\]

Figure 3.6: CTL backwards operators and \(\mu\)-calculus equivalents

3.3 Verification

The model checking problem is defined as follows. Given a Kripke transition system \(T = (S, Act, \rightarrow, I)\) that represents a finite state system and a temporal logic formula \(\Phi\) expressing some desired specification, find the set of all states in \(S\) that satisfy \(\Phi\):

\[\{s \in S \mid s \models \Phi\}\]

Determining the set of all states that satisfy a formula is called the global model checking problem, whereas determining whether one particular state \(s\) satisfies a formula is called the local model checking problem [37, 50]. The two problems are closely related (solving the global model checking problem also gives the solution for the local problem, while solving the local problem for each state gives the solution for the global problem) but local and global model checkers have different applications. For data-flow analysis, a global model checking is of use (because knowledge about all states shall be gained).

There are several approaches to the implementation of model checking [10, 37], that can be distinguished by their abilities to check globally or locally and to check branching-time (concerning the branching behaviour, i.e. different paths, as well as time) or linear-time (not concerned with branching, only with time) logics. The formulas expressing data flow equations are based on a branching-time logic (the \(\mu\)-calculus and hence CTL are branching-time logics). Thus, the semantic (or iterative) approach is introduced here, as it works globally and takes branching-time logic formulas as input.

Remember that, regarding expressiveness, CTL is a subset of the \(\mu\)-calculus. Therefore, CTL formulas are translated into \(\mu\)-calculus formulas and the model checker only works with the latter.

26
The Semantic Approach

The idea behind the semantic approach is to inductively compute the semantics of the formula in question on the given finite model [37]. This generates a global model checker and works well for branching-time logics. Modalities are reduced to their fixed point definitions, and fixed points are computed by applying the Kleene fixed point theorem [25] to the finite domain of the state powerset.

Based on the iterative characterization of fixed points (compare section 3.2), the semantics of $\mu$-calculus formulas can be effectively evaluated on finite state Kripke transition systems. But in general, this is quite difficult, due to the potential interference between least and greatest fixed points. It is shown later in this section that the alternated nesting of least and greatest fixed points requires the introduction of backtracking into the fixed point iteration procedure, which causes an exponential worst-case time complexity of iterative model checking for the full $\mu$-calculus. Whether this is a tight bound for the model checking problem as such is an open problem.

Depending on the structure of the formula, three degrees for the difficulty of the computation can be distinguished. First, there are fixed point subformulas that contain no free variables, and so do not depend on the environment. This is always the case for CTL-like formulas. Here the iteration process can be organized in a hierarchical fashion:

1. Associate all variables belonging to greatest fixed points with the full set of states $S$ and all variables belonging to least fixed points with $\emptyset$.

2. Choose a subformula, $\phi = \mu X. \phi'$ (or $\phi = \nu X. \phi'$), where $\phi'$ is fixed point-free, determine its semantics, and replace it by an atomic proposition $A_\phi$, whose valuation is defined by its semantics.

3. Repeat the second step until the whole formula is proceeded.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{kripke_transition_system.png}
\caption{Kripke Transition System}
\end{figure}

For an example consider the Kripke transition system in figure 3.7 and a CTL formula and its $\mu$-calculus equivalent:

$$\Phi = AG(\text{black}) = \nu X. (\text{black} \land []X)$$

The greatest fixed point is computed by the following iteration steps:
Thus, the model checker correctly computes that $AG(\text{black})$ holds in node 4.

Second, there are fixed point subformulas that contain free variables and so depend on the environment, and the fixed point formulas have the same parity (i.e. they are either all least fixed point formulas or all greatest fixed point formulas). Due to the dependence of the inner fixed point on the variable introduced further outwards, the inner fixed point formula must - at least in principle - be evaluated again and again in each iteration of the outer formula. Fortunately, if the fixed point formulas have the same parity, the problem can be avoided by computing the values of the inner and outer formulas simultaneously, because the value of a fixed point formula depends monotonically on the value of its free variables and the iterations of all formulas proceed in the same direction.

For example, consider again the Kripke transition system in figure 3.7 and the $\mu$-calculus formula

$$\Phi = \nu Z. \nu Y. (\langle a \rangle Z \land \langle b \rangle Y)$$

The fixed point is computed by the following iteration steps:

<table>
<thead>
<tr>
<th>Iteration for $Y,Z$:</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption for $Y,Z$:</td>
<td>S</td>
<td>[2,3]</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\langle a \rangle Z \land \langle b \rangle Y$:</td>
<td>[2,3]</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Thus, the model checker correctly computes that $\Phi$ does not hold for any node.

And third, there are fixed point subformulas that contain free variables and so depend on the environment, and the fixed point formulas do not have the same parity. In this case of mutual dependencies between least and greatest fixed points, the iterations proceed in opposite directions, which excludes a simple monotonic iteration process. Such formulas are called alternating fixed point formulas. They require backtracking (or resets) in the iteration process.

For an example, consider the Kripke transition system in figure 3.7 again, and a $\mu$-calculus formula which intuitively specifies that there is a path consisting of $a$- and $b$-steps with infinitely many $b$-steps:

$$\Phi = \nu Z. \mu Y. (\langle b \rangle Z \lor \langle a \rangle Y)$$

The fixed point is computed by the following iteration steps:

<table>
<thead>
<tr>
<th>Iteration for $Z$:</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assumption for $Z$:</td>
<td>S</td>
<td>[0,2,3,4]</td>
</tr>
<tr>
<td>Iterations for $Y$:</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Assumptions for $Y$:</td>
<td>$\emptyset$</td>
<td>[0,2,3,4]</td>
</tr>
<tr>
<td>$\langle b \rangle Z \lor \langle a \rangle Y$:</td>
<td>[0,2,3,4]</td>
<td>[0,2,3,4]</td>
</tr>
</tbody>
</table>

Thus, the model checker correctly computes that $\Phi$ holds for all states except 1.
**Complexity**  
Due to the nested iteration, the complexity of model checking formulas with alternation is high. A careful implementation of the iterative approach leads to an asymptotic worst-case running time of

$$O((|T| * |\phi|)^{ad})$$

where $|T|$ is the number of states and transitions in the model structure and $|\phi|$ is the size of the formula (measured, say, by the number of operators). $ad$ refers to the *alternation depth* of the formula, which essentially is the number of non-trivial alternations between least and greatest fixed points. (The alternation depth of an alternation-free formula is taken to be 1.)

Although the precise definition of alternation depth is not uniform in the literature, all definitions share the intuitive idea of the above stated model checking complexity. By exploiting monotonicity, a better time-complexity can be achieved, but at the cost of a large storage requirement. It is an open problem to determine the precise complexity of $\mu$-calculus model checking; it might well turn out to be polynomial (it is known to be in the intersection of the classes NP and co-NP \(^1\)).

Alternation-free formulas can be checked efficiently in time $O(|T| * |\phi|)$. This holds in particular for all CTL-like formulas, as they unfold to alternation-free fixed point formulas [3, 11, 16, 37].

---

\(^1\)Co-NP is the complement of the complexity class NP. In simple terms, it is the class of problems for which efficiently verifiable proofs of *no* instances, sometimes called *counterexamples*, exist. $P$ is a subset of both NP and co-NP. NP and co-NP are thought to be unequal. If a problem can be shown to be in both NP and co-NP, that is generally accepted as strong evidence that the problem is probably not NP-complete (since otherwise NP = co-NP). [12]
4 Data-Flow Analysis via Model Checking

Data flow analysis (DFA) is concerned with the automatic identification of program points enjoying specific properties, like for example, liveness of variables or equivalence of program terms. Typically, DFA algorithms are constructed for a given program property of interest and therefore have the following functionality:

$$\text{DFA algorithm for a property} : \text{programs} \rightarrow \text{program points enjoying the property}$$

Model checking is concerned with the automatic identification of those states of a finite system satisfying a specific modal or temporal formula that expresses, for example, properties like deadlock or liveness. Typically, model checkers are parameterized on the formula of interest and therefore have the following functionality:

$$\text{model checker} : \text{modal formulas} \times \text{model} \rightarrow \text{states satisfying the argument formula}$$

Identifying programs with models, program points with states and program properties with modal formulas, model checkers can be seen as DFA algorithms that have the program property of interest as a parameter. [45]

Figure 4.1 (taken from [36]) portrays the process of DFA via Model Checking.

![Diagram of DFA-MC process]

Figure 4.1: DFA-MC
Thus, two transformations have to be made: programs have to be turned into appropriate models, and DFA equations have to be translated into modal specifications. The solving of the equation (verification) is simply done by a model checker.

### 4.1 From Programs to Program Models

Slight variants of Kripke transition systems work well for modeling sequential imperative programs for data flow analysis purposes, as they are able to concisely express the implied predicate transformer scheme. Two possibilities, that are also available in the used framework, are introduced in the following. [43, 44, 45]

One possibility, that is closely related to classical control flow graphs, is that nodes contain the statements and the predicates, while the edges represent the control flow (conditional or unconditional branching).

An appropriate program model $P$ is a quadruple $(S, \rightarrow, B, \lambda)$, where

1. $S$ is a finite set of nodes or program states (representing a single statement of the program), containing one start node (head) and one or more end nodes (tail).
2. $\rightarrow \subseteq S \times \{\text{true}, \text{false}, \text{default}\} \times S$ is a set of labeled transitions, which define the control flow of $P$.
3. $B$ is a set of atomic propositions.
4. $\lambda$ is a function $\lambda: S \rightarrow 2^B$ that labels states with subsets of $B$.

Another possibility is that the statements are pushed from the nodes into the outgoing edges, and thus nodes express the predicates or results of the considered analysis, and edges labeled with the statements express the nodes’ interdependencies. Note that pushing the statements downwards, like it is done here, results in a precondition model, while pushing the statement upwards, i.e. into the incoming edges, yields a postcondition model. Such a precondition program model $P$ is as a quintuple $(S, \text{Act}, \rightarrow, B, \lambda)$, where

1. $S$ is a finite set of nodes or program states, containing one start node (head) and one or more end nodes (tail).
2. $\text{Act}$ is a set of actions.
3. $\rightarrow \subseteq S \times \text{Act} \times S$ is a set of labeled transitions, which define the control flow of $P$.
4. $B$ is a set of atomic propositions.
5. $\lambda$ is a function $\lambda: S \rightarrow 2^B$ that labels states with subsets of $B$.

Both variants are equivalent after all, but depending on what is to be analysed, one may be more convenient to use than the other.
4.2 From DFA Equations to Modal Specifications

Having a specification language at disposal, DFA equations (together with the information what kind of solution is desired) can be translated into $\mu$-calculus formulas. According to [42], the encodings are straightforward: Simple union and intersection translate into disjunction and conjunction, respectively; “big” unions and intersections on the predecessor and successor states translate into diamond and box modalities, respectively; a forwards analysis uses overlined modalities (because it calculates information about histories) and a backwards analysis uses unlined modalities (because it calculates information about futures); and the least and greatest fixed-point solutions of the equations are stated explicitly by the $\mu$- and $\nu$-operators.

This section deals with the translation of the equations for those data flow problems that were introduced in chapter 2. If possible, the equations are also translated into CTL.

Live Variables

The equation for live variables, equation 2.1, translates into the following formula, specifying those states on which variable $x$ is live.

$$
\text{isLive}(x) = \mu Z. (\text{isVUsed}(x) \lor \neg \text{isDefined}(x) \land \langle \rangle Z)
$$

$$
= E\Sigma U(\neg \text{isDefined}(x), \text{isVUsed}(x))
$$

(4.1)

where

$$
\text{isVUsed}(x) = \text{isUsed}(x)
$$

Very Busy Expressions

Equation 2.2, the DFA specification for very busy expressions, is expressed in the $\mu$-calculus as follows:

$$
\text{isVBE}(e) = \nu Z. (\text{isEUsed}(e) \lor \neg \text{isModified}(e) \land [\square Z \land \langle \rangle \text{true})
$$

$$
= A\Sigma U(\neg \text{isModified}(e), \text{isEUsed}(e))
$$

(4.2)

where

$$
\text{isEUsed}(e) = \text{isUsed}(e)
$$

It is satisfied on those states where expression $e$ is very busy.
Available Expressions

The DFA equations for available expressions, equations 2.3 and 2.4, translate into the following \( \mu \)-calculus formula:

\[
isAvail(e) = \nu Z. \bigwedge \neg isModified(e) \lor isGen(e) \land \neg \text{true}
= \bigwedge (\nu Z. (isGen(e) \lor \neg isModified(e) \land \bigwedge Z \land \neg \text{true}))
= AX(ASU(\neg isModified(e), isGen(e)))
\]  

(4.3)

where

\[
isGen(e) = isEUed(e) \land \neg isModified(e)
\]

Then, \( isAvail(e) \) holds at those program points where the expression \( e \) is available.

Reaching Definitions

The following formula, specifying those states that are reached by the definition of variable \( a \) at state \( s \), is the \( \mu \)-calculus analogy for the reaching definition equation 2.5.

\[
isReaching(a,s) = \mu Z. \neg (Z \land \neg isPreserved(a,s)) \lor isSDef(a,s)
= \neg (\mu Z. (isSDef(a,s) \lor \neg isPreserved(a,s) \land \neg Z))
= AX(ESU(\neg isPreserved(a,s), isSDef(a,s)))
\]  

(4.4)

where

\[
isPreserved(a,s) = \neg isDefined(a) \land EF(s)
\]

and

\[
isSDef(a,s) = s
\]

Live Definitions

The set of states where a definition of \( a \) at state \( s \) is live can be determined by the following formula:

\[
isLiveDef(a,s) = isReaching(a,s) \land isLive(a)
\]

with \( isReaching(a,s) \) and \( isLive(a) \) like described previously.

Common Subexpressions

For every statement \( s \) of the form \( A = \text{XopY} \) such that \( \text{XopY} \) is available at \( s \), i.e. all states satisfying

\[
isUsed(\text{XopY}) \land isAvail(\text{XopY})
\]  

(4.5)

do the following. Find all definitions which reach \( s \) and which have \( \text{XopY} \) on the right by examining the reaching definitions information at \( s \), and checking for each definition \( d \) that reaches \( s \), whether \( \text{XopY} \) is used:

\[
isUsed(\text{XopY}) \land d
\]  

(4.6)
If these are more than zero, $X_0pY$ is a common subexpression and can be eliminated by means of common subexpression elimination (compare section 2.4).

**UD-Chaining**

In classical DFA, the ud-chains are derived from the RD information. Languages for modal specifications (like CTL and $\mu$-calculus) make it possible to specify ud-chains for the use of a variable $x$ at a point $p$ based on the USE- and DEF-information:

$$udChain(x, p) = isDefined(x) \land \langle \langle Z, (isUsed(x) \land p) \lor (\neg isDefined(x) \land \langle \langle Z) \rangle) \rangle$$

$$= isDefined(x) \land EX(ESU(\neg isDefined(x), isUsed(x) \land p)) \quad (4.7)$$

Intuitively, the formula specifies states on which $x$ is defined and from where at least one path, on which no redefinitions of $x$ occur, leads to the use of $x$ at $p$.

**DU-Chaining**

The du-chaining information is derived from the live variables information in classical DFA. With model checking, it can - like it is the case for ud-chains - be computed using modal specifications different from the original DFA equations. The du-chain for the definition of a name $a$ at a point $p$ (i.e. the set of all statements $s$ such that $s$ uses $a$ and there is a path from $p$ to $s$ on which $a$ is not redefined) is specified by the following formula:

$$duChain(a, p) = isUsed(a) \land \langle \langle X, (isDefined(a) \land p) \lor (\neg isDefined(a) \land \langle \langle X) \rangle) \rangle$$

$$= isUsed(a) \land EX(ESU(\neg isDefined(a), isDefined(a) \land p)) \quad (4.8)$$

**Copy Propagation**

Consider a copy statement $s : a = X$. Two properties are checked for each use $u$ of $a$ (the statement can be eliminated by means of copy propagation if both are satisfied):

1. The ud-chain for $a$ at $u$ consists only of $s$. (This is found out by examining the ud-chaining information.)

2. On every path from $s$ to $u$, there are no assignments to $X$. The formula

$$isCP(s) = uZ, [\langle (X \land \langle \langle \rangle) true \land \neg isCKill(s) \rangle \rangle \land (isCGen(s)) \langle \langle true \rangle \rangle$$

$$= AX(ASU(\neg isCKill(s), isCGen(s))) \quad (4.9)$$

where

$$isCGen(s) = s$$

and

$$isCKill(s) = (isDefined(a) \lor isDefined(X)) \land \neg isCGen(s)$$

computes the set of states $n$ where every path from the initial node to $n$ contains the statement $a = X$ and subsequent to the last occurrence of $a = X$, there are no assignments to $X$. Thus, if $u$ satisfies this specification, the second property is satisfied.
Optimal Computation Points

Consider an expression $e = X \cdot Y$. A modal formula specifying the optimal computation point for $e$ can be derived directly from its DFA equation, 2.7, \cite{44,45}:

$$\text{isOCP}(e) = \text{isSafe}(e) \land \text{isEAR}(e)$$  \hspace{1cm} (4.10)

where

$$\text{isSafe}(e) = ASU(\neg \text{isModified}(e), \text{isUsed}(e))$$
$$\quad = \mu X.(\text{isUsed}(e) \lor \neg \text{isModified}(e) \land []X \land \langle \rangle \text{true})$$  \hspace{1cm} (4.11)

and

$$\text{isEAR}(e) = [\lnot \text{false} \lor \neg (\mu X.((\text{isSafe}(e) \land \neg \text{isModified}(e)) \lor \neg (\text{isModified}(e) \lor [\lnot \text{false} \land []X \land \langle \rangle \text{true}))))$$
$$\quad = [\lnot \text{false} \lor \neg (\mu X.\lnot (\text{isModified}(e) \lor [\text{false}))))$$
$$\quad \land \text{isSafe}(e) \land \neg \text{isModified}(e)))$$  \hspace{1cm} (4.12)

Remember that this analysis should be preceded on an appropriate placement model, i.e. artificial states have to be inserted into the program model like described in section 2.4.

4.3 From DFA Algorithms to Model Checking

Provided that a model checker, which can handle $\mu$-calculus formulas and an appropriate program model, is available, nothing has to be done for this last step (except, of course, giving the model checker input and launching it). While the classical iterative DFA algorithms are written for solving a particular DFA equation, model checkers provide generic fixed point algorithms that can handle different formulas.

Complexity As was revealed in section 3.3, the complexity for the checking with the semantic approach is

$$O((|T| \cdot |\phi|)^{ad})$$

where $|T|$ is the number of states and transitions in the model structure, $|\phi|$ is the size of the formula, and $ad$ refers to the alternation depth of the formula.

This is of course the complexity for the checking of one formula. Section 2.3 outlined, for classical data-flow analysis, how the properties for different elements can be computed simultaneously by using bit vectors. This procedure can be applied for DFA via model checking as well. Then the complexity increases proportionally with the number $n$ of entries in the bit vector:

$$O(n \cdot (|T| \cdot |\phi|)^{ad})$$

35
5 The jABC Framework

This chapter presents the framework that was used for the DFA-MC in this thesis: the Java Agent Building Center (jABC). A model checking Plugin was already available, while a functionality to generate control flow graphs from Java programs was not. An appropriate plugin for this problem was written therefore by the developers at the University of Dortmund: the Soot-based UnitGraph2SibGraph Converter. Both the model checker and the converter plugin are discussed in this chapter. For more detailed information on the jABC framework refer to [38].

5.1 Principles

The Java-based Java Agent Building Center Framework (abbreviated: jABC) picks up the principles that have already been used in the C++-based ABC since 1993 [32] and combines them with new ideas.

The typical feature of the ABC Frameworks is the usage of a graphical, high-level programming layer, where hierarchical, directed graphs can be constructed from special components, Service Independent Building Blocks (SIBs), that represent a unit of source code encoding a particular functionality. It is possible to define what programs do, and in what order, just by building such graphs from SIBs. Thus, an application designer doesn’t necessarily have to have knowledge about "real" programming, provided some programming experts have developed an appropriate set of SIBs before.

Figure 5.1 gives an impression of the jABC framework. It shows the canvas with a SIB-Graph on it, the SIB-Browser and SIB-Properties on the left, as well as the usual menu and toolbar.

5.2 The Model-Checking Plugin

The Model Checking Plugin for jABC provides all functionalities to check the validity of a formula expressed in the modal μ-calculus for a given model, and can also handle formulas based on CTL (for description of these specification languages see section 3.2). In this application, the model has to be represented by a Service Logic Graph (SLG) that is created and edited on the canvas in the main window. Thus, the plugin makes it possible to perform a formal verification for any SLG which is built via SIBs.
Figure 5.1: The jABC framework
The language that is understood by the parser can be described by the following Backus-Naur form (BNF):

\[
f ::= \ AP | \\
    \ TRUE | \ FALSE | \\
    \ f \&\& \ f | \ f || \ f | \\
    \ \neg \ f | \ IMP(f, f) | \\
    \ BOX(f) | \ BOX(CONSTRAINT, f) | \\
    \ BOX_BACK(f) | \ BOX_BACK(CONSTRAINT, f) | \\
    \ DIA(f) | \ DIA(CONSTRAINT, f) | \\
    \ DIA_BACK(f) | \ DIA_BACK(CONSTRAINT, f) | \\
    \ MIN(FIX, f) | \ MAX(FIX, f) | \\
    \ AG(f) | \ EG(f) | \\
    \ AF(f) | \ EF(f) | \\
    \ AX(f) | \ EX(f) | \\
    \ ASU(f, f) | \ ESU(f, f) | \\
    \ AWU(f, f) | \ EWU(f, f) \\
\]

\[
CONSTRAINT ::= BRANCH, CONSTRAINT | BRANCH \\
BRANCH ::= true | false | ... \\
AP ::= "apple" | "banana" | "pear" | ... \\
FIX ::= x | y | z | a | ... \\
\]

\(AP\) is short for an atomic proposition; atomic propositions should be enclosed in double quotes. The boolean connectives \(\land, \lor, \neg, \text{and } \implies\) are represented by \(\&\&, ||, \neg, \text{and } \IMP,\) respectively.

\(BOX\) and \(BOX\_BACK\) are the forward and backward box operators, \([a]\) and \([\overline{a}]\), whereas \(DIA\) and \(DIA\_BACK\) are the forward and backward diamond operators, \(\langle a \rangle\) and \(\langle \overline{a} \rangle\). \(CONSTRAINT\) describes the actions (BRANCHes) the box/diamond modalities refer to; it might be empty.

\(MIN/MAX\) represent the fixed point operators \(\mu/v\). \(FIX\) stands for a fixed point variable and must be exactly one character long. \(AG, EG, AF, \ldots\) describe the appropriate CTL operators.

Figure 5.2 illustrates what it looks like when a modal formula is verified on a SIB-Graph using the jABC Model Checking Plugin. Note that the jABC model checker works globally, i.e. the formula is checked for each node in the SIB-Graph.
Figure 5.2: Model Checking Plugin

Figure 5.3: UnitGraph2SibGraph Plugin
5.3 The UnitGraph2SibGraph Plugin

As stated before, for the purpose of DFA-MC a possibility to generate SIB-Graphs from Java-programs was needed. Soot is able to create control flow graphs from Java source code. These graphs are not SIB-Graphs and so can not be used as input for the jABC model checker. But it is possible to generate modelcheckable SIB-Graphs by fetching the data from the Soot Unit-Graph and converting it into a SIB-Graph.

As the UnitGraph2SibGraph Plugin is based on Soot, a brief introduction to the latter is given in this section, before describing the plugin further.

Soot

Soot [19, 53] is a framework for analyzing and transforming Java bytecode, with the aim of simplifying the process of developing new optimizations.

Soot is implemented in Java and provides a set of Java APIs for optimizing Java bytecode directly. It supports four intermediate representations for Java bytecode, each of which can be transformed into another: Baf (a streamlined representation of bytecode which is simple to manipulate), Jimple (a typed three-address intermediate representation suitable for optimization), Shimple (and SSA variation of Jimple) and Grimp (an aggregated version of Jimple suitable for decompilation). Jimple will be described further, as it is the three-address representation that is used in this work.

Optimizing Java bytecode in Soot consists of transforming Java bytecode subsequently to Baf, Jimple, Grimp, back to Baf and then to bytecode, and while in each representation, performing some appropriate optimization. Soot makes it possible to perform intraprocedural optimizations as well as whole program optimizations and annotations of Java bytecode.

Soot can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode. Soot is free software and is licensed under the GNU Lesser General Public License. The Soot Framework is an evolving research project at McGill University, Canada. The latest release is version 2.2.1.

The Jimple Representation  Jimple is one of the intermediate representations that are supported by Soot. Like all intermediate representations provided by Soot, it was developed to allow analyses to be performed on bytecode at the most appropriate level.

Jimple is a typed three-address code representation of bytecode, which is an ideal form for performing optimizations and analyses. Essentially, the stack (as it appears in Baf) has been eliminated and replaced by additional local variables (prefixed with $). The operators in Jimple are untyped, but the local variables are given explicit primitive, class, or interface types.

Figure 5.4 gives an example of a piece of Java code and its Jimple form [53].

Unit Graphs  This paragraph describes how a control-flow graph can be generated using Soot. First, the concepts of the SootMethod and Body classes have to be described. Very
briefly, the code that Soot works with is contained in the so-called SootClasses, which are similar to the usual Java classes, but allow of intermediate representations like Jimple. To these classes are then added some SootMethods (declaring method name, parameters, return type...), that again have to be equipped with Bodies (the actual source code). Having the SootClasses for the programs in question, several actions are possible, for example the creation of a control-flow graph.

Soot provides the UnitGraph class to describe the notion of a control-flow graph. There are two types of UnitGraphs: ExceptionalUnitGraph and BriefUnitGraph. The exceptional graph contains all of the edges in the brief graph, plus edges corresponding to potential exceptions being thrown. The UnitGraph class implements the DirectedGraph interface, which captures the essential features of directed graphs (on Directed objects). Now, given a Body b, an ExceptionalUnitGraph can be created by invoking new ExceptionalUnitGraph(b).

For data-flow analysis, each node in the control-flow graph is required to contain some data. In Soot, this data is represented as flow sets (a set of facts). Soot computes the DEF- and USE-sets for each Unit in the flow graph during the creation process (and besides it is possible to add further information to these flow sets).

The Converter Plugin

Having a UnitGraph created by Soot, it has to be converted into a jABC-compatible SIB-Graph. Each node of the UnitGraph provides the necessary information (e.g. branches, caption) via get() methods. Thus, the converter only needs to take a number of "Default Modelcheckable SIBs" (SIBs that don’t contain code, but hold a name, their branching information, and a set of atomic propositions), equip them with the information it can gather from the UnitGraph and set the branches accordingly.
With respect to the different program models that appear in literature, the converter offers to variants for the labeling of the SIB graph (compare figure 5.3). The first, called node action, labels the nodes with the statements and the branches with true/false (in case of conditional branches) or default (in case of unconditional branches). The atomic properties at the nodes refer to the program point where the statement is executed. The second, called branch action, moves the statements from the nodes into the outgoing branches. As the atomic properties remain in the nodes, a precondition model is created. A postcondition variant is not (yet) available, but as the models are actually equivalent, this is hardly of concern.

Furthermore, it is possible to instruct the converter to add certain atomic propositions to the SIBs (for that, the user can give a Java BeanShell Script that is interpreted by the converter at runtime). As mentioned above, Soot computes the DEF- and USE-sets for its UnitGraphs, so it is useful to give some code to the converter that adds the DEF- and USE-sets to each of the SIBs:

```java
import soot.Unit;
import java.util.*;
HashSet ap = new HashSet();
for(it = unit.getDefBoxes().iterator(); it.hasNext(); ){
    ap.add("isDef(" + it.next().getValue() + ")");
}
for(it = unit.getUseBoxes().iterator(); it.hasNext(); ){
    ap.add("isUse(" + it.next().getValue() + ")");
}
return ap;
```
6 Example

As it would be far too lengthy and confused to present all results that were gained during the work on this thesis, the procedure of the analysis is discussed in detail only for one small, though interesting enough, program. The program "fib.java" (Uglified Fibonacci Number Computing) that computes a sequence of Fibonacci Numbers, but additionally contains several superfluous statements, seemed appropriate for this purpose. It is introduced in section 6.1.

As the DFA formulas assume that there is a certain set of pre-computed atomic propositions (i.e. the DEF-, USE-, and MOD-information) for each program point, these had to be computed before starting with the "higher level" program analysis. Section 6.2 deals with this problem. Then, step by step, the following analyses were performed:

1. Live Variables
2. Very Busy Expressions
3. Reaching Definitions
4. Live Definitions
5. Available Expressions
6. Common Subexpressions
7. UD-Chains
8. DU-Chains
9. Propagatable Copies
10. Optimal Computation Points

The procedure and results are presented in section 6.3, before section 6.4 illustrates how the gathered information can be used for code optimization.

6.1 Program: Uglified Fibonacci Number Computing

The Fibonacci numbers form a sequence defined recursively by:

\[
Fib(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
Fib(n-1) + Fib(n-2) & \text{if } n > 1 
\end{cases}
\]
The computation of Fibonacci Numbers can be computed recursively as well as iteratively. For intraprocedural analysis, however, the iterative method is the better one (because with recursion there are many callings of methods whose interferences cannot be analyzed by means of intraprocedural analyses). This iterative version is additionally uglified (i.e. contains several obsolete instructions). The contents of the files "fib.java" (the original Java program) and "fib.jimple" (its Jimple form) are given in the following.

fib.java

public class fib {
    public static void main (String[] args) {
        int fib1 = 1;
        int fib2 = 1;
        int fib;
        int x = 0;

        System.out.println("Ich bin ein dummes Programm...");

        int a = 100;
        int b = 100;
        long i = 1000;
        a = 100-99;
        b = 100-99;

        i = (int)(i*3);

        for (i = 0; i<60; i=i+3)
        {
            fib = fib1 + fib2;
            System.out.println(fib1 + fib2);
            fib = fib1 + fib2;
            for (fib1 = 0; fib1 < x; fib1++)
            {
                System.out.println("Überflüssig");
            }

            fib1 = fib2;
            fib2 = fib;
        }
    }
}
fib.jimple
1: public class fib extends java.lang.Object
2: {
3:
4:    public void <init>()
5:    {
6:        fib this;
7:        this := @this: fib;
8:        specialinvoke this.<java.lang.Object: void <init>()>();
9:        return;
10:    }
11: }
12: public static void main(java.lang.String[])
13: {
14:    java.lang.String[] args;
15:    int fib1, fib2, x, fib, $11, $12;
16:    byte a, b, $b3;
17:    long i, $10;
18:    java.io.PrintWriter $r0;
19:    boolean a, b;
20:    args := @parameter0: java.lang.String[];
21:    fib1 = 1;
22:    fib2 = 1;
23:    x = 0;
24:    $r0 = <java.lang.System: java.io.PrintWriter out>;
25:    virtualinvoke $r0.<java.io.PrintWriter:
26:    void println(java.lang.String)>("Ich bin ein dummies Programm...");
27:    a = 100;
28:    b = 100;
29:    i = 1000L;
30:    a = 1;
31:    b = 1;
32:    $10 = i * 3L;
33:    $11 = (int) $10;
34:    i = (long) $11;
35:    i = 0L;
36:    goto label3;
37:    label0:
38:    fib = fib1 + fib2;
42:    $r0 = <java.lang.System: java.io.PrintStream out>;
43:    $i2 = fib1 + fib2;
44:    virtual invoke $r0.<java.io.PrintStream: void println(int)>($i2);
45:    fib = fib1 + fib2;
46:    fib1 = 0;
47:    goto label2;
48:    label1:
49:    $r0 = <java.lang.System: java.io.PrintStream out>;
50:    virtual invoke $r0.<java.io.PrintStream: 
51:        void println(java.lang.String)>("\u00dcberf\u00fcssig");
52:    fib1 = fib1 + 1;
53:    label2:
54:    if fib1 < x goto label1;
55:    fib1 = fib2;
56:    fib2 = fib;
57:    i = i + 3L;
58:    label3:
59:    $b3 = i cmp 60L;
60:    if $b3 < 0 goto label0;
61:    return;
62: } }
63: 
64: 
65: 
66: }
67: }

In order to cover all variables, expressions, and copy statements, a list of them is made before beginning with the analyses. They usually form the bit vectors, but as a bit-vector framework is currently not available within the jABC model checker, the properties have to be checked for separately for every quantity.

List of variables: fib,fib1,fib2,x,$i1,$i2,a,b,$b3,i,$l0,$r0,args

List of expressions: i * 3L,fib1 + fib2,i + 3L,fib1 + 1

List of copy statements: $i1 = (int) $l0,i = (long) $i1,fib1 = fib2,fib2 = fib
6.2 Program Model and Basic Properties

Figure 6.1 displays a node-action program model for "fib.jimple" (it is equivalent to the graph that was created by the converter plugin, but a screenshot out of the jABC was not very readable, so a substitute version was drawn). Note that only lines 23 through 65 are relevant for the graph model.

**isDef, isUsed, and isMod**

For the most part, DEF, USE, and MOD are presupposed for the formulas given in literature on DFA and are therefore not regarded as results of analyses. The SIBGraph in this application, however, has to be equipped with these properties first. As was revealed in section 5.3, Soot already provides the DEF- and USE-sets for its Units. The UnitGraph2SibGraph Converter is thus instructed (via the Beanshell script) to transfer them into the atomic properties isDef and isUsed. isMod is simply the disjunction of the isDefs of the expression's operators

\[ isMod(a + b) = isDef(a) \lor isDef(b) \]

and can thus be computed straightforwardly.

The following table portrays the isDef-, isUsed, and isMod-information for the fib-model. Like further tables that will appear, its columns refer to the property in question, while its rows indicate the variable/expression/statement... that is of concern. For example, the entry \{41, 45\} at first row, first column means that isDef(fib) holds at states 41 and 45.

A representation similar to bit-vector results may seem more convenient at first glance, but although bit-vectors can be stored efficiently, they are quite lengthy when printed out. Therefore the described structure was chosen for the tables; it seemed to be as less confused as possible.
6 Example

Figure 6.1: Node-Action Program Model for fib.jimple
### 6.3 Analyses and Results

**Live Variables**

The nodes on which `a` is live are determined by the following formula (compare 4.1):

\[
\text{MIN}(z, \ "\text{isUsed}(a)\" \ || \ !\text{isDef}(a) \ &\ & \text{DIA}(z))
\]

Verification yields the result that `a` isn't live at any state. Accordingly, the formula specifying `a` is dead,

\[
!\text{MIN}(z, \ "\text{isUsed}(a)\" \ || \ !\text{isDef}(a) \ &\ & \text{DIA}(z))
\]

yields that `a` is dead at all states.
<table>
<thead>
<tr>
<th>v</th>
<th>isLive(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>46–58</td>
</tr>
<tr>
<td>fib1</td>
<td>25–45, 47–56, 58–63</td>
</tr>
<tr>
<td>fib2</td>
<td>26–57, 59–63</td>
</tr>
<tr>
<td>x</td>
<td>27–63</td>
</tr>
<tr>
<td>$i1</td>
<td>36</td>
</tr>
<tr>
<td>$i2</td>
<td>44</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$b3</td>
<td>63</td>
</tr>
<tr>
<td>i</td>
<td>32–34, 38–63</td>
</tr>
<tr>
<td>$t0</td>
<td>35</td>
</tr>
<tr>
<td>$t0</td>
<td>28, 43, 44, 51</td>
</tr>
<tr>
<td>args</td>
<td></td>
</tr>
</tbody>
</table>

**Very Busy Expressions**

According to formula 4.2, the nodes on which \(i + 3L\) is very busy are specified by:

\[
\text{MAX}(z, \ "\text{isUsed}(i + 3L)" \ \| \ (\!\!\text{isMod}(i + 3L)" \ \&\& \ \text{BOX}(z) \ \&\& \ \!\!\text{tail}\))
\]

which results in the states 41–60.

<table>
<thead>
<tr>
<th>e</th>
<th>isVBE(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i * 3L</td>
<td>32–34</td>
</tr>
<tr>
<td>i + 3L</td>
<td>41–60</td>
</tr>
<tr>
<td>fib1 + fib2</td>
<td>41–45</td>
</tr>
<tr>
<td>fib1 + 1</td>
<td>50–52</td>
</tr>
</tbody>
</table>

**Available Expressions**

Formula 4.3 specifies the set of states on which an expression is available. Apart from isDef, isUsed, and isMod, it requires that isGen is computed beforehand. Considering the expression \(i \ast 3L\) for an example, this is done as follows.

"isUsed(i \ast 3L)" \&\& \!"isMod(i \ast 3L)"

specifies isGen(i \ast 3L). Having added this property to the states fulfilling this formula, the states at which isAvail(i \ast 3L) holds can be computed:

\[
\text{MAX}(z, \ \text{BOX}_{\text_BACK}((\!\!\text{head}" \ \&\& \ z \ \&\& \ \!\!\text{isMod}(i \ast 3L)")) \ \| \ (\!\!\text{isGen}(i \ast 3L)")) \ \&\& \ \!\!\text{head}
\]

The model checker computes that \(i \ast 3L\) is available at states 35 and 36 (\(i + 3L\) is available after its computation at 34, but as \(i\) is redefined at 37, it is not available any longer).
### 6 Example

<table>
<thead>
<tr>
<th>( e )</th>
<th>isAvail(( e ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \cdot 3l )</td>
<td>35, 36</td>
</tr>
<tr>
<td>( i + 3l )</td>
<td></td>
</tr>
<tr>
<td>( \text{fib}1 + \text{fib2} )</td>
<td>42-46</td>
</tr>
<tr>
<td>( \text{fib}1 + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

### Reaching Definitions

According to formula 4.4, the states which a definition, say the definition of \( \text{fib} \) at 41, are defined by

\[
\text{MIN}(z, \text{DIA\_BACK}((z \& \& \text{"isPreserved(fib)" || "41"}))
\]

where \( \text{isPreserved(fib)} \) is computed by

\text{"isDef(fib)" \&\& \text{MIN}(x, \text{DIA\_BACK}(x) || \text{isDef(fib)})}

<table>
<thead>
<tr>
<th>( a,s )</th>
<th>isReaching(( a,s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fib}, 41 )</td>
<td>42-45</td>
</tr>
<tr>
<td>( \text{fib}, 45 )</td>
<td>41,46-65</td>
</tr>
<tr>
<td>( \text{fib}1, 24 )</td>
<td>25-46</td>
</tr>
<tr>
<td>( \text{fib}1, 46 )</td>
<td>47,50-52,55,57</td>
</tr>
<tr>
<td>( \text{fib}1, 52 )</td>
<td>50-52,55,57</td>
</tr>
<tr>
<td>( \text{fib}1, 57 )</td>
<td>41-46,58-65</td>
</tr>
<tr>
<td>( \text{fib}2, 25 )</td>
<td>26-58,62-65</td>
</tr>
<tr>
<td>( \text{fib}2, 58 )</td>
<td>41-65</td>
</tr>
<tr>
<td>( x, 26 )</td>
<td>27-65</td>
</tr>
<tr>
<td>( $i1, 35 )</td>
<td>36-65</td>
</tr>
<tr>
<td>( $i2, 43 )</td>
<td>41-65</td>
</tr>
<tr>
<td>( a, 29 )</td>
<td>30-32</td>
</tr>
<tr>
<td>( a, 32 )</td>
<td>33-65</td>
</tr>
<tr>
<td>( b, 30 )</td>
<td>31-33</td>
</tr>
<tr>
<td>( b, 33 )</td>
<td>34-65</td>
</tr>
<tr>
<td>( $b3, 62 )</td>
<td>41-65</td>
</tr>
<tr>
<td>( i, 31 )</td>
<td>32-36</td>
</tr>
<tr>
<td>( i, 36 )</td>
<td>37</td>
</tr>
<tr>
<td>( i, 37 )</td>
<td>38--65</td>
</tr>
<tr>
<td>( i, 59 )</td>
<td>41-65</td>
</tr>
<tr>
<td>( $10, 34 )</td>
<td>35-65</td>
</tr>
<tr>
<td>( $r0, 27 )</td>
<td>28-42,62-65</td>
</tr>
<tr>
<td>( $r0, 42 )</td>
<td>41-50,55-65</td>
</tr>
<tr>
<td>( $r0, 50 )</td>
<td>41,42,50-65</td>
</tr>
<tr>
<td>( \text{args}, 23 )</td>
<td>24-65</td>
</tr>
</tbody>
</table>
**Live Definitions**

The set of states where the definition of \( \text{fib} \) at state 45 is live, i.e. \( \text{isLiveDef(fib, 41)} \) is computed by

"\text{isReaching(fib, 45)}" \& \& "\text{isLive(fib)}"

The result for this is the set \([46, 50-58]\).

<table>
<thead>
<tr>
<th>(a, \text{s} )</th>
<th>(\text{isLiveDef(a, s)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{fib, 41})</td>
<td></td>
</tr>
<tr>
<td>(\text{fib, 45})</td>
<td>41, 50-58</td>
</tr>
<tr>
<td>(\text{fib1, 24})</td>
<td>25-45, 62, 63</td>
</tr>
<tr>
<td>(\text{fib1, 46})</td>
<td>47-55</td>
</tr>
<tr>
<td>(\text{fib1, 52})</td>
<td>50-55</td>
</tr>
<tr>
<td>(\text{fib1, 57})</td>
<td>41-45, 58-63</td>
</tr>
<tr>
<td>(\text{fib2, 25})</td>
<td>26-57, 62, 63</td>
</tr>
<tr>
<td>(\text{fib2, 58})</td>
<td>41-57, 59-63</td>
</tr>
<tr>
<td>(x, 26)</td>
<td>27-63</td>
</tr>
<tr>
<td>($i1, 35)</td>
<td>36</td>
</tr>
<tr>
<td>($i2, 43)</td>
<td>44</td>
</tr>
<tr>
<td>(a, 29)</td>
<td></td>
</tr>
<tr>
<td>(a, 32)</td>
<td></td>
</tr>
<tr>
<td>(b, 30)</td>
<td></td>
</tr>
<tr>
<td>(b, 33)</td>
<td></td>
</tr>
<tr>
<td>($b3, 62)</td>
<td>63</td>
</tr>
<tr>
<td>(i, 31)</td>
<td>32-34</td>
</tr>
<tr>
<td>(i, 36)</td>
<td></td>
</tr>
<tr>
<td>(i, 37)</td>
<td>38-63</td>
</tr>
<tr>
<td>(i, 59)</td>
<td>41-63</td>
</tr>
<tr>
<td>($10, 34)</td>
<td>35</td>
</tr>
<tr>
<td>($r0, 27)</td>
<td>28</td>
</tr>
<tr>
<td>($r0, 42)</td>
<td>43, 44</td>
</tr>
<tr>
<td>($r0, 50)</td>
<td>51</td>
</tr>
<tr>
<td>(\text{args}, 23)</td>
<td></td>
</tr>
</tbody>
</table>

**Common Subexpressions**

For answering the question whether \(\text{fib1} + \text{fib2}\) is an expression that can be eliminated, the first step is to find all statements \(s\) of the form \(a = \text{fib1} + \text{fib2}\) such that \(\text{fib1} + \text{fib2}\) is available at \(s\). This is the set \([43, 45]\), computed by

"\text{isUsed(fib1 + fib2)}" \& \& "\text{isAvail(fib1 + fib2)}"
Second step is to determine, for all elements \( s \) of this set, all definitions that reach \( s \) and which have \( \text{fib1} + \text{fib2} \) on the right. This can be done by examining the reaching definitions information at \( s \): for each definition \( d \) that reaches \( s \), it has to be checked whether \( \text{fib1} + \text{fib2} \) is used, i.e., by

"isUsed(fib1 + fib2)" & & "d"

The results are \([41]\) for \( s = 43 \) and \([41,43]\) for \( s = 45 \).

Thus, \( \text{fib1} + \text{fib2} \) is a common subexpression at states 41, 43 and 45, and the transformation for common subexpression elimination can start as given in chapter 2. \( \text{fib1} + \text{fib2} \) is also the only common subexpression in \( \text{fib.jimple} \).

**UD-Chains**

The formula for computing the ud-chain for the use of \( \text{fib1} \) at state 55, i.e., \( \text{udChain(fib1, 55)} \), is (compare 4.7):

"isDef(fib1)" & & EX(MIN(x, ("isUsed(fib1)" & & "55") ||
      !("isDef(fib1)" & & DIA(x))))

The computed ud-chain consists of the states 46 and 52.

<table>
<thead>
<tr>
<th>u,s</th>
<th>udChain(u,s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib, 58</td>
<td>45</td>
</tr>
<tr>
<td>fib1, 41</td>
<td>24, 57</td>
</tr>
<tr>
<td>fib1, 43</td>
<td>24, 57</td>
</tr>
<tr>
<td>fib1, 45</td>
<td>24, 57</td>
</tr>
<tr>
<td>fib1, 52</td>
<td>46, 52</td>
</tr>
<tr>
<td>fib1, 55</td>
<td>46, 52</td>
</tr>
<tr>
<td>fib2, 41</td>
<td>25, 58</td>
</tr>
<tr>
<td>fib2, 43</td>
<td>25, 58</td>
</tr>
<tr>
<td>fib2, 45</td>
<td>25, 58</td>
</tr>
<tr>
<td>fib2, 57</td>
<td>25, 58</td>
</tr>
<tr>
<td>x, 55</td>
<td>26</td>
</tr>
<tr>
<td>$i1, 36</td>
<td>35</td>
</tr>
<tr>
<td>$i2, 44</td>
<td>43</td>
</tr>
<tr>
<td>$b3, 63</td>
<td>62</td>
</tr>
<tr>
<td>i, 34</td>
<td>31</td>
</tr>
<tr>
<td>i, 59</td>
<td>37, 59</td>
</tr>
<tr>
<td>i, 62</td>
<td>37, 59</td>
</tr>
<tr>
<td>$10, 35</td>
<td>34</td>
</tr>
<tr>
<td>$r0, 28</td>
<td>27</td>
</tr>
<tr>
<td>$r0, 44</td>
<td>42</td>
</tr>
<tr>
<td>$r0, 51</td>
<td>50</td>
</tr>
</tbody>
</table>
DU-Chains

Consider the definition of fib2 at line 58. According to formula 4.8 The du-chain for this definition is computed by

\[
\text{"isUsed(fib2)" \\& DIA_BACK(MIN(x, ("isDef(fib2)" \\& "58") || \\(!"isDef(fib2)" \\& DIA_BACK(x))))}
\]

which results in duChain(fib2, 58) = \{41,43,45,57\}.

<table>
<thead>
<tr>
<th>d,s</th>
<th>duChain(d,s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib, 41</td>
<td></td>
</tr>
<tr>
<td>fib, 45</td>
<td>58</td>
</tr>
<tr>
<td>fib1, 24</td>
<td>41, 43, 45</td>
</tr>
<tr>
<td>fib1, 46</td>
<td>52, 55</td>
</tr>
<tr>
<td>fib1, 52</td>
<td>52, 55</td>
</tr>
<tr>
<td>fib2, 57</td>
<td>41, 43, 45</td>
</tr>
<tr>
<td>fib2, 25</td>
<td>41, 43, 45, 57</td>
</tr>
<tr>
<td>fib2, 58</td>
<td>41, 43, 45, 57</td>
</tr>
<tr>
<td>x, 26</td>
<td>55</td>
</tr>
<tr>
<td>$i1$, 35</td>
<td>36</td>
</tr>
<tr>
<td>$i2$, 43</td>
<td>44</td>
</tr>
<tr>
<td>a, 29</td>
<td></td>
</tr>
<tr>
<td>a, 32</td>
<td></td>
</tr>
<tr>
<td>b, 30</td>
<td></td>
</tr>
<tr>
<td>b, 33</td>
<td></td>
</tr>
<tr>
<td>i, 31</td>
<td>34</td>
</tr>
<tr>
<td>i, 36</td>
<td></td>
</tr>
<tr>
<td>i, 37</td>
<td>59, 62</td>
</tr>
<tr>
<td>i, 59</td>
<td>59, 62</td>
</tr>
<tr>
<td>$r0$, 34</td>
<td>35</td>
</tr>
<tr>
<td>$r0$, 27</td>
<td>28</td>
</tr>
<tr>
<td>$r0$, 42</td>
<td>43</td>
</tr>
<tr>
<td>$r0$, 50</td>
<td>51</td>
</tr>
<tr>
<td>args, 23</td>
<td></td>
</tr>
</tbody>
</table>

Copy Propagation Information

For an example, consider the question whether the statement 35: \$i1 = (\text{int}) \$10 can be eliminated by means of copy propagation. It can, if two properties are satisfied (compare section 2.4).

First, the udChain for every use of \$i1 must only consist of 35. This is the case, as \$i1 is used only once: in line 36 (compare to use-results).
Second, on every path from 35 to a use of \$i1, there are no assignments to \$10. Obviously this is also the case, but it shall be confirmed using formal specifications.

\[
\text{MAX}(x, \text{BOX\_BACK}((x && !"head" && !"isCkll($i1 = (int) $10)" ||
  "isCgen($i1 = (int) $10)")) && !"head"
\]

where isCgen($i1 = (int) $10) is computed by

"$i1 = (int) $10"

(using the SIB name as atomic proposition) and isCkll($i1 = (int) $10) by

("isDef($i1)" || "isDef($10)" && !"isCgen($i1 = (int) $10)"

confirms this, as it is evaluated to true for all states after 35 and thus includes the use of $i1 at 36.

<table>
<thead>
<tr>
<th>$</th>
<th>propagatable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>true</td>
</tr>
<tr>
<td>36</td>
<td>false</td>
</tr>
<tr>
<td>57</td>
<td>false</td>
</tr>
<tr>
<td>58</td>
<td>false</td>
</tr>
</tbody>
</table>

**Optimal Computation Points**

Remember that, as revealed in section 2.4, determining optimal computation points requires an appropriate placement model. The artificial states can not (yet) be inserted automatically, so this has done manually on the jABC canvas. Then, to determine the optimal computation points for the expression fib1 + fib2, formulas 4.10, 4.11, and 4.12 have to be transferred into the syntax of the jABC model checker.

isSAFE(fib1 + fib2) is computed by

\[
\text{MIN}(x, "isUsed(fib1 + fib2)" || (!("isMod(fib1 + fib2)" || tail) &&
  \text{BOX}(x) && \text{DIA}(\text{TRUE})))
\]

isEAR(fib1 + fib2) by

"head" || !BOOX\_BACK(MIN(x, (isSAFE(fib1 + fib2) && !"isMod(fib1 + fib2)"

| head || !BOOX\_BACK(\text{MIN}(x, \text{isSAFE(fib1 + fib2)} && !"isMod(fib1 + fib2)"

and, finally, isOCP(fib1 + fib2) by

"isEAR(fib1 + fib2)" && "isSAFE(fib1 + fib2)"

The state satisfying the OCP-property is 41.

The optimal computation points for all expressions of the program are
### 6.4 Derived Optimizations

Finally, figure 6.2 shows how the results gathered in the previous section can be used for code transformations. Red and crossed trough parts denote (parts of) statements that are removed, green parts denote (parts of) statements that are inserted.

The following observations can be made regarding the transformations in `fib.jimple`, which are straightforward from the definitions of the optimizations (compare chapter 2).

- Dead code is recognized, and can simply be eliminated.
- Copies which can be eliminated by copy propagation are made out.
- Not all possible optimizations are covered. For example, the statement at line 34, which becomes dead after its only use at line 36 is eliminated, is not recognized as such, because the framework does not (yet) support transformations.
- Common subexpressions are recognized, and can be eliminated by applying common subexpression elimination.
- Not every transformation is an optimization, it is even possible that after the transformations more operations than before take place. For example, consider the optimal computations points at lines 49 and 56, where actually one computation and one assignment is replaced by one computation and two assignments.

This is a general problem with automatic code optimization [1, 2]. Compilers often try to avoid such transformations (by consulting additional information), but it can’t be guaranteed that every possible case is recognized.

<table>
<thead>
<tr>
<th>e</th>
<th>isOCP(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fib1 + fib2</code></td>
<td>41</td>
</tr>
<tr>
<td><code>fib1 + 1</code></td>
<td>50</td>
</tr>
<tr>
<td><code>i * 3L</code></td>
<td>32</td>
</tr>
<tr>
<td><code>i + 3L</code></td>
<td>57</td>
</tr>
</tbody>
</table>
Figure 6.2: Derived Optimizations
7 Further Results

The previous chapter discussed in detail how DFA-MC with the jABC framework is done and which information is received. This chapter introduces some further programs from different areas of application and compares them with regard to possible optimizations and therewith the preceding DFA results. The first program to be presented, Router, was written at the University of Verona, the remaining, some matrix multiplication and string algorithm implementations, originated in exercises for lectures held at the University of Göttingen. For all programs (and especially for the implementations of algorithms), it is important to note that a compiler cannot improve the algorithm itself, but only its implementation. In order to reduce complexity, the algorithm itself has to be improved.

7.1 Router Simulation

The package Router [18] is a system for the processing of information. It was developed by F. Fummi and F. Banterle at the University of Verona not only for the purpose of a routing simulation, but also for serving as a benchmark for optimizing compilers. The package consists of the following classes:

- **Router**
  Implementation of a router that provides a queue, a routing table, an output and several optional functionalities.

- **Queue**
  Implementation of a FIFO queue for buffering packets.

- **packet_t**
  A data structure containing the information destination, id, data, and checksum. A string representation of the packet information is used for sending.

- **RouterThread**
  A class providing a thread that continuously processes the packets arrived at the queue and visualizes the following routing process.

- **PacketGenerator**
  A class providing a thread that continuously generates packets and sends them to the router.

- **Consumer**
  A class providing a thread that continuously consumes packets received from the router.
7 Further Results

• RouterTest

A class containing a main-method that creates four PacketGenerators, four Consumers, and one RouterThread and thus simulates a routing system.

The classes contained in the package Router consist of many small methods, thus intraprocedural analyses can only cover small parts of the system. As most operations are on objects, and only few computations take place, analyses that aim at computations don’t find much here, either. The source for analyses are thus mostly copy and assignment statements. Besides some copies that can be eliminated by copy propagation (in Router and Queue), there are some dead assignments (or dead variables) that can also be removed from the program (in packet_t, RouterThread, and RouterTest), and even a few common subexpressions that can be eliminated (in Router and Consumer).

7.2 Strassen Algorithm: Fast Matrix Multiplication

The Strassen algorithm [51], named after the mathematician Volker Strassen, is used for matrix multiplication and is faster than the standard matrix multiplication algorithm. Standard matrix multiplication takes $n^{\log_{2}8} = n^{3}$ multiplications, Strassen’s algorithm reduces the number of multiplications to $n^{\log_{2}7} \approx n^{2.807}$ (additions are not counted, because they are much faster than multiplications [12]).

The Strassen algorithm uses a divide-and-conquer strategy for the multiplication of two matrices, thus there is plenty administration overhead before the actual computations and again the intraprocedural procedure only allows of analyses of small parts of the program. There are many copy statements, but hardly any can be eliminated by copy propagation, as at most part they are to local temporaries in preparation of some special invoke operation. Many expressions can be found due to the multiplications and additions, but also due to the coping with indexes. Several can be eliminated by means of common subexpression elimination, for others better (i.e. optimal) computations points can be determined.

7.3 Some Algorithms on Strings

The string algorithm implementations dealt with in this section are both written for the purpose of processing DNA sequences, i.e. for an application in bioinformatics. They can, however, be used for similar string problems (that are not based on DNA fragments) after few modifications.

GreedyKGO

GreedyKGO is an implementation of an algorithm for finding the shortest common superstring for a set of strings, which uses a greedy strategy. A string $S$ is a superstring of a string $T$, if $T$ is a substring of $S$. Now, KGO (abbreviation for German "kürzester gemeinsamer Oberstring") means "shortest common superstring", i.e. a string that is
7 Further Results

1. a superstring of a set of strings and
2. the shortest such string.

The problem is NP-complete, so the greedy strategy does not solve the problem in general, but computes satisfying results in most cases.

It works as follows: Within the set of strings for which the shortest common superstring is to be found, those two strings with the greatest overlap (prefix of one string identical to the suffix of another) are determined and accordingly united to one string. This procedure is repeated until only one string is left. As this implementation computes a superstring for a set of DNA fragments that could originate in both DNA strands, not only the given fragments but also their reverse complements (i.e. in reverse order and with the complementary bases) are considered for computing the greatest overlap. Accordingly, a string might have to be reversed before uniting it with another.

As this implementation uses many structures the Java library provides, such as String objects and their functionalities for the operations on DNA fragments, and Vectors for holding the sets, only few computations are done "by hand" and thus little is recognizable for an analysis. What can be found are some common subexpressions and a few better computation points for expressions that modify indexes.

Viterbi Algorithm

The Viterbi algorithm, named after its developer Andrew Viterbi, is a dynamic programming algorithm for finding the most likely sequence of hidden states — known as the Viterbi path — that result in a sequence of observed events, especially in the context of hidden Markov models. The Viterbi algorithm was originally conceived as an error-correction scheme for noisy digital communications links. It is now also commonly used in, for example, information theory, speech recognition, and bioinformatics. Details would be beyond the scope of this thesis, a description of the algorithm can be found in [17]. This implementation of the Viterbi algorithm originates in bioinformatics. It computes the most likely alignment between two given DNA sequences.

Typical operations in this implementations are similar to those in the previous one (i.e. String operations, operations on objects), but as the Viterbi algorithm has to compute many probabilities, there are also many expressions throughout the program. Therefore, several common subexpressions and better computation points can be determined.
8 Conclusion

This thesis gave an introduction to the basic principles of intraprocedural data-flow analysis via model checking and showed how it can be translated into action by using the jABC framework. This conclusion gives a summary before discussing some difficulties and limitations that partly imply directions for future work.

Summary

Chapter 2 revealed basic principles of intraprocedural data-flow analysis. The notion of control-flow graphs was introduced, followed by a description of four basic data-flow problems: Available Expressions, Reaching Definitions, Live Variables and Very Busy Expressions. After a brief introduction to the approximate, iteratively computed solutions of data-flow equations, the chapter ended with an overview of selected further data-flow analysis problems and derived data-flow information.

Chapter 3 gave an introduction to model checking, describing each of the three basic tasks of model checking, i.e. modeling, specification, and verification. Three types of models, all directed graph structures, were discussed: Kripke structures, labeled transition systems and Kripke transition systems Two specification languages were introduced: the modal \( \mu \)-calculus, a low-level temporal logic, and the Computation Tree Logic CTL, a high-level temporal logic, which is a subset of the \( \mu \)-calculus regarding expressiveness. Finally, one algorithm for verification was presented: the semantic approach, that calculates fixed points iteratively.

After that, chapter 4 revealed the connection between data-flow analysis and model checking: model checkers can be seen as DFA algorithms that have the program property of interest as a parameter. It was discussed what appropriate program models look like and from what they can be derived, how data-flow equations and information about the desired solution translate into model checking formulas, and how the classical iterative data-flow algorithms are substituted by the model checker.

Then, chapter 5 presented the jABC framework, especially those functionalities that are relevant for the process of DFA-MC: the Model Checking Plugin, which provides a model checker for the modal \( \mu \)-calculus, and the UnitGraph2SibGraph Plugin, which is based on Soot and creates jABC-compatible flow graphs from Java classes.

Chapter 6 gave an example for the process of DFA-MC with the jABC. A flow graph was created for the program fib.java, an uglified implementation for Fibonacci Number Computing, and afterwards the previously discussed data-flow problems were investigated on this graph, ending with an overview of possible optimizations. Finally, chapter 7 revealed results for further programs.
Discussion and Future Work

As literature on program analysis via model checking mainly consists of articles or conference contributions, unfortunately the notation and the definitions of properties are hardly standardized. Above all, there were great difficulties in finding out details, while the general ideas became quickly clear. After all, these starting difficulties could coped with, and others such as the frequent out-of-memory-errors of the JVM or problems with the jABC framework could also be solved.

As soon as results were available, it was tried to verify their correctness using Soot. This is not possible in any case, as the Soot framework provides functionalities for optimizing transformations, not for the analyses alone. Among the results that can be checked directly are reaching definitions (due to Soot’s ReachingDefsTagger), live, or dead, variables (Dead Assignment Eliminator), propagatable copies (Copy Propagator) and common subexpressions (Common Subexpression Eliminator). They confirmed the results received by DFA-MC. Furthermore, the correctness of information like, for example, ud- and du-chains can be guessed by examining transformations like the Local Facker and Local Splitter, that are based on du-ud-webs. Other information is quite well "hidden" behind Soot’s transformations and don’t allow of satisfactory comparisons, yet other information is not available at all.

This shows, on the other hand, that the process of data-flow analysis via model checking is more flexible compared to classical DFA. Extending Soot with further, even small, functionality requires the implementation of an entire class or at least method, while with DFA-MC only a new modal specification is needed.

There are restrictions in the current setting, however (partly providing input for future work). The process of DFA-MC with the jABC is currently quite arduous, as every property has to be checked separately and the results have to be annotated to the graph manually. A solution to the latter problem is currently being developed: the next version of the jABC Model Checking Plugin will include a functionality to add a certain atomic proposition to all nodes satisfying a formula. A solution for the former requires the implementation for a bit-vector functionality and is at least thought of.

Another limitation is that the analyses do not go very deep: while optimizing compilers analyze the procedure, transform it, analyze it again and so on, the framework so far only provides means for one step of analyses. Hence it would be useful to equip the framework with a functionality to perform transformations on the graph (trying to do this by hand is not efficient and above all error-prone).

A general problem with intraprocedural program analysis is that it does not take into account the quite expensive procedure calls. Interprocedural analysis can enable more efficient optimizations regarding this, and is actually desired for a language like Java, that is designed for the use of many, often small, procedures. [20] suggests that interprocedural data-flow analysis can be achieved by suitable (but careful) preparation with a divide-and-conquer strategy (a procedure-by-procedure analysis) and restricting attention to intraprocedural data-flow analysis.
Acknowledgements

Many thanks to Prof. Dr. Tiziana Margaria and Prof. Dr. Bernhard Steffen for supervision, to Marc Njoku, Marco Bakera and Clemens Renner for programming and technical support, and to Florian Tegeler and Christian Dickmann for benchmarks, proofreading and helpful comments.
Bibliography


Bibliography


65


# List of Figures

1.1 Compilation steps ............................................. 4

2.1 CFG example .................................................. 7
2.2 Live variables example ........................................ 8
2.3 Very busy expressions example ................................ 9
2.4 Available expressions example ................................. 10
2.5 Reaching definitions example .................................. 11
2.6 The four types of basic DFA problems ....................... 12
2.7 Common subexpressions example ............................ 13
2.8 Common subexpression elimination example ................ 14
2.9 Use-Definition Chains example ............................... 14
2.10 Definition-Use Chains example ............................. 15
2.11 Copy Propagation example ................................... 16
2.12 Optimal computation points example ...................... 17

3.1 Kripke structure .............................................. 20
3.2 Labeled Transition System .................................... 21
3.3 Kripke Transition System ..................................... 22
3.4 Semantics of the $\mu$-calculus ............................... 23
3.5 CTL forward operators and $\mu$-calculus equivalents ...... 25
3.6 CTL backward operators and $\mu$-calculus equivalents .. 26
3.7 Kripke Transition System .................................... 27

4.1 DFA-MC ......................................................... 30

5.1 The jABC framework .......................................... 37
5.2 Model Checking Plugin ....................................... 39
5.3 UnitGraph2SibGraph Plugin ................................. 39
5.4 Original Java and Jimple .................................... 41

6.1 Node-Action Program Model for fib_jimple ................ 48
6.2 Derived Optimizations ....................................... 57