JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Daniela Ritirc, BSc

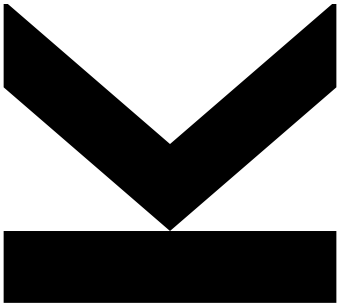
Angefertigt am
Research Institute for Symbolic Computation

Beurteiler
Univ.-Prof. DI Dr. Franz Winkler

Mitbetreuung
A. Univ.-Prof. DI Dr. Wolfgang Schreiner

Jänner 2016

Formally Modeling and Analyzing Mathematical Algorithms with Software Specification Languages & Tools



Masterarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Masterstudium

Computermathematik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 26. Jänner 2016 *Daniela Ritirc, BSc*

Abstract

In this thesis the behaviour of software specification languages and tools on mathematical algorithms shall be investigated. The main goal is to investigate how tools which have been designed for modeling and analyzing software in other application contexts can be applied to mathematical algorithms. For this purpose, two different mathematical algorithms, namely the DPLL method and Dijkstra's Shortest Path Algorithm are selected. Furthermore five well-known software specification languages are selected: JML, Alloy, TLA/PlusCal, VDM and Event-B. It shall be examined how far the algorithms can be modeled and how far model checking respectively verification succeeds. The goal of the thesis is not a proper verification/check of every model with every tool but a survey of the potential as well as the difficulties of the usage of software specification languages for the analysis of mathematical algorithms.

As a starting point for each algorithm a formal specification is derived and the algorithms are supplied in pseudo-code. A Java prototype is implemented for each algorithm which is then specified by JML annotations. Furthermore the algorithms are modelled in TLA/PlusCal, Alloy, VDM and Event-B and for each language the appropriate analysis supported by the tool is selected (visualizing, model checking, verification).

The main result of the thesis is that each tool shows some success when it is used for specifying and analyzing mathematical algorithms, because modeling the algorithms succeeded in every language. In TLA, VDM and Alloy it was possible to completely model check the specifications. Furthermore it was possible to visualize the algorithms in Alloy. In JML and Event-B it was possible to verify major parts of the model; however some branches of the proofs could not be completed by the included provers.

Zusammenfassung

In dieser Masterarbeit wird die Anwendung von Softwarespezifikationssprachen und Tools für mathematische Algorithmen untersucht. Das Hauptziel ist es zu untersuchen wie diese Tools, welche für die Modellierung und Analyse von Software in anderen Anwendungsbereichen konzipiert sind, für die Anwendung für mathematische Algorithmen geeignet sind. Zu diesem Zweck werden zwei verschiedene mathematische Algorithmen, nämlich DPLL-Algorithmus und Dijkstra-Algorithmus, ausgewählt. Die verwendeten Softwarespezifikationssprachen sind JML, TLA, Alloy, VDM und Event-B. Es wird ermittelt wie weit die Modellierung der gewählten Algorithmen möglich ist und wie weit Model Checking beziehungsweise Verifikation gelingt. Das Ziel dieser Arbeit ist nicht die vollständige Verifikation von jedem Modell in jedem Tool, sondern eine genaue Auflistung der Potentiale und Schwierigkeiten von der Anwendung von Softwarespezifikationssprachen auf mathematische Algorithmen.

Zu Beginn wird für jeden Algorithmus eine formale Spezifikation definiert und der Algorithmus wird in Pseudo-Code bereitgestellt. Für jeden Algorithmus wird ein Prototyp in Java implementiert, für welchen eine JML Spezifikation entwickelt wird. Weiters werden die Algorithmen in TLA/PlusCal, Alloy, VDM und Event-B modelliert und es wird für jede Spezifikationssprache die vom zugehörigen Tool unterstützte Analysemethode ausgewählt (Visualisierung, Model Checking, Verifikation).

Das Fazit dieser Thesis ist, dass in jeder Sprache ein gewisser Erfolg erkennbar ist, wenn sie für die Modellierung und Analyse von mathematischen Algorithmen verwendet wird, da die Modellierung der Algorithmen in jeder Sprache erfolgreich war. In TLA, VDM und Alloy war ein kompletter Model Check der Algorithmen möglich. Weiters war es in Alloy möglich die Algorithmen zu visualisieren. In JML und Event-B konnte ein Großteil des Models verifiziert werden; jedoch konnten einige Beweise von den inkludierten Beweisern nicht abgeschlossen werden.

Contents

1. Introduction **2**

2. Software Specification Languages & Tools **5**
2.1. Java Modeling Language . . . 5
2.2. Temporal Logic of Actions . . . 8
2.3. Alloy . . . 13
2.4. Vienna Development Method . . . 16
2.5. Event-B . . . 18
2.6. Comparison of the Languages and Tools . . . 22

3. DPLL Algorithm **24**
3.1. Informal Problem and Recursive Algorithm . . . 24
3.2. Iterative Algorithm and Correctness . . . 26
3.3. Formal Specification . . . 27
3.3.1. Input Condition . . . 28
3.3.2. Output Condition . . . 28
3.4. Java Program . . . 29
3.5. JML . . . 31
3.5.1. Version 1 . . . 31
3.5.2. Version 2 . . . 33
3.5.3. Version 3 . . . 35
3.6. TLA . . . 39
3.7. Alloy . . . 43
3.7.1. Model 1 . . . 43
3.7.2. Model 2 . . . 45
3.7.3. Model 3 . . . 47
3.8. VDM . . . 48
3.8.1. Recursive Algorithm . . . 48
3.8.2. Iterative Algorithm . . . 53
3.9. Event-B . . . 56
3.9.1. Abstract Model . . . 56
3.9.2. Refined Model . . . 61
3.10. Summary of the Results . . . 64

4. Dijkstra's Shortest Path Algorithm **66**
4.1. Informal Problem . . . 66
4.2. Formal Specification . . . 68
4.2.1. Input Condition . . . 68
4.2.2. Output Condition . . . 69
4.2.3. Correctness of the Algorithm . . . 70

4.3. Java Program . . . 75
4.4. JML . . . 76
4.4.1. Version 1 . . . 76
4.4.2. Version 2 . . . 77
4.5. TLA . . . 79
4.6. Alloy . . . 82
4.6.1. Model 1 . . . 82
4.6.2. Model 2 . . . 86
4.7. VDM . . . 87
4.8. Event-B . . . 93
4.9. Summary of the Results . . . 98

5. Conclusion **99**

Bibliography **102**

A. Model Listings for DPLL Algorithm **106**
A.1. Java . . . 106
A.2. JML . . . 108
A.2.1. Version 1 . . . 108
A.2.2. Version 2 . . . 113
A.2.3. Version 3 . . . 119
A.3. TLA . . . 126
A.4. Alloy . . . 128
A.4.1. Model 1 . . . 128
A.4.2. Model 2 . . . 129
A.4.3. Model 3 . . . 130
A.5. VDM . . . 132
A.5.1. Recursive Model . . . 132
A.5.2. Iterative Model . . . 133
A.6. Event-B . . . 135
A.6.1. Context: VarDefinition . . . 135
A.6.2. Context: ProgramCounter . . . 135
A.6.3. Machine: Algorithm . . . 136
A.6.4. Context: VarDefRefined . . . 137
A.6.5. Machine: AlgorithmRef . . . 138

B. Model Listings for Dijkstra's Algorithm **140**
B.1. Java . . . 140
B.2. JML . . . 141
B.2.1. Version 1 . . . 141
B.2.2. Version 2 . . . 145
B.3. TLA . . . 151
B.4. Alloy . . . 154
B.4.1. Model 1 . . . 154
B.4.2. Model 2 . . . 155
B.5. VDM . . . 156

B.6. Event - B 159
B.6.1. Context: VarDefinition 159
B.6.2. Context: ProgramCounter 159
B.6.3. Machine: Algorithm 160
B.6.4. Generated Proof Obligations 163

List of Figures

2.1. Output from Java KeY for the method `search` 8
2.2. Setting up a model in the TLA Toolbox 12
2.3. Output of the TLC Model Checker 12
2.4. Output of the Alloy Analyzer 15
2.5. Personalized visualization of a model in the Alloy Analyzer 16
2.6. Proof obligation generation for class Stack 18
2.7. Output for the first machine 20
2.8. Output for the refined machine 22
2.9. Comparison of the Languages and Tools 23

3.1. Proving containsEmptyClause with the simplified specification in KeY . . 37
3.2. Proof tree of DPLLiter in KeY 37
3.3. Output for every function in KeY 39
3.4. Setting up the model in the TLC Model Checker 42
3.5. Output of the TLC Model Checker 43
3.6. Instance of a formula which is satisfiable 45
3.7. Run of the DPLL algorithm in Alloy 47
3.8. Output of the Alloy Analyzer 48
3.9. Result after executing all test cases 51
3.10. Result of a specific test case 51
3.11. Generated proof obligations 52
3.12. Generated proof obligations 55
3.13. State diagram of the program counter 57
3.14. Generated proof obligations 59
3.15. Proving perspective 60
3.16. Generated proof obligations 64
3.17. Results of the Languages and Tools for DPLL method 65

4.1. Example for a graph 68
4.2. Setting up the model in the TLC Model Checker 81
4.3. Example Graph 84
4.4. A visualization of a run of Dijkstra's Algorithm in Alloy 85
4.5. Output of the Alloy Analyzer 87
4.6. Result after executing all test cases 90
4.7. Result of a specific test case 90
4.8. Generated proof obligations 91
4.9. Proof obligation types 92
4.10. State diagram of the program counter 94
4.11. Unproved proof obligations 96
4.12. Proof tree for "VnotinC/chooseV_2/INV" 97

4.13. Proof tree for “TerminationOnV/chooseU_4/INV” 97
4.14. Results of the Languages and Tools for Dijkstra’s Algorithm 98

B.1. Generated proof obligations 164
B.2. Generated proof obligations - continued 165
B.3. Generated proof obligations - continued 166
B.4. Generated proof obligations - continued 167

1. Introduction

A life without software is no longer imaginable, software and programs are used nearly in every area of life. But it doesn't suffice to only program software, software must be trustworthy. Testing a software can only guarantee the correctness of several behaviours, but nothing can be said about scenarios which were not tested. Thus "formal methods" were introduced to perform appropriate analysis of software. Formal methods are techniques for designing a system with discrete mathematical models and using mathematical proofs to ensure correct behaviour.

To apply formal methods on a system it is first necessary to model and to specify the system. "Modelling a system" means that the system is transferred into an abstract mathematical model. A "specification of a model" is a description of the properties of the system; it describes what the system is supposed to fulfill. The aim is to show that the model satisfies its specification. Both the model of the system and the specification are defined with the help of a software specification language (also called "software modeling language"). Today there exist several specification languages for concrete programming languages like the Java Modeling Language (JML) [34], which describes a Java program with Java annotations directly in the source code. There also exist abstract modeling languages like Temporal Logic of Actions (TLA) [38, 30], Vienna Development Method (VDM) [10], Alloy [24] or Event-B [10, 2]. TLA combines temporal logic with a logic of actions, whereas VDM supports the description of data and functionality. Alloy models the system as a collection of constraints describing a set of structures. Event-B depends on set theory and first order logic and makes use of refinement to represent systems at different abstraction levels. When a system is fully transferred into a formal model, model checking and verification can be applied. "Model checking" means a systematically exhaustive investigation of every possible execution of the system in a finite scope whether the model satisfies its specification. "Verification" ensures that the model satisfies its specification by deriving theorems which guarantee the correctness of the model and subsequently formally proving these theorems with the help of automatic or interactive provers. In some tools it is also possible to simulate and visualize formal models. A "simulation" is an imitation of the system over time. A "visualization" is a graphical illustration of a run of the model.

Algorithms are techniques for solving problems. Generally an algorithm is a set of step-by-step operations which are performed to gain the desired result of a given input. Therefore algorithms need to be correct and effective, which means that each operation is possible in practice. Furthermore algorithms have to terminate after a finite amount of steps and the sequence of operations is unique for each input. A "mathematical algorithm" is an algorithm which solves problems in various mathematical domains such as algebra, geometry or logic. Mathematical algorithms operate on abstract mathematical types, like matrices, polynomials, graphs or logic formulas. The behaviour of abstract types is defined by a set of laws, properties and relationships.

The main goal of this thesis is to investigate how tools designed for modeling, simulating and analyzing software in other application contexts can be applied to mathematical algorithms. It shall be examined how mathematical problems can be specified and how far corresponding mathematical algorithms can be modeled and analyzed. In order to make the investigation more representative, two different mathematical algorithms, namely the "Davis-Putnam-Logemann-Loveland (DPLL) method" [6] and "Dijkstra's Shortest Path Algorithm" [15] are selected. Furthermore five well-known software specification languages are selected: JML [34], TLA/PlusCal [38, 30], VDM [10], Alloy [24] and Event-B [10, 2]. In each language a formal model for each algorithm shall be developed, which then is simulated, visualized, model checked and verified as far as possible in the corresponding tool. The goal of the thesis is not a proper verification of every model with every tool, although it would be a nice result, but a detailed breakdown of the potential as well as the difficulties of the usage of software specification languages for the analysis of mathematical algorithms.

To approach the goal of the thesis we have first derived for each algorithm a formal specification and the algorithms are described in pseudo-code. For each algorithm we implemented a Java prototype which we specified by JML annotations which we statically checked with ESC/Java2. We then tried to verify the algorithm with Java KeY as far as the built-in automatic proof strategies succeeded (we did not attempt to use interactive proofs in Java KeY, because of the limited interaction capabilities of the prover). For the specification in TLA, we used the algorithmic language PlusCal, which is automatically translated into TLA by the PlusCal translator inside the TLA Toolbox; we then applied the TLC Model Checker to model check the TLA model. In Alloy, we derived at least two models for each algorithm, because in Alloy it is possible to gain visualizations of a model as well as to apply model checking. After modelling the algorithms in VDM, we model checked the algorithms and investigated the generated proof obligations. For the specifications in Event-B, it was necessary to define the algorithms with several events and invariants. After setting up the models we used automatic and interactive proofs to verify the models as far as possible. All specifications which were derived for this thesis can be found at [44] and in the appendix of the thesis.

The main result of the the thesis is that we notice that each of the five used languages shows some success when it is used for specifying and analyzing mathematical algorithms. We are able to completely model and specify the algorithms in every tool, although modelling the algorithms in Alloy is very exhausting due to the specifics of the language. Alloy is the only language where instances of the algorithms can be visualized. In Alloy, TLA/PlusCal and VDM we are able to completely model check the algorithms. In TLA and VDM setting up the models for the model checkers is simple, whereas in Alloy the problem occurs that the scope for model checking has to be restricted for every defined object in the model, which may get very complex. In VDM it is also possible to generate proof obligations, they are too weak and some of them are only placeholders; thus VDM cannot really be used for verifying a model. In Java KeY we only can verify the low-level functions of the algorithms; when the specification gets to expressive, Java KeY is not able to complete the proofs. In Event-B we are able to gain an almost complete verification for Dijkstra's Algorithm, whereas for the DPLL method several proofs

can not be completed.

The remainder of the thesis is structured as follows: In Chapter 2, a description of each software specification language and its corresponding tool is given. In Chapter 3, the DPLL method is introduced; it is shown how it is modeled in each language and how far model checking and verifying is possible in each tool. The structure of Chapter 4 is the same as for Chapter 3, but now Dijkstra's Algorithm is modelled and analyzed. In Chapter 5, the capabilities and limitations of each tool are described and a short comparison of the tools is given.

2. Software Specification Languages & Tools

In this chapter each used software specification language and its corresponding tool is described. The description part covers the main characteristics of the language as well as application examples. To be more descriptive, a small case example is implemented which shows the syntax and some properties of the language. The instruction of the use of the corresponding tool highlights its main features.

Up to now there exist some case examples, where two or more of the software specification languages are compared. In [3] an overview of several specification formalisms is given, including a core summary on how to model in VDM and Event-B. In [7] VDM and B are compared by an example of a communication protocol; in [51] the differences of Alloy and Spin (which is not used in this thesis) are stressed out with a case study of Chord. In the appendix of [24] a scheme for recordable hotel-door locks was modeled in Alloy, Event-B and VDM.

There also exist a few papers where mathematical algorithms were modelled with a software specification language. Graph decomposition was modelled with VDM [22]; a simple shortest path verification was done in Isabelle/HOL [42]. In [36] a formal verification of a SAT solver based on the DPLL method is shown in Isabelle/HOL.

2.1. Java Modeling Language

The Java Modeling Language (JML) [46] differs from the other software specification languages considered in this thesis, since the model is not translated into an abstract model. Specifying a model in JML is done by providing concrete Java annotations comments in the source code files [34]. The Java compiler ignores these annotations since they are disguised as Java comments. It is possible to specify Java classes as well as interfaces with JML.

JML includes an expression language, which covers logical connectives, quantifiers as well as primitive operators and functions. With the help of these expressions, one may formulate in JML pre- and postconditions and assertions in the style of the Hoare calculus [35]. Pre- and postconditions are indicated by the keywords `requires` and `ensures`. An `assignable` clause describes which fields can be modified by a method. If the precondition is fulfilled by the caller of the method, the method needs to guarantee the postcondition upon its return. This approach is called “design by contract” [39]. Assertions specify a property which should always be true. When the source code includes a loop, a loop invariant and a termination term may be included by the keywords `loop_invariant` and `decreases`. A loop invariant is a property which is true before and

after every iteration of the loop. The termination term denotes a value which decreases with every iteration of the loop but never becomes negative.

JML has a rich tool support including type checkers, run-time assertion checkers and extended static checkers. ESC/Java2 [27] is an extended static checker, which is able to find violations of pre- and postconditions by static analysis. Although ESC/Java2 is not sound and complete, it is able to detect many programming errors. ESC/Java2 is currently maintained by the KindSoftware Research Group at the Technical University of Denmark. In this thesis version 2.0.5 of the tool is used. KeY [26] is a further tool for JML which may be used to verify a JML-annotated Java program in a sound way. KeY is a joint project of Karlsruhe Institute of Technology, Chalmers University of Technology and TU Darmstadt. In this thesis, the platform independent version of KeY 2.4.1 is used.

To introduce the syntax of JML a program for finding an integer value in an array is annotated:

```
1  public class Search
2  {
3
4  /*@ public normal_behavior
5  @ requires a!=null && a.length>0;
6
7  @ assignable \nothing;
8
9  @ ensures (0 <= \result && \result <a.length && a[\result]==x &&
10 @    (\forall int j; 0<=j && j<\result; a[j]!=x)) ||
11 @    (\result ==-1 && (\forall int i; 0<=i && i<a.length; a[i]!=x));
12 @*/
13     public static int search(int []a, int x){
14         int pos = -1;
15         int i =0;
16
17 /*@ loop_invariant
18 @   a!=null && a.length>0 &&
19 @   0 <= i && i <= a.length &&
20 @   (\forall int j; 0 <= j && j < i; a[j] != x) &&
21 @   (pos == -1 || (pos == i && i < a.length && a[pos] == x));
22
23 @ decreases pos == -1 ? a.length-i : 0;
24 @ assignable pos, i; (*only for Java KeY*)
25 @*/
26         while(pos==-1 && i<a.length){
27             if(a[i]==x)
28                 pos = i;
29             else
30                 i++;
31         }
32         return pos;
33     }
34 }
```

This Java class consists only of one function, which needs to be annotated. The preconditions for this function state, that the argument pointer must refer to an existing array which has to have at least one element. Since the input array and the input value are not changed, nothing needs to be assignable. This function has two possible outcomes. If the input value is found in the array, the result has to be in the range of the input array. Furthermore the function needs to guarantee that all values which are at positions smaller than the result are unequal to the input value. If the input value is not in the

input array, the result shall be -1 and all values in the input array have to be different from the input value.

Since the function includes a loop, a loop invariant is specified. The loop invariant states that the preconditions of the method are true before and after every iteration. The range of the increasing value i is specified and it always has to be true that already visited positions are unequal to the input value. The variable *pos* can have two different values: It is -1 as long as the input value is not found; if the input value is found, *pos* is equal to the current visited position which is smaller than the length of the input array. The termination term asks if *pos* is still -1. If so *a.length-i* decreases, otherwise the termination term is set to 0, because the end of the loop is reached. This loop invariant includes an *assignable* clause, which declares which variables are changed in the loop. This clause is necessary for Java KeY. For ESC/Java2 it has to be excluded, otherwise an error message is given.

When ESC/Java2 is applied on this method, the following output is generated.

```
1  ESC/Java version ESCJava-2.0.5
2      [0.028 s 8383672 bytes]
3
4  Search ...
5    Prover started:0.0080 s 10691144 bytes
6      [0.364 s 10269984 bytes]
7
8  Search: search(int[], int) ...
9      [0.126 s 10404656 bytes] passed
10
11 Search: Search() ...
12     [0.0090 s 10565760 bytes] passed
13   [0.5 s 10566640 bytes total]
```

The function passed immediately without any error or warning.

Applying Java KeY on this method leads to the output depicted in Figure 2.1. It states that the proof of the method succeeded.

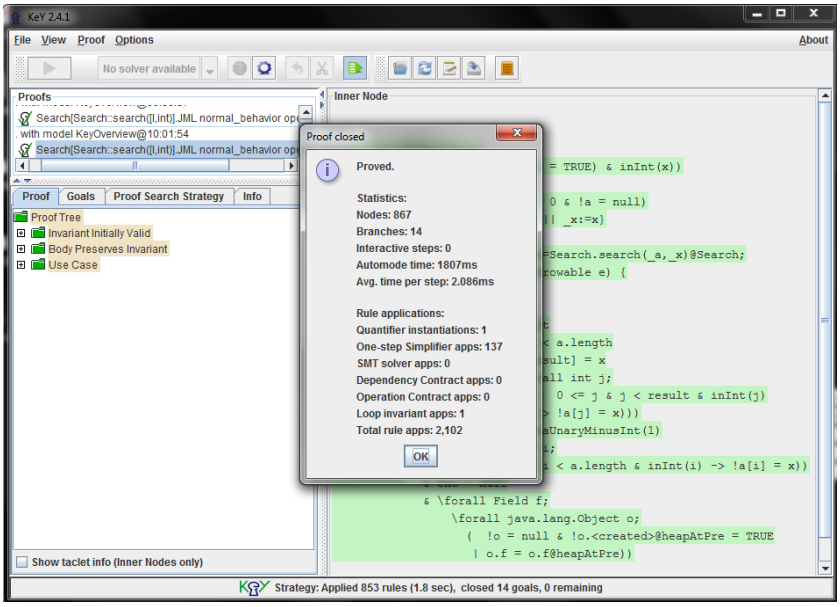


Figure 2.1.: Output from Java KeY for the method `search`

2.2. Temporal Logic of Actions

The Temporal Logic of Actions (TLA) [30] was developed by Leslie Lamport in 1994. TLA combines temporal logic with a logic of actions which makes it possible to describe a whole system with a single mathematical formula [29]. The new idea of TLA was that the user can use formulas with primed and unprimed variables to directly describe the state-transition relations of a system. In TLA the transition system and all properties are stated in the same logic. Thus assertions that a system meets its specification can be simply expressed by logical implication.

Generally a TLA specification for a whole system is of the form:

$$Spec == Init \wedge \Box[Next]_v \wedge Fairness$$

A TLA formula specifies all possible initial states and the transitions which a system is allowed to do. The box operator in front of the *Next* transition means that after the initial state always a possible *Next* step is taken. The *Next* transition itself is a disjunction of all possible transitions. The transitions which the system can make at a specific point can be restricted by inclusion of preconditions. Furthermore every transition needs to determine all changes of the variables. If a variable remains unchanged during a step, this has to be specified explicitly. TLA also contains the possibility of stuttering steps, which means that the system may take transitions where the variables of a system are not changed. This is indicated by $\_v$ following the *Next* transition, where v is a placeholder for a sequence of the variables of a system. To avoid infinitely many stuttering steps, fairness conditions have to be added to the system. TLA makes use of two fairness conditions namely weak fairness and strong fairness. Weak fairness conditions warrant that an event occurs eventually if it is permanently enabled after some point, whereas strong fairness conditions assure that an event occurs eventually if it is infinitely often

enabled.

The TLA Toolbox [47] is an open-source IDE for writing and analyzing specifications in TLA+ and PlusCal. TLA+ [38] is a combination of TLA and mathematical set theory and is used to describe and reason about distributed algorithms. PlusCal [31] is an algorithm language which can describe sequential and concurrent algorithms. It can be translated into a TLA+ specification using the PlusCal translator inside the TLA toolbox. The TLA Toolbox can be used for visualizing pretty-printed versions of the modules, running the PlusCal translator and running the TLC model checker. The TLC model checker runs fully automatically. It builds a finite state model for checking the invariance properties of a model. Therefore it generates the set of all possible initial states and performs a breadth-first search. The execution of the TLC model checker stops when all state transitions lead to already discovered states or a violation of a property is found. In this thesis the TLA Toolbox 1.4.8 for Windows is used.

To go into more detail about the language of PlusCal and TLA, it is now shown how to translate the Euclidean algorithm for finding the GCD of two numbers into PlusCal, which then will be automatically translated into a TLA model. This model is based on [45].

```
1  ------------ MODULE euclid_pluscal --------------
2
3  EXTENDS Naturals
4
5  CONSTANT MAXINT (* Maximal integer *)
6  (* PlusCal options (-termination) *)
7
8  (*
9  --algorithm EuclidAlg {
10   variables u \in 1..MAXINT; (* 1st integer *)
11             v \in 1..MAXINT; (* 2nd integer *)
12             u_init = u;
13             v_init = v;
14
15 {
16     while (u /= 0) {
17       if (u < v) {
18         u := v || v := u
19       };
20       u := u - v
21     };
22 }}
23 *)
```

Every model starts with the keyword `MODULE` and the name of the file. In this example the range operator `..` is used, therefore the module of the natural numbers needs to be included. Since this algorithm computes the GCDs of all pairs of natural numbers in a given range, a maximal integer value needs to be specified. The maximal integer value is defined as a constant, so it can be specified in the TLC model checker.

Since the Euclidean algorithm for finding the GCD of two numbers is a finitely executing algorithm, the PlusCal option for termination needs to be added, otherwise a deadlock would be reported. The PlusCal options as well as the algorithm itself need to

be written as comments, otherwise a parsing error is reported.

Every PlusCal algorithm has to start with the keyword `-algorithm` and the name of the algorithm. Then the variables of the algorithm need to be declared. In this case there are generally two variables u and v and their starter values are saved in *u_init* and *v_init*. The core of the algorithm happens in the while loop. While u is not zero the smaller value from u and v is subtracted from the other one. PlusCal is able to swap two variables in parallel without the use of a dummy element. If at some point u is zero, then the GCD of u and v is equal to v. This algorithm can now automatically be translated into a TLA model:

```
\* BEGIN TRANSLATION
VARIABLES u, v, u_init, v_init, pc

vars == << u, v, u_init, v_init, pc >>

Init == (* Global variables *)
        /\ u \in 1..MAXINT
        /\ v \in 1..MAXINT
        /\ u_init = u
        /\ v_init = v
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ IF u /= 0
               THEN /\ IF u < v
                          THEN /\ /\ u' = v
                                  /\ v' = u
                          ELSE /\ TRUE
                               /\ UNCHANGED << u, v >>
                    /\ pc' = "Lbl_2"
               ELSE /\ pc' = "Done"
                    /\ UNCHANGED << u, v >>
         /\ UNCHANGED << u_init, v_init >>

Lbl_2 == /\ pc = "Lbl_2"
         /\ u' = u - v
         /\ pc' = "Lbl_1"
         /\ UNCHANGED << v, u_init, v_init >>

Next == Lbl_1 \/ Lbl_2
          \/ (* Disjunct to prevent deadlock on termination *)
             (pc = "Done" /\ UNCHANGED vars)

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)

\* END TRANSLATION
```

At first the variables are defined but not instantiated. There is now one more variable *pc* which defines a program counter of the model. All these variables are also defined as a sequence, which will be used later in the Next transition to allow stuttering steps. Then the initial state of the system is defined. So u and v can be between 1 and the predefined maximum integer. *U_init* and *v_init* are again set to the starter variables of u and v and the program counter is set to *Lbl_1*.

The system is now able to perform two steps, either a *Lbl_1* step or a *Lbl_2* step.

Steps are always conjunctions of several conditions, which all need to be executed such that the step is true. Which step is taken in this model is decided by the value of *pc*. For performing a *Lbl_1* step, the program counter needs to be equal to *Lbl_1*. This step decides now if the loop of the algorithm is entered again. If so it is then checked which variable is the smaller one. If it is u, then u and v are swapped otherwise u and v remain unchanged. The program counter is then set to *Lbl_2*. If the loop is not entered again, which means that u is equal to zero the program counter is set to *Done* and the variables remain unchanged. If the loop was entered, the system is only able to perform a *Lbl_2* step, which means that the new variable of u is equal to u-v and the program counter is again set to *Lbl_1*.

To generate the specification of a system, all transition states are composed in the *Next* transition, which is a disjunction of all possible steps which the system may take. Here a final step is added, where the program counter has reached the *Done* state and all variables remain the same. As previously mentioned, the whole specification of a system is a conjunction of the initial state, the next transition and fairness conditions. In this example only weak fairness is needed, which guarantees that a *Next* step is taken, when it is permanently enabled.

In this example it is furthermore checked, that the algorithm terminates and that it is correct:

```
1  Termination == <>(pc = "Done")
2
3  p | q == \E d \in 1..q : q = p * d
4  Divisors(q) == {d \in 1..q : d | q}
5  Maximum(S) == CHOOSE x \in S : \A y \in S : x >= y
6  GCD(p,q) == Maximum(Divisors(p) \cap Divisors(q))
7
8  IsCorrect == [](pc = "Done" => v = GCD(u_init, v_init))
```

Therefore two properties are added, which then will be checked in the TLC model checker. The termination property verifies that at some point the program counter is equal to *Done*. For the second property, which should check that in the end v is really the GCD of *u_init* and *v_init* the properties of a GCD need to be specified. Then it can be checked that always when the program counter is set to *Done*, v is the GCD of *u_init* and *v_init*.

Before running the TLC model checker all constants of the model need to be specified. Furthermore the properties and invariants, which someone wants to check, need to be selected. The declaration of the model can be seen in Figure 2.2.

After setting up the model the TLC model checker can be applied. The TLC model checker does not detect any error in the model, as depicted in Figure 2.3, which means that all properties are correct in the specified range.

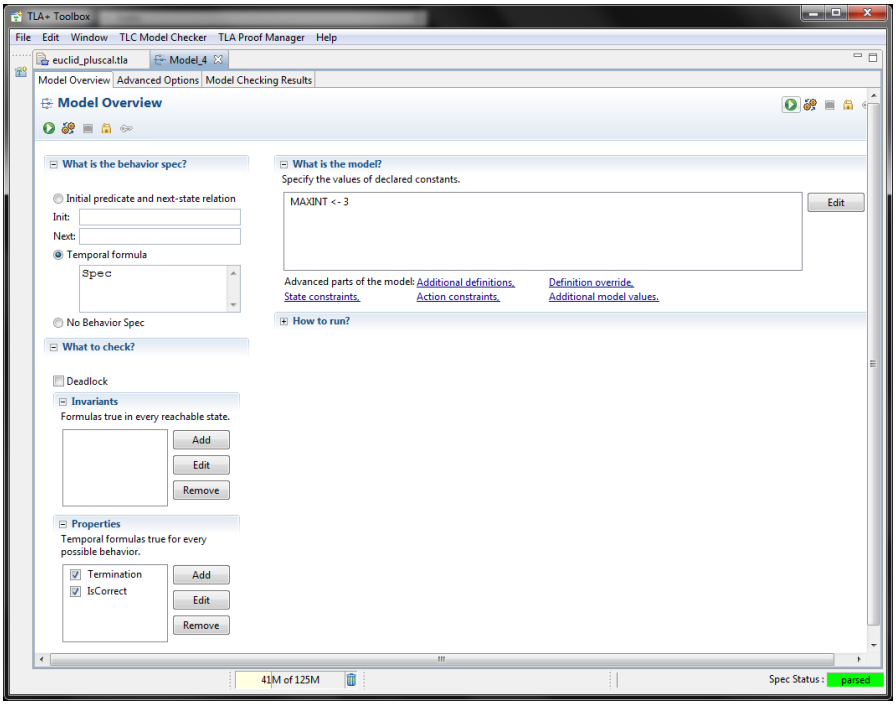


Figure 2.2.: Setting up a model in the TLA Toolbox

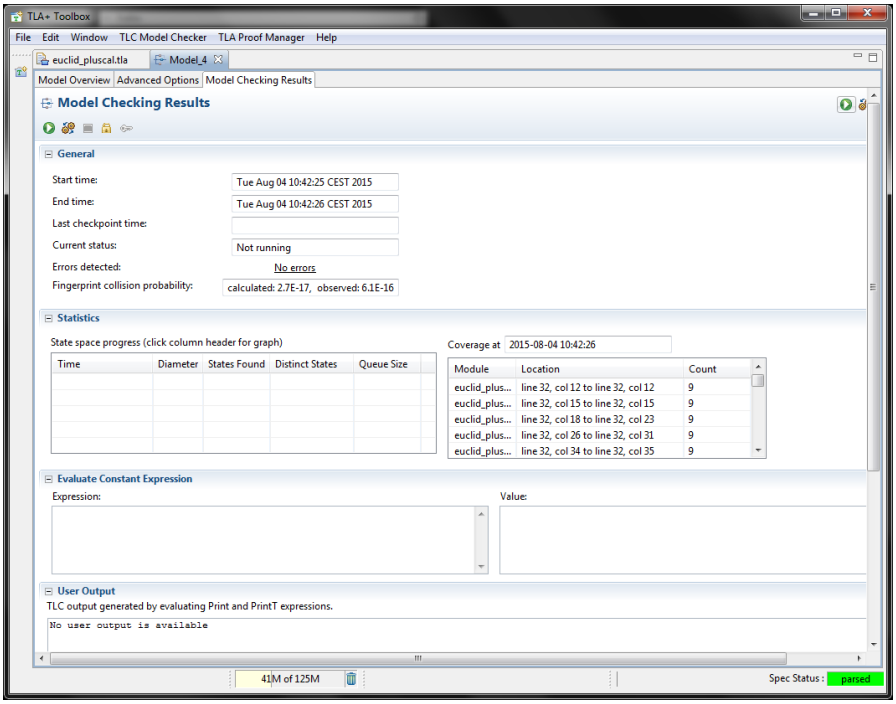


Figure 2.3.: Output of the TLC Model Checker

2.3. Alloy

Alloy [24] is a product of the Software Design Group at the Massachusetts Institute of Technology, which is lead by Professor Daniel Jackson [24]. The first prototype of Alloy emerged in 1997 as a successor of Nitpick. Up to now the performance and scalability of Alloy has successively increased. It is used for example in the "Equals Checker", which is a tool for checking problems related to the *Object.equals* method in Java [5].

Alloy can be used to describe and analyze all types of systems, but it is mostly used on systems that involve complex structured states. Examples for applications are name servers, access control, instant messaging and network configuration protocols. One big advantage of Alloy compared to other formal specification systems is that it allows the analysis of systems which can change dynamically. For example the leader election phase of the Firewire protocol was analyzed in Alloy. Another great benefit of Alloy is that it is possible to incrementally analyze a model, which means that the analysis can be done on a very small model, which is then refined stepwise.

The language of Alloy [50] is based on relational first order logic with built-in operators for sets and integer arithmetic. Generally all components of a model are depicted as relations. Properties of the system can be expressed with several operators. Boolean expressions are used in Alloy to phrase states and executions of a model. This means that structural constraints can be described intuitively with sets and relations. The language itself is highly influenced by the Z notation [24].

The Alloy Analyzer [4] is a self-contained executable that extends the Alloy library and also includes the Kodkod model finder [48] and several SAT solvers. The Alloy Analyzer translates all constraints of the Alloy model into a Boolean formula, which is then passed to a SAT solver. The solution is then translated back into the language of Alloy. All models of Alloy are analyzed in a user-defined scope that bounds the size of domains. Generally the Alloy Analyzer is designed to perform finite scope checks by trying to find a model in the given scope which makes the formula true. So the Alloy Analyzer is not a model checker, it is technically a model finder.

The analysis of a model is done fully automatically in the Alloy Analyzer. For checking a given property the Alloy Analyzer negates the assertion and searches for a model which will be a counterexample of the claim. If the Alloy Analyzer finds no counterexample, this does not automatically mean that the property is correct, it only guarantees that in the given scope no counterexample was found. But the larger the scope, the more unlikely it is to miss a counterexample. By explicitly modeling traces in Alloy, it is also possible to check LTL properties. This works only well in bounded systems, since the Alloy Analyzer can only perform finite scope checks. In this thesis the platform independent version of Alloy 4.2 is used.

To introduce the structure of a model and to be more accurate about the language of Alloy the Königsberg bridge problem is modeled [50]. The Königsberg bridge problem is a finite state problem in graph theory. The question is, if it is possible to walk around in Königsberg by using all bridges exactly once. Leonhard Euler proved in 1736,

that it is not possible and a deadlock is reached. Since the Königsberg bridge problem is a finite state problem, the Alloy Analyzer is able to check all possible models.

```
module Bridges

abstract sig Direction{}
one sig North, East, South, West extends Direction {}

abstract sig Bridge { connects: set Direction }{#connects = 2}

one sig Bridge1 extends Bridge {}{ connects = North + West }
one sig Bridge2 extends Bridge {}{ connects = North + West }
one sig Bridge3 extends Bridge {}{ connects = North + East }
one sig Bridge4 extends Bridge {}{ connects = East + West }
one sig Bridge5 extends Bridge {}{ connects = East + South }
one sig Bridge6 extends Bridge {}{ connects = South + West }
one sig Bridge7 extends Bridge {}{ connects = South + West }

sig Path { firstStep: Step }

sig Step {
    from, to: Direction,
    via: Bridge,
    nextStep: lone Step}
   {via.connects = from + to }

fact {
    all curr: Step, next: curr.nextStep | next.from = curr.to
}

fun steps (p:Path): set Step {
    p.firstStep.*nextStep
}

pred path() {
    some p:Path | steps[p].via = Bridge
}

run path for 7 but exactly 1 Path
```

Each new model in Alloy starts with the declaration of a `module`. Then the objects of the model, namely bridges, directions, steps over a bridge and the path have to be introduced. This is done by the keyword `sig`.

An abstract signature of an object means that it is not possible to instance it directly, therefore the extends keyword is used for the signatures *North, East, South and West*. The `extends` keyword can be seen as dividing the set *Direction* into disjoint subsets *North, East, South* and *West*. `One sig` limits the number of each subset to exactly one. *Bridge* is again an abstract object. But a *Bridge* has now a property called *connects*. *Connects* can be seen as a function from *Bridge* to the set *Direction*. The second pair of curly brackets appends a fact to the definition of *Bridge*. This fact states that a *Bridge* can only connect exactly two directions. The seven bridges of Königsberg are now set up where each definition includes the fact which subset of *Direction* is connected by which *Bridge*. The last two objects of the model are a *Path* and a *Step*. A *Path* simply starts with a *firstStep* of type *Step*. The declaration of *Step* describes that each *Step* has to start at one side of a *Bridge* and end on the other side of the *Bridge*. A *Step* can either be followed by one new *Step* or not, which is denoted by the keyword `lone`. Furthermore a fact needs to be added, which ensures that the destination point of one *Step* is the

starting point of the next *Step*.

Finally the behaviour of the model can be set up with functions and predicates. The function *steps* takes a *Path* as input and returns a set of *Steps*. From the input *Path p* the *firstStep* and all following *Steps* shall be returned. Therefore the operator * is used, which defines the reflexive transitive closure. In this model a possible *Path* should be returned, therefore the predicate *path()* is defined, which determines if there exists some *Path* such that the *Steps* of a *Path* have crossed all *Bridges*.

After setting up the model it is now possible to run the model. The restriction *for 7 but exactly 1 Path* guarantees that it is searched for a model where up to 7 steps on exactly one path are taken. The Alloy Analyzer is not able to find such a model which suffices the predicate path, thus the Königsberg bridge problem has no solution. The output can be seen in Figure 2.4.

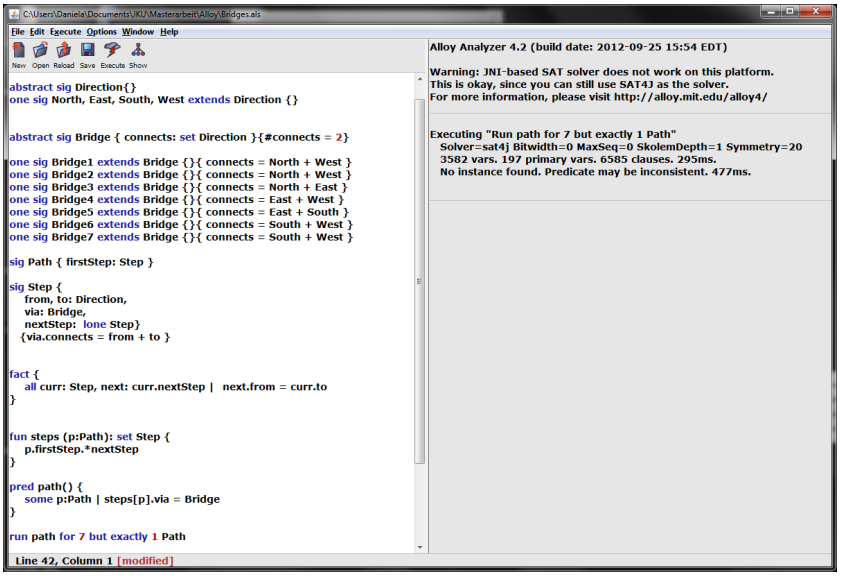


Figure 2.4.: Output of the Alloy Analyzer

If the number of steps is increased to 8, which means that it is allowed to cross one bridge twice, a model is found. The visualization of the model can be seen in Figure 2.5. The visualization is personalized, such that only specific relations are shown.

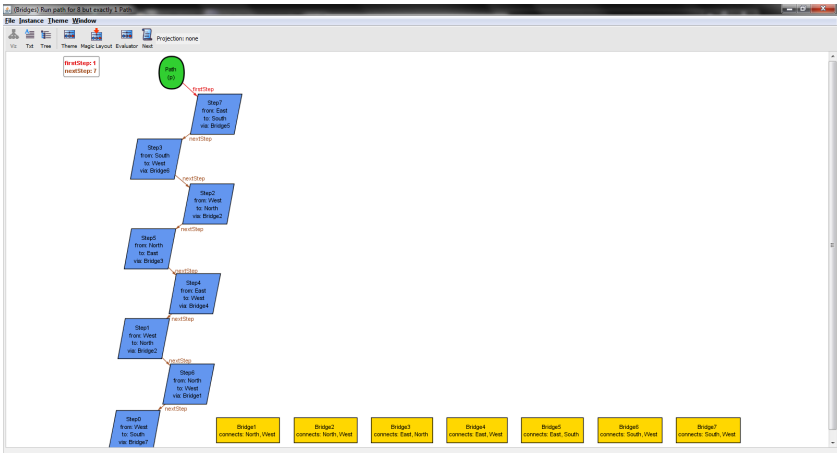


Figure 2.5.: Personalized visualization of a model in the Alloy Analyzer

2.4. Vienna Development Method

VDM is a formal language developed around 1970 at the IBM laboratories in Vienna. It is a collection of techniques for modeling, specifying and designing systems. The VDM Specification Language (VDM-SL) is the standard definition language of VDM. VDM-SL was standardized by ISO in 1996. VDM++ [19] is an extension of VDM-SL including class and object concepts, which allow to specify object oriented systems. VDM has a rich language which can be used in a wide variety of applications. The language of VDM [3] includes basic data types like Boolean values, natural numbers and structureless tokens as well as compound types like sets, sequences and maps. VDM also allows to model operations and functions. Functions and operators can be defined both implicitly and explicitly in VDM. Thus VDM supports modeling and analysis of software systems at different levels of abstraction. VDM is used in the formal definition of Ada and in the development of the compiler for Ada. Furthermore VDM is used in the development of a portable compiler for CHILL [9].

Overture [41] is one of several tool supports for VDM. Overture is an open source IDE on top of the Eclipse platform. It is developed by research scientists and students from Denmark, England, The Netherlands and Japan. In Overture, immediate syntax checking and type checking is possible. Furthermore automatic proof obligation generation and test generation support are available. Automatic proof support and visualization for execution traces is under development. In this thesis Overture 2.2.6 for Windows is used.

To give an example for the language of VDM++ the abstract data type “Stack” with the two basic operations “pop” and “push” is specified [11]:

```
1  class Stack
2
3  types
4  public val = int;
5
6  instance variables
7  stack : seq of val;
8
```

```
9  operations
10 public Stack: () ==> Stack
11     Stack() ==
12         (stack:=[]);
13
14 public Push : val ==> seq of val
15     Push(newval) ==
16         (stack := [newval]^stack;
17         return stack)
18     post stack = [newval]^stack~;
19
20 public Pop : () ==> val
21     Pop() ==
22         def res = hd stack in
23         (stack := tl stack;
24         return res)
25     pre stack <> []
26     post stack~ = [RESULT]^stack;
27
28 end Stack
```

This model begins with the keyword *class* and its name *Stack*. Subsequently the data type which the stack is able to store is defined. *Types* can be seen as placeholders, because when the data type of the values of the stack is changed, only the *types* section of the model needs to be modified. The internal variables of a model are defined in the section *instance variables*. In this class it only needs to be specified that a stack is a sequence of type *val*.

The class consists of one constructor and the two basic operations "push" and "pop". Constructors have the same name of the class and return a new instance of the class. The constructor in this example has no input argument and returns an empty stack. The *Push* operation adds an item at the top of the stack. It takes a *val* as input and returns a sequence of *vals*. Inside the method the item *newval* is appended at the front of *stack* and *stack* is returned. The postcondition of this operator states, that the new value of *stack* is the old value of *stack* with the added value *newval* at the beginning. Postconditions are added to the proof obligations, which then need to be proved by the user. The operation *Pop* which removes the first element of the *stack* takes no input and returns a *val*. Inside the method a variable *res* is defined which is equal to the head of the *stack*. The value *stack* is set to its tail, which means that the head of *stack* is removed. The head is then returned. Pop needs a precondition, which ensures that the *stack* is not empty. The postcondition states that the old value of *stack* is the new value of *stack* with *RESULT* at the beginning. *RESULT* is a keyword in VDM and represents the value which is returned, in this operation *RESULT=res*.

In Overture four proof obligations are created for the model, which is depicted in Figure 2.6. One proof obligation is created for the operation "Push(val)" and states that the "operation establishes postcondition". The remaining three proof obligations are generated for the operation "Pop()" and declare that neither the head or the tail of stack are a non-empty sequence and that the postcondition is established after the execution. Up to now the proof obligations cannot be exported or analyzed within the Overture tool, but in future versions a proof component in Overture shall be available [33].

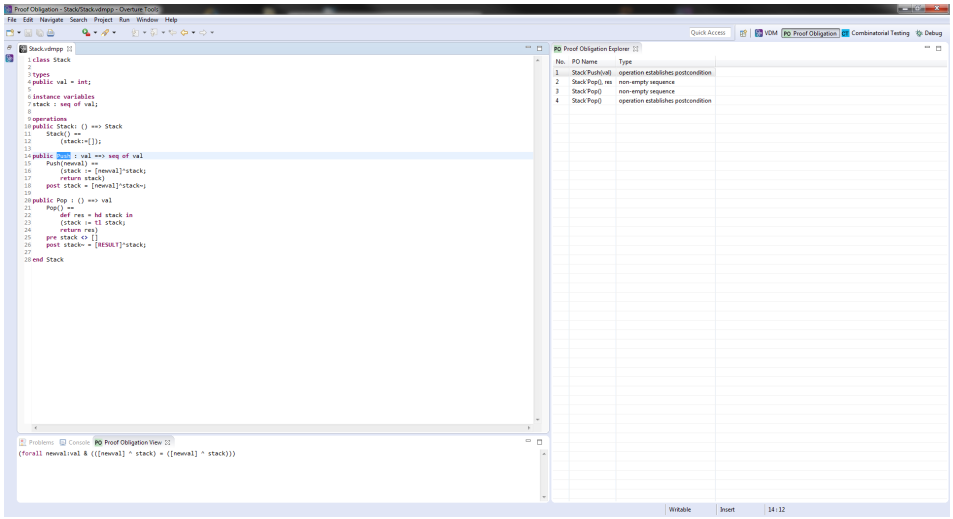


Figure 2.6.: Proof obligation generation for class Stack

2.5. Event-B

Event-B [10, 2] is an advancement of the classical B method. The B method includes the B language, the concept of refinements and several proving techniques in the related tools. The B method was first described by Jean-Raymond Abrial in 1996 in The B-Book [1]. Classical B is related to Z [32] and thus based on Zermelo-Fraenkel set theory including the axiom of choice. Event-B, which was also defined by Abrial, has a simpler notation than classical B and describes a system with events. Event-B and Rodin are used in several industrial projects like in the development of a cruise control system from Bosch or in a train control system developed by Siemens Transportation [23].

Event-B is a proof-based development method which includes the concept of refinements. This means that the first model of a system is a very abstract one, only the core concept of the system is specified in the first model. Details are added piecewise by creating a sequence of more concrete models. So each successor model refines the previous one. Each new model comes up with stronger invariants and new proof obligations which ensure the correctness of the system. The core concept of refinement is that it preserves proved properties, which makes it easier to gain a very detailed proved model in the end. If the refining steps are omitted, it would be a lot harder to prove the correctness of the model. Refining is also useful to transform a very abstract model into a more concrete model. The last model is so close to an implementation, that it can be transformed into a program with the help a plug-in, called Code Generation Feature [17].

The Rodin Platform [16] is an open-source Eclipse-based IDE and can be expanded with several plug-ins. Within the Rodin Platform the use of refinement to represent systems at different abstraction levels and the usage of mathematical proofs is possible. More precisely for each refinement level the Rodin Platform generates the proof obligations automatically, which can then be solved automatically or interactively. By installing different plug-ins it is also possible to gain graphical visualizations or animations of the model. The Rodin Platform is a joint project of ETH Zürich, Systerel, Clearsy,

Heinrich-Heine-Universität Düsseldorf, the University of Newcastle and the University of Southampton. In this thesis Rodin 3.1.0 for Windows is used.

To explain the language of Event-B and to show a refinement step the system of a traffic light at a crossing is specified. The first model is very abstract, the values for the two traffic lights of cars and pedestrians are simply modelled with boolean values. In the refined model the traffic lights will be modelled with colours [25].

```
MACHINE
       mac

VARIABLES
      cars_go
      peds_go

INVARIANTS
      inv1:  cars_go ∈ BOOL
      inv2:  peds_go ∈ BOOL
      inv3:  ¬(cars_go = TRUE ∧ peds_go = TRUE)

EVENTS
   INITIALISATION:
        THEN
           act1:  cars_go := FALSE
           act2:  peds_go := FALSE
        END

   set_peds_go:
        WHERE
           grd1:  cars_go = FALSE
        THEN
           act1:  peds_go := TRUE
        END

       set_peds_stop: not extended ordinary
        THEN
           act1:  peds_go := FALSE
        END

    set_cars:
        ANY
           new_value
        WHERE
           grd1:  new_value ∈ BOOL
           grd2:  new_value = TRUE ⇒ peds_go = FALSE
        THEN
           act1:  cars_go := new_value
        END

    END
```

This is the first model, called machine *mac*. It consists of two variables, one for the cars and the other one for pedestrians. Both variables are from the data type bool, which is ensured by the first two invariants of the system. The third invariant ensures that at no time pedestrians and cars are allowed to move at the same time.

This machine has four events. The first event is the initialisation of the machine, where both variables are set to false. The second event changes the variable of the pedestrians to true, but only when the cars are not allowed to move. The next event sets the value of the variable for the pedestrians to false. This can be done independently from the value

of the cars, so a guard is not necessary. The last event changes the value of the variable for the cars. It could be again done with two events, as it is done for the pedestrians. But it is modelled differently, because in the refining model three colours for the traffic light for cars are introduced. Therefore the boolean variable *new_value* is introduced including the restriction that *new_value* is only true when pedestrians have to stop. The variable for the cars is then set to this *new_value*.

In the Rodin Platform all automated generated proof obligations for the first machine are proved automatically, which can be seen in Figure 2.7.

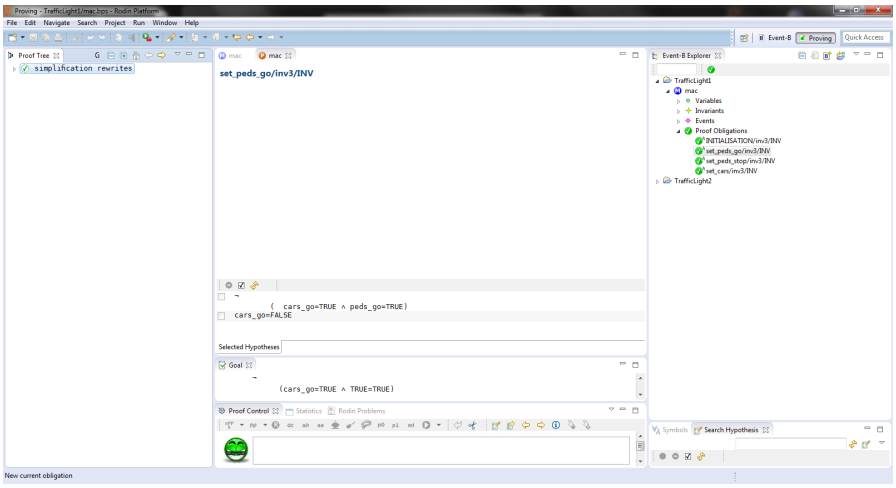


Figure 2.7.: Output for the first machine

Before the model can be refined, a new context needs to be added, where the set of colours is implemented:

```
1  CONTEXT
2      ctx1
3  SETS
4      COLOURS
5  CONSTANTS
6      red
7      yellow
8      green
9  AXIOMS
10     axm1:  partition(COLOURS,{red},{yellow},{green})
11 END
```

In this context *ctx1* a set called *COLOURS* and the constants *red, yellow, green* are defined. Furthermore an axiom is added, which states that the set *COLOURS* is a partition of *red, yellow, green*.

```
1  MACHINE
2     mac1
3  REFINES
4     mac
5  SEES
6     ctx1
7
8  VARIABLES
9     peds_colour
10    cars_colour
```

```
11
12 INVARIANTS
13     inv4:  peds_colour ∈ {red, green}
14     gluing_p:  peds_go = TRUE ⇔ peds_colour = green
15     inv5:  cars_colour ⊆ COLOURS
16     gluing_c:  cars_go = TRUE ⇔ green ∈ cars_colour
17
18 EVENTS
19     INITIALISATION:
20         THEN
21            act1: peds_colour := red
22            act2:  cars_colour := {red}
23         END
24
25     set_peds_green:
26         REFINES
27             set_peds_go
28         WHERE
29            grd1:  green ∉ cars_colour
30         THEN
31            act1:  peds_colour := green
32         END
33
34     set_peds_red:
35         REFINES
36             set_peds_stop
37         THEN
38            act1:  peds_colour := red
39         END
40
41     set_cars_colours:
42         REFINES
43             set_cars
44         ANY
45            new_value_colours
46         WHERE
47            grd1:  new_value_colours ⊆ COLOURS
48            grd2:  green ∈ new_value_colours ⇒ peds_colour = red
49            grd3:  cars_colour = {yellow} ⇒ new_value_colours = {red}
50            grd4:  cars_colour = {red} ⇒ new_value_colours = {red, yellow}
51            grd5:  cars_colour = {red, yellow} ⇒ new_value_colours = {green}
52            grd6:  cars_colour = {green} ⇒ new_value_colours = {yellow}
53         WITH
54            new_value: new_value = TRUE ⇔ green ∈ new_value_colours
55         THEN
56            act1:  cars_colour := new_value_colours
57         END
58  END
```

Machine *mac1* is now a refinement of machine *mac* with usage of the context *ctx1*. This machine has two new variables *peds_colour* and *cars_colour*, where *peds_colour* is an element of the set *{red, green}* and *cars_colour* is a subset of *COLOURS*. *Cars_colour* can not be a single element from the set, because the traffic light for cars can show red and yellow at the same time.

To map the variables from the first machine to this machine two invariants, so-called *gluing invariants* are added. Both gluing invariants map the states where pedestrians respectively cars are allowed to move from the old machine to the new one.

The initialisation event and the events for the pedestrians of *mac1* are simply adjusted by changing the properties of the variables of the system. The event for setting the colours for the traffic light of the cars needs more guards in the refined model, since for

each state of the traffic light, the successor state needs to be defined. Additionally the *new_value* from the first machine needs to be linked to the *new_value_colours*.

All automated generated proof obligations are again proved automatically by the Rodin Platform, as depicted in Figure 2.8.

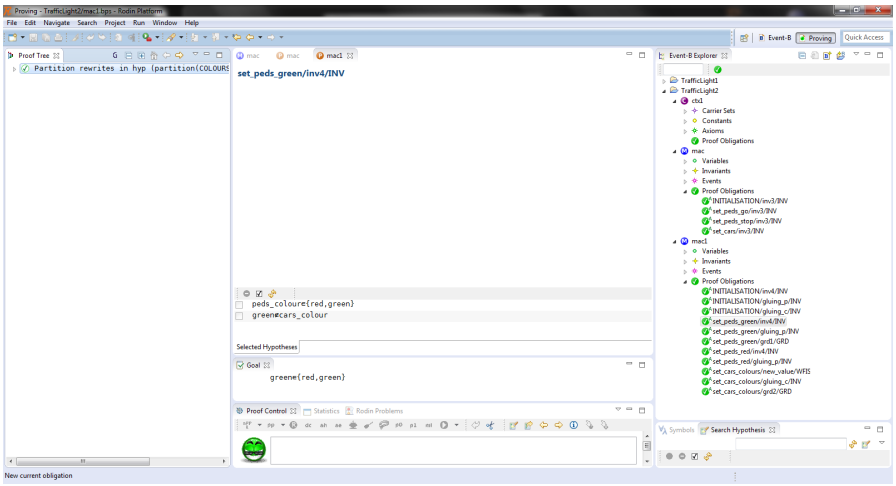


Figure 2.8.: Output for the refined machine

2.6. Comparison of the Languages and Tools

A short summary of the software specification languages and tools considered in this thesis is given in Figure 2.9. The summary includes the main characteristics of the languages. Furthermore the corresponding tools are denoted and the major tool support is highlighted.

	Main Characteristics	Tool	Tool Support
JML	Direct specification of a system with Java annotations in source code	ESC/Java2	Extended static checker for finding specification mistakes
		KeY	Verification of automatically generated proof obligations
Alloy	Everything is described as a relation	Alloy Analyzer	Includes Kodkod model finder and several SAT solvers for finding models in a finite scope

TLA	Combination of temporal logic with a logic of actions	TLA Toolbox	Includes TLC model checker and PlusCal translator
VDM	Extension of the standardized VDM-SL by object-oriented concepts	Overture	Combinatorial testing and generating proof obligations which then need to be solved by the user
Event-B	Development of a series of more and more accurate models	Rodin	Automatically generates proof obligations which then can be solved automatically and interactively

Figure 2.9.: Comparison of the Languages and Tools

There exist some case examples, where two or more of the methods are compared. In [3] an overview of several specification formalisms is given, including a core summary on how to model in VDM and Event-B. In [7] VDM and B are compared by an example of a communication protocol; in [51] the differences of Alloy and Spin (which is not used in this thesis) are stressed out with a case study of Chord. In the appendix of [24] a scheme for recordable hotel-door locks was modeled in Alloy, Event-B and VDM.

There also exist a few papers where mathematical algorithms were modelled with a software specification language. Graph decomposition was modelled with VDM [22]; a simple shortest path verification was done in Isabelle/HOL [42]. In [36] a formal verification of a SAT solver based on the DPLL method is shown in Isabelle/HOL.

3. DPLL Algorithm

This chapter deals with the Davis-Putnam-Logeman-Loveland algorithm. At first the algorithm is informally introduced and a pseudo-code is given. The informal part is followed by a formal specification, where the input and output conditions are outlined. A Java prototype of the algorithm is developed, which is then specified in JML and analyzed by ESC/Java2 and KeY. Subsequent the algorithm is modelled in TLA, Alloy, VDM and Event-B. For each specification language the set-up of the model is declared and the output of the corresponding tool is shown.

3.1. Informal Problem and Recursive Algorithm

The Davis-Putnam-Logeman-Loveland (DPLL) algorithm [6, 37] for solving the propositional satisfiability (SAT) problem is a core algorithm in computer science and computational logic. The SAT problem is the problem of determining if there exists an interpretation under which a given propositional formula in conjunctive normal form is satisfiable. A formula in conjunctive normal form is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a positive variable or its negation. The SAT problem seeks for a way that all variables of the given propositional formula can consistently be replaced by "true" or "false" such that the formula evaluates to "true".

The DPLL algorithm is a refinement of the Davis-Putnam (DP) [21] algorithm, which was developed in 1960 [14] for checking the validity of a formula. The DP algorithm makes use of a resolution-based decision procedure, which can be very time consuming. The DPLL method was developed by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland in 1962 [13]. It makes use of a *splitting rule* and thus is one of the first algorithms for solving the SAT problem efficiently.

Today the DPLL method is still the basis algorithm for the majority of modern complete SAT solvers (e.g. MiniSAT, zChaff) [37]. The DPLL method is also used in automated theorem proving or satisfiability modulo theories (SMT). A SMT problem is a problem of deciding if a given logical formula is true with respect to a background theory expressed in first-order logic.

The DPLL method is a backtracking based search algorithm. A literal from the formula is chosen and a truth value is assigned to it. This simplifies the formula and the remaining formula is again checked for satisfiability by assigning a truth value to another literal from the formula. This is done until all literals of the formula are assigned. If the simplified formula is satisfiable, then the original formula is satisfiable too. If it is not satisfiable, the recursive check is repeated by assigning the opposite truth value to the literal. This method is known as the *splitting rule*. The splitting rule replaced the elimination rule of the original Davis-Putnam algorithm and thus facilitated the original

algorithm, because the problem is now divided into two smaller subproblems instead of solving a single subproblem, as it was done in the Davis-Putnam algorithm.

The selection of a literal is critical for the performance of the algorithm. Therefore the DPLL method includes two rules, which enhance the performance of the algorithm. If those rules are omitted the algorithm is still sound and complete, but the execution of the algorithm can get time consuming. The worst case performance of the DPLL method is $\mathcal{O}(2^n)$. The performance increasing rules which are used at each step of the algorithm are called *unit propagation* and *pure literal elimination*:

- *Unit propagation* is executed when a clause consists of only one unassigned literal. This unit clause can only be evaluated to true by assigning the necessary truth value to make the corresponding literal true.
- *Pure literal elimination* is applied, when a literal occurs in only one state, either positive or negative, in the formula. This literal is then called pure. A pure literal can directly be assigned in such a way that all clauses containing the pure literal evaluate to true. Thus all clauses containing the pure literal do not have to be considered anymore and can be neglected.

Satisfiability of a formula is detected when the original formula is reduced to an empty formula. This means that all variables are assigned in such way that all clauses of the original formula are satisfied. On the other hand unsatisfiability of the original formula can be detected when a clause becomes the empty clause. This means that all variables of the clause have been assigned in such a way that the corresponding literals are false. Unsatisfiability of the original formula can only be detected after exhaustive search, since it must be guaranteed that no assignment of the variables evaluates the formula to true.

The following pseudo-code summarizes the algorithm (adapted from [37]):

Algorithm DPLL(Φ) recursive

Input: A formula Φ
Output: A truth value indicating if the formula is satisfiable

```
1: if Φ is empty then
2:     return true
3: else if Φ contains an empty clause then
4:     return false
5: end if
6: for every unit clause l in Φ do
7:     Φ ← unit-propagation(l, Φ)
8: end for
9: for pure literal l in Φ do
10:    Φ ← pure-literal-elimination(l, Φ)
11: end for
12: select a variable v occurring in Φ
13: if DPLL(substitute(Φ, v, true))=true then
14:    return true
15: else
```

```
16:     return DPLL(substitute(Φ, v, false))
17: end if
```

In *unit-propagation(l, Φ)*, where l is a unit clause, the unit propagation rule is applied on Φ. The unit clause in Φ is replaced with *true*. Every other occurrence of the literal l is replaced with *true* and every occurrence of $\neg l$ is replaced with *false*. The updated set of clauses is returned.

In *pure-literal-elimination(l, Φ)*, where l is a pure literal in Φ, the pure literal elimination rule is applied on Φ. Every occurrence of the literal l is assigned with *true*. The updated set of clauses is returned.

In *substitute(Φ, v, bool val)* every occurrence of v is replaced by the boolean value *val*. Whenever v evaluates to *true*, the whole clause will be omitted. Whenever v evaluates to *false*, the literal will be omitted.

The functions *unit-propagation(l, Φ)* and *pure-literal-elimination(l, Φ)* are optimizations of the DPLL method. The algorithm is still sound and complete, when the optimizing functions are dropped, but the algorithm may need more time. In the following only the basic form of the algorithm without the performance increasing rules is considered.

3.2. Iterative Algorithm and Correctness

Since some formal modelling languages do not support recursions, a pseudo-code for an iterative implementation of the DPLL method based on [8] is given. The refining rules *unit propagation* and *pure literal elimination* are here neglected.

Algorithm DPLL(Φ) iterative

Input: A formula Φ
Output: A truth value indicating if the formula is satisfiable

```
 1: stack ∈ empty
 2: while true do
 3:   if Φ is empty then
 4:     return true
 5:   else if Φ contains an empty clause then
 6:     if stack.isEmpty() then
 7:       return false
 8:     end if
 9:     Φ ← stack.pop()
10:   else
11:     select a variable v occurring in Φ
12:     stack.push(substitute(Φ, v, false))
13:     Φ ← substitute(Φ, v, true)
14:   end if
15: end while
```

The partial correctness of the algorithm is implied by its core invariant which states, that the original formula is satisfiable if and only if one of the formulas of the stack evaluates to true or the current formula Φ evaluates to true. Furthermore the formula at the top of the stack has the same number of literals as Φ, given that the stack is nonempty. Formalizations of this invariant are given in Section 3.5.3 (for JML) and in Section 3.9.1 (for Event-B).

The termination term of the loop is defined by the number of unvisited knots in the binary search tree of the search space of the formula. Each vertex in the tree represents a formula and the out edges of the vertex represent the two possible assumptions made for a specific literal occurring in the formula in the vertex. Each path from the root to a leaf of the tree is a possible valuation of the formula. The search tree for a formula in n variables has $2^{n+1} - 1$ knots. After a substitution of the current formula Φ is made, the formula added to the stack and the new formula Φ have one variable less than the old Φ. This means that the original search space is divided into two smaller subtrees, both containing $2^n - 1$ knots. Since $2 * (2^n - 1) = 2^{n+1} - 2$, the termination term decreases with every substitution of the formula. A formalization of the termination term is given in Section 3.5.3 (for JML).

3.3. Formal Specification

In this section a formal specification of the DPLL method is given. The input and output of the algorithm including input and output conditions are outlined. For each condition the underlying terms are defined.

Algorithm DPLL

Input: *(F, n)*
Input condition: $n \geq 1 \wedge F \in \text{Formula}_n$
Output: s
Output condition: $s = 1 \Leftrightarrow (F, n)$ is satisfiable

---

In the informal description of the DPLL method, the input of the algorithm was the formula Φ. In the formal specification Φ is replaced by F and a second parameter n is added, which specifies the range of the literals occurring in F. The output value s can be mapped to the truth value which indicates if the formula is satisfiable.

For better readability the following type definitions are used:

- Var := $\mathbb{N}$
- RawLiteral := $\mathbb{Z}$
- RawClause := $\mathbb{P}(\mathbb{Z})$
- RawFormula := $\mathbb{P}(\mathbb{P}(\mathbb{Z}))$

3.3.1. Input Condition

At first a predicate *consistent* is defined, which ensures that a literal does not occur positive and negative inside a clause.

$$\text{consistent} \subseteq \text{RawLiteral} \times \text{RawClause}$$
$$\text{consistent}(l, c) :\Leftrightarrow l \in \mathbb{Z} \wedge \neg(l \in c \wedge -l \in c)$$

Before the definition of a formula can be given, the sets of literals and clauses need to be defined. A literal is an integer which is either positive or negative.

$$\text{literal} \subseteq \text{Var} \times \text{RawLiteral}$$
$$\text{literal}_n(l) :\Leftrightarrow l \in \mathbb{Z} \wedge (0 < l \leq n \vee -n \leq l < 0)$$

$$\text{Literal: Var} \to \text{RawClause}$$
$$\text{Literal}_n := \{l \in \mathbb{Z} \mid \text{literal}_n(l)\}$$

Based on the set of literals, the definition of a clause can be given. A clause is a set of literals which is consistent for all literals.

$$\text{clause} \subseteq \text{Var} \times \text{RawClause}$$
$$\text{clause}_n(c) :\Leftrightarrow c \in \mathbb{P}(RawClause) \wedge \forall v \in \mathbb{Z} : consistent(v, c)$$

$$\text{Clause: Var} \to \text{RawFormula}$$
$$\text{Clause}_n := \{c \in \mathbb{P}(Literal_n) \mid \text{clause}_n(c)\}$$

With the definition of the set of clauses, the definition of a formula can be derived. A formula is a set of clauses.

$$\text{formula} \subseteq \text{Var} \times \text{RawFormula}$$
$$\text{formula}_n(f) :\Leftrightarrow f \in \mathbb{P}(Clause_n)$$

$$\text{Formula: Var} \to \mathbb{P}(\text{RawFormula})$$
$$\text{Formula}_n := \{f \in \mathbb{P}(RawFormula) \mid \text{formula}_n(f)\}$$

3.3.2. Output Condition

For the specification of the output condition, the definition of a valuation is needed. The definition is exactly the same as the definition of $clause_n(v)$ but for increasing the readability a new predicate is introduced.

$$\text{valuation} \subseteq \text{Var} \times \text{RawClause}$$
$$\text{valuation}_n(v) :\Leftrightarrow clause_n(v)$$

$$\text{Valuation: Var} \to \text{RawFormula}$$
$$\text{Valuation}_n := Clause_n$$

A formula is satisfiable when a valuation exists such that the formula is true in the valuation.

$$(\text{F, n}) \text{ is satisfiable} :\Leftrightarrow \exists v \in Valuation_n : v \vDash F$$

A formula is satisfied by a valuation when each clause is satisfied by the valuation.

$$\vDash \subseteq \text{RawFormula} \times \text{RawFormula}$$
$$v \vDash F :\Leftrightarrow \forall c \in F : v \vDash c$$

A clause is satisfied by a valuation when a literal is satisfied by the valuation.

$$\vDash \subseteq \text{RawFormula} \times \text{RawClause}$$
$$v \vDash c :\Leftrightarrow \exists l \in c : v \vDash l$$

A literal is satisfied by a valuation when it occurs in the valuation.

$$\vDash \subseteq \text{RawFormula} \times \text{RawLiteral}$$
$$v \vDash l :\Leftrightarrow l \in v$$

3.4. Java Program

From the listed pseudo-codes in Section 3.1 a Java program of the DPLL algorithm is implemented. The Java program contains both, the iterative and the recursive algorithm of the DPLL method.

To keep the implementation of the algorithm as simple as possible, it is decided to implement the data types of the algorithm with integers and arrays. This means that a literal is implemented as an integer, a clause is defined as an array of integers and a formula is an array of arrays of integers. The order of the literals inside the arrays does not matter for the implementation of the algorithm. Furthermore the performance increasing rules *unit propagation* and *pure literal elimination* are neglected. This makes the algorithm less efficient but it is then easier to give a JML specification for the Java program.

The recursive implementation of the algorithm takes an array of arrays of integers and an integer as input. The array of arrays of integers represents the original formula which shall be checked for satisfiability. The integer n represents the maximum value of the literals. The integer n is not necessary for the recursive algorithm and it is only passed to other functions. It will be used in the JML specification, thus it is already added to the Java implementation of the recursive algorithm.

The output of the function is a Boolean value which is true when the formula is satisfiable and false when the formula is unsatisfiable.

Inside the function several helper functions are called, which are fully listed in Ap-

pendix A.1. At first it is checked if the formula is empty, which would mean that all literals have been assigned in such way that all clauses are satisfied and thus the original formula is satisfiable. It the formula is not empty it is then checked if it contains an empty clause, which means that all literals in a clause have been assigned is such way that the clause reduces to false. If the formula is neither empty nor contains an empty clause, a substitution of the formula can be done. Since the performance increasing rules *unit propagation* and *pure literal elimination* are neglected, any remaining literal in the formula can be chosen. In this case the first integer in the first array of the formula is selected as the literal, which shall be substituted. Due to the implementation of the function *substitute* this integer always exists and is not already substituted.

In the function *substitute* a new formula object is created. All clauses of the original formula, which do not contain *val* are copied to the new formula. Afterwards all clauses of the new formula which contain *-val* are replaced by new clauses, where *-val* is deleted. If the function *substitute* is not called with the Boolean value *true*, but with *false*, *val* is exchanged with *-val*. This guarantees that a new formula object is returned, where all positions in the array are unequal to null and hence the position *formula[0][0]* always exist.

After selecting the literal, which shall be substituted, the depth-first search can be executed by recursive method calls.

```
1  public static boolean DPLLrec(int[][]formula, int n){
2      if(isEmptyFormula(formula, n)) return true;
3
4      else if(containsEmptyClause(formula, n)) return false;
5
6      else{
7          int val = formula[0][0];
8
9          if(DPLLrec(substitute(formula,val,true,n),n)==true)
10                 return true;
11
12         else{return(DPLLrec(substitute(formula,val,false,n),n));}
13     }
14 }
```

The iterative implementation of the algorithm has the same input and output as the recursive implementation. The integer n is now used in the implementation as a boundary condition.

The iterative method call needs a stack, which holds all substituted formulas in the depth-first search, which still need to be checked for satisfiability. It is decided to implement the stack as an array of formulas instead of using the Java utility class *Stack*. This is done, because it is then easier to specify the iterative algorithm with JML. The size of the stack is known, because at most n substitutions of the original formula can be done.

In the while-loop it is again checked if the current formula is empty, which indicates satisfiability. If the current formula is not empty but it contains an empty clause, two cases can take place. If the stack is empty, which means that either all valuations of the original formula have been checked or already the original formula contains an empty clause, unsatisfiability is detected and the function returns *false*. If the stack is not

empty, the current formula is replaced by the formula on top of the stack. If the current formula is not empty and does not contain an empty clause, a substitution is done. The literal which shall be substituted is again the first integer in the first array of the formula. The new formula, where all occurrences of *val* are substituted by *false* is put on the stack and the new formula where all occurrences of *val* are substituted by *true* becomes the current formula which is then investigated in the next iteration of the loop.

```
1  public static boolean DPLLiter( int[][]formula, int n){
2      int[][][] stack = new int[n][][];
3      int inStack = 0;
4      int[][]copy_formula = formula;
5
6      while(true){
7          if(isEmptyFormula(copy_formula,n)) return true;
8
9          else if(containsEmptyClause(copy_formula,n)){
10             if(inStack==0) return false;
11             inStack--;
12             copy_formula = stack[inStack];
13         }
14
15         else{
16             int val = copy_formula[0][0];
17             stack[inStack]=substitute(copy_formula,val,false,n);
18             inStack++;
19             copy_formula = substitute(copy_formula,val,true,n);
20         }
21      }
22 }
```

3.5. JML

The JML specification of the Java program is done in three stages to make it more comprehensible. The first version of the JML specification is very fundamental and only deals with preventing errors in the specification and avoiding precondition warnings. The second version introduces the predicates of the input and output conditions and the postconditions of the functions are specified. The first and second version are analyzed with ESC/Java2. The third version introduces invariants and termination terms of the loops. The third version is analyzed with Java KeY.

3.5.1. Version 1

The first version is specified in such a way, that ESC/Java2 does not highlight any errors and warnings concerning the preconditions of the functions. So postconditions of functions are only specified, when it is needed to get rid of a precondition warning of another function.

The main goal is to verify the JML specifications of the functions for the iterative and recursive algorithm. It can be seen that both the iterative and recursive algorithm have the same specification, because both functions take the same input and return the same output. In this version only the input of the functions is specified in JML with *requires* clauses to avoid precondition warnings. The output conditions will be specified in Version 2.

```
 /*@ public normal_behavior
    @ requires formula != null;
    @ requires formula.length > 0;
    @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
    @ requires n>=0;
@*/
public static boolean DPLLiter(int[][]formula, int n)

/*@ public normal_behavior
    @ requires formula != null;
    @ requires formula.length > 0;
    @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
    @ requires n > 0;
@*/
    public static boolean DPLLrec(int[][]formula, int n)
```

The function *substitute*, which is called inside the iterative and the recursive algorithm needs a specification of the postcondition, because in the recursive function *DPLLrec*, *DPLLrec* is called with a substituted formula. Thus a specification of the postcondition is needed, otherwise the preconditions of *DPLLrec* are not ensured.

```
/*@ public normal_behavior
    @ requires formula != null;
    @ requires formula.length > 0;
    @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);

    @ ensures \result !=null;
    @ ensures \result.length > 0;
    @ ensures (\forall int i; 0<=i && i< \result.length; \result[i]!=null);
@*/
    public static int[][] substitute (int[][]formula, int lit, boolean val, int n)
```

The JML specification of the remaining functions can be seen in Appendix A.2.1.

Applying ESC/Java2 on the full specification yields seven warnings. Four warnings concern a "possible violation of a postcondition" and they appear for the functions *substitute* and *deleteClause*. The other three warnings state that an "Array Index is possibly too large" and concern the functions *DPLLiter* and *DPLLrec*.

It can be seen that due to the specification of the postcondition for *substitute* no warning concerning a precondition arises for *DPLLiter* nor *DPLLrec*.

```
Dpll: DPLLiter(int[][], int) ...
------------------------------------------------------------------------
./DPLL1.java:27: Warning: Array index possibly too large (IndexTooBig)
                            int val = copy_formula[0][0];
                                                      ^
Execution trace information:
    Reached top of loop after 0 iterations in "./DPLL1.java", line 17, col 2.
    Executed else branch in "./DPLL1.java", line 20, col 8.
    Executed else branch in "./DPLL1.java", line 26, col 7.

------------------------------------------------------------------------
./DPLL1.java:28: Warning: Array index possibly too large (IndexTooBig)
                            stack[inStack]=substitute(copy_formula ...
                                 ^
Execution trace information:
    Reached top of loop after 0 iterations in "./DPLL1.java", line 17, col 2.
    Executed else branch in "./DPLL1.java", line 20, col 8.
```

```
18      Executed else branch in "./DPLL1.java", line 26, col 7.
19
20  ----------------------------------------------------------------------
21      [0.141 s 10835512 bytes] failed
22
23
24
25  Dpll: DPLLrec(int[][], int) ...
26  ----------------------------------------------------------------------
27  ./DPLL1.java:50: Warning: Array index possibly too large (IndexTooBig)
28                          int val = formula[0][0];
29                                    ^
30  Execution trace information:
31      Executed else branch in "./DPLL1.java", line 47, col 7.
32      Executed else branch in "./DPLL1.java", line 49, col 6.
33
34  ----------------------------------------------------------------------
35      [0.106 s 11745336 bytes] failed
```

The complete output of ESC/Java2 is listed in Appendix A.2.1.

3.5.2. Version 2

The second version of the JML specification refines the first version by introducing the predicates of the formal specification of the algorithm as *static pure model functions*. The predicates cover the definition of the input condition from Chapter 3.3.1 and the output condition from Chapter 3.3.2. Furthermore some helper functions are defined which help to specify the postconditions of the implemented functions. These helper functions describe the properties "literal is in a clause", "literal is in a formula", "clause is in a formula" and "a literal is removed from a clause". With the predicates and the helper function all postconditions of the functions are specified.

The main goal is again to verify the specifications of the methods for the iterative and recursive algorithm. Due to the introduction of predicates the preconditions of both functions are now described with a model function. The postconditions are also specified with a model function.

```
1   /*@ public normal_behavior
2       @ requires validformula(formula,n);
3       @ requires n>=0;
4
5       @ assignable \nothing;
6
7       @ ensures \result == satisfiable(formula,n);
8   @*/
9       public static boolean DPLLiter(int[][]formula, int n)
10
11
12  /*@ public normal_behavior
13      @ requires validformula(formula,n);
14      @ requires n>=0;
15
16      @ assignable \nothing;
17
18      @ ensures \result == satisfiable(formula,n);
19  @*/
20      public static boolean DPLLrec(int[][]formula, int n)
```

The predicate *validformula* meets the definition of a formula from Chapter 3.3.1 and states that the formula exists and that each clause in the formula must be valid. For the validity of a clause a further predicate is introduced which states that each literal in the clause is in the specified range and that a literal can not occure both positive and negative in a clause. The full specification is listed in Appendix A.2.2.

```
1   @ ensures \result <==> f!=null && (\forall int i; 0<=i && i<f.length; validclause(f[i],n));
2   @ public static pure model boolean validformula(int[][] f,int n);
```

The result of the pure model function *satisfiable* states if a valuation in the set of all valuations exists, such that the valuation satisfies the formula.

```
1   @ ensures \result <==> (\exists int[] v; valuation(v,n); ValuationSatFormula(f,v,n));
2   @ public static pure model boolean satisfiable(int[][] f, int n);
```

To get ESC/Java2 running this specification, two specifications have to be modified. First the postcondition of the function *DPLLrec*, which state that the result of the function ensures if the given formula is satisfiable or not, needs to be commented out, otherwise ESC/Java2 gets stuck at this function and runs into an infinite loop. The second modification concerns the definition of *validClause*.

```
1  @ ensures \result <==>
2    @     (\forall int i; 0<=i && i<c.length; validliteral(c[i],n)) &&
3    @     (\forall int j; 0<=j && j<=n; consistent(j,c)) &&
4    @     c!=null;
5    @ public static pure model boolean validclause(int[] c,int n);
```

ESC/Java2 gets stuck already in the first function *DPLLiter* with the full specification of *validClause*. Therefore the specification is weakened to *c != null* and the part which states that every literal in a clause needs to be valid and consistent is neglected.

Applying ESC/Java2 on the modified specification yields six warnings. One warning already appeared in the first version and highlights that an "Array Index is possibly too large" in the function *DPLLiter*. The other warnings appear for the postconditions of the functions *DPLLiter*, *deleteClause*, *substitute* and *deleteLit*. For the function *DPLLiter* it could not be shown that the result ensures if the formula is satisfiable or not. The other postconditions warnings arise when a postcondition includes a `\forall` expression and the range expression is a predicate defined as a *static pure model function*.

```
1  Dpll: deleteLit(int[], int, int) ...
2  ------------------------------------------------------------------------
3  ./DPLL2.java:130: Warning: Postcondition possibly not established (Post)
4          }
5          ^
6  Associated declaration is "./DPLL2.java", line 122, col 3:
7          @ ensures (\forall int l; validliteral(l,n); LitInClause(l, \r ...
8            ^
9  Execution trace information:
10     Executed else branch in "./DPLL2.java", line 128, col 7.
11     Executed return in "./DPLL2.java", line 128, col 7.
12
13 ------------------------------------------------------------------------
```

ESC/Java2 is able to check the specifications of the lower level functions, where it is not necessary to use a model function in a quantified expression. Thus the specifica-

tions of the functions *isEmptyClause*, *isEmptyFormula*, *containsEmptyClause*, *search*, *CutFormula* and *CopyClause* are checked by ESC/Java2. The full output of ESC/Java2 is listed in Appendix A.2.2.

```
Dpll: CopyClause(int[], int, int) ...
    [0.031 s 16522104 bytes] passed
```

3.5.3. Version 3

The third version refines the second version by including all invariants and termination terms for loops. Furthermore the recursive function *DPLLrec* is assigned with a `measured_by` clause, specifying the termination term of the recursion.

The `measured_by` clause defines the maximal number of recursions for the input. Since a formula in m literals can only be substituted m times, the depth of the recursions is given by the number of literals in the formula. The model function *numVar* counts the number of literals occurring in the formula. The input variable n can not be used as a boundary, since n defines the range of the literals and not the current number of occurring literals.

```
/*@ public normal_behavior
    @ requires validformula(formula,n);
    @ requires n>=0;

    @ assignable \nothing;

    @ measured_by numVar(formula,n);

    @ ensures \result == satisfiable(formula,n);
@*/
    public static boolean DPLLrec(int[][]formula, int n)
```

The specification of the pre- and postconditions for *DPLLiter* remains the same as in Version 2 of the JML specification, but a loop invariant including a termination term needs to be added.

A loop invariant describes all properties which need to hold before and after every iteration of the loop. Since none of the input variables is changed, the preconditions of the method are repeated in the loop invariant, because they still hold. Furthermore all properties concerning local variables are added. It holds that *stack* always exists and that its length is at most n. The counter *inStack* is always positive and *copy_formula* is a valid formula. It also holds that every formula which is put onto the stack is a valid formula. The next property concerns the satisfiability of the original formula and states that the original formula is satisfiable if and only if the current formula is satisfiable or one of the formulas on the stack is satisfiable. Furthermore the number of literals of the formulas on the stack decreases and the current formula has the same number of literals as the formula on top of the stack.

The termination term is defined by the number of unvisited knots of the search space of each formula on the stack including the current formula.

```
 /*@ public normal_behavior
```

```
2       @ requires validformula(formula,n);
3       @ requires n>=0;
4
5       @ assignable \nothing;
6
7       @ ensures \result == satisfiable(formula,n);
8   @*/
9       public static boolean DPLLiter(int[][]formula, int n){
10          int[][][] stack = new int[n][][];
11          int inStack = 0;
12          int[][]copy_formula = formula;
13
14  /*@ loop_invariant
15      @ validformula(formula,n) && n>=0 &&
16      @
17      @ stack!=null && stack.length<=n &&
18      @ inStack>=0 &&
19      @ validformula(copy_formula,n) &&
20      @ copy_formula.length <= formula.length &&
21      @
22      @ (\forall int i; 0<=i && i<inStack; validformula(stack[i],n)) &&
23      @
24      @ satisfiable(formula,n) <==>
25      @     (satisfiable(copy_formula,n) ||
26      @     (\exists int i; 0<=i && i<inStack; satisfiable(stack[i],n))) &&
27      @
28      @ (\forall int i; 0<=i && i<inStack-1; numVar(stack[i],n)>=numVar(stack[i+1],n)) &&
29      @
30      @ inStack > 0 ==> numVar(copy_formula,n) == numVar(stack[inStack-1],n);
31
32      @ decreases (\sum int i; 0<=i && i<inStack;
33      @   power2(numVar(stack[i],n)+1)-1) + power2(numVar(copy_formula,n)+1)-1;
34
35      @ assignable inStack, stack[*], copy_formula[*];
36  @*/
37          while(true){
38              ...
39          }
40      }
```

This JML specification is not analyzed with ESC/Java2 anymore. It is analyzed with Java KeY.

At first Java KeY is applied on the full specification. Since some functions cannot be proved immediately, a second attempt where the predicate *validClause(c,n)* is again weakened to $c! = null$ is performed. The output for every function is listed in Figure 3.3.

As an example of a successful and an unsuccessful result, the outputs of KeY for the functions *containsEmptyClause* and *DPLLiter* are shown. The weakened specification of *containsEmptyClause* can be proved by KeY, which can be seen in Figure 3.1.

The specification of the function *DPLLiter* can not be proved by KeY. It can be seen in Figure 3.2, that the proof tree splits into several branches. The prover is not able to apply proper rules on those branches and thus cuts the problem into subproblems. It is also tried do apply the solvers Yices, Z3 and Simplify, which also does not help to further the proof. It is not tried to apply interactive proofs in KeY.

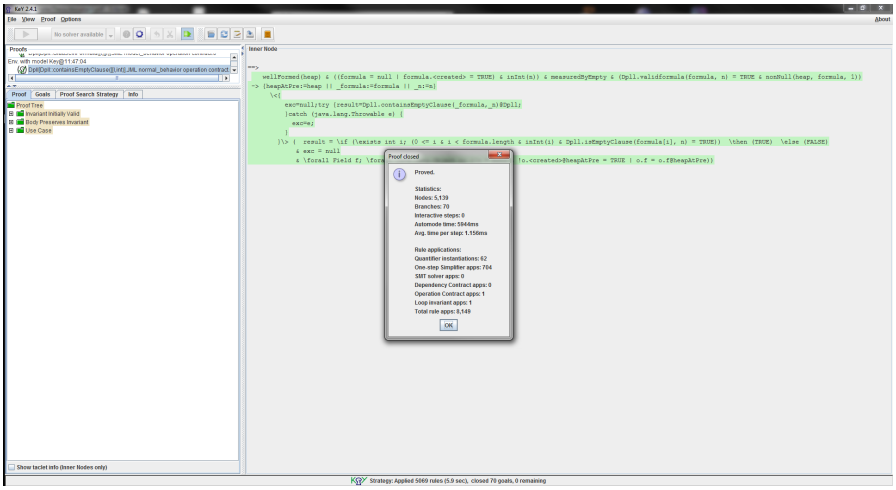


Figure 3.1.: Proving containsEmptyClause with the simplified specification in KeY

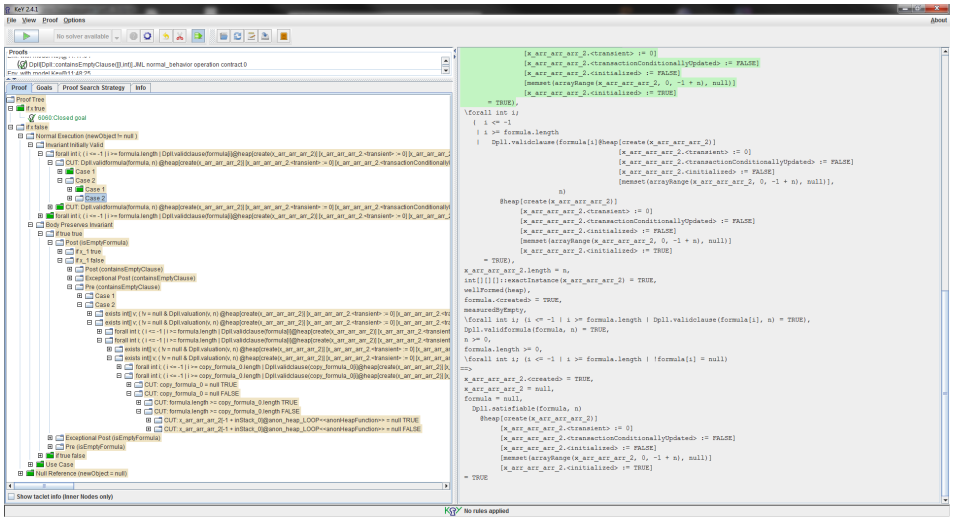


Figure 3.2.: Proof tree of DPLLiter in KeY

Running Java KeY on all functions leads to the following output for the specification, which can be seen in Figure 3.3. The table of functions is organized in ascending order beginning with stand-alone functions and ending with the functions, which execute the whole algorithm.

Function	Output of Java KeY	Output of Java KeY with a simplification of *validClause*
isEmptyClause	Proved immediately	
isEmptyFormula	Proved immediately	
search	Proved immediately	

CutFormula	The part of the proof tree, where $j >= 0$, cannot be proved. In the normal execution of this part it can neither be proved that the invariant is initially valid, the body preserves the invariant nor that the invariant implies the postcondition (use case). This means that no part of the normal execution is proved.	Proved immediately
CopyClause	No part of the normal execution, where *clause.length* $>= 0$, can be proved.	Proved immediately
containsEmptyClause	It can be proved that the invariant is initially valid and the use case is verified. It is not proved that the body preserves the invariant.	Proved immediately
deleteLit	Applying only the *ensures* clause containing the predicate *LitInClause* the branches "Exceptional Post(search)" and "Pre(search)" are proved. The branch "Post(search)" is not proved. It is also attemted to write the postcondition without predicates by using quantifiers. This approach also leads to the same output. If the method *search* would be defined as a *pure* function without any header, the proof of *deleteLit* succeeds.	A branch of the "Post(search)" proof tree, where *pos==-1* is proved too. In the other branch of "Post(search)" the branches "Exceptional Post(CopyClause)" and "Pre(CopyClause)" are proved. "Post(CopyClause)" is not proved.

deleteClause	In the normal execution of the function the branch where the length of the formula is greater than 0 can not be proved.	The simplification does not help to further the proof.
substitute	In both execution branches, where *val* is either true or false, the condition "Post(deleteClause)" is not proved. In both branches the conditions "Exceptional Post(deleteClause)" and "Pre(deleteClause)" can be proved.	The simplification does not help to further the proof.
DPLLrec	The branches "Exceptional Post(isEmptyFormula)" and "Pre(isEmptyFormula)" are proved. The condition "Post(isEmptyFormula)" can not be proved.	The simplification does not help to further the proof.
DPLLiter	In the normal execution branch, where $n > 0$, only the use case can be proved. It is not proved that the invariant is initially valid and that the body preserves the invariant.	The simplification does not help to further the proof.

Figure 3.3.: Output for every function in KeY

3.6. TLA

The specification of the DPLL method in TLA is done with sets. At first the set of literals is modelled as a set of integers ranging from *-RangeLit* to *RangeLit* excluding zero. *RangeLit* is a constant in the module and its value has to be defined in the model of the TLC Model Checker. Based on the set of literals the set of clauses can be created by taking all subsets of the set of literals excluding those sets where a literal occurs both positive and negative. Based on the set of clauses the set of all formulas is defined. A single formula is then an element of the set of all formulas.

To make model checking more traceable, the specifications of the set of clauses and

the set of formulas include a restriction on the cardinality of these sets. This is done with the help of the two constants *ClauseLength* and *ClauseNumber*. *ClauseLength* limits the maximal number of literals in a clause and *ClauseNumber* limits the maximal number of clauses in a formula. The values of both constants are again defined in the TLC Model Checker.

The TLA specification of the DPLL method covers only the iterative algorithm. Thus a stack is defined. The stack is modelled as an array of length *RangeLit*, because the maximum number of formulas occurring in the stack is the highest value of a literal. Each formula in the stack contains one literal less than its predecessor, because a substitution of a literal is done before the formula is put onto the stack.

For better readability the function *substitute* is predefined as a "macro". The macro *substitute(F,n)* substitutes all occurrences of n in F by deleting all clauses containing n and deleting all occurrences of the literal $-n$ in the remaining clauses.

With the definition of the macro and the variables, the pseudo-code of the iterative DPLL method from Section 3.1 can be implemented straight forward in PlusCal and is listed below.

```
1  --algorithm Dpll{
2
3  variables
4      Literals = {l \in -RangeLit..RangeLit: (l > 0 \/ l<0)};
5      Clause = {c \in SUBSET(Literals): Cardinality(c)<=ClauseLength
6                /\ \A l \in 1..RangeLit: \neg(l \in c /\ -l \in c)};
7
8      FormulaSet = {f \in SUBSET(Clause): Cardinality(f)<=ClauseNumber};
9
10     Formula \in FormulaSet;
11     CFormula = Formula;
12     OFormula = Formula;
13
14     emptyformula = FALSE;
15     emptyclause = FALSE;
16     sat = 0;
17     lit = 0;
18
19     inStack = 0;
20     stack = [l \in 1..RangeLit |-> {}];
21
22
23 macro substitute(F,n){
24     F:= {c\{-n}: c \in F\{cl \in F: n \in cl}}
25 }
26
27 {
28    while(sat = 0){
29         emptyformula:= Formula={};
30         emptyclause := \E c \in Formula: c={};
31
32         if(emptyformula = TRUE) {sat := 1;}
33
34         else if(emptyclause = TRUE){
35             if(inStack = 0) {sat := -1;}
36
37             else{
38             Formula:=stack[inStack];
39             inStack:=inStack-1;}
```

```
40          }
41
42          else{
43              lit := CHOOSE l \in Literals: \E c \in Formula : l \in c;
44              CFormula := Formula;
45
46              substitute(Formula,lit);
47              substitute(CFormula,-lit);
48
49              inStack:= inStack+1;
50              stack[inStack]:=CFormula;
51          };
52      };
53  }
54  }
```

The PlusCal algorithm is translated by the TLA Toolbox into a TLA model. The complete listing of the model including the TLA translation is given in Appendix A.3.

Two properties of the model shall be checked by the TLC model checker. The first property *Termination* guarantees that the algorithm terminates. The second property *IsCorrect* ensures that the output of the algorithm meets the output condition of the formal specification in Section 3.3.2. Therefore all predicates of the output condition are specified. The property *isCorrect* states, that whenever the algorithm terminates, *sat* is equal to 1 if and only if the original formula is satisfiable.

It can be seen that a new variable *pc* occurs in the definitions of the properties. This variable is a program counter which is automatically added by the translation of the PlusCal algorithm into a TLA model. A program counter indicates which step of the model can be executed at each state of the model. The program counter *pc* has three states in this model: *"Lbl_1"*, *"Lbl_2"* and *"Done"*. The *"Lbl_1"* step deals with the while loop of the algorithm and the *"Lbl_2"* step is in charge of the execution of the else branch inside the loop where substitutions of the formula are done. The *"Done"* state of the program counter is reached whenever the value of *sat* is unequal to 0. After the program counter is set to *"Done"* all steps are disabled and the algorithm terminates.

```
1   \* WHAT TO CHECK
2
3   \* Algorithm terminates
4   Termination == <>(pc = "Done")
5
6   \* Algorithm satisfies output condition
7   Valuation == {c \in SUBSET(Literals): Cardinality(c)<=ClauseLength /\
8                 \A l \in 1..RangeLit: \neg(l \in c /\ -l \in c)}
9
10  ValuationSatLit(v,l) == v \in Valuation /\ l \in v
11
12  ValuationSatClause(v,c) == \E l \in Literals: l \in c /\
13                             ValuationSatLit(v,l)
14
15  ValuationSatFormula(v,f) == \A c \in f: ValuationSatClause(v,c)
16
17  Satisfiable(f) == \E v \in Valuation: ValuationSatFormula(v,f)
18
19
20  IsCorrect == [](pc = "Done" => (sat = 1 <=> Satisfiable(OFormula)))
```

Before the TLC model checker can be applied on the module, the model needs to be set up. This means that all constants occurring in the module need to be specified. Furthermore the "behaviour spec", which is a formula describing the behaviour of the system, needs to be defined. Finally all invariants and properties which shall be checked can be listed in "What to check?". For this model the behaviour of the system is described by the temporal formula *Spec* occurring in the TLA model. The constants are set to 3 for *ClauseLength* and *RangeLit* and to 4 for *ClauseNumber*. It shall be checked, that the algorithm does not run into a deadlock. Therefore it is necessary to include the PlusCal option for termination, to prevent an error in the TLC Model Checker. Furthermore the two specified properties *Termination* and *IsCorrect* shall be checked. The settings of the model in the TLC model checker can be seen in Figure 3.4.

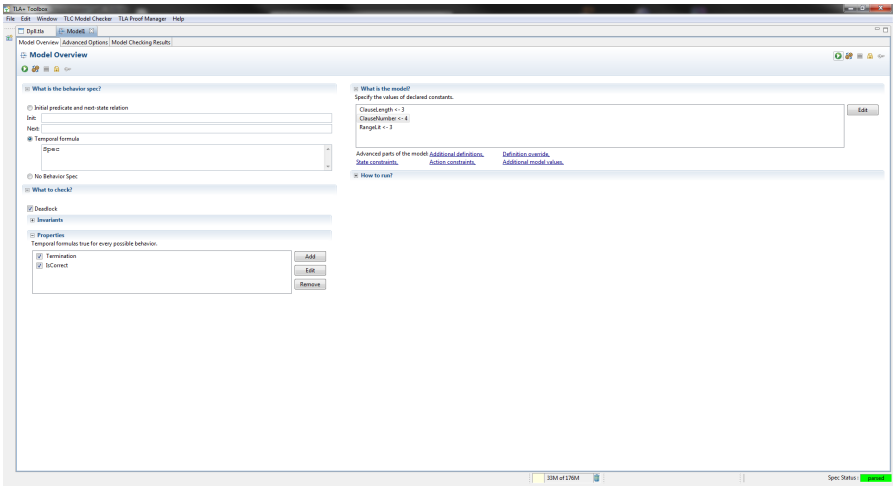


Figure 3.4.: Setting up the model in the TLC Model Checker

After setting up the model, the TLC Model Checker is applied. The TLC Model Checker runs through all formulas, satisfying the definitions of the constants. This means that all formulas up to four clauses, where the clauses contain literals from the set $\{-3, -2, -1, 1, 2, 3\}$ and have a maximum length of three, are checked.

The TLC Model Checker does not detect any error in the model, which can be seen in Figure 3.5. This means that all properties are correct in the specified range. After running the TLC Model Checker, the correctness of the properties is not guaranteed for all formulas. It is only ensured that they are correct in the given range. Since the checked set of formulas is extensive, the confidence level is high that the properties hold for all formulas.

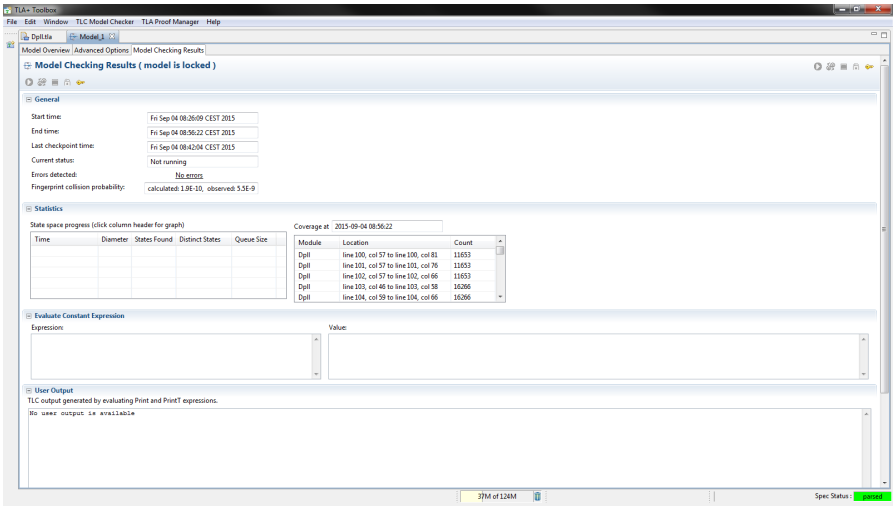


Figure 3.5.: Output of the TLC Model Checker

3.7. Alloy

The specification of the DPLL algorithm in Alloy is done in three models. The first model introduces the input and output conditions of the DPLL algorithm and returns all instances of formulas in the given literal range, which are satisfiable. The second model defines the DPLL method. The instances of this model show successful runs of the DPLL algorithm. The third model is a combination of the first and the second model which checks the assertion that the output of the DPLL algorithm satisfies the output conditions.

3.7.1. Model 1

The first model defines the input and output conditions of the algorithm and introduces all objects which occur in the algorithm. The model shall return all instances of formulas which are satisfiable. At the beginning of the model the occurring objects are defined. Since Alloy discourages the use of integers in modelling (although they are supported in a limited way), all literals are defined explicitly. The occurring literals in this model are A and B with their negations nA and nB. The set of literals is defined as an abstract set, because only the explicit defined literals shall occur in a formula. So in this model the clauses in the formula can only contain up to two literals.

The set of clauses includes a relation *LitInClause* which maps a set of literals to a clause. Furthermore a `fact` is added directly in the definition of the signature, which ensures the consistency of a clause. A formula is defined as a set of clauses by introducing the corresponding relation in the signature of the formula. Furthermore two `facts` concerning the specification of a clause are added. `Facts` are constraints which are assumed to always hold in a module. The first `fact` states that a clause always belongs to a formula and the second one states, that whenever two clauses contain the same literals they are the same objects. Although not strictly required, these facts are introduced, in order to restrict the models to “compact” ones that do not contain superfluous clauses or duplicate formulas.

The definition of a valuation is the nearly the same as the definition of a clause, but in a valuation every literal has to occur, either positive or negative. A fact, which states that valuations are unique, is added in the model.

After all objects are introduced the predicate *satisfiable* is defined. It takes a formula as input and looks for a valuation that satisfies the input formula.

The model shall find instances which satisfy the predicate *satisfiable* in the range of 4. This means that at most 4 objects of each signature can occur in an instance. The restriction `but 1 formula, 1 valuation` states that only one formula shall be investigated in each new instance and that the predicate shall only return one valuation. This is done for an easier comprehension of the instances.

```
module runSatisfiable

//Literals
abstract sig literal {}
one sig A, nA, B, nB extends literal {}

//Clause
sig clause {LitInClause: set literal}{
    not A + nA in LitInClause and
    not B + nB in LitInClause
}

//Formula
sig formula{ ClauseInFormula: set clause}{}


//Facts about clauses
fact allClausesBelongToAFormula {
       all c:clause | some f:formula | c in f.ClauseInFormula
}

fact clausesAreUnique {
  all c1:clause | all c2: clause |
       c1.LitInClause = c2.LitInClause => c1 = c2
}


//Valuation
sig valuation {LitInValuation: set literal}{
    not A + nA in LitInValuation and
    (A in LitInValuation or
    nA in LitInValuation) and
    not B + nB in LitInValuation and
    (B in LitInValuation or
    nB in LitInValuation)
 }


fact valuationsAreUnique {
  all v1,v2: valuation | v1.LitInValuation = v2.LitInValuation => v1 = v2
}


//satisfiable
pred satisfiable(f:formula){
   some v:valuation | all c : f.ClauseInFormula | some l:c.LitInClause | l in v.LitInValuation
}

```

```
49
50 run satisfiable for 4 but 1 formula, 1 valuation
```

The Alloy Analyzer finds several instances of formulas which satisfy the predicate *satisfiable*. An example for such an instance can be seen in Figure 3.6. The formula consists of three clauses {nB}, {A, nB} and {nA, B}. The valuation which satisfies the formula is {nA, nB}. The objects *formula* and *valuation* both contain a comment *satisfiable_* which indicates the relation of those objects given by the predicate *satisfiable*.

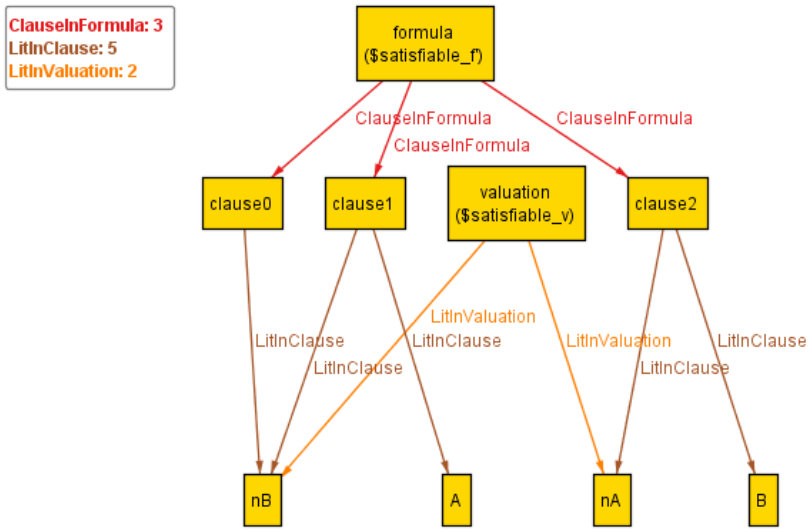


Figure 3.6.: Instance of a formula which is satisfiable

3.7.2. Model 2

The second model deals with the implementation of the DPLL algorithm. The specifications of the objects *literal*, *clause* and *formula* remain the same as in the first model. The only difference is that the set of literals is expanded to three literals A, B and C.

To show the several states of the algorithm, where each state corresponds to an iteration of the while-loop in the iterative algorithm from Section 3.1, it is necessary to introduce an object *state*. A *state* is mapped to exactly one *formula*. Since the *state* objects indicate states of the algorithm, they need to be ordered, which is done by using the `util/ordering module` for modeling ordered states.

The initial state of the algorithm is described in the `fact initialState`. It assorts that the first formula can only contain up to four clauses. The restriction on the number of clauses for the original formula is necessary, because the number of clause objects is limited. If the restriction of clauses in the original formula is neglected, the algorithm might not be able to create new clauses and terminates permanently in a state that violates the postcondition. With a total number of maximal 10 clauses, the limit of the clauses in the original formula is set to 4, because in each new state at least one clause is omitted and all the remaining clauses can change and thus become new clause objects.

The algorithm may terminate with a formula containing an empty clause, thus a total number of 10 clauses is necessary for a formula with four clauses in three literals.

The algorithm itself is modelled as a `fact`. It defines the relation between a *state s* and its successor *state s'*. Whenever the current formula in *state s* is empty or includes an empty clause the formula is not changed any more and thus the states remain the same. Otherwise a substitution of the current formula in *state s* is done by using the predicate *substitute*.

In *substitute* a literal *l* which occurs in the input formula *f* is selected. At first all clauses of *f* which do not include the literal *l* are selected in the set *cs0*. In the set *cs1* the literal *l.negate* is deleted from all clauses occurring in *cs0*. This is done by the use of the "dot join-operator" [24], which represents relational composition. The set of clauses from the output formula *f'* is then set to *cs1*.

The algorithm is called by the predicate *solve* which seeks for formulas where the last state of the algorithm returns the empty formula or a formula with an empty clause. If the *currentFormula* of the last *state* is an empty set, the algorithm found a way to assign all literals in such a way that the formula evaluates to true. If the last formula contains an empty clause, the algorithm assigned all literals in such a way that the formula evaluates to false, which makes it invalid.

```
1  module runDpll
2
3  open util/ordering[state] as ord
4
5  //States
6  sig state{currentFormula: one formula}
7
8
9  //Substitute
10 pred substitute[f, f': one formula]{
11     one l: f.ClauseInFormula.LitInClause | {
12          let cs0 = { c: f.ClauseInFormula | not l in c.LitInClause } |
13          let cs1 = cs0.{c1:clause, c2:clause | c2.LitInClause = c1.LitInClause-l.negate} |
14            f'.ClauseInFormula = cs1
15     }
16 }
17
18 //Initial State
19 fact initialState {
20    let s0 = ord/first | #s0.currentFormula.ClauseInFormula <5
21 }
22
23 //Algorithm
24 fact DPLL{
25     all s: state, s': ord/next[s] {
26         (#s.currentFormula.ClauseInFormula = 0 or
27         some c: s.currentFormula.ClauseInFormula | #c.LitInClause = 0) =>
28              s' = s
29         else
30             substitute[s.currentFormula ,s'.currentFormula]
31     }
32 }
33
34 //run Algorithm
35 pred solve{
36     #ord/last.currentFormula.ClauseInFormula = 0
37     or some c: ord/last.currentFormula.ClauseInFormula |
```

```
38          #c.LitInClause = 0
39  }
40
41  run solve for 4 but 10 clause
```

The complete model can be seen in Appendix A.4.2.

An example for a run of the algorithm can be seen in Figure 3.7. This model has been customized by hiding several relations to make the output more clearly. It can be seen that the first state *state0* consists of a formula with 4 clauses {{B},{B, nC},{A, nC},{nA,C}}. In the step from *state0* to *state1* the literal B is selected and thus all clauses containing B are deleted. In the next step the literal nA is selected and thus only a clause containing nC remains. The last literal is selected in the last step and thus a state containing an empty formula is reached. So it can be seen that the original formula is satisfiable with the valuation {nA, B, nC}.

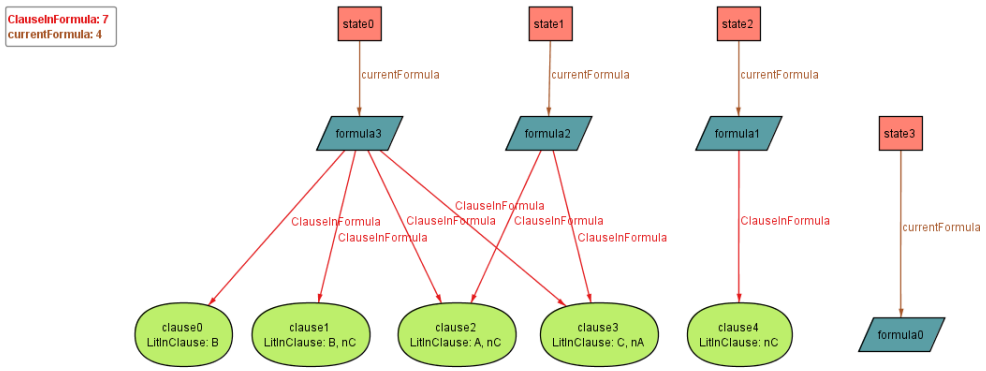


Figure 3.7.: Run of the DPLL algorithm in Alloy

3.7.3. Model 3

The third model is a combination of the first and the second model. It contains all the previous definitions of the objects *literal*, *clause*, *formula*, *state* and *valuation* including their facts for uniqueness and belongings. Furthermore the predicate *satisfiable* as well as the whole algorithm is listed in this model. The complete model is listed in Appendix A.4.3.

The goal of this model is to check the output condition of the algorithm from Section 3.3.2. Therefore an assertion is specified, which states that the algorithm terminates with an empty formula if and only if the predicate *satisfiable* is fulfilled. This means that a valuation exists which makes the formula true. Furthermore the assertion states that the algorithm terminates either in a state which consists of an empty formula or a formula containing an empty clause.

```
1  assert solve{
2    (#ord/last.currentFormula.ClauseInFormula = 0 <=> satisfiable[ord/first.currentFormula])
3     and
4    (#ord/last.currentFormula.ClauseInFormula = 0 or
5     some c: ord/last.currentFormula.ClauseInFormula | #c.LitInClause = 0)
```

```
6 }
7
8 check solve for 8 but 5 state, 10 clause
```

This assertion is checked by the Alloy Analyzer. The Alloy Analyzer does not find a counterexample, which can bee seen in Figure 3.8. This means that all formulas in the

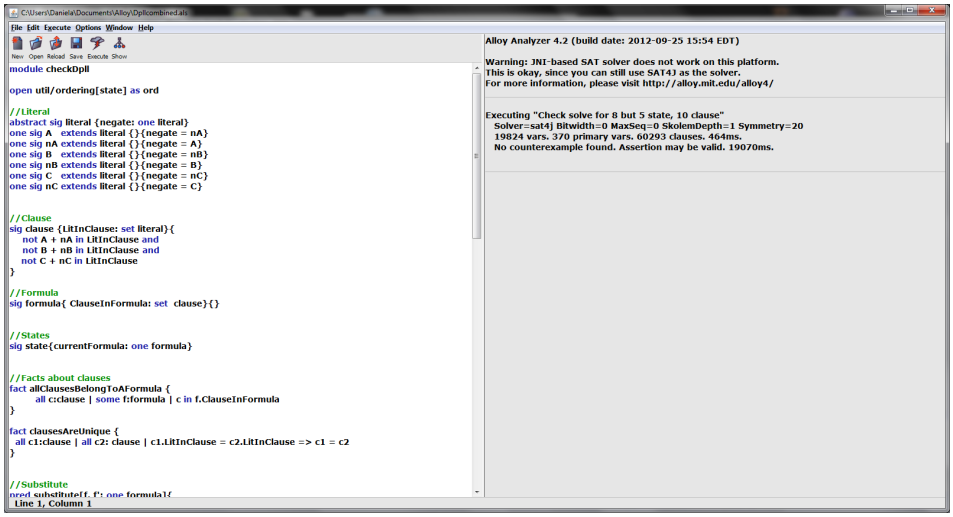


Figure 3.8.: Output of the Alloy Analyzer

given range do not violate the assertion. It can not be guaranteed that the assertion is valid for all formulas, it can only be guaranteed for the given range. Since the range is quite comprehensive, the confidence level is high that the assertion holds for all formulas.

It is odd that the Alloy Analyzer needs a restriction to 5 states, because 4 states should be enough for three literals. After the original state, where all three literals may occur, each successor state has one literal less. If the restriction is set to 4 states, the Alloy Analyzer returns counterexamples which do not violate the assertion. This deviation in the number of states is also noticed for formulas in two, respectively four literals.

3.8. VDM

Since it is possible to model recursive functions in VDM, it is decided to model both the iterative and the recursive algorithm. Both models are described in VDM++ to make use of object-oriented design. After setting up the models, for each model a corresponding `trace` is defined, which simulates an execution of the model. All `traces` are analyzed in the *Combinatorial Testing* perspective in Overture, which generates and runs all possible test cases. Furthermore proof obligations for both algorithms are generated in Overture.

3.8.1. Recursive Algorithm

Due to the rich specification language of VDM, all occurring types in the model can be described with sets. Invariants, which are added directly after an introduction of a type, add restrictions on the corresponding type.

```
public Clause = set of Literal
inv c == forall l in set c & not (l in set c and -l in set c);

public Literal = int
inv l == l <= n and l>=-n and l <> 0;

public Valuation = set of Literal
inv v == (forall l in set {1,...,n} &
            (l in set v or -l in set v) and (not (-l in set v and l in set v)));
```

A clause is defined as a set of literals, together with the invariant that a literal must not occur both positively and negatively in a clause. A literal is defined as an integer of the set $\{-n, \ldots, n\} \backslash \{0\}$. The integer n is a constant, which is defined in the model. Constants are defined in the section `values`. This model contains two constants n and *numOfClause*. The variable n defines the border for the literals and *numOfClause* restricts the number of clauses in a formula. The constant *numOfClause* is used for setting up test cases of the model.

Unfortunately it is not possible to introduce named predicates for the invariants to increase the readability. If a predicate is defined the invariants are still type checked, but the combinatorial testing fails. Therefore it is decided to define all properties of the invariants explicitly.

```
values
  public n: int = 2;
  public numOfClause: int = 4;
```

In this model the formula which shall be analyzed is not defined globally, it is defined locally in the execution of a trace.

After all types and constants are defined, the algorithm is modelled as a function.

```
public Dpllrec: set of Clause -> bool
Dpllrec(f) ==
    if f = {}
        then true
    elseif exists c in set f & c = {}
        then false
    else
         let l in set {-n,...,n}\{0} be st (exists c in set f & l in set c) in (
            if Dpllrec(substitute(l,f)) = true
               then true
            else Dpllrec(substitute(-l,f)))
pre validFormula(f)
post RESULT = satisfiable(f)
measure varInFormula;
```

The function *Dpllrec* takes a set of clauses as input and returns a boolean value. The algorithm is directly adapted from the pseudo code listed in Section 3.1. The complete listing of the model can be seen in Appendix A.5.1.

The input formula f of the algorithm must be a valid formula. This is guaranteed by adding a corresponding precondition after the definition of the function. The post-condition of this function ensures that the resulting boolean value matches the definition when a formula is satisfiable, given in Section 3.3.2. Both predicates *validFormula(f)*

and *satisfiable(f)* are defined as functions.

```
public validFormula: set of Clause -> bool
validFormula(f)==
   forall c in set f & validClause(c);
```

The output of the function *validFormula* states if all clauses in the formula are valid clauses. Clauses are valid if and only if the literals in the clause are in the specified range and do not occur both positively and negatively in the clause.

```
public satisfiable: set of Clause -> bool
satisfiable(f) ==
 let pcl = power({1,...,n} union {-n,...,-1}) in (
      let cl = {c | c in set pcl & (forall l in set {1,...,n} &
             (l in set c or -l in set c) and (not (-l in set c and l in set c)))} in (
        exists val in set cl & valuationSatFormula(f,val)));
```

The function *satisfiable* returns if a valuation exists, which satisfies the formula. This means that for all clauses in the formula there exists a literal in the clause which also occurs in the valuation. The set of all valuations *cl* needs to be defined explicitly in the function *satisfiable*. Therefore the power set of $\{-n, \ldots, n\} \backslash \{0\}$ is generated, but only those elements from the power set are taken where every literal is included either positively or negatively.

Since the function *Dpllrec* is a recursive function, a `measure` function which defines the recursion depth is added. The function *varInFormula* returns the number of literals occurring in the formula. After each substitution the formula contains one literal less, therefore the number of literals defines the maximum number of recursive calls.

```
traces
S1: let pcl = power({1,...,n} union {-n,...,-1}) in (
      let cl = {c | c in set pcl & (forall val in set c &
          (not (-val in set c and val in set c)))}
      in (
       let cl1 = power(cl) in (
          let f in set cl1 be st card(f)<=numOfClause in Dpllrec(f))))
```

At the end of the model a `trace` is defined, which describes an execution of the model. Therefore the set of all clauses needs to be defined in the trace. A formula f is then a subset of the set of clauses, where the cardinality of f is less or equal to the predefined constant *numOfClause*. The restriction on the number of clauses is done to make model checking more traceable. The recursive function *Dpllrec* is then called with f as the input formula.

In the combinatorial testing perspective all possible executions of the traces are performed. This yields to 3610 different test cases. None of the test cases detects a run-time error, in particular a violation of the precondition or postcondition of *Dpllrec*, which is indicated by the green ticks, which can be seen in Figure 3.9. An example for a certain test case can be seen in Figure 3.10. In this test case the algorithm is called with the input formula $\{\{-1, 2\}, \{-1\}, \{-2, -1\}, \{-2, 1\}\}$ and the result is *true*, which means that the input formula is satisfiable. This can easily be checked, because the valuation $\{-2, -1\}$ satisfies the formula.

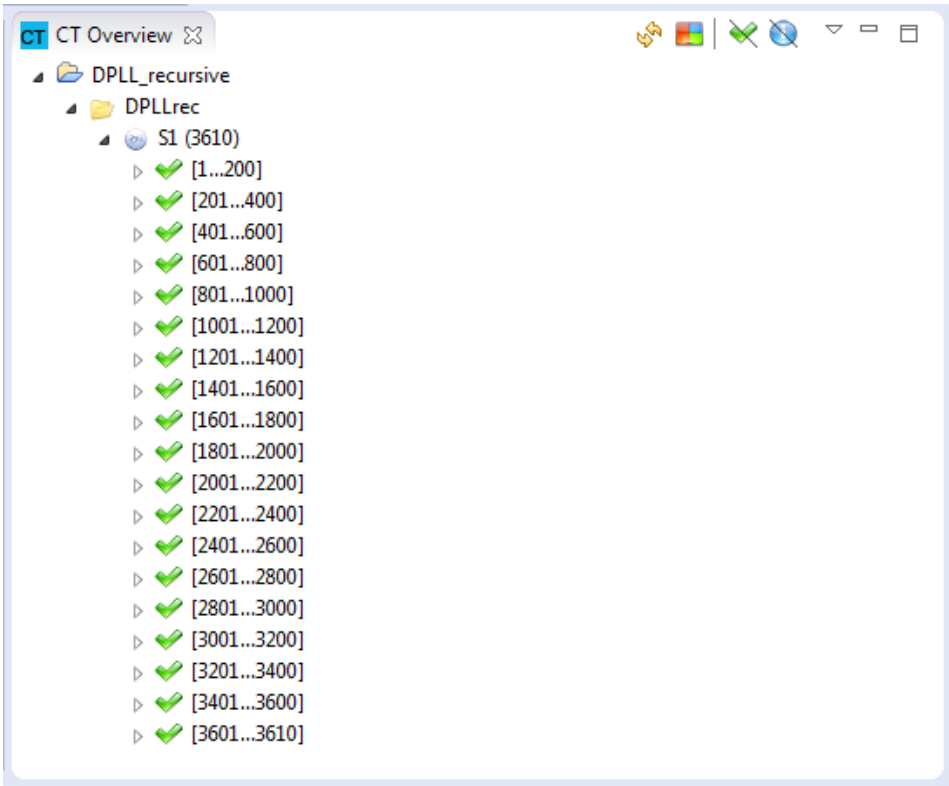


Figure 3.9.: Result after executing all test cases

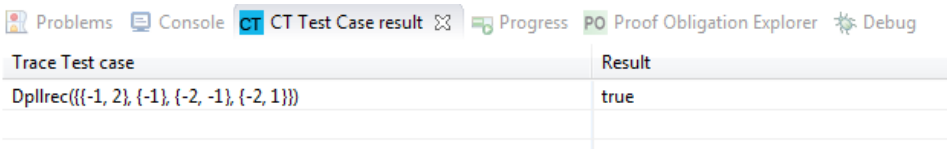


Figure 3.10.: Result of a specific test case

In Overture it is not only possible to check the model with the help of traces, it is also possible to generate proof obligations. These proof obligations shall be proved by the user to ensure the correctness of the model. Up to now the proof obligations are only generated, but it is not possible to export them for proving by an automated theorem prover or interactive proving assistant. A proof component in Overture shall be available in future version of Overture[33]. All proof obligations, which are generated for the recursive model can be seen in Figure 3.11.

PO Proof Obligation Explorer

No.	PO Name	Type
1	DPLLrec`Clause	type invariant satisfiable
2	DPLLrec`Literal	type invariant satisfiable
3	DPLLrec`Valuation	type invariant satisfiable
4	DPLLrec`Dpllrec(set of (Clause))	function establishes postcondition
5	DPLLrec`Dpllrec(set of (Clause))	let be st existence
6	DPLLrec`Dpllrec(set of (Clause))	legal function application
7	DPLLrec`Dpllrec(set of (Clause))	recursive function
8	DPLLrec`Dpllrec(set of (Clause))	type compatibility
9	DPLLrec`Dpllrec(set of (Clause))	legal function application
10	DPLLrec`Dpllrec(set of (Clause))	recursive function
11	DPLLrec`Dpllrec(set of (Clause))	type compatibility
12	DPLLrec`satisfiable(set of (Clause))	type compatibility

Figure 3.11.: Generated proof obligations

The first three proof obligations, which are of the type "type invariant satisfiable" are generated to ensure the type invariants, which are defined in the `types` section of the model. The fourth proof obligation is of the type "function establishes postcondition" and is created for the function *Dpllrec*. This proof obligation is shown below:

```
1 (forall f:set of (Clause) & (pre_Dpllrec(f) => post_Dpllrec(f,
2     (if (f = {})
3         then true
4      elseif (exists c in set f & (c = {}))
5         then false
6      else let l in set ({-n, ... ,n} \ {0}) be st (exists c in set f & (l in set c)) in (
7            if (Dpllrec(substitute(l, f)) = true)
8               then true
9            else Dpllrec(substitute(-l, f)))))))
```

In the Appendix of [33] a full explanation of all proof obligation categories is given, which states for this category: "Whenever an explicit operation has a post-condition there is an implicit proof obligation generated to remind the user that you have to ensure that the explicit body of the operation satisfies the post-condition for all possible inputs." Therefore this proof obligation is only a placeholder for a proof obligation which needs to be defined by the user.

The fifth proof obligation of type "let be st existence" is generated for the `let be st` statement inside the function *Dpllrec* and is used to ensure that at least one value fulfills the statement.

The next proof obligation is created for the method call *Dpllrec(substitute(l,f))* and

shall guarantee a legal function application. A proof obligation of this type is created whenever a function is called within another function and it shall ensure that the list of input arguments satisfies the preconditions of the called function.

The proof obligation of type "recursive function" is also created for the method call *Dpllrec(substitute(l,f))* and shall ensure that the recursive function terminates:

```
(forall f:set of (Clause) & (validFormula(f) =>
  ((not (f = {})) => ((not (exists c in set f & (c = {}))) =>
     (forall l in set ({-n, ... ,n} \ {0}) &
       ((exists c in set f & (l in set c)) =>
          (varInFormula(f) > varInFormula(substitute(l, f)))))))))
```

This proof obligation states that when a recursive function call is executed in the algorithm, the result of the `measure` function is smaller than the result of the `measure` function applied on the original input.

Proof obligation number 8 is also generated for the method call *Dpllrec (substitute(l,f))* and shall ensure that the input value l of the function *substitute(l,f)*, which is created by the `let l in set {-n,...,n}\{0 } be st (exists c in set f & l in set c)` statement, implies the invariant of the type *literal*. This needs to be ensured, because the function *substitute(l,f)* takes a literal as an input variable and the variable l is not explicitly defined as a literal.

The next three proof obligations are similar to the proof obligations number 6-8, because they are generated for the method call *Dpllrec(substitute(-l,f))*. The last proof obligation is again of type "type compatibility" and shall ensure that defined variable *val* in the function *satisfiable* matches the input type of *valuationSatFormula*.

3.8.2. Iterative Algorithm

The types and constants in the model of the iterative algorithm do not differ from the types and constants in the model of the recursive algorithm. The formula, which shall be analyzed for satisfiability, is now defined as a global variable instead of a local variable. This is done, because an invariant of the iterative algorithm is defined and it is not possible to define an invariant for a local variable. Therefore a section `instance variables` is added, where the formula is defined as a set of clauses. A second formula *OldFormula* is defined to store the original formula. Furthermore the stack, which is needed in the iterative algorithm is defined as a sequence of set of clauses. Initially all formulas and the stack are empty.

```
 instance variables
  public Formula : set of Clause := {};
  public OldFormula: set of Clause := {};
  public stack : seq of (set of Clause) := [];

inv validFormula(Formula) and validFormula(OldFormula) and
     (forall f in set elems stack & validFormula(f)) and
     n>=0 and numOfClause >=0 and len(stack)<=n and
     (forall i in set {1,...,len(stack)-1} & varInFormula(stack(i+1))>=varInFormula(stack(i))) and
     card(Formula)<=card(OldFormula) and
     OldFormula <> {} => (satisfiable(OldFormula) <=>
        (satisfiable(Formula) or (exists f in set elems stack & satisfiable(f))))
```

The invariant, which is added for the instance variables, matches the loop invariant which is defined for the iterative algorithm in the third model of the JML specification, given in Section 3.5.3. All formulas occurring as variables or occurring in the stack shall be valid. Furthermore the constants shall be greater than zero and the maximum number of formulas in the stack is smaller than n. It must hold, that the number of variables of the formulas in the stack is decreasing and that the number of clauses in a formula is smaller than the number of clauses in the original formula. The last property concerns the satisfiability of the original formula, which is satisfiable if and only if the current formula is satisfiable or one of the formulas in the stack is satisfiable.

Unfortunately it is not possible to name the invariant, as it can be done for type invariants. This invariant is actually a loop invariant, but it is not possible to annotate loops in VDM; therefore this invariant is added as a global invariant. Furthermore it is not possible to add a termination term to a while loop.

Since the formula, which shall be investigated, is now defined as a global variable, an operation is added, which allocates the formula. The other functions are the same as for the recursive function, only the algorithm itself as well as *substitute*, *varInFormula* and *setFormula* are now defined as operations instead of functions. Otherwise it would not be possible to change instance variables in the methods.

The iterative algorithm is directly adapted from the pseudo code listed in Section 3.1. Since the formula is a global variable, no input is needed and the precondition refers to the global variable. The postcondition states that the result of the algorithm is equal to the satisfiabilty of the original formula.

```
1  public Dplliter: () ==> bool
2  Dplliter() ==
3  (
4      while true do
5      (
6      if Formula = {}
7          then return true
8      elseif exists c in set Formula & c = {}
9          then (
10         if stack = [] then
11                return false
12         else (
13             Formula := hd stack;
14             stack := tl stack)
15         )
16     else
17         let l in set {-n,...,n}\{0} be st (exists c in set Formula & l in set c) in(
18             stack := [substitute(l, Formula)]^stack;
19             Formula := substitute(-l, Formula);)
20     )
21 )
22 pre validFormula(Formula)
23 post RESULT = satisfiable(Formula~);
```

The predicates for the input and output conditions of the iterative algorithm are the same as for the recursive algorithm. The `trace`, which simulates a run of the iterative algorithm is adapted to the iterative algorithm. The full model can be seen in Appendix A.5.2.

As for the recursive algorithm 3610 test cases in the combinatorial testing perspective are created and none of them detects a violation of pre- or postconditions.

PO Proof Obligation Explorer

No.	PO Name	Type
1	DPLLiter`Clause	type invariant satisfiable
2	DPLLiter`Literal	type invariant satisfiable
3	DPLLiter`Valuation	type invariant satisfiable
4	DPLLiter`inv_DPLLiter	state invariant initialized
5	DPLLiter`inv_DPLLiter	state invariant satisfiable
6	DPLLiter`Dplliter()	while loop termination
7	DPLLiter`Dplliter()	state invariant holds
8	DPLLiter`Dplliter()	non-empty sequence
9	DPLLiter`Dplliter()	state invariant holds
10	DPLLiter`Dplliter()	non-empty sequence
11	DPLLiter`Dplliter()	let be st existence
12	DPLLiter`Dplliter()	state invariant holds
13	DPLLiter`Dplliter()	state invariant holds
14	DPLLiter`Dplliter()	operation establishes postcondition
15	DPLLiter`setFormula(set of (Clause))	state invariant holds
16	DPLLiter`setFormula(set of (Clause))	state invariant holds
17	DPLLiter`satisfiable(set of (Clause))	type compatibility

Figure 3.12.: Generated proof obligations

Overture creates 17 proof obligations for the iterative algorithm, which can be seen in Figure 3.12. The first three proof obligations are again created for the type invariants. The next two proof obligations are generated for the state invariant. The proof obligation of type “state invariant initialized” initializes the invariant and the proof obligation of type “state invariant satisfiable” shall be proved to ensure that variables exist which satisfy the invariant.

The proof obligation which is created for the termination of the while loop is only a placeholder, because the user guide [33] lists for this type of proof obligations, that it is a reminder to ensure that a while loop will terminate.

Six of the remaining proof obligations are of type “state invariant holds”. They are created whenever an assignment is made to a part of the instance variables. Thus four of these proof obligations are created for the function *Dplliter()* and the other two are created for the function *setFormula(c)*. These proof obligations state that the precondition of the method implies that the state invariant still holds with an assignment made to an instance variable. These proof obligations are not really strong, because only the precondition of the corresponding function is considered. But for the iterative algorithm also the if-conditions should be added to the left side of the implication. Thus it can be said that the Overture Tool does not include a complete calculus.

This can also be seen for the proof obligations of type “non-empty sequence” which

are both created for the operations `hd stack` and `tl stack`. Both proof obligations state that from the precondition it shall follow that the stack is non-empty.

```
(validFormula(Formula) => (stack <> []))
```

The nested if-then-else statements inside the loop are completely neglected by the proof obligations generator, but they are necessary to proof that the stack is non-empty. When the if-then-else statements are considered, the proof obligation is of the following form and can easily be verified.

```
(validFormula(Formula) and (exists c in set Formula & c = {}) and
    not (stack = []) => (stack <> []))
```

The proof obligations of type “let be st existence”, “operation establishes postcondition” and “type compatibility” are the same proof obligations which are already generated for the recursive algorithm.

3.9. Event-B

The DPLL algorithm in Event-B is specified in two models to show the concept of refinement. The first model which is called “abstract model”, introduces the algorithm based on a definition of formulas as sets (of clauses). The second model is a refinement of the first model and formulas are defined as arrays instead of sets. The Rodin Platform automatically generates proof obligations for the abstract and refined model, which can then be solved automatically or interactively. In Event-B only the iterative algorithm is considered.

3.9.1. Abstract Model

Before the algorithm is defined in a `machine`, a `context` needs to be defined. In a `context` all user-specified data types are introduced. New data types are defined as `constants` and their properties are specified in the `axioms` section.

```
    CONTEXT
        VarDefinition
    CONSTANTS
        n
        Literal
        Clause
        Formula
        SetOfFormulas
        SetOfClause
        Valuation
    AXIOMS
       defConstant:  n = 3
       TypeLit:  Literal ⊆ ℤ
       validLiteral: ∀ l · (l ∈ Literal ⇒( l ≤ n ∧ l ≥ -n ∧ l ≠ 0))
       TypeClause: Clause ⊆ Literal
       consistent: ∀ l · (l ∈ Clause ⇒( ¬(l ∈ Clause ∧ -l ∈ Clause)))
       TypeSetOfClause: SetOfClause = ℙ(Clause)
       TypeFormula: Formula ⊆ SetOfClause
       TypeSetOfFormulas: SetOfFormulas = ℙ(Formula)
       TypeVal:  Valuation = {v | v ⊆ Literal ∧∀ l · (l ∈ Literal ⇒ ((l ∈ v ∨ -l ∈v) ∧¬ (l ∈
           v ∧ -l ∈ v)))}
       axm1: finite(Formula)
  END
```

The variable n defines the maximum value of a literal and is set to 3. The set of literals is then defined as a subset of the integers, where all elements are in the range $[-3, 3]$ excluding 0.

Based on the set of literals a clause is defined as a subset of the set of literals including the restriction, that a clause needs to be consistent. This means that no literal occurs both positively and negatively in a clause. A formula is defined as a subset of the set of all clauses.

A valuation is a subset of literals, where each literal has to occur, either positively or negatively. The last axiom *axm1* states that a formula is a finite set. This property is necessary for proving.

A second `context` is added for defining program counters. In the context *Program-Counter* a set PC is defined, which contains the elements "init", "done" and "lbl". The full context can be seen in Appendix A.6.2. In Figure 3.13 all occurring states of the program counter are displayed in a state diagram. The transitions in the state diagram mark the corresponding events in the specification of the algorithm.

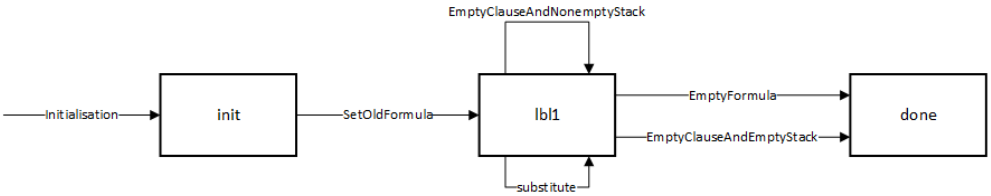


Figure 3.13.: State diagram of the program counter

The algorithm itself is modeled as a `machine`. At first all occurring variables in the algorithm have to be defined. This is done by adding them to the `variables` section in the model. The data types of the variables are defined as invariants in the `invariants` section.

```
1  INVARIANTS
2     inv1:  f ∈ SetOfFormulas
3     inv2:  stack ∈ 1..n → SetOfFormulas
4     inv3:  inStack ≥ 0 ∧ inStack ≤ n
5     inv4:  sat ∈ {−1, 0, 1}
6     inv5:  pc ∈ PC
7     inv6:  Oformula ∈ SetOfFormulas
8     inv7:  n≥0
9     inv8:  card(f)≤card(Oformula)
10    inv9:  ∀ i · (i ∈ 1..inStack ⇒ stack(i) ∈ SetOfFormulas)
11    inv10: ∀ i · (i ∈ 1..inStack-1 ⇒ card(stack(i+1)) ≤ card(stack(i)))
12    inv11: pc ≠ init ⇒
13           (( ∃ v· ( v ∈ Valuation ∧ (∀ c · (c ∈ Oformula ⇒ (∃ l ·(l ∈ c ∧ l ∈ v)))))) ⇔
14              ((∃ v· ( v ∈ Valuation ∧ (∀ c · (c ∈ f ⇒ (∃ l ·(l ∈ c ∧ l ∈ v)))))) ∨
15              (∃ i · (i ∈ 1..inStack ∧ (∃ v· ( v ∈ Valuation ∧
16                  (∀ c · (c ∈ stack(i) ⇒ (∃ l ·(l ∈ c ∧ l ∈ v))))))))))
17    inv12: ∀ i · (i ∈ 1..inStack ⇒ card(stack(i))≤card(Oformula))
18    goal:  pc = done ⇒ (sat = 1 ⇔ (∃ v· ( v ∈ Valuation ∧ (∀c · (c ∈ Oformula ⇒
19             (∃ l ·(l ∈ c ∧ l ∈ v)))))))
```

In this model six variables are needed, which are defined by the first six invariants. The variable f defines the formula, which shall be investigated for satisfiability; its original state is stored in *Oformula*. The stack for the iterative algorithm is modelled as a function $stack : [1..n] \rightarrow SetOfFormulas$. The current number of formulas in the stack is stored in *inStack*. The variable *sat* is equal to 0 until a state is reached, where it can be decided if the formula is satisfiable (=1) or not (=−1). Furthermore a program counter *pc* is added to the model.

The remaining invariants are global invariants of the algorithm and match the loop invariant which is defined for the iterative algorithm in the third model of the JML specification, given in Section 3.5.3. The border of the literals shall be greater than 0 and the number of clauses in the formula f is smaller than the number of clauses in the original formula. Furthermore all formulas occuring in the stack shall be valid and their cardinality decreases. The invariant *inv11* states that the original formula is satisfiable if and only if the current formula is satisfiable or one of the formulas in the stack is satisfiable. Unfortunately it is not possible to define predicates for the invariants, because the system rejects such definitions. Defining predicates would increase the readability of *inv11*. The last invariant *goal* states that whenever the algorithm is finished, sat=1 if and only if the original formula is satisfiable.

The state transitions of the algorithm are defined by several `events`. For each conditional branch in the pseudo-code, which is listed in Section 3.1, an event is defined. An event may consist of `guards`, `event parameters`, `witnesses` and `actions`. `Guards` indicate which event of the machine can be executed at each state of the model. `Event parameters` are local variables inside an event and their properties need to be added as `guards`. `Witnesses` are only used in refined models, since they connect variables and event parameters of the events of the abstract and refined models. `Actions` describe the changes of the variables.

```
1  substitute:
2    ANY
3      lit
4      cF
5    WHERE
6      grd1:  f ≠ ∅
7      grd2:  pc = lbl1
8      grd3:  ∀ c ·(c ∈ f ⇒ c ≠ ∅)
9      grd4:  lit ∈ Literal ∧ (∃ c ·(c ∈ f ∧ lit ∈ c))
10     grd5:  cF  ∈ SetOfFormulas ∧ cF = f
11     grd6:  inStack < n
12   THEN
13     act1:  f:= {c· c ∈ f\{cl· cl ∈ f ∧ lit ∈ cl|cl}|c\{-lit}}
14     act2:  inStack := inStack +1
15     act3:  stack(inStack+1) := {c ·c ∈ cF\{cl ·cl ∈ cF ∧-lit ∈ cl|cl}| c\{lit}}
16   END
```

The event *substitute* consists of two event parameters, six guards and three actions. The event parameter *lit* defines the literal which shall be substituted in the formula. Therefore it has to occur in the current formula *f*. This property is defined by *grd4*. The second event parameter *cF* is an exact copy of the current formula *f*, which is defined by *grd5*.

The remaining four guards state, that the event *substitute* can only be executed when the current formula is not empty and does not contain an empty clause. Furthermore the program counter needs to have the value "lbl1" and *inStack* needs to be smaller than *n*.

When the event is executed, the new value of *f* is the old formula *f*, where each clause containing *lit* is deleted and *-lit* is deleted from all remaining clauses. The opposite is done to *cF* and the resulting formula is added to the stack. The number of formulas in the stack is increased by one.

The full machine, with all occurring events can be seen in Appendix A.6.3.

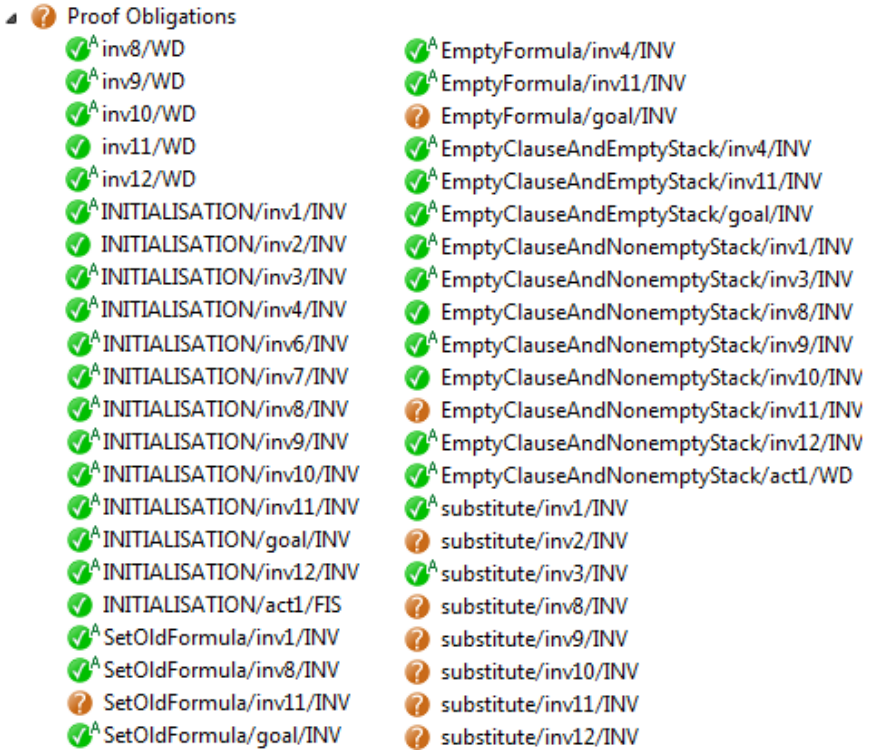


Figure 3.14.: Generated proof obligations

After the model is set up, the Rodin Platform generates proof obligations for the partial correctness of the model (termination is not considered), which can be seen in Figure 3.14. For this model 44 proof obligations are generated. 30 proof obligations are directly proved by the Rodin Platform, which is indicated by the green "A". So for about two-third of the proof obligations no further investigation is necessary. From the remaining 14 proof obligations 5 can be proved by the use of external automatic provers in the Rodin platform. The remaining 9 proof obligations cannot be proved by the use of automatic provers.

Except for its well-definedness and the initialisation event, every other proof obligation concerning *inf11* fails due to its complicated form. For the event *substitute* most invariant proofs cannot be completed, because the formulas are updated with nested

sets and it can not be shown that the updated variables still fulfill the corresponding invariants.

In Figure 3.15 the proving perspective for the proof obligation “EmptyFormula/goal/INV” can be seen. This proof obligation cannot be proved by automatic provers.

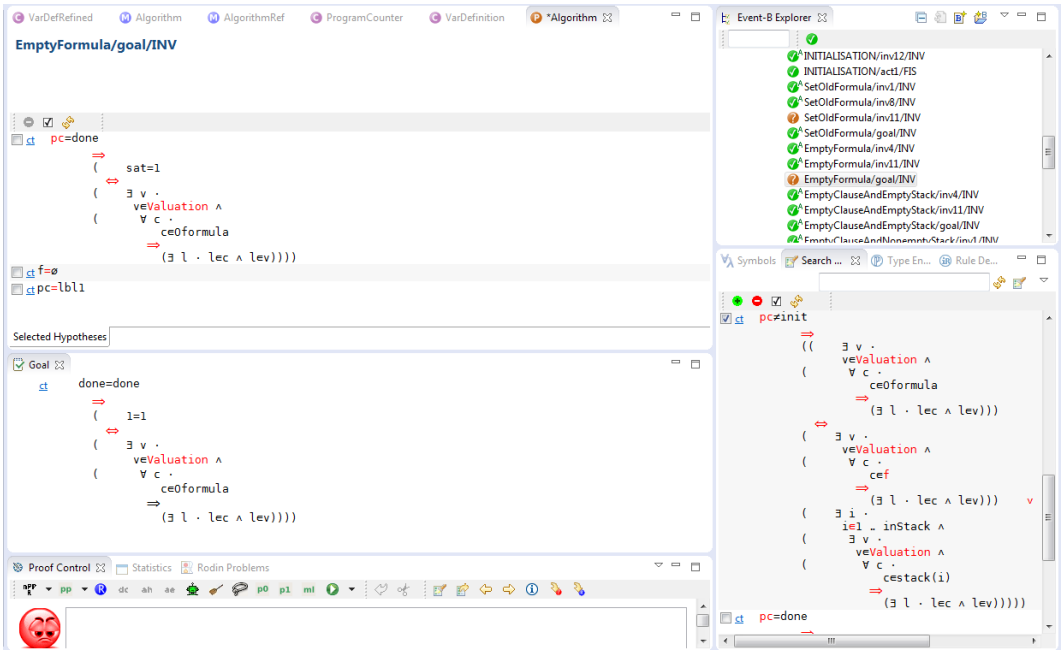


Figure 3.15.: Proving perspective

Proof obligations of type “event/invariantlabel/INV” are created, when it needs to be guaranteed that an invariant is preserved after the execution of an event. In this case the proof obligation shall ensure that the invariant *goal* is still valid after the execution of the event *EmtpyFormula*. The event is listed below.

```
1  EmptyFormula:
2     WHERE
3        grd1:   f = ∅
4        grd2:   pc = lbl1
5     THEN
6        act1:   sat := 1
7        act2:   pc := done
8     END
```

It can be seen that this event only occurs when f is the empty formula and the program counter has the value “lbl1”. After the event the variable *sat* is equal to 1 and the program counter has the value “done”. In Figure 3.15 it can be seen that the goal is to prove the invariant *goal*, where the updated values for *sat* and *pc* are plugged in. Thus the goal simplifies to

$$\exists\, v \cdot (\, v \in \text{Valuation} \land (\forall c \cdot (c \in \text{Oformula} \Rightarrow (\exists\, l \cdot l \in c \land l \in v)))).$$

The Rodin Platform selects automatically three hypotheses, namely the two guards and the invariant itself.

It is possible to add more hypotheses from a list, which can be seen in the lower right corner of Figure 3.15. In this case the invariant *inf11* is selected, because with the knowledge of *inf11* it should easily be possible to prove the goal. Since *pc* is equal to "lbl1", the equivalence in *inf11* holds. The formula *f* is the empty set, therefore the right side of the equivalence reduces to true and thus the left side of the equivalence has to be true. The left side of the equivalence is the goal, which shall be proved.

Unfortunately the Rodin Platform is not able to complete the proof after the hypothesis is added to the set of selected hypothesis. The newly added hypothesis simply vanishes.

It is also possible to make an interactive proof in the Rodin platform by clicking on the red symbols, which occur in the proving perspective. Since the manual [25] does not include a proper explanation of the interactive prover, it is not attempted to apply interactive proofs on the remaining proof obligations.

Another possibility for proving the remaining proof obligations would be to make us of plug-ins for SMT solvers or by exporting the formulas to the interactive proving assistant Isabelle/HOL [18].

3.9.2. Refined Model

In the second machine it is decided to model the formula *f* as an array instead of a set. Thus the second model is a refinement of the first model, because it is more detailed than the first model. The core concept of refinement is that it preserves proved properties, which makes it easier to gain a more detailed proved model. The second model is called "concrete machine" and the first model is called "abstract machine".

To ensure that the concrete machine is a refinement of the abstract machine, two properties need to hold. The first property is called "guard strengthening" and guarantees that an event in the concrete machine can only occur, when the corresponding event in the abstract machine occurs. The second property, called "correct refinement" ensures that whenever an event in the concrete machine occurs, the corresponding event in the abstract machine occurs in such a way that the "gluing invariant" is still valid. Gluing invariants are invariants which create a link from variables of the abstract machine to the newly introduced variables in the concrete machine.

Before the refinement step of the machine is done, a new `context` is introduced, which defines the new data type of a formula.

```
1  CONTEXT
2      VarDefRefined
3  EXTENDS
4      VarDefinition
5
6  CONSTANTS
7      m
8      FormulaRef
9      SetOfFormulaRef
10
11 AXIOMS
12     axm1:  m = 5
13     axm2:  FormulaRef = 1..m → {Clause ∪ {0}}
```

```
14      axm3:  SetOfFormulaRef = {f | f ∈ FormulaRef}
15  END
```

It can be seen that this context is an extension of the context *VarDefinition*. A new constant m is added, which defines the number of clauses in a formula and it is set to 5. Since dynamic arrays are not implemented in Event-B, a formula is defined as a total function $FormulaRef : [1..m] \rightarrow \{Clause \cup \{0\}\}$. The set $\{0\}$ is added to the range of the function, because it is necessary to differ between an empty clause and a deleted clause. So whenever a clause is deleted during a substitution, the corresponding position in the function is set to $\{0\}$. This means that a formula is the empty formula if and only if all positions in the array are equal to $\{0\}$. If a position in the array is equal to $\emptyset$, the formula contains an empty clause at this position. This is not the most elegant solution of modelling a formula as an array, but it is the simplest version (since dynamic arrays are not implemented in Event-B, otherwise the machine would get too complex).

After the new context is set up, the refinement step can be done. The variables *f*, *Oformula* and *stack* of the abstract machine are replaced by new variables.

```
1   INVARIANTS
2     inv1:  f1  ∈ SetOfFormulaRef
3     inv2:  Oformula1 ∈ SetOfFormulaRef
4     inv3:  stack1 ∈ 1..n → SetOfFormulaRef
5     glu1:  ∀ c · (c ∈ SetOfClause ⇒ (c ∈ f ⇔ (∃ i ·(i ∈ 1..m ∧ f1(i) = c))))
6     glu2:  ∀ c · (c ∈ SetOfClause ⇒ (c ∈ Oformula ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1(i) = c))))
7     glu3:  ∀ c · (c ∈ SetOfClause ⇒ (∀ i · (i ∈ 1..n ⇒ ( c ∈ stack(i) ⇔
8                 (∃ j · (j ∈ 1..m ∧ stack1(i)(j) = c))))))
9     glu4:  f = ∅ ⇔ (∀ i · (i ∈ 1..m ⇒ f1(i) = {0}))
10    glu5:  Oformula = ∅ ⇔ (∀ i · (i ∈ 1..m ⇒ Oformula1(i) = {0}))
11    glu6:  ∀ i · (i ∈ 1..n ⇒ ( stack(i) = ∅ ⇔ (∀ j · (j ∈ 1..m ∧ stack1(i)(j) = {0}))))
12    glu7:  ∅ ∈ f ⇔ (∃ i ·(i ∈ 1..m ∧ f1(i) = ∅))
13    glu8:  ∅ ∈ Oformula ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1(i) = ∅))
14    glu9:  ∀ i · (i ∈ 1..n ⇒ ( ∅ ∈ stack(i) ⇔ (∃ j · (j ∈ 1..m ∧ stack1(i)(j) = ∅))))
```

Each new variable has to be defined by new invariants in the `invariants` section. Furthermore the "gluing invariants" need to be defined. So for each new variable a link to the corresponding variable of the abstract machine has to be defined. Whenever a clause *c* is an element of the abstract formula *f*, there must exist a position in the refined formula *f1*, which is equal to *c*. This is defined by the invariant *glu1*. During the proving process it is discovered, that this property must explicitly be defined for the empty set, which is done in *glu7*. Furthermore it needs to be defined that *f* is the empty formula if and only if all positions in *f1* are equal to $\{0\}$. This property is defined in *glu4*.

The remaining gluing invariants are similarly defined for *Oformula1* and for each position in *stack1*.

Each event of the abstract model has to be modified by replacing the variables *f*, *Oformula* and *stack* and adapting their properties. In the concrete model `witnesses` may be necessary for events. Witnesses have to be defined when either an event in the abstract machine assigns a variable non-deterministically and the variable is not part of the concrete machine any more, or when an abstract event has an event parameter, which is not part of the concrete event.

```
INITIALISATION:
   WITH
     Oformula': ∀ c · (c ∈ SetOfClause ⇒ (c ∈ Oformula' ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1'(i)=c))))
   THEN
     act1:  Oformula1 :∈ SetOfFormulaRef
     act2:  stack1 := λi·i ∈ 1..n | (λj · j ∈ 1..m | {0})
     act3:  inStack := 0
     act4:  sat := 0
     act5:  pc := init
     act6:  f1 := λi · i ∈1..m | {0}
   END
```

In the abstract event *INITIALISATION*, the variable *Oformula* is assigned non-deterministically and thus a witness is necessary in the concrete event. In this case the witness is a modification of a gluing invariant. Furthermore it can be seen, that the actions for the newly introduced variables are adapted, whereas the other actions remain unchanged.

The full machine, with all occurring events can be seen in Appendix A.6.5.

After the model is set up, the Rodin Platform generates proof obligations, which can be seen in Figure 3.16. For this model 65 proof obligations are generated. 27 proof obligations are directly proved by the Rodin Platform, indicated by the green "A". From the remaining proof obligations 7 can be proved by the use of external automatic provers in the Rodin platform. The remaining 31 proof obligations cannot be proved by the use of automatic provers. It is again not attempted to perform interactive proofs.

All remaining proof obligations of the types "label/WD", "event/actionlabel/WD" respectively "event/identifier/WWD" fail, because it cannot be shown that a variable is an element of *SetOfFormulaRef*. The same holds for all open proof obligations concerning the event *INITIALISATION*.

In the proofs of the proof obligations for the event *EmptyClauseAndNonemptyStack* the error where a hypothesis is added to the set of selected hypotheses and immediately vanishes occurs again.

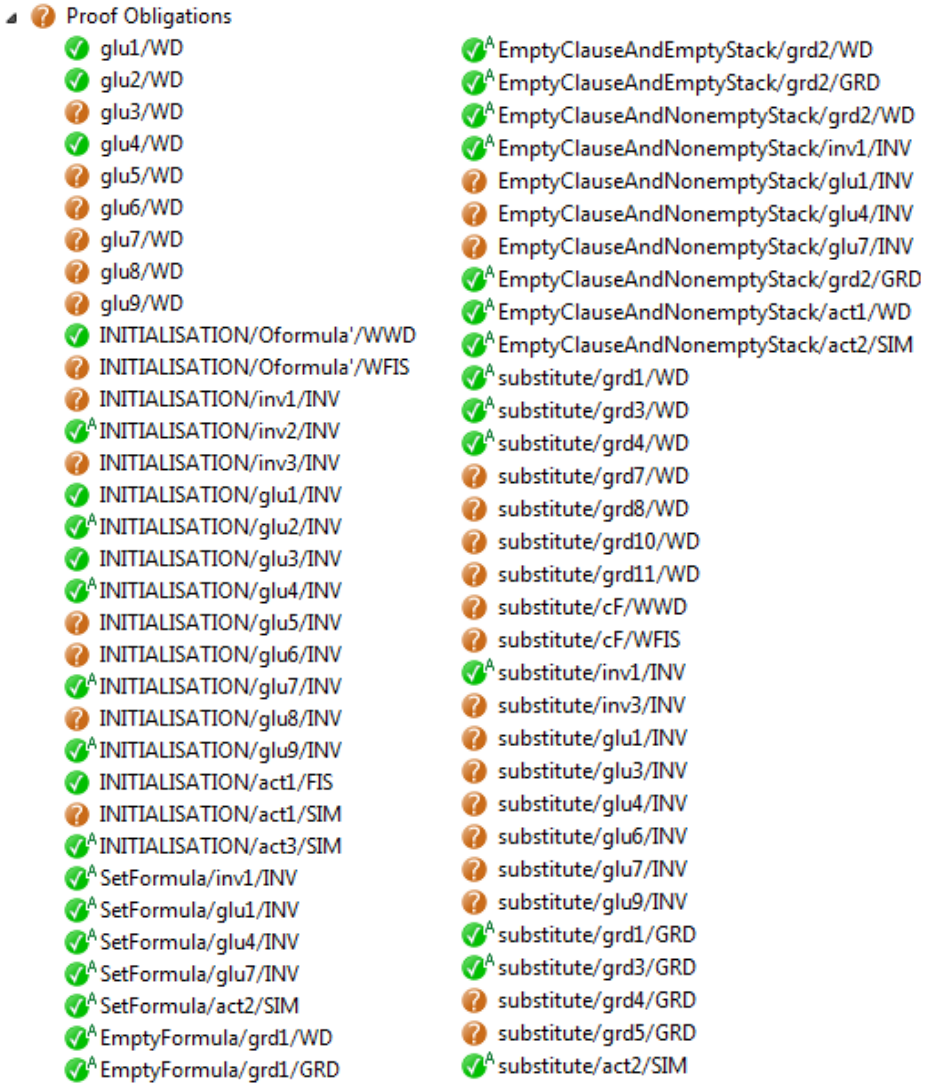


Figure 3.16.: Generated proof obligations

3.10. Summary of the Results

A short summary of the results of the application of the software specification languages and tools for the DPLL method is given in Figure 3.17.

Language	Result
JML	The specification of the DPLL method is easy because Java annotations are added to the source code. It is possible to check the lower level functions with ESC/Java2. For more complex functions ESC/Java2 is not able to check correct annotations and returns warnings. In Java KeY the annotations of the easier functions can be verified, whereas the automatic prover fails to complete the proofs for more complex functions.

TLA	The usage of PlusCal makes it very easy to specify the DPLL method in TLA. Setting up the model for the TLC Model Checker is very simple, because only the values of the constants of the model have to be defined. In the given scope the TLC Model Checker is able to model check the algorithm. The only drawback is, that TLA does not support recursion, thus only the iterative version of the DPLL method can be specified.
Alloy	Specifying the DPLL method in Alloy is very complicated, because everything is based on relations. Furthermore the scope for the model has to be defined with care, because it has to be defined for each object occurring in the model. When the scope is too small, the termination conditions of the algorithm may not be fulfilled. The Alloy Analyzer is able to model check the defined assertions and furthermore it is possible to gain visualizations of the run of the algorithm.
VDM	Specifying the DPLL method in VDM is easy due to the expressive language. Furthermore VDM supports recursion. For model checking it is necessary to define a `trace`, which imitates the run of the algorithm. In the given scope Overture is able to model check the system. It is also possible to generate proof obligations for the model, but the calculus is incomplete and some of the proof obligations are only placeholders.
Event-B	Due to the expressive language specifying the DPLL method in Event-B is easy. The Rodin Tool automatically creates proof obligations for the defined events and invariants. The easier proof obligations are automatically shown by the included provers. In more complex proof obligations, where nested logical expressions are included, the provers are not able to verify the proof obligations, although they are correct. It is noticed that the concept of refinements can only be used in a very limited way for algorithms, because only the data types can be refined.

Figure 3.17.: Results of the Languages and Tools for DPLL method

4. Dijkstra's Shortest Path Algorithm

In this chapter Dijkstra's Shortest Path Algorithm is specified and analyzed. At first the algorithm is informally introduced and a pseudo-code is given. The informal part is followed by a formal specification, where the input and output conditions are outlined. A Java prototype of the algorithm is listed, which is then specified in JML and analyzed by ESC/Java2. Subsequent the algorithm is modelled in TLA, Alloy, VDM and Event-B. For each specification language the set-up of the model is declared and the output of the corresponding tool is shown.

4.1. Informal Problem

Dijkstra's Shortest Path Algorithm is an algorithm for finding the shortest path from a given vertex to every other vertex in a graph with weighted edges. The algorithm can be applied to undirected as well as to directed graphs. The graph is represented by the set of vertices and a mapping of each pair of vertices to the weight of the connecting edge (infinity, if there is none). In Dijkstra's Shortest Path Algorithm all edges must have positive weights. If the graph is undirected, each pair of vertices is represented by a set.

Dijkstra's Shortest Path Algorithm was developed by Edsger W. Dijkstra in 1959 [15]; it is often referred to as Dijkstra's Algorithm. This algorithm improves the shortest path algorithms given by Ford in 1956 and Berge in 1958 and achieves a run-time of $\mathcal{O}(n^2)$. Furthermore it has an easier implementation than the method of Dantzig, which was developed in 1958 and has the same run-time [43]. Today Dijkstra's Algorithm is used in network routing protocols, like OSPF[40] and IS-IS[20].

Dijsktra's Algorithm is an example for a greedy algorithm. At first, all nodes are marked as unvisited, and their distances and predecessors in the shortest path are undefined. The starting node is marked as the current node and its distance is set to zero. From the starting node the distance to each neighbour node is calculated and stored. Each neighbour is added to the set of connected vertices and the predecessor of each neighbour is set to the starting node. Then the given node is marked as visited. Visited nodes will never be visited again. The new current node is the node in the set of connected vertices, which has not been visited and has minimal distance. The distances to all unvisited neighbour nodes of the current node are calculated by adding the weight of the edge to the distance of the current node. If a distance is already assigned to an unvisited neighbour node, the new calculated distance is compared to the current assigned distance and the smaller one is chosen. If the distance is updated, the predecessor is set to the current node. The current node is then marked as visited and a new current node is selected. This procedure is repeated until all connected nodes are marked as visited [12].

The following pseudo-code summarizes the algorithm (adapted from [12]):

Algorithm Dijkstra(n, s, w)

Input: A partial function w mapping edges of a graph to a weight,
source vertex s

Output: A triple *(C, dist, pre)*, where C is the set of connected vertices, the list *dist* defines the shortest distances for all connected vertices, and the list *pre* stores for each connected vertex its predecessor in the shortest path.

```
1: C ← {s}
2: Q ← Vertex_n
3: dist[s] ← 0
4: pre[s] ← s
5: while Q ∩ C ≠ ∅ do
6:    choose u ∈ Q ∩ C: ∀u' ∈ Q ∩ C: dist[u]≤ dist[u']
7:    Q ←Q\{u}
8:    for  all v ∈ Q: {u,v} ∈ dom(w) do
9:       length ← dist[u] + w({u,v})
10:      if v ∈ C then
11:         if length < dist[v] then
12:            dist[v] ← length
13:            pre[v]← u
14:         end if
15:      else
16:         dist[v] ← length
17:         pre[v]← u
18:         C ← C ∪ {v}
19:      end if
20:   end for
21: end while
22: return  C, dist, pre
```

The algorithm returns three objects. The first object is the set of all connected nodes. Since the algorithm can be applied to disconnected graphs, only those vertices which are connected to the starting vertex will be visited and their distances and predecessors will be computed. The second object is a list where for each vertex the length of the shortest path from the starting vertex to the vertex is stored. The third object is a second list which returns for each vertex its predecessor in the shortest path. In the lists only those entries of the vertices, which are an element of the set of all connected nodes are well-defined.

The next current node in Dijsktra's algorithm is selected as the node with minimal distance, which has not been visited yet. This yields a run-time of $\mathcal{O}(n^2)$, where n defines the cardinality of the set of vertices. If any other node, which has not been visited yet but is known to be connected to the graph, is selected as the new current node the algorithm is not correct anymore, because visited nodes are not considered any more. A counterexample is given below.

In Figure 4.1 a simple example of a graph can be seen. The start vertex is *node 1* and it is assumed, that *node 2* and *node 4* have already been visited. If then *node 3* is selected to be the current node, the connection to *node 4* will not be considered anymore and the length the shortest path from *node 1* to *node 4* is computed to be 10, although a shorter path of length 4 exists.

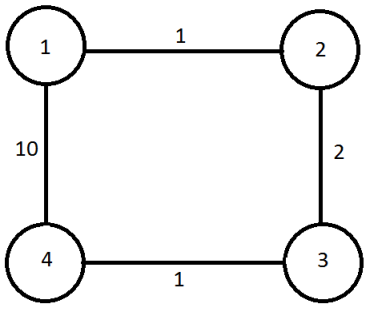


Figure 4.1.: Example for a graph

An invariant of the algorithm is that for each visited vertex the calculated distance is already the distance of the shortest path from the start vertex to this vertex. Furthermore for each vertex, which has been added to the set of connected vertices, the stored distance is the length of the shortest path from the starting vertex, considering only those vertices, which have already been visited.

4.2. Formal Specification

In this section a formal specification of Dijkstra's Algorithm is given. The input and output of the algorithm including input and output conditions are outlined.

Algorithm Dijkstra

Input: *(n, s, w)*
Input condition: $n \geq 0 \wedge s \in \text{Vertex}_n \wedge w \in \text{wgraph}_n$
Output: *C, dist, pre*
Output condition: (C, dist, pre, w, s, n) is shortest path

4.2.1. Input Condition

Before the definition of a weighted graph is given, the sets of vertices and edges need to be defined. A vertex is a natural number smaller than or equal to a given border.

$$\text{vertex} \subseteq \mathbb{N} \times \mathbb{N}$$
$$\text{vertex}_n(v) :\Leftrightarrow v \leq n$$

$$\text{Vertex}: \mathbb{N} \to \mathbb{P}(\mathbb{N})$$
$$\text{Vertex}_n := \{v \in \mathbb{N} \mid \text{vertex}_n(v)\}$$

Based on the set of vertices, the definition of an edge is given. An edge is a set of exactly two vertices.

$$\text{edge} \subseteq \mathbb{N} \times \mathbb{P}(\mathbb{N})$$
$$\text{edge}_n(e) :\Leftrightarrow e \in \mathbb{P}(\text{Vertex}_n) \wedge |e| = 2$$

$$\text{Edges: } \mathbb{N} \to \mathbb{P}(\mathbb{P}(\mathbb{N}))$$
$$\text{Edges}_n := \{e \in \mathbb{P}(\mathbb{N}) \mid \text{edge}_n(e)\}$$

A weighted graph is defined as a partial function from the set of edges to the natural numbers.

$$\text{wgraph}_n\text{: Edges}_n \mapsto \mathbb{N}$$

4.2.2. Output Condition

For the specification of the output condition several predicates are needed. The predicate *neighbour* defines, that two vertices are neighbours in a graph if and only if they are elements of the domain of the graph.

$$\text{neighbour} \subseteq \text{wgraph}_n \times \mathbb{P}(\mathbb{N}) \times \mathbb{P}(\mathbb{N})$$
$$\text{neighbour(w,u,v)} : \Leftrightarrow \{\text{u,v}\} \in \text{dom(w)}$$

A *distance* is defined as a partial function, which maps elements from the set of vertices to natural numbers including 0.

$$\text{distance: Vertex}_n \mapsto \mathbb{N}_0$$

The function *predecessor* defines the predecessor of a vertex in the shortest path.

$$\text{predecessor: Vertex}_n \mapsto \text{Vertex}_n$$

With the help of the predicates the output condition can be defined. The output condition is fulfilled, when C is the set of all linked vertices and *dist* and *pre* define the shortest distances respectively shortest paths from the given vertex to all linked vertices.

$$\text{is shortest path} \subseteq \mathbb{P}(\mathbb{N}) \times \text{distance} \times \text{predecessor} \times \text{wgraph}_n \times \mathbb{N} \times \mathbb{N}$$
$$\text{(C, dist, pre, w, s, n) is shortest path: } \Leftrightarrow$$
$$\quad \text{SetOfLinkedVertices(C, s, w, n)} \wedge$$
$$\quad \text{DefinesShortestDistances(dist, pre, C, s, w)}$$

The set C is the set of all linked vertices if and only if the starting vertex is included in C and for all vertices in C all their neighbours are also elements of C. Furthermore for each vertex in C exists another vertex in C such that they are neighbours.

$$\text{SetOfLinkedVertices} \subseteq \mathbb{P}(\mathbb{N}) \times \mathbb{N} \times \text{wgraph}_n \times \mathbb{N}$$
$$\text{SetOfLinkedVertices(C, s, w, n)}: \Leftrightarrow$$
$$C \subseteq \text{Vertex}_n \wedge s \in C \wedge$$
$$(\forall c \in C : \forall v \in \text{Vertex}_n : \text{neighbour}(w, v, c) \Rightarrow v \in C) \wedge$$
$$(\forall c \in C \backslash \{s\} : \exists c_1 \in C : \text{neighbour}(w, c, c_1))$$

The outputs *dist* and *pre* define the shortest path and its length from the given vertex to each vertex which is included in C. The distance of the starting vertex s has to be zero. For each vertex in $C\backslash\{s\}$ its predecessor is chosen and those two vertices have to be neighbours and the difference of their distances is the weight of their common edge. Furthermore for each vertex in C it must hold that its distance is smaller than or equal to the sum of the distance and the weight of the common edge of each neighbour node.

$$\text{DefinesShortestDistances} \subseteq \text{distance} \times \text{predecessor} \times \mathbb{P}(\mathbb{N}) \times \mathbb{N} \times \text{wgraph}_n$$
$$\text{DefinesShortestDistances(dist, pre, C, s, w)} : \Leftrightarrow$$
$$dist[s] = 0 \wedge$$
$$(\forall v \in C \backslash \{s\} : \text{let } pre[v] = v_1 \text{ in neighbour}(w, v, v_1) \wedge$$
$$dist[v] = dist[v_1] + w(\{v, v_1\})) \wedge$$
$$\forall v_1, v_2 \in C : \text{neighbour}(w, v_1, v_2) \Rightarrow dist[v_1] \leq dist[v_2] + w(\{v_1, v_2\})$$

4.2.3. Correctness of the Algorithm

We start by showing, that the predicate "DefinesShortestDistances" really defines the shortest distances from the starting vertex to each connected vertex. For this purpose, some predicates are defined.

A path up to length c from a given vertex s to a vertex v is a sequence of up to c vertices such that all succeeding vertices are neighbours and the first vertex is s and the last vertex is v.

$$\text{path}_c \subseteq \mathbb{N} \times \text{Vertex}_n^n \times \text{Vertex}_n \times \text{Vertex}_n \times \text{wgraph}_n$$
$$\text{path}_c\text{(p, s, v, w)} : \Leftrightarrow \exists m \in \mathbb{N} : 1 \leq m \leq c \wedge p \in \text{Vertex}_n^m \wedge$$
$$\forall 0 \leq i < m - 1 : \text{neighbour}(w, p_i, p_{i+1}) \wedge p_0 = s \wedge p_{m-1} = v$$

The predicate *IsShortestDist* defines, that for all linked vertices and all paths up to length c connecting the start vertex s and a vertex v, the sum of the weights of the edges on the path is greater than or equal to the assigned minimal distance *dist* of v. Furthermore for each linked vertex exists a path up to length c, such that the sum of the weighs of the edges on the path is equal to the minimal distance.

$$\text{IsShortestDist} \subseteq \text{distance} \times \text{predecessor} \times \mathbb{P}(\mathbb{N}) \times \mathbb{N} \times \text{wgraph}_n \times \mathbb{N} \times \mathbb{N}$$
$$\text{IsShortestDist(dist, pre, C, s, w, n, c)} : \Leftrightarrow$$
$$(\forall v \in C : \forall p : path_c(p, s, v, w) \Rightarrow \textstyle\sum_{0 \leq i < |p|-1} w(\{p_i, p_{i+1}\}) \geq dist[v]) \wedge$$
$$(\forall v \in C : \exists p : path_c(p, s, v, w) \wedge \textstyle\sum_{0 \leq i < |p|-1} w(\{p_i, p_{i+1}\}) = dist[v] \wedge$$
$$\forall 0 \leq i < |p| - 1 : pre[p_{i+1}] = p_i)$$

The following predicate defines, that for all possible inputs and outputs of the algorithm it holds that whenever the cardinality of C is equal to c, *IsShortestDist* follows

from the output condition *DefinesShortestDistances*.

$$
\begin{aligned}
&\mathrm{F} \subseteq \mathbb{N} \\
&\mathrm{F(c)} : \Leftrightarrow \forall C \subseteq \mathrm{Vertex}_n, s \in \mathrm{Vertex}_n, n \in \mathbb{N}, w \in \mathrm{wgraph}_n, \\
&\quad dist \in \mathrm{distance}, pre \in \mathrm{predecessor}: \\
&\quad |C| = c \Rightarrow (\mathrm{definesShortestDistances(dist,\ pre,\ C,\ s,\ w)} \Rightarrow \\
&\qquad \mathrm{IsShortestDist(dist,\ pre,\ C,\ s,\ w,\ n,\ c)})
\end{aligned}
$$

We now show that $F(c)$ holds for all $c \in \mathbb{N}$.

Lemma: $\forall c \in \mathbb{N} : F(c)$

Proof: The proof is done by induction on c. Let s, n, w, *dist*, *pre* be arbitrary but fixed.

Base case: Let $c = 0$ and assume *DefinesShortestDistances(dist, pre, C, s, w)* holds. Since $c = 0 \Rightarrow C = \emptyset$, the domain for the universal quantifiers in *IsShortestDist(dist, pre, C, s, w, n, c)* is the empty set. Thus *IsShortestDist(dist, pre, C, s, w, n, c)* reduces to "true" and the statement holds for $c = 0$.

Inductive step: Let $c \in \mathbb{N}$ and assume $F(c)$ holds. It needs to be shown that $F(c+1)$ holds. Let C be such that $|C| = c + 1$ and assume *DefinesShortestDistances(dist, pre, C, s, w)* holds.

It needs to be shown that

$$\forall v \in C : \ \forall p : path_c(p, s, v, w) \Rightarrow \textstyle\sum_{0 \le i < |p|-1} w(\{p_i, p_{i+1}\}) \ge dist[v]$$

and

$$\forall v \in C : \ \exists p : (path_c(p, s, v, w) \ \wedge \ \textstyle\sum_{0 \le i < |p|-1} w(\{p_i, p_{i+1}\}) = dist[v] \ \wedge \forall\ 0 \le i < |p| - 1 : pre[p_{i+1}] = p_i).$$

At first it is shown that

$$\forall v \in C : \ \forall p : path_c(p, s, v, w) \Rightarrow \textstyle\sum_{0 \le i < |p|-1} w(\{p_i, p_{i+1}\}) \ge dist[v].$$

Therefore let $v_0 \in C$ and p' be arbitrary but fixed such that $path_{c+1}(p', s, v_0, w)$ holds. It needs to be shown that $\sum_{0 \le i < |p'|-1} w(\{p'_i, p'_{i+1}\}) \ge dist[v_0]$ holds.

Let $C_0 := \{v \in C | \ \exists i : 0 \le i \le |p'| - 2 \wedge p'_i = v\}$ be the set of vertices in p' excluding the last vertex v_0. It holds that $|C_0| = |C| - 1$.

Since each weight is greater than 0, it can never occur that $dist[v_0] = dist[v_0] + w(\{v_0, v_j\}) + \ldots + w(\{v_{j+h}, v_0\})$. Thus a vertex can not occur more than once in a path and therefore $v_0 \notin C_0$.

C_0 suffices *DefinesShortestDistances(dist, pre, C_0, s, w)*, because $C_0 \subset C$ and it is assumed that *DefinesShortestDistances(dist, pre, C, s, w)* holds.

Using the induction hypothesis C_0 suffices *IsShortestDist(dist, pre, C_0, s, w, n, c)*.

Thus

$$\sum_{0\leq i<|p'|-1} w(\{p'_i, p'_{i+1}\}) \overset{\text{split sum}}{=}$$

$$\sum_{0\leq i<|p'|-2} w(\{p'_i, p'_{i+1}\}) + w(\{p'_{|p'|-2}, v_0\}) \overset{\text{induction hypothesis}}{\geq}$$

$$dist[p'_{|p'|-2}] + w(\{p'_{|p'|-2}, v_0\}) \overset{\text{definesShortestDistances}}{\geq} dist[v_0]$$

which completes the first part of the proof.

In the second branch of the proof, it needs to be shown that

$$\forall v \in C: \ \exists p : path_c(p, s, v, w) \wedge \textstyle\sum_{0\leq i<|p|-1} w(\{p_i, p_{i+1}\}) = dist[v] \wedge \\ \forall 0 \leq i < |p| - 1 : pre[p_{i+1}] = p_i.$$

Let $v_0 \in C$ be arbitrary but fixed and let $C_0 := C \backslash \{v_0\}$. Thus $|C_0| = |C| - 1 = c$. C_0 suffices *IsShortestDist*$(dist, pre, C_0, s, w, n, c)$ because of the same reasons as in the first part of the proof.

From *DefinesShortestDistances(dist, pre, C, s, w)* it follows that $pre[v_0] \neq v_0$ and thus $pre[v_0] \in C_0$.

Since C_0 suffices *IsShortestDist*$(dist, pre, C_0, s, w, n, c)$ it holds that

$$\exists p : path_c(p, s, pre[v_0], w) \wedge \sum_{0\leq i<|p|-1} w(\{p_i, p_{i+1}\}) = dist[pre[v_0]] \wedge \\ \forall\ 0 \leq i < |p| - 1 : pre[p_{i+1}] = p_i.$$

Let p be this path.

Let $\bar{p}$ be the path in C, such that $\bar{p} \in \text{Vertex}_n^{|p|+1} \wedge \bar{p}_{|p|} = v_0 \wedge \forall 0 \leq i \leq |p| - 1 : \bar{p}_i = p_i$. So $\bar{p}$ is the path p including v_0 added at the end.

It needs to be shown that $\bar{p}$ satisfies $path_{c+1}(\bar{p}, s, v_0, w) \wedge \sum_{0\leq i<|\bar{p}|-1} w(\{\bar{p}_i, \bar{p}_{i+1}\}) = dist[v_0] \wedge \forall 0 \leq i < |\bar{p}| - 1 : pre[\bar{p}_{i+1}] = \bar{p}_i$.

At first it is shown that $\bar{p}$ suffices $path_{c+1}(\bar{p}, s, v_0, w)$.

Since $path_c(p, s, pre[v_0], w)$ holds because of the induction hypothesis, it holds that $|p| \leq c$ and thus $|\bar{p}| \leq c + 1$. It follows from the definition of $\bar{p}$ that, $\bar{p} \in \text{Vertex}_n^{\bar{p}}$ and that $\bar{p}_0 = s$ and $\forall 0 \leq i < |\bar{p}| - 2 : neighbour(\bar{p}_{i+1}, \bar{p}_i)$.

Since $\bar{p}_{|\bar{p}|-2} = pre[v_0]$ and $\bar{p}_{|\bar{p}|-1} = v_0$ it holds that $\forall 0 \leq i < |\bar{p}| - 1 : neighbour(\bar{p}_{i+1}, \bar{p}_i)$. Thus $path_{c+1}(\bar{p}, s, v_0, w)$ holds.

It is now shown that $\sum_{0\leq i<|\bar{p}|-1} w(\{\bar{p}_i, \bar{p}_{i+1}\}) = dist[v_0]$.

$$\sum_{0\leq i<|\bar{p}|-1} w(\{\bar{p}_i, \bar{p}_{i+1}\}) \overset{\text{split sum}}{=}$$

$$\sum_{0\leq i<|\bar{p}|-2} w(\{\bar{p}_i, \bar{p}_{i+1}\}) + w(\{\bar{p}_{|\bar{p}|-2}, v_0\}) \overset{\text{definition of } \bar{p}}{=}$$

$$\sum_{0\leq i<|p|-1} w(\{p_i, p_{i+1}\}) + w(\{pre[v_0], v_0\}) \overset{\text{induction hypothesis}}{=}$$

$$dist[pre[v_0]] + w(\{pre[v_0], v_0\}) \overset{\text{definesShortestDistances}}{=} dist[v_0].$$

It remains to show that $\forall 0 \leq i < |\bar{p}| - 1 : pre[\bar{p}_{i+1}] = \bar{p}_i$.

$$\begin{aligned}
\forall 0 \leq i < |\bar{p}| - 1 : pre[\bar{p}_{i+1}] = \bar{p}_i &\overset{\text{extract } \bar{p}_{|\bar{p}|-1}}{\Leftrightarrow} \\
\forall 0 \leq i < |\bar{p}| - 2 : pre[\bar{p}_{i+1}] = \bar{p}_i \wedge pre[\bar{p}_{|\bar{p}|-1}] = \bar{p}_{|\bar{p}|-2} &\overset{\text{definition of } \bar{p}}{\Leftrightarrow} \\
\forall 0 \leq i < |p| - 1 : pre[p_{i+1}] = p_i \wedge pre[v_0] = p_{|p|-1} &\overset{\text{induction hypothesis}}{\Leftrightarrow} \\
\mathbb{T} \wedge pre[v_0] = pre[v_0] &\Leftrightarrow \mathbb{T}.
\end{aligned}$$

Thus the second branch of the proof is completed and thus $F(c+1)$ holds. □.

We now show that the algorithm listed in the pseudo code is correct.

Lemma: *Dijkstra(n, s, w)* terminates and the output *C, dist, pre* of *Dijkstra(n, s, w)* satisfies *(C, dist, pre, w, s, n) is shortest path.*

Proof: At first it is shown that *Dijkstra(n, s, w)* terminates. Initially the set $Q = \text{Vertex}_n$; thus Q is a finite set. In each iteration of the while-loop exactly one vertex is removed from Q. The while-loop terminates when $Q \cap C = \emptyset$, thus the while-loop iterates at most n times. In each iteration of the nested for-loop exactly one vertex of Q is investigated, thus the for-loop terminates after $|Q|$ iterations.

It is now shown that the output *C, dist, pre* of *Dijkstra(n, s, w)* satisfies *(C, dist, pre, w, s, n) is shortest path.* At first *SetOfLinkedVertices(C, s, w, n)* is shown. Initially s is added to C and the set $Q = \text{Vertex}_n$. In the algorithm only vertices from Q are added to C and thus $C \subseteq \text{Vertex}_n$. In the run of the algorithm, vertices are added to C whenever they are neighbours of the current selected node u and have not been added to C yet. This guarantees that all neighbours of nodes in C are also in C and that for each node which is added to C a node exists, which is already in C. Thus *SetOfLinked-Vertices(C, s, w, n)* holds.

The proof for *DefinesShortestDistances(dist, pre, C, s, w)* is done by induction on the set of visited nodes V. In the given pseudo-code V is equal to the set $Vertex_n \setminus Q$. The following invariants shall hold at each time

- $dist[s] = 0$
- $\forall v \in V \backslash \{s\} : \exists v_1 \in V : neighbour(w, v, v_1) \wedge dist[v] = dist[v_1] + w(\{v, v_1\}) \wedge pre[v] = v_1$
- $\forall v_1, v_2 \in V : neighbour(w, v_1, v_2)) \Rightarrow dist[v_1] \leq dist[v_2] + w(\{v_1, v_2\})$

Initialisation: Initially $V = \{\}$ and $dist[s] = 0$ and thus the invariants are true. Since the distance of s is not changed anymore in the algorithm the first invariant holds throughout the induction steps.

Induction step: Assume that the invariants hold for k vertices $\{v_1 = s, \ldots, v_k\}$. It has to be shown that the invariants still hold after node v_{k+1} is added to V.

Let v_{k+1} be the node in $C \setminus V$, which has minimal distance and thus is added to V. Since $v_{k+1} \in C$, there must exist a node $v_i \in V : i \in \{1 \ldots k\}$ such that $dist[v_{k+1}] = dist[v_i] + w(\{v_{k+1}, v_i\})$ and $pre[v_{k+1}] = v_i$, otherwise $v_{k+1} \notin C$. Thus the second invariant holds.

For every node $v_j \in V : j \in \{1 \ldots k\}$ such that v_j and v_{k+1} are connected it holds that $dist[v_{k+1}] \leq dist[v_j] + w(\{v_j, v_{k+1}\})$. This is guaranteed by the "if v ∈ C"-branch in the algorithm, because if v_{k+1} is linked to more than one vertices in V, this branch ensures, that the distance between the starting point s and v_{k+1} is set to the smallest one.

Thus when v_{k+1} is added to V, the current distance $dist[v_{k+1}]$ is the distance of the shortest path considering only nodes in V. Thus the third invariant holds. The remaining question is, if a shorter path from s to v_{k+1} exists, which includes nodes, which have not been added to V yet.

It can be proved by contradiction, that no such path exists:

Let $\delta(v)$:="length of the shortest path from the given node s to the node v". Suppose $v_{k+2} \in C$ is the first vertex to be added to V, such that $dist[v_{k+2}] \neq \delta(v_{k+2})$. Since $v_{k+2} \in C$ there must exist a shortest path from s to v_{k+2}. Let $p = (s, v_i, v_j, v_{k+2})$ be the shortest path from s to v_{k+2}, such that $v_i \in V$ and $v_j \notin V$. Since v_{k+2} is the first vertex to be added to V, such that $dist[v_{k+2}] \neq \delta(v_{k+2})$, it holds that $dist[v_i] = \delta(v_i)$.

When v_i was added to V, the distance of v_j was updated, such that $dist[v_j] = \delta(v_j)$. Since v_{k+2} is the successor of v_j in the path p it holds that $dist[v_j] = \delta(v_j) \leq \delta(v_{k+2}) \leq dist[v_{k+2}]$. On the other hand it holds that $dist[v_{k+2}] \leq dist[v_j]$, because Dijkstra's Algorithm always selects the node in C which has minimal distance to be added to V.

Thus $dist[v_j] = \delta(v_j) = \delta(v_{k+2}) = dist[v_{k+2}]$, which contradicts the hypothesis. So when a node is added to V, its current distance is already the distance of its shortest path.

Termination: After the algorithm terminates $V = C$ and thus the invariants hold for all vertices, which are connected and thus the output is correct. □

4.3. Java Program

From the pseudo-code listed in Section 4.1 a Java program is implemented.

The occurring data types in the algorithm are implemented as integers and arrays. This is done to keep the implementation of the algorithm and the following JML specification as simple as possible. The set of vertices is modelled by an integer n, which defines the cardinality of the set of vertices. The occurring vertices are labelled with numbers from 0 to $n-1$. Each set occurring in the pseudo-code is modelled as an array of length n. If a node is an element of a set the corresponding position in the array is set to 1, otherwise to 0.

The algorithm takes a two-dimensional array *weight* and two integers s and n as input values. The integer n defines the cardinality of the set of vertices. The second integer s defines the start vertex of the graph. The two-dimensional array *weight* models the weighted graph. The array *weight* defines a quadratic and symmetric matrix of dimension n. For instance if the value *weight[0][1]* is greater than 0, then vertex 0 is connected to vertex 1 and the value of *weight[0][1]* determines the weight of their common edge. Since a vertex is not allowed to be connected to itself, the diagonal entries have to be 0.

The output of the algorithm is an array of arrays of integers of size 3. Each entry is of size n. The first entry defines the set of linked vertices. Each vertex, which is linked to the starting vertex is marked with 1, vertices which are not linked are marked with 0. The second entry stores the shortest distances of the linked vertices to the start vertex and the third entry stores the predecessor of each vertex in the shortest path from the start vertex. In the second and third array only those entries, where the corresponding entry in the first array is equal to 1 are well defined. Entries, where the corresponding entry in the first array is equal to 0 are dummy-elements.

In the algorithm all necessary arrays which implement sets and lists are defined and initialised at the beginning of the method. The core of the algorithm is the while-loop which is executed as long as a connected but still unvisited node exists. In the loop the node which is connected but still unvisited and has the shortest distance is selected and stored in the variable u. Then for each unvisited neighbour of u the distance, where u is the predecessor in the path, is calculated. If a neighbour is already connected, the calculated distance is compared to the assigned distance and if necessary updated. If a neighbour is not connected, it is added to the set of connected vertices and the calculated distance is assigned. If the distance of a neighbour node is changed, the value of the predecessor is set to u.

```
1  public static int[][] DijkstraAlg(int weight[][], int s, int n){
2      ...
3
4      while(existsMin(Q,C)){
5          u = returnMinDist(Q,C,dist);
6          Q[u]=0;
7
8          for(int v =0; v<n; v++){
9              if (Q[v]==1 && weight[u][v]>0){
10                 newdist = dist[u]+weight[u][v];
```

```
11
12                 if(C[v]==1){
13                     if(newdist<dist[v]){
14                         dist[v]=newdist;
15                         pre[v] = u;
16                     }
17                 }
18                 else{
19                     dist[v]=newdist;
20                     pre[v] = u;
21                     C[v]=1;
22                 }
23             }
24         }
25     }
```

The complete implementation of the algorithm including the initialisation part and including all helper functions can be seen in Appendix B.1.

4.4. JML

The specification in JML is performed in two stages. The first version of the JML specification introduces the pre- and postconditions. The second version is an extension of the first version and introduces invariants and termination terms of the loops. Both version are analyzed with ESC/Java2.

A verification with Java KeY is not attempted, since a complete analysis of Dijkstra's Algorithm with Java KeY can be found in [28]. In his master thesis Volker Klasen tried to verify five different implementations of Dijkstra's Algorithm. Four of them are abstract implementations, which can be verified with Java KeY using interactive proofs. The fifth implementation is a concrete implementation and could not be shown completely, since two proofs could not be completed with Java KeY.

4.4.1. Version 1

The first version introduces all pre- and postconditions of the functions. The predicates of the formal specification of the algorithm are defined as *static pure model functions*. The pre- and postconditions of the function *DijkstraAlg* match the input condition from chapter 4.2.1 and the output condition from chapter 4.2.2.

The main goal is to verify the JML specification of the function for the algorithm.

```
1  /*@ public normal_behavior
2  @ requires weight != null;
3  @ requires DefWeight(weight, n);
4  @ requires n > 0;
5  @ requires 0 <= s && s < n;
6
7  @ assignable \nothing;
8
9  @ ensures \result.length == 3;
10 @ ensures (\forall int i; 0<=i && i< 3; \result[i]!=null && \result[i].length==n);
11 @ ensures SetOfLinkedVertices(\result[0], s, weight, n);
12 @ ensures DefinesShortestDist(\result[1], \result[2], \result[0], s, weight, n);
13 @*/
14 public static int[][] DijkstraAlg(int weight[][], int s, int n)
```

The *requires* clauses define that *weight* is a quadratic and symmetric matrix, where each diagonal entry is equal to 0. Furthermore the input variable n, which defines the cardinality of the set of vertices needs to be greater than 0 and the input variable s needs to be in the range of the vertices.

The *ensures* clauses define, that the output *result* is a two dimensional array of size $3 \times n$. The first entry shall define the set of linked vertices. The second entry stores the shortest distances from the given vertex to each vertex and the third entry stores the predecessor of a vertex in the shortest path.

The JML specification of the remaining functions can be seen in Appendix B.2.1.

Applying ESC/Java2 with its default settings on the full specification returns no warning, which can be seen in Appendix B.2.1. To affirm the result of ESC/Java2 small errors were added in the program code or in the annotation of the program, which were all detected by ESC/Java2.

4.4.2. Version 2

The second version of the JML specification extends the first version by including all invariants and termination terms for loops.

To increase the readability of the loop invariants in the function *DijkstraAlg*, it is tried to define several predicates as *static pure model functions*. Unfortunately ESC/Java2 returns then the warning "VC too big". Thus all annotations are stated explicitly, also the annotations of the method itself have to be stated explicitly.

The function *DijkstraAlg* needs two loop invariants, one for the while-loop and a second one for the nested for-loop. The loop invariant for the while-loop can be seen below.

```
1  /*@
2  /* Precondition of the function */
3  @ loop_invariant weight != null;
4  @ loop_invariant weight.length == n;
5  @ loop_invariant (\forall int i; 0<=i && i<weight.length; weight[i]!=null && weight[i].length==n);
6  @ loop_invariant (\forall int i; 0<=i && i<n;
7  @      (\forall int j; 0<=j && j<n; weight[i][j] == weight[j][i] && weight[i][j]>=0));
8  @ loop_invariant (\forall int i; 0<=i && i<n;weight[i][i] == 0);
9  @ loop_invariant n>0 && 0 <= s && s < weight.length;
10
11 /* Range of local variables */
12 @ loop_invariant C!= null && C.length == n;
13 @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==0 || C[i]==1);
14
15 @ loop_invariant Q!= null && Q.length == n;
16 @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 || Q[i]==1);
17
18 @ loop_invariant dist!= null && dist.length == n;
19 @ loop_invariant (\forall int i; 0<=i && i<n; dist[i]>=-1);
20
21 @ loop_invariant pre!= null && pre.length == n;
22 @ loop_invariant (\forall int i; 0<=i && i<n; -1<=pre[i] && pre[i]<n);
23
24 @ loop_invariant C[s]==1 && dist[s]==0;
25 @ loop_invariant -1<=u && u<n;
26 @ loop_invariant newdist>=-1;
```

```
27
28 /* Set Of Visited Linked Vertices*/
29 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0;
30 @     (\forall int j; 0<=j && j<n; neighbour(weight,i,j)==> C[j]==1 ));
31 @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
32 @     (\exists int j; 0<=j && j<n && C[j]==1; neighbour(weight,i,j)));
33
34 /* Defines Shortest Dist Of Visited Nodes*/
35 @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
36 @     (\exists int j; 0<=j && j<n && Q[j]==0; pre[i]==j &&
37 @       neighbour(weight,i,j) && dist[i]==dist[i]+weight[i][j]));
38 @ loop_invariant(\forall int i; 0<=i && i<n && C[i]==1;
39 @     (\forall int j; 0<=j && j<n && C[j]==1;
40 @       neighbour(weight,i,j) ==> dist[j]<=dist[i]+weight[i][j]));
41
42 /* Visited implies Connected */
43 @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 ==> C[i]==1);
44
45 /* Connected implies defined predecessor and distance */
46 @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==1 ==> (dist[i]>=0 && pre[i]!=-1));
47
48 /* Distance of visited nodes is shorter than the distance of unvisited but connected nodes */
49 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0;
50 @     (\forall int j; 0<=j && j<n && Q[j]==1 && C[j]==1; dist[i]<=dist[j]));
51
52 @ decreases (\num_of int i; 0<=i && i<n; Q[i]==1);
53 @*/
54 while(existsMin(Q,C))
```

Since none of the input variables is changed the precondition of the function needs to hold. Furthermore each local variable needs to be in its specified range. The output conditions *SetOfLinkedVertices* and *DefinesShortestDist* need to hold for all visited nodes and thus the loop invariants *Set Of Visited Linked Vertices* and *Defines Shortest Dist Of Visited Nodes* are modifications of the output condition. Furthermore each visited node needs to be an element of C and for all elements of C the corresponding distance and predecessor is defined. It also holds that the distance of the visited nodes is shorter than the distances of the connected but unvisited nodes.

In each iteration of the loop one node is visited and thus the number of unvisited decreases with each iteration of the loop.

The loop invariant of the nested for-loop is basically the same as for the while-loop. The only difference is that some statements of *Set Of Visited Linked Vertices* and *Defines Shortest Dist Of Visited Nodes* do not hold for the node u. Thus those properties are modified by excluding u from the domain and by defining a new property which holds for those neighbours of u, which have already been investigated in the run of the for-loop.

```
1 /* Set Of Visited Linked Vertices*/
2 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0 && i!=u;
3 @     (\forall int j; 0<=j && j<n; neighbour(weight,i,j)==> C[j]==1));
4 @ loop_invariant (\forall int i; 0<=i && i<v; neighbour(weight,u,i)==> C[i]==1 );
5 @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
6 @    (\exists int j; 0<=j && j<n && C[j]==1; neighbour(weight,i,j)));
```

The complete listing of the JML specification can be seen in Appendix B.2.2.

The output of ESC/Java2 yields 8 warnings. Six warnings concern the loop invari-

ants of the function *DijkstraAlg* and state that " Warning: Loop invariant possibly does not hold (LoopInv)". The warnings for the loop invariant arise when the corresponding invariant contains a nested quantified formula. The remaining two warnings concern the termination term of the while loop and state that the "loop variant function may not lead to loop exit" and that the "loop variant function possible not decreased".

The output of ESC/Java2 including all warnings can be seen in Appendix B.2.2.

4.5. TLA

The specification of Dijkstra's Algorithm in TLA is based on sets. The set of vertices is defined as a set of natural numbers ranging from 1 to a given constant n. The set of edges is created by taking all elements of the power set of the set of vertices which have a cardinality equal to 2.

The set *weights* defines all combinations of edges and weights as tuples. The first element in the tuple is an element of the set of edges and the second element in the tuple is a number in the range [0, maxWeight]. The variable *maxWeight* is a constant in the module and defines the maximum weight of an edge; its value has to be defined in the model of the TLC Model Checker. If the weight is equal to 0, it means that the two corresponding vertices are not linked. The set *Wgraph* consists of all elements of $\mathbb{P}$*(weights)*, which have a cardinality of $\frac{n*(n-1)}{2}$. The value $\frac{n*(n-1)}{2}$ is the number of subsets with two elements of a set with cardinality n. Furthermore it must hold that for each edge exists a corresponding tuple in each element of *Wgraph*. A weighted graph *wgraph* is an element of the set *Wgraph*.

The starting vertex s is chosen non-deterministically from the set of vertices. Initially the set of connected vertices contains s and the set of unvisited nodes Q contains all vertices.

The lists *dist* and *pre* are defined as arrays of length n and initially they are filled with dummy elements.

With the definition of the variables, the pseudo-code of Dijkstra's Algorithm from Section 4.1 is implemented straight forward in PlusCal.

```
1  --algorithm Dijkstra{
2
3  variables
4      Vertex = {l \in 1..n : TRUE};
5      Edge = {e \in SUBSET(Vertex): Cardinality(e)=2};
6
7      weights = [e : Edge , w : 0..maxWeight];
8      Wgraph = {wg \in SUBSET(weights): Cardinality(wg)=((n*(n-1))\div 2)
9                    /\ \A edge \in Edge: \E tup \in wg : tup.e =edge};
10     wgraph \in Wgraph;
11
12     s \in Vertex;
13
14     C = {s};
15     Q = Vertex;
16
17     dist = [ver \in Vertex |-> 0];
```

```
18      pre = [ver \in Vertex |-> -1];
19
20      u = -1;
21      vtup = <<{0,0},0>>;
22      v = 0;
23      length = -1;
24
25  {
26  while(Q \cap C # {}){
27      u := CHOOSE v1 \in Q \cap C: \A v2 \in Q \cap C : dist[v1]<=dist[v2];
28      Q := Q\{u};
29      v := 1;
30
31      while(v<=n){
32        if( v \in Q /\ \E tup \in wgraph : tup.e ={u,v} /\ tup.w >0){
33              vtup := CHOOSE tup \in wgraph: tup.e ={u,v};
34              length := dist[u] + vtup.w;
35
36              if(v \in C){
37                  if(length<dist[v]){
38                      dist[v]:= length;
39                      pre[v]:=u;
40                  }
41              }
42
43              else{
44                  dist[v]:= length;
45                  pre[v]:=u;
46                  C := C \cup {v};
47             };
48         };
49         v := v+1;
50  };
51  };
52  }}
```

The PlusCal algorithm is automatically translated into a TLA model. The complete PlusCal algorithm including its translation into TLA is listed in Appendix B.3.

The model shall satisfy two properties. The first property *Termination* ensures that the algorithm terminates. The second property *IsCorrect* shall guarantee that the output of the algorithm satisfies the output condition of the formal specification. Therefore the predicates *SetOfLinkedVertices* and *DefinesShortestDistances* of the output condition are specified. The property *IsCorrect* states, that whenever the algorithm terminates both predicates *SetOfLinkedVertices* and *DefinesShortestDistances* hold.

The definition of the properties include a new variable *pc*, which is automatically added by the translation of the PlusCal specification. The variable *pc* is a program counter and indicates which step of the model can be executed at each state of the model. In the TLA translation the program counter has three states: "*Lbl_1*", "*Lbl_2*" and "*Done*". The states *Lbl_1* and *Lbl_2* deal with intermediate steps in the execution of the algorithm, whereas the state *Done* indicates that the algorithm has terminated.

```
1  \* WHAT TO CHECK
2
3  \* Algorithm terminates
4  Termination == <>(pc = "Done")
5
6  \* Algorithm satisfies output condition
7  neighbour(w,u1,v1) == \E tup \in w: tup.e ={u1,v1} /\ tup.w >0
```

```
8
9   SetOfLinkedVertices(C1,s1,w,n1) ==
10      C1 \in SUBSET(Vertex) /\ s1 \in C1 /\
11      (\A v1 \in C1: \A v2 \in Vertex: neighbour(w, v1,v2) => v2 \in C1) /\
12      (\A v1 \in C1\{s1}: \E v2 \in C1: neighbour(w, v1,v2))
13
14  DefinesShortestDistances(dist1,pre1,C1,s1,w) ==
15      dist[s1]=0 /\
16      (\A v1 \in C1\{s1}: LET v2==pre1[v1] IN (neighbour(w, v1,v2) /\
17          (LET vtup1==CHOOSE tup \in w: tup.e ={v1,v2} IN dist[v1]=dist[v2]+vtup1.w))) /\
18      (\A v1,v2 \in C1: ( (LET vtup1==CHOOSE tup \in w: tup.e ={v1,v2} IN
19         neighbour(w, v1,v2) =>dist[v1]<=dist[v2]+vtup1.w)))
20
21  IsShortestPath(C1,dist1,pre1,w,s1,n1) ==
22      SetOfLinkedVertices(C1,s1,w,n1) /\
23      DefinesShortestDistances(dist1,pre1,C1,s1,w)
24
25  IsCorrect == [](pc = "Done" => (IsShortestPath(C,dist,pre,wgraph,s,n)))
```

Before the TLC Model Checker can be applied, the behaviour of the system and the constants need to be specified. In this model the behaviour of the system is defined by the formula *Spec*, which is defined in the TLA translation. The vertices shall range from 1 to 4 and the weight of the edges shall be smaller than 4. Therefore the constants n is set to 4 and *maxWeight* is set to 3. Finally the two properties *Termination* and *IsCorrect* need to be listed in the "What to check?" section of the TLC model checker. The complete setting of the model can be seen in Figure 4.2.

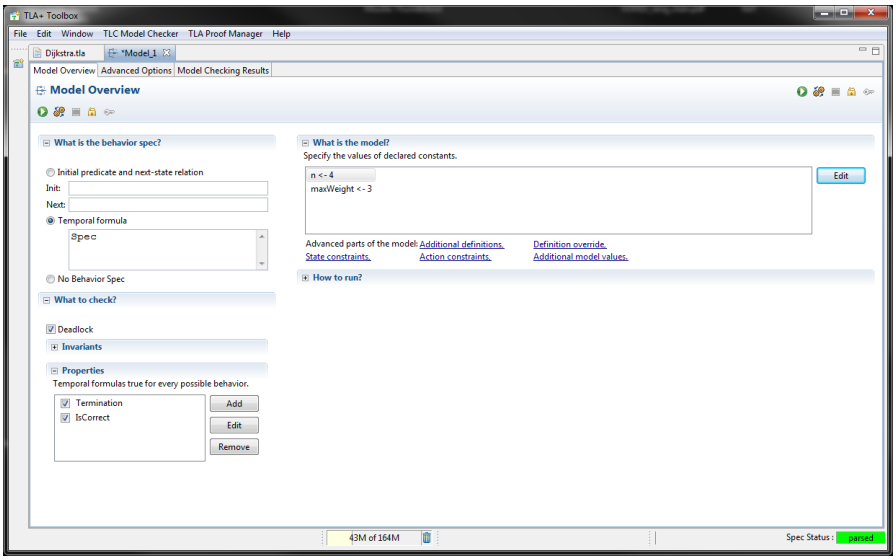


Figure 4.2.: Setting up the model in the TLC Model Checker

The TLC Model Checker does not detect any error in the model, thus all properties are correct in the specified range. It is not guaranteed that the properties are correct for all weighted graphs, but due to the extensive scope, the confidence level is high that the properties hold for all weighted graphs.

4.6. Alloy

Dijkstra's Algorithm in Alloy is specified by two models. The first model introduces the specification of a weighted graph and the algorithm is defined. The instances of this model show executions of Dijkstra's Algorithm. In the second model the output condition, that Dijkstra's Algorithm returns the length of the shortest path is defined and is checked for correctness. In both models it is defined that the weighted graph is fully connected.

4.6.1. Model 1

The first model deals with the specification of a weighted graph and the implementation of Dijkstra's Algorithm. The model shall return successful runs of the algorithm. The specification is based on a module of Jan van Eijck [49]. At the beginning of the model a weighted graph is defined. Therefore the module `util/graph` is included, where several properties of graphs are defined.

At first the nodes of a graph are defined. The definition of a node includes a relation w, which maps two nodes up to one integer. This means that if two nodes are mapped to an integer, the nodes have a common edge and the weight of the edge is the value of the integer. The signature *Root* defines exactly one node in the graph, which shall be the starting vertex of Dijkstra's Algorithm. The relation w has to be symmetric and the weight has to be in the range [1,4]. It was tried to merge the properties "w[x,y]!=Int[0] and gt[w[x,y], Int[-1]]" to "gt[w[x,y], Int[0]]" but then the Alloy Analyzer is not able to find instances of the model.

The function *edge* returns all pairs of nodes from the graph which are connected by the relation w. The fact *Graph* defines some properties for the connected nodes, which shall always hold. The first property "noSelfLoops" defines that a node cannot have an edge to itself and the second property "undirected" defines that edges are symmetric. The third property "weaklyConnected" defines that all nodes in the graph have to be connected, but not every node has to be connected to every other node.

```
1  sig Node { w: Node -> lone Int}
2  one sig Root extends Node {}
3
4  fact wSymm { all x,y: Node | w[x,y] = w[y,x] }
5
6  fact wDef { all x,y: Node | w[x,y]!=Int[0] and gt[w[x,y], Int[-1]] and lt[w[x,y], Int[5]]}
7
8  fun edge: Node -> set Node {
9     { x, y: Node | some w[x,y] }
10 }
11
12 fact Graph {
13      noSelfLoops[edge]
14      undirected[edge]
15      weaklyConnected[edge]
16 }
```

To show the several states of the algorithm, where in each state one node of the graph is selected and the distances to each neighbour node are calculated and if necessary updated, an object *state* is defined. In a *state* the current sets of visited and unvisited

nodes are stored. Furthermore the function d stores the distances from the *Root* node to each node and the function *prev* stores for each node the predecessor in the shortest path. Since the *states* indicate states of the algorithm, they need to be modelled as ordered states, which is done by using the `util/ordering module`.

The initial state of the algorithm is described in the predicate `init`. In the first state of the algorithm the node *Root* is marked as visited and all other nodes are marked as unvisited. The distance of the node *Root* is set to 0 and its predecessor is not defined. Furthermore the distance to each neighbour node of the starting node is computed and its predecessor is set to *Root*. The distances of all other nodes are set to -1 and their predecessors are undefined.

A step of the algorithm is defined in the predicate *Dijkstra*, which describes the relation between two states. A node u is selected from the unvisited nodes of the predecessor state, which has the smallest positive distance. Then for all unvisited neighbour nodes v of u which have a negative distance or their assigned distance is longer than the sum of the distance of u and the weight of the common edge, the distance of v is updated and the predecessor of v is set to u. In the end u is removed from the set of unvisited nodes and added to the set of visited nodes.

The whole algorithm is modelled as a `fact`. In the `fact` it is defined that the predicate *init* has to hold and that for each *state s* and its successor *state s'* the predicate *Dijkstra* has to hold.

The algorithm is called by the predicate *allVisited* which seeks for runs of the algorithms, where in the last state all nodes are marked as visited. Since in each state of the algorithm exactly one node is added to the set of visited nodes, the size of the scope has to be the same for the *Node* objects and the *State* objects.

The algorithm is called for graphs containing up to 4 *Nodes* and thus the run of the algorithm needs up to 4 *States*. Since the graph contains up to 4 nodes and the weight between two nodes is at most 4, the maximum length of a path is 12. Thus the size of the bitwidth for the integers has to be increased to 5, which makes it possible to calculate in the range [-16, 15].

```
1  sig State {
2    visited: set Node,
3    unvisited: set Node,
4    d: Node -> lone Int,
5    prev: Node -> lone Node
6  }
7
8  pred init {
9    let fs = so/first | {
10       fs.visited = Root
11       fs.unvisited = Node - Root
12       fs.d[Root] = Int[0]
13       no fs.prev[Root]
14       all x: (Node - Root) | {
15          x in edge[Root] =>{ fs.d[x] = w[Root,x] and fs.prev[x]=Root}
16          else {fs.d[x] = Int[-1] and no fs.prev[x]}}
17       }
18 }
```

```
19
20 pred Dijkstra [pre, post: State] {
21     some u: pre.unvisited |
22         {pos[pre.d[u]] and
23           all x: pre.unvisited - u | {pos[pre.d[x]] => { integer/lte[pre.d[u],pre.d[x]]}}}
24
25      and all v: Node | {
26         {v in edge[u] and v in pre.unvisited
27          and (neg[pre.d[v]] or ( pos[pre.d[v]] and add[pre.d[u],w[u,v]] < Int[pre.d[v]]))}=>
28            {post.d[v] = add[pre.d[u],w[v,u]] and post.prev[v]=u}
29         else {post.d[v] = pre.d[v] and post.prev[v]= pre.prev[v]} }
30       and post.visited = pre.visited + u
31       and post.unvisited = pre.unvisited - u
32 }
33
34 fact DijkstraAlg {
35      init
36      all s: State - so/last | let s' = so/next[s] |Dijkstra[s,s']
37 }
38
39 pred allVisited {so/last.visited = Node}
40
41 run allVisited for 4 Node, 4 State, 5 Int
```

The complete model can be seen in Appendix B.4.1.

The graph of a run of the algorithm is visualized in Figure 4.3 to make the corresponding output of the Alloy Analyzer more clearly. In this graph each node is connected to each other node of the graph. The states of the run of the algorithm are shown in Figure 4.4.

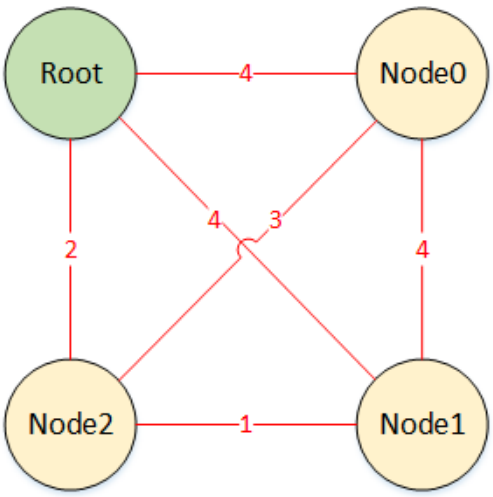


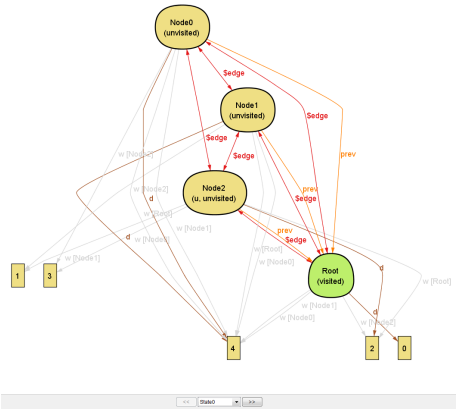
Figure 4.3.: Example Graph

It can be seen that in the first state *State0* the predicate *init* is applied and that *Node2* is selected as the next node, which shall be visited. Since the node *Root* is connected to each remaining node, the distance of every other node in the graph is positive.

In the step from *State0* to *State1*, the updates of the distances and predecessors for all neighbours of *Node2* are done. It can be seen that the length of the path from *Root* to *Node1* via *Node2* is shorter than the distance of the path directly from *Root* to *Node1* and thus an update of the distance and the predecessor of *Node1* is done. *Node1* is

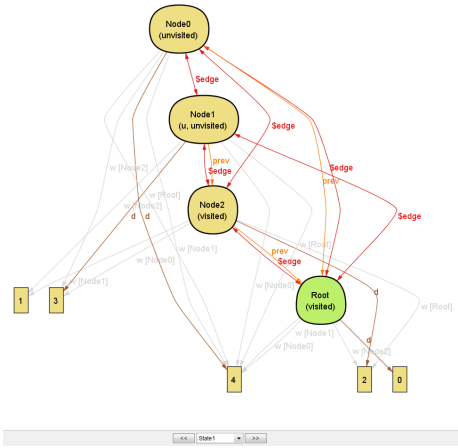
selected as the next node, which shall be visited.

In the states *State2* and *State3* no updates of the distances and predecessors of the nodes are done and the algorithm terminates in *State3*, where all nodes are marked as visited.



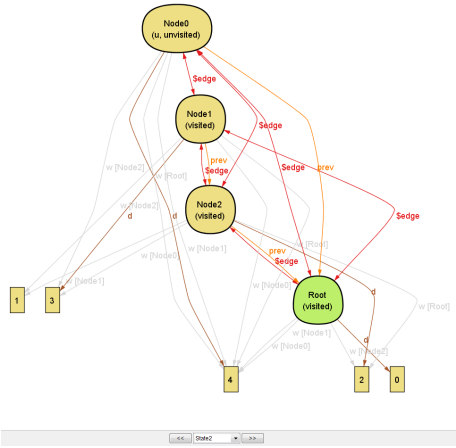
(a) State0

	visited	d	prev
Root	yes	0	-
Node0	no	4	Root
Node1	no	4	Root
Node2	no	2	Root



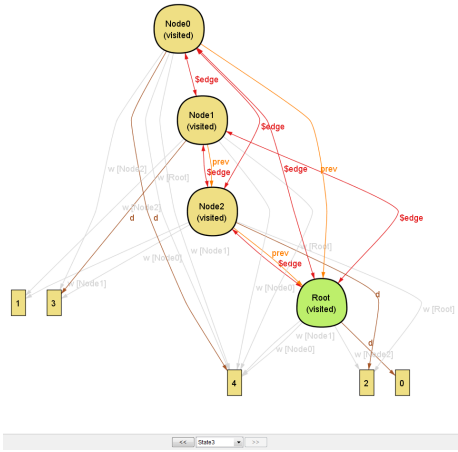
(b) State1

	visited	d	prev
Root	yes	0	-
Node0	no	4	Root
Node1	no	3	Node2
Node2	yes	2	Root



(c) State2

	visited	d	prev
Root	yes	0	-
Node0	no	4	Root
Node1	yes	3	Node2
Node2	yes	2	Root



(d) State3

	visited	d	prev
Root	yes	0	-
Node0	yes	4	Root
Node1	yes	3	Node2
Node2	yes	2	Root

Figure 4.4.: A visualization of a run of Dijkstra's Algorithm in Alloy

4.6.2. Model 2

In the second model the output condition of Dijkstra's Algorithm is added to the first model. The specification of the weighted graph and the implementation of the algorithm remains the same as in the first model. The complete model is listed in Appendix B.4.2.

For the definition of the output condition the specification of a path is needed. A path is modelled as a sequence of *Steps* starting at the node *Root* and ending at a selected node, which is chosen non-deterministically from the remaining nodes in the graph. In a *Step* the starting node and the destination node of the step are stored. The starting node and the destination node of a step have to be linked by an edge in the graph. The node, which is selected to be the last node of the path is stored in *goal*. A *Step* also stores the distance from the root node to the starting node of the step and the distance between the starting node and the destination node of the step. It is defined, that a step is followed by a step if and only if the node *from* is not the destination vertex of the path.

It must hold that the destination node of a step is the starting node of its successor step and the distance between the root node and the starting node of a step is the sum of the distance from the root node to the starting node of the predecessor step and the distance between the starting node and the destination node of the predecessor step. Furthermore the destination node of the whole path must not change. Nodes shall never occur twice in a path, thus it must not be cyclic.

The goal of the model is to check, that when the algorithm is finished, no path from the node *Root* to any node in the graph exists which has a shorter distance than the distance, which is calculated by the algorithm. Since the graph is defined to be fully connected the output condition concerning the set of connected vertices can be neglected.

The assertion is checked "for 49 Step, 16 Path, 4 Node, 4 State, 5 Int", because for 4 nodes exist up to 16 non-cyclic paths starting at the node *Root*. There exists exactly 1 path containing 1 step, 3 paths containing 2 steps, 6 paths containing 3 steps and 6 paths containing 4 steps. Thus a total of 49 steps is needed.

```
1  sig Path { firstStep: Step }{firstStep.from=Root and firstStep.sumdist=Int[0]}
2
3  sig Step {
4      from, to: Node,
5      goal: one Node,
6      currdist: one Int,
7      sumdist: one Int,
8      nextStep: lone Step}
9      {to in edge[from] and currdist=w[from,to] and from=goal => no nextStep}
10
11 fact RelationshipBetweenSteps {
12     all curr: Step, next: curr.nextStep |{
13         next.from = curr.to and
14         next.goal = curr.goal and
15         next.sumdist = add[curr.currdist, curr.sumdist]}}
16
17 fact PathNotCyclic {
18     no s:Step| s.from in s.^nextStep.to}
19
20 fact allNodesBelongToSomeQueue {
21     all s:Step | one p:Path| s in p.firstStep.*nextStep}
22
```

```
23 assert NoShorterPath{
24     so/last.visited = Node and
25     all n: Node| no p:Path |{
26         p.firstStep.goal = n and
27             some s:Step| {s in p.firstStep.*nextStep and s.from = s.goal
28                 and s.sumdist<so/last.d[s.goal]}} }
29
30 check NoShorterPath for 49 Step, 16 Path, 4 Node, 4 State, 5 Int
```

The assertion is checked by the Alloy Analyzer. The Alloy Analyzer does not find a counterexample, which can be seen in Figure 4.5. This means that for all graphs containing up to 4 nodes, the algorithm returns the shortest path from the root node to every other node in the graph. It is not guaranteed that the assertion is valid for all weighted graphs because it is only valid in the given scope, but the confidence level is high that the assertion holds for all weighted graphs.

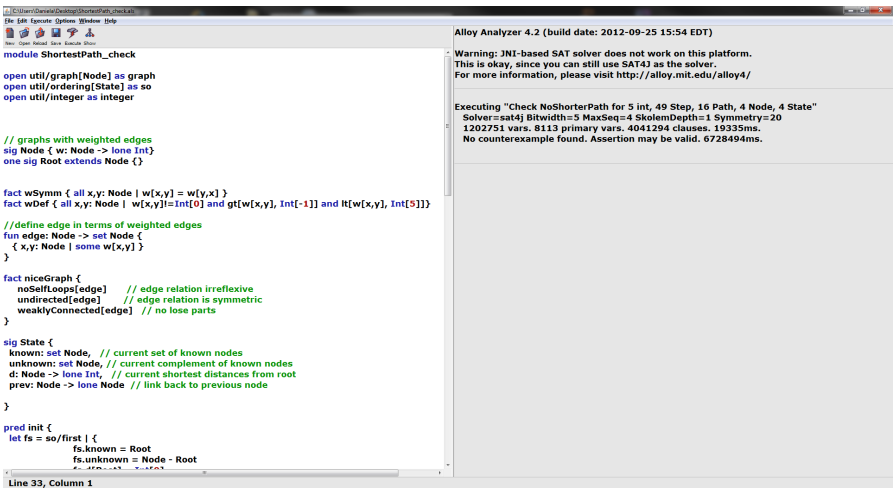


Figure 4.5.: Output of the Alloy Analyzer

4.7. VDM

Dijkstra's Algorithm is specified in VDM++ which allows to use object-oriented design. Before the algorithm is defined all occurring data types have to be specified.

```
1  types
2  public Vertex = int
3  inv v == v >= 1 and v <= n;
4
5  public Edge = set of Vertex
6  inv e == card(e)=2;
7
8  public Weight = int
9  inv w == w >= 0 and w <= maxWeight;
10
11 public Wgraph :: e : Edge
12                  w : Weight;
```

A *Vertex* is defined as an integer in the range [1,n] where n is a constant, which is defined in the model. An *Edge* is defined as a set of exactly two vertices. The weight which is mapped to an edge is an integer greater than or equal to 0 and smaller than

or equal to a constant *maxWeight*. In this model n is equal to 3 and *maxWeight* is set to 2.

The data type *Wgraph* is a `composite type`, which stores an edge and its corresponding weight. The graph in the algorithm is then defined as a set of *Wgraph*.

The graph itself and all occurring sets and maps are defined globally in the section `instance variables`. Initially the weighted graph and the set of connected vertices C are empty sets and the starting vertex s is set to 0. The set of unvisited vertices contains all vertices and the mappings *dist* and *pred* are empty maps.

```
1  instance variables
2    public wgraph : set of Wgraph := {};
3    public s : int := 0;
4    public C: set of Vertex := {};
5    public Q: set of Vertex := {1,...,n};
6    public dist: map Vertex to nat := {|->};
7    public pred: map Vertex to Vertex := {|->};
```

The invariant, which is added for the instance variables is not very comprehensive, because the global defined invariants have to hold at each time of the execution of the system. This means, that in contrast to loop-invariants, which have to hold before and after each iteration of a loop, these global variants have to hold before and after each step in the execution of the algorithm. A possibility for defining more invariants of the system would be to introduce a program counter in the algorithm and reason about the invariants in the different states of the program counter. Since using a program counter in a while-loop based program is not conventional, it is not attempted to add a program counter in the program and thus only the invariants, which hold globally are added in the `inv` section.

```
1  inv DefinesShortestDistofVisited(dist, pred, C, Q, s, wgraph) and
2      VisitedImpliesConnected(C, Q) and LengthOfDist(C, Q, dist)
```

It holds that for each vertex u in the set of connected vertices exists a visited vertex v, such that the distance of u is the sum of the distance of v and the weight of their common edge. Furthermore for each node in the set of connected vertices it holds that its distance is the smallest possible. The second invariant states that whenever a node is marked as visited it is an element of the set of connected vertices and the last invariant states that the distances of visited nodes are smaller than or equal to the distances of unvisited, but connected nodes.

The algorithm is directly adapted from the pseudo code listed in Section 4.1. It takes a set of *Wgraph* and a *Vertex* as an input and returns the set of connected vertices, the map *dist* and the map *pred*.

The precondition of the operation *DijkstraAlg* states that the input graph g is a valid graph, which means that each possible set of exactly two vertices is mapped to a number in the range [0, maxWeight] and that the starting vertex *startV* is an element of the set *Vertex*. The postcondition *IsShortestPath(dist, pred, C, s, wgraph)* matches the definition of the output condition of Dijkstra's Algorithm listed in Section 4.2.2.

```
public DijkstraAlg: set of Wgraph * Vertex ==> (set of Vertex * map Vertex to int * map Vertex to
    Vertex)
DijkstraAlg(g,startV) ==
(
  s:=startV;
  wgraph:= g;
  C := C union {s};
  dist := dist ++ {s|->0};
  pred := pred ++ {s|->s};

  while (C inter Q <> {}) do (
    let u in set C inter Q be st (forall u1 in set C inter Q & dist(u)<=dist(u1)) in (
    Q := Q\{u};

    for all v in set Q do (
        let uv in set wgraph be st uv.e={u,v} in(
        if uv.w > 0 then (
           let length=dist(u)+uv.w in (
             if v in set C then (
               if length < dist(v) then (
                    dist := dist ++ {v|->length};
                    pred := pred ++ {v|-> u};
                )
             )
             else(
                 dist := dist ++ {v|->length};
                   pred := pred ++ {v|-> u};
                   C := C union {v};
             )
         )
     ))
    );
  ));
 return mk_(C,dist,pred);
)
pre validGraph(g) and validStart(startV)
post IsShortestPath(dist, pred, C, s, wgraph);
```

At the end of the model a `trace` is defined, which imitates an execution of the algorithm. Therefore all possible weighted graphs with $n = 3$ vertices and a maximum weight of $maxWeight = 2$ are defined in the set *graph*. An element of *graph* has a cardinality of $\frac{n \cdot (n-1)}{2}$, because this is the number of subsets with exactly two elements of a set with n elements. If a set of two vertices is mapped to 0, this means that those two vertices are not connected. The algorithm is then called with an element of the set *graph* and a starting Vertex $startV$ in the range [1,n]. The complete model can be seen in Appendix B.5.

```
traces
 S1:
 let vert = power({1,...,n}) in (
    let edges = {e | e in set vert & card(e)=2} in (
        let weights = {0,...,maxWeight} in (
           let graphTuple = {mk_Wgraph(e,w) | e in set edges , w in set weights} in (
              let graph={wg| wg in set power(graphTuple) & (card(wg)=n*(n-1)/2 and (forall ed in
                   set edges & (exists tup in set wg & tup.e =ed)))} in (
                  let g in set graph in(
                     let startV in set {1,...,n} in (
                        DijkstraAlg(g,startV))))))))
```

In the combinatorial testing perspective all possible executions of the trace are performed. It is noticed that for each input set g of the algorithm, also all possible permu-

tations of the *Wgraph objects* in g are checked. This yields to a total number of

$$\underbrace{(maxWeight+1)^{\left(\frac{n\cdot(n-1)}{2}\right)}}_{\substack{\text{each edge is mapped to}\\ \text{an element of } \{0,\ldots,\text{maxWeight}\}}} \cdot \underbrace{\left(\frac{n\cdot(n-1)}{2}\right)!}_{\substack{\text{number of}\\ \text{permutations of } g}} \cdot \underbrace{n}_{startV}$$

test cases. Thus 486 test cases are generated for $n = 3$ and $maxWeight = 2$. Unfortunately it is not possible to apply model checking for weighted graphs with more vertices, because for example for the constants $n = 4$ and $maxWeight = 2$ a total of 2 099 520 test cases are generated, which cannot be checked in a reasonable amount of time.

Figure 4.6.: Result after executing all test cases

In Figure 4.6 it can be seen that none of the 486 test cases generates a run-time error, in particular a violation of the postcondition of *DijkstraAlg*. An example for a certain test case can be seen in Figure 4.7. In this test case the weighted graph contains two edges. One edge connects *node2* and *node3* with a weight of 2 and the second edge connects *node1* and *node3* with a weight of 1. The start vertex in this execution is *node2*. In the output of the algorithm it can be seen, that all three nodes are elements of the set of connected vertices. The map *dist* is displayed by the second set of the result and it can be seen that the shortest path from *node2* to *node3* has a length of 2 and from *node2* to *node1* a length of 3. The third set of the result displays the map *prev*, where it can be seen that the predecessor of *node3* is *node2* and the predecessor of *node1* is *node3*.

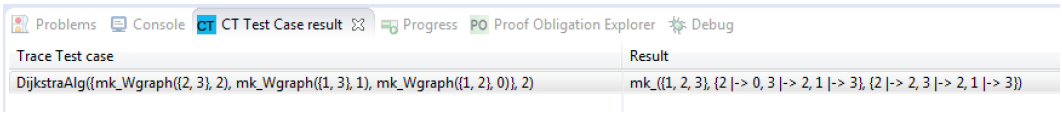


Figure 4.7.: Result of a specific test case

Overture creates 52 proof obligations for the algorithm, which can be seen in Figure 4.8. The proof obligations have nine different types and in Figure 4.9 it is explained under which circumstances which type of proof obligation arises and what needs to be proved. The types of proof obligations are listed in their order of appearance in the `Proof Obligation Explorer`.

PO Proof Obligation Explorer

No.	PO Name	Type
1	Dijkstra`Vertex	type invariant satisfiable
2	Dijkstra`Edge	type invariant satisfiable
3	Dijkstra`Weight	type invariant satisfiable
4	Dijkstra`Q	type compatibility
5	Dijkstra`inv_Dijkstra	state invariant initialized
6	Dijkstra`inv_Dijkstra	state invariant satisfiable
7	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
8	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
9	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
10	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
11	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
12	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
13	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
14	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
15	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	while loop termination
16	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	let be st existence
17	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	legal map application
18	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	legal map application
19	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
20	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	let be st existence
21	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex), length	legal map application
22	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	legal map application
23	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
24	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
25	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
26	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
27	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
28	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
29	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	state invariant holds
30	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	type compatibility
31	Dijkstra`DijkstraAlg(set of (Wgraph), Vertex)	operation establishes postcondition
32	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
33	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
34	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	let be st existence
35	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
36	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
37	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	let be st existence
38	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
39	Dijkstra`DefinesShortestDist(map (Vertex) to (nat), map (Vertex) t...	legal map application
40	Dijkstra`SetOfLinkedVertices(set of (Vertex), Vertex, set of (Wgrap...	type compatibility
41	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	legal map application
42	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	type compatibility
43	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	let be st existence
44	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	legal map application
45	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	legal map application
46	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	type compatibility
47	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	let be st existence
48	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	legal map application
49	Dijkstra`DefinesShortestDistofVisited(map (Vertex) to (nat), map (...	legal map application
50	Dijkstra`LengthOfDist(set of (Vertex), set of (Vertex), map (Vertex)...	legal map application
51	Dijkstra`LengthOfDist(set of (Vertex), set of (Vertex), map (Vertex)...	type compatibility
52	Dijkstra`LengthOfDist(set of (Vertex), set of (Vertex), map (Vertex)...	legal map application

Figure 4.8.: Generated proof obligations

Proof Obligation Type	Description
type invariant satisfiable	Arises for each specified type invariant and has to be proved to ensure that the invariants for elements of the corresponding type are satisfied
type compatibility	Created for each assignment of an instance variable and has to proved to ensure that the assigned value matches the type of the instance variable
state invariant initialized	It needs to be proved that the state invariant holds initially
state invariant satisfiable	It needs to be proved that instance variables exist which satisfy the invariant
state invariant holds	Arises whenever an assignment is made to a an instance variable and it needs to be proved that the invariant still holds after the assignment
while loop termination	Arises for every while loop and it needs to be proved that the loop terminates
let be st existence	Created whenever a let-be-such-that expression is used and it needs to be shown that at least one value fulfills the such-that expression
legal map application	Created whenever a map application is done and it needs to be proved that the argument is in the domain of the map
operation establishes postcondition	Created whenever a function has a postcondition and it needs to be proved that the postcondition holds after the execution of the operation

Figure 4.9.: Proof obligation types

As already mentioned in Section 3.8, the Overture Tool does not include a complete calculus, because the proof obligations are not really strong since no logical rules for reasoning are applied.

For instance for the statement "C := C union {s};" the following proof obligation for proving "type compatibility " is created:

```
1 (forall g:set of (Wgraph), startV:Vertex & ((validGraph(g) and validStart(startV)) =>
2       is_((C union {s}), set of (Vertex))))
```

It is known that s is equal to an element of the set of vertices and that C is the empty set. Thus $C \cup \{s\}$ equals to $\{s\}$, which is a set of *Vertex*.

In the Appendix of [33] an overview of all categories of proof obligations is given and for some categories it is mentioned that the created proof obligations are only place holders to remind the user, that a certain property needs to be proved.

4.8. Event-B

For Dijkstra's Algorithm only one model is introduced in Event-B, because it was noticed in Section 3.9 that the concept of refinement for mathematical algorithms can only be applied for the occurring data types.

Before the algorithm is defined in a `machine`, all occurring data types are introduced in a `context`.

```
1  CONTEXT
2      VarDefinition
3  CONSTANTS
4      n
5      maxWeight
6      Vertex
7      Edge
8      Weight
9      Wgraph
10 AXIOMS
11     axm1:  n = 5
12     axm2:  maxWeight = 3
13     axm3:  Vertex = 1..n
14     axm4:  Edge = {e | e ∈ ℙ(Vertex) ∧ card(e)=2}
15     axm5:  Weight = 0..maxWeight
16     axm6:  Wgraph = Edge → Weight
17 END
```

The variable n defines the maximum number of vertices in a graph and it is set to 5. The second constant *maxWeight* defines the maximum weight of an edge and is set to 3.

The set of vertices is then defined as the set $\{1,\ldots,5\}$. The set of edges is defined by selecting all elements of the power set of the set of vertices which have a cardinality of 2. The set *Weight* is defined as the set $\{0,\ldots,3\}$. A weighted graph is then defined as a total function from the set of edges to the set of weights.

A second `context` is added for defining program counters. In Figure 4.10 all occurring states of the program counter are displayed in a state diagram. The transitions in the state diagram mark the corresponding events in the specification of the algorithm. The complete context is listed in Appendix B.6.2.

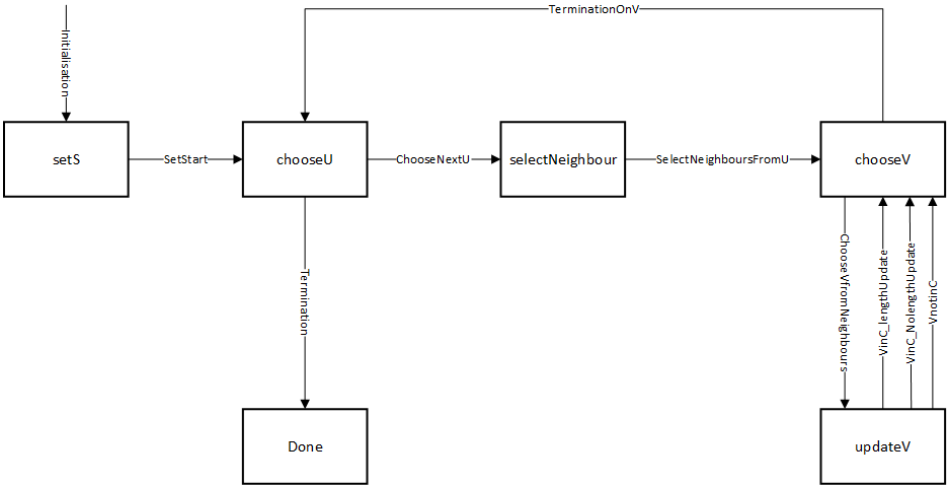


Figure 4.10.: State diagram of the program counter

At the beginning of the `machine` all occurring variables of the algorithm are specified. The data types of the variables are defined as invariants in the `invariants` section.

```
1  INVARIANTS
2    inv1:  wgraph ∈ Wgraph
3    inv2:  s ∈ Vertex
4    inv3:  C ⊆ Vertex
5    inv4:  Q ⊆ Vertex
6    inv5:  dist ∈ Vertex → ℕ
7    inv6:  pre ∈ Vertex → Vertex ∪ {0}
8    inv7:  pc ∈ PC
9    inv8:  u ∈ Vertex ∪ {0}
10   inv9:  v ∈ Vertex ∪ {0}
11   inv10: neighbour ⊆ Vertex
12   inv11: length ∈ ℕ
```

The variable *wgraph* defines a weighted graph and *s* defines the starting vertex of the algorithm. Both variables are selected non-deterministically in the initialisation event. The sets C and Q are subsets of the set of vertices and define the set of connected vertices respectively the set of unvisited vertices. In the map *dist* the length of the shortest path is stored for each vertex and in the map *pre* the predecessor of each vertex in the shortest path is stored. The variable *pc* stores the current state of the program counter.

The node u is the node which is currently visited in the run of the algorithm. In the set *neighbour* all unvisited neighbours of u are stored which have to be investigated. The variable v stores a selected node from the set *neighbour* for which the distance is calculated and if necessary updated. The calculated distance is stored in *length*.

The `invariants` section does not only cover the type invariants, it also covers the invariants for each state of the program counter. For example when the program counter is equal to “chooseU”, which corresponds to the state that an iteration of the while loop of the pseudo code, which is listed in Section 4.1, is completed, the loop invariants of the while loop have to hold.

```
chooseU_1: pc = chooseU ⇒ ( ∀ u1 · (u1 ∈ Vertex\Q ⇒ ∀ u2 · (u2 ∈ Vertex ⇒ (wgraph({u1, u2})>0
    ⇒ u2 ∈ C))))
chooseU_2: pc = chooseU ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
chooseU_3: pc = chooseU ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ (∃ u2 · (u2 ∈ Vertex\Q ∧ pre(u1)=u2 ∧
         wgraph({u1,u2})>0 ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
chooseU_4: pc = chooseU ⇒ (∀ u1 · (u1 ∈ Vertex\Q ⇒ (∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒
    dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
chooseU_5: pc = chooseU ⇒ v = 0 ∧ u = 0 ∧ neighbour = ∅
```

It needs to hold that all neighbours of visited vertices are in the set of connected vertices. For each vertex *u1* in the set of connected vertices which is not the start vertex s exists a vertex in C which is a neighbour of *u1* and furthermore there exists a visited vertex which is a neighbour of *u1* and is defined as the predecessor of *u1* and the distance of *u1* is the distance of its predecessor plus the length of the common edge. It also holds that the distance of each connected vertex is the smallest possible. The last invariant states that u, v and *neighbour* are in their initial state where no node is currently selected to be visited.

Those invariants are adapted for each state of the program counter and for the final state "Done" of the program counter, the corresponding invariants match the definition of the output conditions listed in 4.2.2.

The state transitions of the algorithm are defined by 10 `events`, which describe the changes of the variables. The `guards` of the `events` indicate if an `event` can be executed at a state of the model.

```
ChooseNextU:
   ANY
      u1
   WHERE
      grd1:  pc = chooseU
      grd2:  Q ∩ C ≠ ∅
      grd3:  u1 ∈ Q ∩ C ∧ (∀ u2 · (u2 ∈ Q ∩ C ⇒ dist(u1)≤dist(u2)))
   THEN
      act1:  u := u1
      act2:  Q := Q\{u1}
      act3:  neighbour := ∅
      act4:  pc := selectNeighbour
   END
```

The `event` "ChooseNextU" specifies the part of the algorithm, where the while loop is entered and the current visited node u is selected. The `event` can only be executed when the program counter is equal to "chooseU" and the set $Q \cap C$ is not the empty set. Then the node from the set $Q \cap C$ which has the smallest distance is selected as the next node which shall be visited. The full machine with all occurring events can be seen in Appendix B.6.3.

After the model is set up, the Rodin Platform automatically generates proof obligations for the partial correctness of the system, which can be seen in Appendix B.6.4. For this model 378 proof obligations are generated. 337 proof obligations are directly proved by the Rodin Platform and 12 proof obligations can be proved by interactive proofs. To complete the interactive proofs it was necessary to manually apply case distinctions or to select hypothesis which shall be investigated by the provers. The remaining 31 proofs,

which could not be completed, can be seen in Figure 4.11.

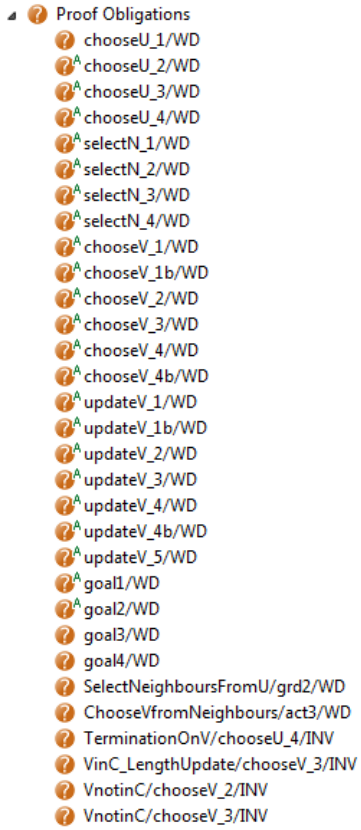


Figure 4.11.: Unproved proof obligations

27 of the unproved proof obligations are well-definedness proof obligations, which is indicated by the ending "WD". These proof obligations are generated whenever the expression "$wgraph(\{node1,\ node2\})$" occurs in a quantified formula or when an expression containing "$wgraph(\{node1,\ node2\})$" is assigned to a variable. In all of those proof obligations it needs to be shown that $wgraph(\{node1,\ node2\}) \in dom(wgraph)$. In the corresponding hypothesis it is known that both *node1* and *node2* are elements of the set *Vertex*. The set *Edge* contains all sets of vertices with exactly two elements, thus the set $\{node1,\ node2\}$ is an element of *Edge*. Since *wgraph* is defined as a total function from *Edge* to *Weight*, the set $\{node1,\ node2\}$ needs to be an element of the domain of *wgraph*.

Several attempts of redefining the set *Edge* and *Wgraph* were made, but the Rodin Platform is not able to complete the well-definedness proof obligations.

In the remaining proof obligations "VinC_LengthUpdate/chooseV_3/INV", "VnotinC/-chooseV_2/INV" and "VnotinC/chooseV_3/INV" up to two branches of the proof tree cannot be completed, because it cannot be shown that "$wgraph(\{node1,\ node2\}) = wgraph(\{node2,\ node1\})$".

An example of such an open branch for the proof obligation "VnotinC/chooseV_2/INV" can be seen in Figure 4.12, where $wgraph(\{u,\ v\}) > 0$ is included in the hypothesis and

it needs to be shown that $wgraph(\{v,\ u\}) > 0$.

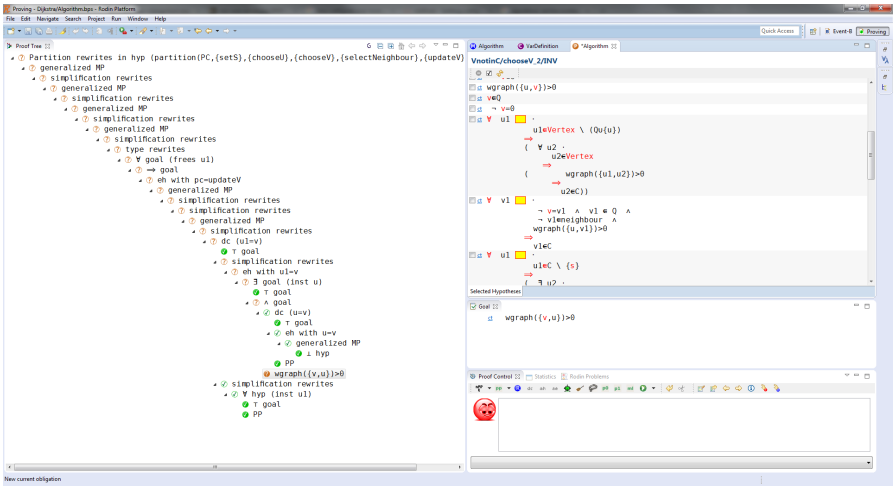


Figure 4.12.: Proof tree for “VnotinC/chooseV_2/INV”

In Figure 4.13 the proof of the proof obligation “TerminationOnV/chooseU_4/INV” can be seen. It is known that $wgraph(\{u1,\ u2\}) > 0$ and $wgraph(\{u2,\ u1\}) > 0 \Rightarrow dist(u1) \leq dist(u2) + wgraph(\{u2,\ u1\})$. Thus the goal can be simplified to

$$
\begin{aligned}
&dist(u2) \leq dist(u1) + wgraph(\{u1,\ u2\}) \Leftrightarrow \\
&dist(u2) \leq dist(u2) + wgraph(\{u2,\ u1\}) + wgraph(\{u1,\ u2\}) \Leftrightarrow \\
&\qquad 0 \leq 2 \cdot wgraph(\{u1,\ u2\})
\end{aligned}
$$

Since the range of $wgraph$ is the set Weight={0,...,maxWeight} the goal is satisfied.

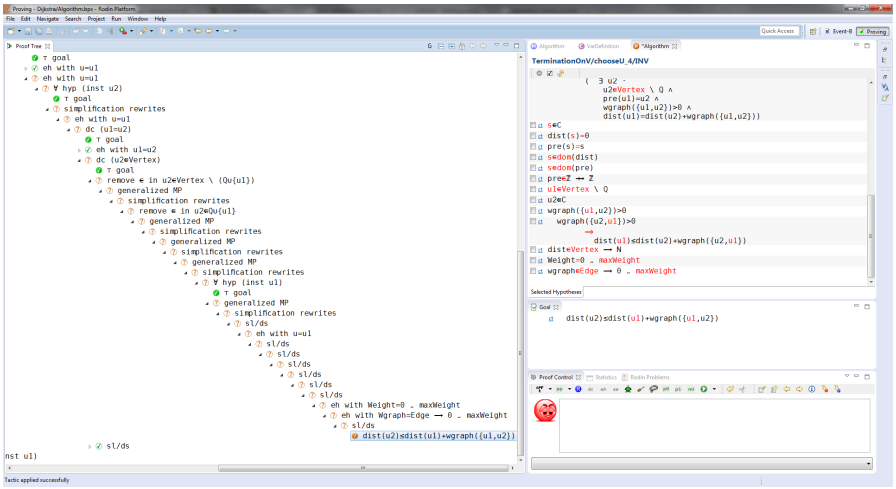


Figure 4.13.: Proof tree for “TerminationOnV/chooseU_4/INV”

Summarized it can be said that all proof obligations would be completed when it can be shown that a set of two vertices is an element of the domain of $wgraph$ and when it can be shown that “$wgraph(\{node1,\ node2\}) = wgraph(\{node2,\ node1\})$”. It was attempted to contact the developers for this problem, but so far no response was given.

4.9. Summary of the Results

A short summary of the results of the application of the software specification languages and tools for Dijkstra's Algorithm is given in Figure 4.14. The difficulty of specifying Dijkstra's Algorithm does not differ from the difficulty of specifying the DPLL method in each tool, which is listed in Section 3.10.

Language	Result
JML	Since the annotations of Dijkstra's Algorithm are very comprehensive, it is not possible to use predicates in the specification, because then ESC/Java2 returns that the verification conditions are too big. Thus it is necessary to state all annotations explicitly. In the first model, where the loop invariants are neglected, it is possible to check the model with ESC/Java2. In the second model, where the loop invariants are added to the specification, ESC/Java2 is not able to check correct annotations and returned warnings for the loop invariants.
TLA	The TLC Model Checker is able to model check the algorithm in the given scope.
Alloy	The Alloy Analyzer is able to model check the defined assertions and furthermore it is possible to gain visualizations of the run of the algorithm.
VDM	In the given scope Overture is able to model check the system. It is noticed that when an input variable of a function is a set, Overture also model checks all permutations of the set. The generated proof obligations for the model give again only a general overview what needs to be shown to verify the model.
Event-B	The easier proof obligations are automatically shown by the included provers. In more complex proof obligations, it is often necessary to manually add hypothesis or to interactively apply proving techniques like case distinction. While most of the verification could be performed (with interactive assistance), some minor proof branches remain open: for user defined data types the problem arises, that the prover cannot show well-definedness; furthermore the provers cannot complete proofs where a set is included in the goal and the hypothesis include the same statement of the goal for a partition of the set.

Figure 4.14.: Results of the Languages and Tools for Dijkstra's Algorithm

5. Conclusion

The main result of the thesis is that it is noticed that with each of the five used languages JML, TLA/PlusCal, VDM, Alloy and Event-B some success can be achieved when it is used for specifying and analyzing a mathematical algorithm, but each language comes with some advantages and disadvantages.

A big advantage of JML is that there is no need to give a formal model of the system, because the specification of a system is directly given by Java annotations in the source code. A further benefit of the syntax and semantics of JML is that it allows the use of recursion and it allows the introduction of loop-invariants. ESC/Java2 is able to detect common run-time errors, but when the specification is too expressive, the verification conditions may get too big, and ESC/Java2 is not able to reason about the specification. This error happened, when the loop invariants are added for Dijkstra's Algorithm. In Java KeY no interactive proofs are attempted and the default settings of the prover are used. It is noticed that all of the simpler specifications for helper functions can be verified by Java KeY, but it is observed, that when the automatically generated proof obligations for the methods get too extensive, the prover starts to cut the problem into numerous subproblems and does not apply proper rules anymore.

The usage of PlusCal is really helpful for defining mathematical algorithms in TLA, because PlusCal supports the usage and manipulation of mathematical objects like sets. Thus modelling an algorithm in TLA is very easy, because after the data types of the occurring variables are defined, the pseudo-code of an algorithm can be defined straight forward in PlusCal and then automatically be translated into a TLA model. The TLC Model Checker checks user-defined properties of the model in a finite scope. If the model checker finds an execution which violates properties of the algorithm, an error message is returned together with the complete execution path which yields the violation. Thus it is very easy to correct the specification. A disadvantage of PlusCal is that it does not support recursive algorithms.

Using the Alloy Analyzer for mathematical algorithms is complicated, because everything is based on relations. An algorithm is modelled with the help of ordered objects which describe the relation of steps in the execution of the algorithm. This means that for example a loop is specified in Alloy by describing the changes of the variables during an iteration of the loop. The Alloy Analyzer can either return instances of the model which satisfy the given properties or it can check user-defined assertions and return counter examples. The instances of the model are visualized by depicting all relations between the objects of the model. A big disadvantage of Alloy is that the scope for the instances has to be defined for each object. This may get very complex when during the execution of an algorithm new objects are created, because the number of these objects has to be included in the scope. If the scope is too small it is not guaranteed that the termination condition is fulfilled and the Alloy Analyzer may return instances, where

the run of an algorithm is not properly executed.

The syntax of VDM includes mathematical objects like sets and functions, which is very helpful for defining mathematical algorithms. Furthermore recursive function calls are supported. VDM provides the usage of user-defined data types, for which type invariants may be included. The pre- and postconditions of functions are directly added after the definition of the function. In VDM it is also possible to model system invariants, but in contrast to loop invariants, system invariants have to hold before and after each step in the execution of the algorithm. Thus the invariants of a VDM specification are not very strong. In Overture it is possible to define a `trace`, which imitates a run of an algorithm. In the definition of the `traces` the user-defined data types cannot be used and the data types of the input variables have to be stated explicitly by using basic data types. A big disadvantage is, that when an input variable of a function is a set, the scope of a `trace` cannot be too big, because also all permutations of sets are model checked. Overture also creates proof obligations of the model, but these proof obligations are not really strong, because no proper calculus is included, and some of them are only place holders. So the generated proof obligations only give a general overview what needs to be shown to verify the model.

The specification of an algorithm in Event-B, is performed with several `events` which describe the changes of the variables. `Guards` which are added to the events restrict which event can be executed at which state of the algorithm. The language of Event-B is very comprehensive and includes mathematical expressions like sets and functions. By adding program counters to the specification, it is possible to add invariants for each state of the program counter. The Rodin Platform automatically generates proof obligations for each event and the corresponding invariants. The easier proof obligations are automatically shown by the included provers. In more complex proof obligations it is often necessary to interactively add hypothesis or to manually apply proving techniques like case distinction. For user defined data types the problem may arise, that the prover cannot show well-definedness, furthermore it is noticed that the provers cannot complete proofs, where a set is included in the goal and the hypothesis include the same statement of the goal for a partition of the set. Moreover the included provers fail, when reasoning about expressive nested logical expressions is necessary. The concept of refinements can only be used in a very limited way for algorithms, because only the data types can be refined.

From the five used languages and tools, TLA, Alloy and VDM are used for model checking and JML and Event-B are used for verification. Although in VDM it is possible to generate proof obligations, it cannot be used for verifying a model, since the proof obligations are too weak.

Summarized it can be said, that Alloy is not the best language for modelling and analyzing mathematical algorithms, because the specification is very complicated and the scope for model checking needs to be defined with caution. Furthermore it is not traceable, which models are considered when an assertion is checked.

From the other two languages TLA and VDM no language can be favoured over the

other in general, because in both languages the modelling part is very easy due to the expressive language and in both languages model checking is comprehensive. A difference in the output of the model checkers is that in VDM all investigated models and their results are visualized in a list, whereas TLA only returns models which violate properties. In TLA a "print"-expression can be included in the specification, which returns the instances of the variables, to make model checking traceable. Since in TLA only the main function of an algorithm is defined as a function, and all helper functions are defined as `macros`, the execution of the algorithm does not have to be defined explicitly, as it needs to be done in VDM. Thus setting up the model is easier in TLA, than in VDM. In VDM pre- and postconditions of functions can directly be added in the specification of the function, whereas in TLA pre- and postconditions have to be defined as invariants for the state of the generated program counter. Since a state of the program counter is always created for the beginning of a loop, it is possible to add loop invariants to TLA, whereas in VDM only global invariants can be added.

When a mathematical algorithm shall be verified Event-B is favoured over JML, because Event-B supports the usage of mathematical expressions, whereas JML is based on Java. Furthermore JML and its tool Java KeY can only be used to verify very easy algorithms, which do not have a very complicated specification. When the specification gets too expressive, Java KeY is not able to complete the proofs.

The main problem of verifying models in Event-B is that when the invariants are too complicated and include nested logical expressions the provers cannot complete proofs, although the specification is correct. Furthermore the included provers often need manual support by adding necessary hypothesis or by applying proof techniques to further the proof.

Among the tools considered Event-B is recommended for verification. For model checking both TLA/PlusCal and VDM are advocated in general; if a recursive algorithm shall be model checked, VDM is recommended.

Bibliography

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, USA, 2010.

[3] V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer-Verlag, London, United Kingdom, 1998.

[4] Alloy. `http://alloy.mit.edu/alloy/`. accessed: January 26, 2016.

[5] Alloy applications. `http://alloy.mit.edu/alloy/applications.html`. accessed: January 26, 2016.

[6] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer Publishing Company, Incorporated, 3rd edition, 2012.

[7] J. Bicarregui and B. Ritchie. Invariants, Frames and Postconditions: A Comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, 1995.

[8] A. Biere and M. Seidl. Propositional Logic: Evaluating the Formulas, Lecture notes in VL Logik, 2014. Institute for Formal Models and Verification, Johannes Kepler University Linz.

[9] D. Bjorner. 32 Years of VDM: From Earliest Days via Adolescence to Maturity. Presented at the IPSJ/SIGSE Software Engineering Symposium, Tokyo, Japan, 2006.

[10] D. Bjorner and M. Henson. *Logics of Specification Languages*. Springer-Verlag, Berlin-Heidelberg, Germany, 2008.

[11] M. Carro. A Short Introduction to Formal Methods. Technical report, Technical University of Madrid, Madrid, Spain, 2005.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, USA, 2009.

[13] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[14] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[15] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[16] Event-B and the Rodin Platform. http://www.event-b.org/. accessed: January 26, 2016.

[17] Event-B Code Geneartion Activity. http://wiki.event-b.org/index.php/Code_Generation_Activity. accessed: January 26, 2016.

[18] Event-B Plugin. http://wiki.event-b.org/index.php/Category:Plugin. accessed: January 26, 2016.

[19] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, USA, 2005.

[20] H. Gredler and W. Goralski. *The Complete IS-IS Routing Protocol*. Springer London, 2005.

[21] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[22] A. Huq and N.T. Ramachandran. VDM Specification of an Algorithm for Graph Decomposition. *Journal of Digital Information Management*, 5(1):347–353, 2007.

[23] Industrial Projects and Research Projects using Event-B. http://wiki.event-b.org/index.php/Industrial_Projects. accessed: January 26, 2016.

[24] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, USA, 2012.

[25] M. Jastram. *Rodin User's Handbook*. CreateSpace Independent Publishing Platform, North Charleston, USA, 2014.

[26] KeY Project: Integrated Deductive Software Design. http://www.key-project.org/. accessed: January 26, 2016.

[27] KindSoftware: Esc/Java2. http://kindsoftware.com/products/opensource/ESCJava2/docs.html. accessed: January 26, 2016.

[28] V. Klasen. Verifying Dijkstra's Algorithm with KeY. Master's thesis, University Koblenz-Landau, Germany, 2010.

[29] L. Lamport. Specifying Concurrent Systems with TLA+. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, pages 183–247. IOS Press, Amsterdam, The Netherlands, 1999.

[30] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2002.

[31] L. Lamport. The PlusCal Algorithm Language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 36–60. Springer-Verlag, Berlin, Heidelberg, 2009.

[32] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[33] P. G. Larsen, K. Lausdahl, P. Tran-Jørgensen, J. Coleman, S. Wolff, and L. D. Cuoto. Overture VDM-10 Tool Support: User Guide. Technical report, Aarhus University, Department of Engineering, Arhus, Denmark, 2015.

[34] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[35] G. T. Leavens and Y. Cheon. Design by Contract with JML. Technical report, Dept. of Computer Science, Iowa State University, Ames, USA, Dept. of Computer Science, University of Texas at El Paso, El Paso, USA, 2006.

[36] F. Maric. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[37] F. Maric and P. Janicic. Formal Correctness Proof for DPLL Procedure. *Informatica*, 21(1):57–78, 2010.

[38] S. Merz. The Specification Language TLA+. In D. Bjorner and M. C. Henson, editors, *Logics of Specification Languages*, pages 401–451. Springer-Verlag, Berlin-Heidelberg, Germany, 2008.

[39] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[40] J. Moy. *OSPF Complete Implementation*. Pearson Education, 2000.

[41] Overture Tool. `http://overturetool.org/`. accessed: January 26, 2016.

[42] C. Rizkallah. A Simpl Shortest Path Checker Verification. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2014.

[43] A. Schrijver. On the History of the Shortest Path Problem. *Documenta Mathematica*, Extra Volume(ISMP):155–167, 2012.

[44] Specifications used in the Thesis. `http://thesis.ritirc.at`. accessed: January 26, 2016.

[45] TLA+ Proof System. `https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/The_example.html`. accessed: January 26, 2016.

[46] The Java Modeling Language (JML). `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`. accessed: January 26, 2016.

[47] The TLA Toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`. accessed: January 26, 2016.

[48] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[49] J. van Eijck. `http://homepages.cwi.nl/~jve/`. personal communication: December 10, 2015.

[50] E. Wong, M. Herrmann, and O. Tayeb. A Guide To Alloy. Technical report, Imperial College London, Department of Computing, London, United Kingdom, 2008.

[51] P. Zave. A Practical Comparison of Alloy and Spin. *Formal Aspects of Computing*, 27(2):239–253, 2015.

A. Model Listings for DPLL Algorithm

A.1. Java

In this Appendix the Java prototype for the DPLL Algorithm from Section 3.4 is listed.

```
1  public class DPLL {
2
3  // Calls the DPLL method iteratively
4      public static boolean DPLLiter( int[][]formula, int n){
5          int[][][] stack = new int[n][][];
6          int inStack = 0;
7          int[][]copy_formula = formula;
8
9          while(true){
10             if(isEmptyFormula(copy_formula,n)) return true;
11
12             else if(containsEmptyClause(copy_formula,n)){
13                 if(inStack==0) return false;
14                 inStack--;
15                 copy_formula = stack[inStack];
16
17             }
18             else{
19                 int val = copy_formula[0][0];
20                 stack[inStack]=substitute(copy_formula,val,false,n);
21                 inStack++;
22                 copy_formula = substitute(copy_formula,val,true,n);
23             }
24         }
25     }
26
27 // Calls the DPLL method recursively
28     public static boolean DPLLrec(int[][]formula, int n){
29         if(isEmptyFormula(formula, n)) return true;
30
31         else if(containsEmptyClause(formula ,n)) return false;
32
33         else{
34             int val = formula[0][0];
35             if(DPLLrec(substitute(formula,val,true ,n),n)==true)
36                 return true;
37             else{ return(DPLLrec(substitute(formula,val,false ,n),n));}
38         }
39     }
40
41 // Substitutes the literal lit in the formula by the value val
42     public static int[][] substitute (int[][]formula, int lit, boolean val, int n){
43         int l = lit;
44         if(val== false) l=-l;
45
46         int[][]copy_formula = deleteClause(formula, l, n);
47         for(int i= 0; i<copy_formula.length;i++){
48             copy_formula[i]=deleteLit(copy_formula[i], -l, n);
49         }
50         return copy_formula;
51     }
52
53 // Deletes all clauses containing lit in the formula
```

```
54      public static int[][] deleteClause(int[][] formula, int lit, int n ){
55          int[][] copy_formula = new int[formula.length][];
56          int j = 0;
57          int i = 0;
58
59          while(i<formula.length){
60              int pos = search(formula[i], lit,n);
61
62              if(pos == -1){
63                  copy_formula[j]=formula[i];
64                  j++;
65              }
66              i++;
67          }
68          return CutFormula(copy_formula, j, n);
69      }
70
71  // Deletes a possible occurence of lit in the clause
72      public static int[] deleteLit(int[] clause, int lit, int n ){
73          int pos = search(clause,lit, n);
74
75          if(pos==-1) return clause;
76          else return CopyClause(clause, pos, n);
77      }
78
79  // Copies a clause except of the given position pos
80      public static int[] CopyClause(int []clause, int pos, int n){
81          int[] copy_clause = new int[clause.length-1];
82          int i;
83          for (i=0; i<pos; i++) copy_clause[i] = clause[i];
84          for (i=pos; i<clause.length-1; i++) copy_clause[i] = clause[i+1];
85
86          return copy_clause;
87      }
88
89  // Cuts a formula after j clauses
90      public static int[][] CutFormula(int [][]formula, int j, int n){
91          int[][] copy_formula = new int[j][];
92          int i = 0;
93          while(i<j){
94              copy_formula[i]= formula[i];
95              i++;
96          }
97          return copy_formula;
98      }
99
100 // Searches for a literal in a clause
101     public static int search(int []clause, int lit, int n){
102         int pos = -1;
103         int i =0;
104         while(pos==-1 && i<clause.length){
105             if(clause[i]==lit){
106                 pos = i;
107             }
108             i++;
109         }
110         return pos;
111     }
112
113  // Searches if a formula contains an empty clause
114     public static boolean containsEmptyClause(int[][]formula, int n){
115         int i = 0;
116         boolean val= false;
117         while( i<formula.length && val==false){
118             if(isEmptyClause(formula[i],n)) val=true;
119             else i++;
120         }
```

```
121         return val;
122     }
123 
124 // Checks if the formula is empty
125     public static boolean isEmptyFormula(int[][]formula, int n){
126         return formula.length==0;
127     }
128 
129 // Checks if a clause is empty
130     public static boolean isEmptyClause(int[] clause, int n){
131         return clause.length==0;
132     }
133 }
```

A.2. JML

A.2.1. Version 1

JML Specification

The first version of the JML specification of the DPLL Algorithm from Section 3.5.1 is shown.

```
1  public class DPLL {
2  
3  /*@ public normal_behavior
4      @ requires formula != null;
5      @ requires formula.length > 0;
6      @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
7      @ requires n>=0;
8  @*/
9  public static boolean DPLLiter(int[][]formula, int n){
10         int[][][] stack = new int[n][][];
11         int inStack = 0;
12         int[][]copy_formula = formula;
13 
14         while(true){
15             if(isEmptyFormula(copy_formula,n)) return true;
16 
17             else if(containsEmptyClause(copy_formula,n)){
18                 if(inStack==0) return false;
19                 inStack--;
20                 copy_formula = stack[inStack];
21 
22             }
23             else{
24                 int val = copy_formula[0][0];
25                 stack[inStack]=substitute(copy_formula,val,false,n);
26                 inStack++;
27                 copy_formula = substitute(copy_formula,val,true,n);
28             }
29         }
30     }
31 
32 /*@ public normal_behavior
33     @ requires formula != null;
34     @ requires formula.length > 0;
35     @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
36     @ requires n > 0;
37 @*/
38     public static boolean DPLLrec(int[][]formula, int n){
39         if(isEmptyFormula(formula,n)) return true;
40 
41         else if(containsEmptyClause(formula,n)) return false;
```

```
42
43          else{
44              int val = formula[0][0];
45              if(DPLLrec(substitute(formula,val,true,n),n)==true)
46                  return true;
47              else{ return(DPLLrec(substitute(formula,val,false,n),n));}
48          }
49      }
50
51  /*@ public normal_behavior
52      @ requires formula != null;
53      @ requires formula.length > 0;
54      @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
55
56      @ ensures \result !=null;
57      @ ensures \result.length > 0;
58      @ ensures (\forall int i; 0<=i && i< \result.length; \result[i]!=null);
59  @*/
60      public static int[][] substitute (int[][]formula, int lit, boolean val, int n){
61          int l = lit;
62          if(val== false) l=-l;
63
64          int[][]copy_formula = deleteClause(formula, l,n);
65          for(int i= 0; i<copy_formula.length;i++){
66              copy_formula[i]=deleteLit(copy_formula[i], -l,n);
67          }
68
69          return copy_formula;
70      }
71
72  /*@ public normal_behavior
73      @ requires formula != null;
74      @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
75
76      @ ensures \result !=null;
77      @ ensures (\forall int i; 0<=i && i< \result.length; \result[i]!=null);
78  @*/
79      public static int[][] deleteClause(int[][] formula, int lit, int n ){
80          int[][] copy_formula = new int[formula.length][];
81          int j = 0;
82          int i = 0;
83          while(i<formula.length){
84              int pos = search(formula[i], lit,n);
85
86              if(pos == -1){
87                  copy_formula[j]=formula[i];
88                  j++;
89              }
90              i++;
91          }
92
93          return CutFormula(copy_formula, j,n);
94      }
95
96  /*@ public normal_behavior
97      @ requires clause != null;
98  @*/
99      public static int[] deleteLit(int[] clause, int lit, int n){
100         int pos = search(clause,lit,n);
101
102         if(pos==-1) return clause;
103         else return CopyClause(clause,pos,n);
104     }
105
106 /*@ public normal_behavior
107     @ requires clause != null;
108     @ requires 0<=pos && pos<clause.length;
```

```
109 @*/
110 public static int[] CopyClause(int []clause, int pos, int n){
111         int[] copy_clause = new int[clause.length-1];
112         int i;
113         for (i=0; i<pos; i++) copy_clause[i] = clause[i];
114         for (i=pos; i<clause.length-1; i++) copy_clause[i] = clause[i+1];
115         return copy_clause;
116     }
117 
118 /*@ public normal_behavior
119     @ requires formula!=null;
120     @ requires 0<=j && j<=formula.length;
121 @*/
122     public static int[][] CutFormula(int [][]formula, int j,int n){
123         int[][] copy_formula = new int[j][];
124         int i = 0;
125 
126         while(i<j){
127             copy_formula[i]= formula[i];
128             i++;
129         }
130         return copy_formula;
131     }
132 
133 /*@ public normal_behavior
134     @ requires clause!=null;
135     @ ensures -1 <= \result && \result < clause.length;
136 @*/
137     public static int search(int []clause, int lit, int n){
138         int pos = -1;
139         int i =0;
140         while(pos==-1 && i<clause.length){
141             if(clause[i]==lit){
142                 pos = i;
143             }
144             else i++;
145         }
146         return pos;
147     }
148 
149 /*@ public normal_behavior
150     @ requires formula != null;
151     @ requires (\forall int i; 0<=i && i< formula.length; formula[i]!=null);
152 @*/
153     public static boolean containsEmptyClause(int[][]formula, int n){
154         int i = 0;
155         boolean val= false;
156 
157         while( i<formula.length && val==false){
158             if(isEmptyClause(formula[i],n)) val=true;
159             else i++;
160         }
161         return val;
162     }
163 
164 /*@ public normal_behavior
165     @ requires formula != null;
166 @*/
167     public static boolean isEmptyFormula(int[][]formula, int n){
168         return formula.length==0;
169     }
170 
171 /*@ public normal_behavior
172     @ requires clause != null;
173 @*/
174     public static boolean isEmptyClause(int[] clause, int n){
175         return clause.length==0;
```

```
176     }
177 }
```

Output of ESC/Java2

The output of ESC/Java2 for the first version of the JML specification from Section 3.5.1 is shown.

```
1   Dpll ...
2     Prover started:0.0080 s 10382488 bytes
3       [0.416 s 10610224 bytes]
4
5   Dpll: DPLLiter(int[][], int) ...
6   ------------------------------------------------------------------------
7   ./DPLL1.java:27: Warning: Array index possibly too large (IndexTooBig)
8                               int val = copy_formula[0][0];
9                                                      ^
10  Execution trace information:
11      Reached top of loop after 0 iterations in "./DPLL1.java", line 17, col 2.
12      Executed else branch in "./DPLL1.java", line 20, col 8.
13      Executed else branch in "./DPLL1.java", line 26, col 7.
14
15  ------------------------------------------------------------------------
16  ./DPLL1.java:28: Warning: Array index possibly too large (IndexTooBig)
17                              stack[inStack]=substitute(copy_formula ...
18                                  ^
19  Execution trace information:
20      Reached top of loop after 0 iterations in "./DPLL1.java", line 17, col 2.
21      Executed else branch in "./DPLL1.java", line 20, col 8.
22      Executed else branch in "./DPLL1.java", line 26, col 7.
23
24  ------------------------------------------------------------------------
25      [0.141 s 10835512 bytes] failed
26
27  Dpll: DPLLrec(int[][], int) ...
28  ------------------------------------------------------------------------
29  ./DPLL1.java:50: Warning: Array index possibly too large (IndexTooBig)
30                       int val = formula[0][0];
31                                         ^
32  Execution trace information:
33      Executed else branch in "./DPLL1.java", line 47, col 7.
34      Executed else branch in "./DPLL1.java", line 49, col 6.
35
36  ------------------------------------------------------------------------
37      [0.106 s 11745336 bytes] failed
38
39  Dpll: substitute(int[][], int, boolean, int) ...
40  ------------------------------------------------------------------------
41  ./DPLL1.java:78: Warning: Postcondition possibly not established (Post)
42          }
43          ^
44  Associated declaration is "./DPLL1.java", line 64, col 3:
45          @ ensures \result.length > 0;
46            ^
47  Execution trace information:
48      Executed then branch in "./DPLL1.java", line 70, col 18.
49      Reached top of loop after 0 iterations in "./DPLL1.java", line 73, col 2.
50      Executed return in "./DPLL1.java", line 77, col 2.
51
52  ------------------------------------------------------------------------
53  ./DPLL1.java:78: Warning: Postcondition possibly not established (Post)
54          }
55          ^
56  Associated declaration is "./DPLL1.java", line 65, col 3:
57          @ ensures (\forall int i; 0<=i && i< \result.length; \result[i ...
```

```
58            ^
59  Execution trace information:
60      Executed then branch in "./DPLL1.java", line 70, col 18.
61      Reached top of loop after 0 iterations in "./DPLL1.java", line 73, col 2.
62      Reached top of loop after 1 iteration in "./DPLL1.java", line 73, col 2.
63      Executed return in "./DPLL1.java", line 77, col 2.
64
65  ------------------------------------------------------------------------
66      [0.053 s 10795288 bytes] failed
67
68  Dpll: deleteClause(int[][], int, int) ...
69  ------------------------------------------------------------------------
70  ./DPLL1.java:105: Warning: Postcondition possibly not established (Post)
71          }
72          ^
73  Associated declaration is "./DPLL1.java", line 86, col 3:
74          @ ensures \result !=null;
75            ^
76  Execution trace information:
77      Reached top of loop after 0 iterations in "./DPLL1.java", line 94, col 2.
78      Executed return in "./DPLL1.java", line 104, col 2.
79
80  ------------------------------------------------------------------------
81  ./DPLL1.java:105: Warning: Postcondition possibly not established (Post)
82          }
83          ^
84  Associated declaration is "./DPLL1.java", line 87, col 3:
85          @ ensures (\forall int i; 0<=i && i< \result.length; \result[i ...
86            ^
87  Execution trace information:
88      Reached top of loop after 0 iterations in "./DPLL1.java", line 94, col 2.
89      Executed return in "./DPLL1.java", line 104, col 2.
90
91  ------------------------------------------------------------------------
92      [0.035 s 10931696 bytes] failed
93
94  Dpll: deleteLit(int[], int, int) ...
95      [0.012 s 11368160 bytes] passed
96
97  Dpll: CopyClause(int[], int, int) ...
98      [0.012 s 10830176 bytes] passed
99
100 Dpll: CutFormula(int[][], int, int) ...
101     [0.014 s 11183792 bytes] passed
102
103 Dpll: search(int[], int, int) ...
104     [0.0090 s 11625440 bytes] passed
105
106 Dpll: containsEmptyClause(int[][], int) ...
107     [0.012 s 11060664 bytes] passed
108
109 Dpll: isEmptyFormula(int[][], int) ...
110     [0.0020 s 11166232 bytes] passed
111
112 Dpll: isEmptyClause(int[], int) ...
113     [0.0060 s 11270648 bytes] passed
114
115 Dpll: Dpll() ...
116     [0.0040 s 11432576 bytes] passed
117   [0.823 s 11433456 bytes total]
118 7 warnings
```

A.2.2. Version 2

JML Specification

The second version of the JML specification of the DPLL Algorithm from Section 3.5.2 is listed in this Appendix.

```
1  public class DPLL {
2
3  /*@ public normal_behavior
4      @ requires validformula(formula,n);
5      @ requires n>=0;
6
7      @ assignable \nothing;
8
9      @ ensures \result == satisfiable(formula,n);
10 @*/
11     public static boolean DPLLiter(int[][]formula, int n){
12         int[][][] stack = new int[n][][];
13         int inStack = 0;
14         int[][]copy_formula = formula;
15
16         while(true){
17             if(isEmptyFormula(copy_formula,n)) return true;
18
19             else if(containsEmptyClause(copy_formula,n)){
20                 if(inStack==0) return false;
21                 inStack--;
22                 copy_formula = stack[inStack];
23
24             }
25             else{
26                 int val = copy_formula[0][0];
27                 stack[inStack]=substitute(copy_formula,val,false,n);
28                 inStack++;
29                 copy_formula = substitute(copy_formula,val,true,n);
30
31             }
32         }
33     }
34
35 /*@ public normal_behavior
36     @ requires validformula(formula,n);
37     @ requires n>=0;
38
39     @ assignable \nothing;
40
41     @ ensures \result == satisfiable(formula,n);
42 @*/
43     public static boolean DPLLrec(int[][]formula, int n){
44         if(isEmptyFormula(formula,n)) return true;
45
46         else if(containsEmptyClause(formula,n)) return false;
47
48         else{
49             int val = formula[0][0];
50             if(DPLLrec(substitute(formula,val,true,n),n)==true)
51                 return true;
52             else{ return DPLLrec(substitute(formula,val,false,n),n);}
53
54         }
55     }
56
57 /*@ public normal_behavior
58     @ requires validformula(formula,n);
59
60     @ assignable \nothing;
```

```
61
62      @ ensures validformula(\result,n);
63      @ ensures (\forall int[] c; validclause(c,n); ClauseInFormula(c, \result) <==>
64      @   ((ClauseInFormula(c,formula) && !LitInClause(lit, c) && !LitInClause(-lit, c))) ||
65      @    (\exists int[]c1; validclause(c1,n) && ClauseInFormula(c1,formula);
66      @         c == remove(val, lit, c1,n)));
67  @*/
68      public static int[][] substitute (int[][]formula, int lit, boolean val, int n){
69          int l = lit;
70          if(val== false) l=-l;
71
72          int[][]copy_formula = deleteClause(formula, l,n);
73          for(int i= 0; i<copy_formula.length;i++){
74              copy_formula[i]=deleteLit(copy_formula[i], -l,n);
75          }
76          return copy_formula;
77      }
78
79  /*@ public normal_behavior
80      @ requires validformula(formula,n);
81
82      @ assignable \nothing;
83
84      @ ensures validformula(\result,n);
85      @ ensures (\forall int[] c; validclause(c,n);
86      @    ClauseInFormula(c, \result) <==> (ClauseInFormula(c,formula) && !LitInClause(lit, c)));
87  @*/
88      public static int[][] deleteClause(int[][] formula, int lit, int n ){
89          int[][] copy_formula = new int[formula.length][];
90          int j = 0;
91          int i = 0;
92          while(i<formula.length){
93              int pos = search(formula[i], lit,n);
94
95              if(pos == -1){
96                  copy_formula[j]=formula[i];
97                  j++;
98              }
99              i++;
100         }
101         return CutFormula(copy_formula, j,n);
102     }
103
104 /*@ public normal_behavior
105     @ requires validclause(clause,n);
106
107     @ assignable \nothing;
108
109     @ ensures \result!=null;
110     @ ensures (\forall int l; validliteral(l,n); LitInClause(l, \result)
111     @       <==> (l!=lit && LitInClause(l, clause)));
112 @*/
113     public static int[] deleteLit(int[] clause, int lit, int n){
114         int pos = search(clause,lit,n);
115
116         if(pos==-1) return clause;
117         else return CopyClause(clause,pos,n);
118     }
119
120 /*@ public normal_behavior
121     @ requires validclause(clause,n);
122     @ requires 0<=pos && pos<clause.length;
123
124     @ assignable \nothing;
125
126     @ ensures validclause(\result,n);
127     @ ensures (\forall int i; 0<=i && i<pos; clause[i]==\result[i]);
```

```
128     @ ensures (\forall int i; pos<i && i<clause.length; clause[i]==\result[i-1]);
129 @*/
130     public static int[] CopyClause(int []clause, int pos, int n){
131         int[] copy_clause = new int[clause.length-1];
132         int i;
133         for (i=0; i<pos; i++) copy_clause[i] = clause[i];
134         for (i=pos; i<clause.length-1; i++) copy_clause[i] = clause[i+1];
135         return copy_clause;
136     }
137 
138 /*@ public normal_behavior
139     @ requires validFormulaUpToJ(formula,n,j);
140     @ requires 0<=j && j<=formula.length;
141 
142     @ assignable \nothing;
143 
144     @ ensures validformula(formula,n);
145     @ ensures \result.length == j;
146     @ ensures (\forall int i; 0<=i && i<j; formula[i]==\result[i]);
147 
148 @*/
149     public static int[][] CutFormula(int [][]formula, int j,int n){
150         int[][] copy_formula = new int[j][];
151         int i = 0;
152 
153         while(i<j){
154             copy_formula[i]= formula[i];
155             i++;
156         }
157         return copy_formula;
158     }
159 
160 /*@ public normal_behavior
161     @ requires validclause(clause,n);
162 
163     @ assignable \nothing;
164 
165     @ ensures -1 <= \result && \result < clause.length;
166     @ ensures \result ==-1 <==> (\forall int i; 0<=i && i<clause.length; clause[i]!=lit);
167 @*/
168     public static int search(int []clause, int lit, int n){
169         int pos = -1;
170         int i =0;
171         while(pos==-1 && i<clause.length){
172             if(clause[i]==lit){
173                 pos = i;
174             }
175             else i++;
176         }
177         return pos;
178     }
179 
180 /*@ public normal_behavior
181     @ requires validformula(formula,n);
182 
183     @ assignable \nothing;
184 
185     @ ensures (\result <==> (\exists int i; 0<=i && i<formula.length;
186     @       isEmptyClause(formula[i],n)));
187 @*/
188     public static boolean containsEmptyClause(int[][]formula, int n){
189         int i = 0;
190         boolean val= false;
191 
192         while( i<formula.length && val==false){
193             if(isEmptyClause(formula[i],n)) val=true;
194             else i++;
```

```
195         }
196         return val;
197     }
198 
199 /*@ public normal_behavior
200     @ requires formula != null;
201 
202     @ assignable \nothing;
203 
204     @ ensures (\result <==> (formula.length == 0));
205 @*/
206     public /*@ pure @*/ static boolean isEmptyFormula(int[][]formula, int n){
207         return formula.length==0;
208     }
209 
210 /*@ public normal_behavior
211     @ requires validclause(clause,n);
212 
213     @ assignable \nothing;
214 
215     @ ensures (\result <==> (clause.length==0));
216 @*/
217     public /*@ pure @*/ static boolean isEmptyClause(int[] clause, int n){
218         return clause.length==0;
219     }
220 
221 
222 // Input condition
223 /*@
224 
225   @ ensures \result <==> ((1<=l && l<=n) || (-n<=l && l<=-1));
226   @ public static pure model boolean validliteral(int l, int n);
227 
228   @ ensures \result <==> !(LitInClause(l,c) && LitInClause(-l,c));
229   @ public static pure model boolean consistent(int l, int[] c);
230 
231   @ ensures \result <==>
232   @     (\forall int i; 0<=i && i<c.length; validliteral(c[i],n)) &&
233   @     (\forall int j; 0<=j && j<=n; consistent(j,c)) &&
234   @     c!=null;
235   @ public static pure model boolean validclause(int[] c,int n);
236 
237   @ ensures \result <==> (\forall int i; 0<=i && i<f.length; validclause(f[i],n)) && f!=null;
238   @ public static pure model boolean validformula(int[][] f,int n);
239 
240   @ ensures \result <==> (\forall int i; 0<=i && i<j; validclause(f[i],n)) && f!=null;
241   @ public static pure model boolean validFormulaUpToJ(int[][] f,int n, int j);
242 
243 @*/
244 
245 // Output condition
246 /*@
247   @ ensures \result <==>
248   @     (\forall int i; 0<=i && i<v.length; validliteral(v[i],n)) &&
249   @     (\forall int j; 0<=j && j<=n; consistent(j,v));
250   @ public static pure model boolean valuation(int []v, int n);
251 
252   @ ensures \result <==> (valuation(v,n) && LitInClause(l,v));
253   @ public static pure model boolean ValuationSatLit(int l, int[]v, int n);
254 
255   @ ensures \result <==>
256   @    (\exists int l; validliteral(l,n)&&LitInClause(l,c); ValuationSatLit(l,v,n));
257   @ public static pure model boolean ValuationSatClause(int[] c, int[]v, int n);
258 
259   @ ensures \result <==>
260   @     (\forall int[] c; validclause(c,n)&&ClauseInFormula(c,f); ValuationSatClause(c,v,n));
261   @ public static pure model boolean ValuationSatFormula(int[][] f, int[]v, int n);
```

```
262
263   @ ensures \result <==> (\exists int[] v; valuation(v,n); ValuationSatFormula(f,v,n));
264   @ public static pure model boolean satisfiable(int[][] f, int n);
265 @*/
266
267 // Helper functions
268 /*@
269   @ ensures \result <==> (\exists int i; 0<=i && i<c.length; c[i]==l);
270   @ public static pure model boolean LitInClause(int l, int[] c);
271
272   @ ensures \result <==> (\exists int i; 0<=i && i<f.length; LitInClause(l,f[i]));
273   @ public static pure model boolean LitInFormula(int l, int[][] f);
274
275   @ ensures \result <==>
276   @     (\exists int i; 0<=i && i<f.length;
277   @           (\forall int j; 0<=j && j<c.length;
278   @                LitInClause(c[j],f[i])) && c.length == f[i].length);
279   @ public static pure model boolean ClauseInFormula(int [] c, int[][] f);
280
281   @ ensures (\forall int l; validliteral(l,n); LitInClause(l,\result) <==>
282   @           (LitInClause(l,c) && ((val==false && l!=lit) ||
283   @           (val==true && l!=-lit))));
284   @ public static pure model int[] remove(boolean val, int lit, int[]c, int n);
285 @*/
286
287 }
```

Output of ESC/Java2

The output of ESC/Java2 for the second version of the JML specification from Section 3.5.2 is shown.

```
1
2  Dpll ...
3    Prover started:0.0080 s 11231608 bytes
4      [0.515 s 10601856 bytes]
5
6  Dpll: DPLLiter(int[][], int) ...
7  ------------------------------------------------------------------------
8  ./DPLL2.java:29: Warning: Array index possibly too large (IndexTooBig)
9                               stack[inStack]=substitute(copy_formula ...
10                                   ^
11 Execution trace information:
12     Reached top of loop after 0 iterations in "./DPLL2.java", line 18, col 2.
13     Executed else branch in "./DPLL2.java", line 21, col 8.
14     Executed else branch in "./DPLL2.java", line 27, col 7.
15
16 ------------------------------------------------------------------------
17 ./DPLL2.java:35: Warning: Postcondition possibly not established (Post)
18         }
19         ^
20 Associated declaration is "./DPLL2.java", line 10, col 3:
21         @ ensures \result == satisfiable(formula,n);
22           ^
23 Execution trace information:
24     Reached top of loop after 0 iterations in "./DPLL2.java", line 18, col 2.
25     Executed then branch in "./DPLL2.java", line 19, col 38.
26     Executed return in "./DPLL2.java", line 19, col 38.
27
28 ------------------------------------------------------------------------
29     [0.396 s 11566872 bytes] failed
30
31 Dpll: DPLLrec(int[][], int) ...
32     [0.29 s 12535056 bytes] passed
33
```

```
Dpll: substitute(int[][], int, boolean, int) ...
------------------------------------------------------------------------
./DPLL2.java:81: Warning: Possible violation of modifies clause (Modifies)
                      copy_formula[i]=deleteLit(copy_formula[i], -l, ...
                      ^
Associated declaration is "./DPLL2.java", line 66, col 3:
       @ assignable \nothing;
         ^
Execution trace information:
    Executed then branch in "./DPLL2.java", line 77, col 18.
    Reached top of loop after 0 iterations in "./DPLL2.java", line 80, col 2.

------------------------------------------------------------------------
./DPLL2.java:85: Warning: Postcondition possibly not established (Post)
       }
       ^
Associated declaration is "./DPLL2.java", line 69, col 3:
       @ ensures (\forall int[] c; validclause(c,n); ClauseInFormula( ...
         ^
Execution trace information:
    Executed then branch in "./DPLL2.java", line 77, col 18.
    Reached top of loop after 0 iterations in "./DPLL2.java", line 80, col 2.
    Reached top of loop after 1 iteration in "./DPLL2.java", line 80, col 2.
    Executed return in "./DPLL2.java", line 84, col 2.

------------------------------------------------------------------------
    [2.032 s 15565176 bytes] failed

Dpll: deleteClause(int[][], int, int) ...
------------------------------------------------------------------------
./DPLL2.java:112: Warning: Postcondition possibly not established (Post)
       }
       ^
Associated declaration is "./DPLL2.java", line 95, col 3:
       @ ensures (\forall int[] c; validclause(c,n); ClauseInFormula( ...
         ^
Execution trace information:
    Reached top of loop after 0 iterations in "./DPLL2.java", line 101, col 2.
    Executed then branch in "./DPLL2.java", line 104, col 16.
    Reached top of loop after 1 iteration in "./DPLL2.java", line 101, col 2.
    Executed return in "./DPLL2.java", line 111, col 2.

------------------------------------------------------------------------
    [0.15 s 15986088 bytes] failed

Dpll: deleteLit(int[], int, int) ...
------------------------------------------------------------------------
./DPLL2.java:130: Warning: Postcondition possibly not established (Post)
       }
       ^
Associated declaration is "./DPLL2.java", line 122, col 3:
       @ ensures (\forall int l; validliteral(l,n); LitInClause(l, \r ...
         ^
Execution trace information:
    Executed else branch in "./DPLL2.java", line 128, col 7.
    Executed return in "./DPLL2.java", line 128, col 7.

------------------------------------------------------------------------
    [0.038 s 16548648 bytes] failed

Dpll: CopyClause(int[], int, int) ...
    [0.031 s 16522104 bytes] passed

Dpll: CutFormula(int[][], int, int) ...
    [0.076 s 16523440 bytes] passed

Dpll: search(int[], int, int) ...
```

```
    [0.012 s 15976344 bytes] passed

Dpll: containsEmptyClause(int[][], int) ...
    [0.035 s 16270064 bytes] passed

Dpll: isEmptyFormula(int[][], int) ...
    [0.0030 s 16385000 bytes] passed

Dpll: isEmptyClause(int[], int) ...
    [0.0050 s 16646488 bytes] passed

Dpll: Dpll() ...
    [0.0040 s 15561312 bytes] passed

Dpll: validliteral(int, int) ...
    [0.0 s 15563248 bytes] passed immediately

Dpll: consistent(int, int[]) ...
    [0.0 s 15565280 bytes] passed immediately

Dpll: validclause(int[], int) ...
    [0.0 s 15567312 bytes] passed immediately

Dpll: validformula(int[][], int) ...
    [0.0 s 15569592 bytes] passed immediately

Dpll: validFormulaUpToJ(int[][], int, int) ...
    [0.0 s 15572504 bytes] passed immediately

Dpll: valuation(int[], int) ...
    [0.0 s 15574536 bytes] passed immediately

Dpll: ValuationSatLit(int, int[], int) ...
    [0.0 s 15577088 bytes] passed immediately

Dpll: ValuationSatClause(int[], int[], int) ...
    [0.0 s 15580000 bytes] passed immediately

Dpll: ValuationSatFormula(int[][], int[], int) ...
    [0.0030 s 15583080 bytes] passed immediately

Dpll: satisfiable(int[][], int) ...
    [0.0 s 15585360 bytes] passed immediately

Dpll: LitInClause(int, int[]) ...
    [0.0 s 15587392 bytes] passed immediately

Dpll: LitInFormula(int, int[][]) ...
    [0.0 s 15589736 bytes] passed immediately

Dpll: ClauseInFormula(int[], int[][]) ...
    [0.0 s 15592136 bytes] passed immediately

Dpll: remove(boolean, int, int[], int) ...
    [0.0010 s 15595608 bytes] passed immediately
  [3.594 s 15596488 bytes total]
6 warnings
```

A.2.3. Version 3

In this Appendix the third version of the JML specification of the DPLL Algorithm from Section 3.5.3 is listed.

```
public class DPLL {
```

```
2
3   /*@ public normal_behavior
4       @ requires validformula(formula,n);
5       @ requires n>=0;
6
7       @ assignable \nothing;
8
9       @ ensures \result == satisfiable(formula,n);
10
11  @*/
12      public static boolean DPLLiter(int[][]formula, int n){
13          int[][][] stack = new int[n][][];
14          int inStack = 0;
15          int[][]copy_formula = formula;
16
17  /*@ loop_invariant
18      @ validformula(formula,n) && n>=0 &&
19      @ stack!=null && stack.length<=n && inStack>=0 &&
20      @ validformula(copy_formula,n) &&
21      @ copy_formula.length <= formula.length &&
22      @ (\forall int i; 0<=i && i<inStack; validformula(stack[i],n)) &&
23      @ satisfiable(formula,n) <==>
24      @     (satisfiable(copy_formula,n) ||
25      @     (\exists int i; 0<=i && i<inStack; satisfiable(stack[i],n))) &&
26      @ (\forall int i; 0<=i && i<inStack-1; numVar(stack[i],n)>=numVar(stack[i+1],n)) &&
27      @ inStack>0 ==> numVar(copy_formula,n) == numVar(stack[inStack-1],n);
28
29      @ decreases (\sum int i; 0<=i && i<inStack;
30      @   power2(numVar(stack[i],n)+1)-1) + power2(numVar(copy_formula,n)+1)-1;
31
32      @ assignable inStack, stack[*], copy_formula[*];
33  @*/
34          while(true){
35              if(isEmptyFormula(copy_formula,n)) return true;
36
37              else if(containsEmptyClause(copy_formula,n)){
38                  if(inStack==0) return false;
39                  inStack--;
40                  copy_formula = stack[inStack];
41
42              }
43              else{
44                  int val = copy_formula[0][0];
45                  stack[inStack]=substitute(copy_formula,val,false,n);
46                  inStack++;
47                  copy_formula = substitute(copy_formula,val,true,n);
48              }
49          }
50      }
51
52  /*@ public normal_behavior
53      @ requires validformula(formula,n);
54      @ requires n>=0;
55
56      @ assignable \nothing;
57
58      @ measured_by numVar(formula,n);
59
60      @ ensures \result == satisfiable(formula,n);
61  @*/
62      public static boolean DPLLrec(int[][]formula, int n){
63          if(isEmptyFormula(formula,n)) return true;
64
65          else if(containsEmptyClause(formula,n)) return false;
66
67          else{
68              int val = formula[0][0];
```

```
69              if(DPLLrec(substitute(formula,val,true,n),n)==true)
70                  return true;
71              else{ return DPLLrec(substitute(formula,val,false,n),n);}
72
73          }
74      }
75
76
77  /*@ public normal_behavior
78      @ requires validformula(formula,n);
79
80      @ assignable \nothing;
81
82      @ ensures validformula(\result,n);
83      @ ensures (\forall int[] c; validclause(c,n); ClauseInFormula(c, \result) <==>
84      @   ((ClauseInFormula(c,formula) && !LitInClause(lit, c) && !LitInClause(-lit, c))) ||
85      @      (\exists int[]c1; validclause(c1,n) && ClauseInFormula(c1,formula);
86      @               c == remove(val, lit, c1,n)));
87  @*/
88      public static int[][] substitute (int[][]formula, int lit, boolean val, int n){
89          int l = lit;
90          if(val== false) l=-l;
91
92          int[][]copy_formula = deleteClause(formula, l,n);
93
94  /*@ loop_invariant
95      @ validformula(formula, n) && validformula(copy_formula,n) &&
96      @ copy_formula != null && copy_formula.length <= formula.length &&
97      @ 0 <= i && i <= copy_formula.length &&
98      @ (\forall int j; 0<=j && j<i; (\forall int l1; validliteral(l1,n);
99      @       LitInClause(l1, copy_formula[j]) <==>
100     @       (l1!=-l && LitInClause(l1,\old(copy_formula[j])))));
101
102     @ decreases copy_formula.length-i;
103
104     @ assignable i, copy_formula[*];
105 @*/
106         for(int i= 0; i<copy_formula.length;i++){
107             copy_formula[i]=deleteLit(copy_formula[i], -l,n);
108         }
109
110         return copy_formula;
111     }
112
113
114
115 /*@ public normal_behavior
116     @ requires validformula(formula,n);
117
118     @ assignable \nothing;
119
120     @ ensures validformula(\result,n);
121     @ ensures (\forall int[] c; validclause(c,n);
122     @     ClauseInFormula(c, \result) <==>
123     @       (ClauseInFormula(c,formula) && !LitInClause(lit, c)));
124 @*/
125     public static int[][] deleteClause(int[][] formula, int lit, int n ){
126         int[][] copy_formula = new int[formula.length][];
127         int j = 0;
128         int i = 0;
129
130 /*@ loop_invariant
131     @ validformula(formula,n) &&
132     @ 0 <= i && i <= formula.length && 0<=j && j<=i &&
133     @ (\forall int k; 0<=k && k<j;
134     @    ClauseInFormula(formula[k], copy_formula) <==>
135     @       (ClauseInFormula(formula[k],formula) && !LitInClause(lit, formula[k])));
```

```
136
137     @ decreases formula.length-i;
138
139     @ assignable j, i, copy_formula[*];
140 @*/
141         while(i<formula.length){
142             int pos = search(formula[i], lit,n);
143
144             if(pos == -1){
145                 copy_formula[j]=formula[i];
146                 j++;
147             }
148             i++;
149         }
150         return CutFormula(copy_formula, j,n);
151     }
152
153
154 /*@ public normal_behavior
155     @ requires validclause(clause,n);
156
157     @ assignable \nothing;
158
159     @ ensures validclause(\result,n);
160 //    @ ensures
161 //      ((\forall int i; 0<=i && i< clause.length; clause[i]!=lit) &&
162 //         \result == clause) ||
163 //         (\result.length == clause.length-1 &&
164 //            (\exists int i; 0<=i && i< clause.length; clause[i]==lit &&
165 //            (\forall int j; 0<=j && j< i; \result[j]==clause[j]) &&
166 //            (\forall int j; i<=j && j< clause.length-1;
167 //              \result[j]==clause[j+1])));
168    @ ensures (\forall int l; validliteral(l,n); LitInClause(l, \result) <==> (l!=lit &&
        LitInClause(l, clause)));
169 @*/
170     public static int[] deleteLit(int[] clause, int lit, int n){
171         int pos = search(clause,lit,n);
172
173         if(pos==-1) return clause;
174         else return CopyClause(clause,pos,n);
175     }
176
177 /*@ public normal_behavior
178     @ requires validclause(clause,n);
179     @ requires 0<=pos && pos<clause.length;
180
181     @ assignable \nothing;
182
183     @ ensures validclause(\result,n);
184     @ ensures \result.length == clause.length-1;
185     @ ensures (\forall int i; 0<=i && i<pos; \result[i]==clause[i]);
186     @ ensures (\forall int i; pos<=i && i<clause.length-1; \result[i]==clause[i+1]);
187 @*/
188     public static int[] CopyClause(int []clause, int pos, int n){
189         int[] copy_clause = new int[clause.length-1];
190         int i;
191
192 /*@ loop_invariant
193     @ validclause(clause, n) && 0 <= pos && pos < clause.length &&
194     @ copy_clause != null && copy_clause.length == clause.length-1 &&
195     @ 0 <= i && i <= pos &&
196     @ (\forall int j; 0<=j && j<i; copy_clause[j] == clause[j]);
197
198     @ decreases clause.length-i;
199
200     @ assignable i, copy_clause[*];
201 @*/
```

```
202         for (i=0; i<pos; i++) copy_clause[i] = clause[i];
203
204     /*@ loop_invariant
205         @ validclause(clause, n) && 0 <= pos && pos < clause.length &&
206         @ copy_clause != null && copy_clause.length == clause.length-1 &&
207         @ pos <= i && i <= clause.length-1 &&
208         @ (\forall int j; 0<=j && j<pos; copy_clause[j] == clause[j]) &&
209         @ (\forall int j; pos<=j && j<i; copy_clause[j] == clause[j+1]);
210
211         @ decreases clause.length-i;
212
213         @ assignable i, copy_clause[*];
214     @*/
215         for (i=pos; i<clause.length-1; i++) copy_clause[i] = clause[i+1];
216
217         return copy_clause;
218     }
219
220     /*@ public normal_behavior
221         @ requires validFormulaUpToJ(formula,n,j);
222         @ requires 0<=j && j<=formula.length;
223
224         @ assignable \nothing;
225
226         @ ensures validformula(formula,n);
227         @ ensures \result.length == j;
228         @ ensures (\forall int i; 0<=i && i<j; formula[i]==\result[i]);
229     @*/
230         public static int[][] CutFormula(int [][]formula, int j,int n){
231             int[][] copy_formula = new int[j][];
232             int i = 0;
233
234     /*@ loop_invariant
235         @ validFormulaUpToJ(formula,n,j) && 0<=j && j<=formula.length &&
236         @ 0<=i && i<=j &&
237         @ (\forall int k; 0 <= k && k < i; copy_formula[k]==formula[k]);
238
239         @ decreases j-i;
240
241         @ assignable copy_formula[*],i;
242     @*/
243             while(i<j){
244                 copy_formula[i]= formula[i];
245                 i++;
246             }
247             return copy_formula;
248         }
249
250     /*@ public normal_behavior
251         @ requires validclause(clause,n);
252
253         @ assignable \nothing;
254
255         @ ensures (0 <= \result && \result < clause.length && clause[\result]==lit &&
256         @   (\forall int j; 0<=j && j<\result; clause[j]!=lit)) ||
257         @   (\result ==-1 && (\forall int i; 0<=i && i<clause.length; clause[i]!=lit));
258     @*/
259         public /*@pure@*/ static int search(int []clause, int lit, int n){
260             int pos = -1;
261             int i =0;
262
263     /*@ loop_invariant
264         @ validclause(clause,n) && 0 <= i && i <= clause.length &&
265         @ (\forall int j; 0 <= j && j < i; clause[j] != lit) &&
266         @ (pos == -1 || (pos == i && i < clause.length && clause[pos] == lit));
267
268         @ decreases pos == -1 ? clause.length-i : 0;
```

```
269
270     @ assignable pos, i;
271 @*/
272         while(pos==-1 && i<clause.length){
273             if(clause[i]==lit){
274                 pos = i;
275             }
276             else i++;
277         }
278         return pos;
279     }
280
281 /*@ public normal_behavior
282     @ requires validformula(formula,n);
283
284     @ assignable \nothing;
285
286     @ ensures (\result <==> (\exists int i; 0<=i && i<formula.length;
287     @   isEmptyClause(formula[i],n)));
288 @*/
289     public static boolean containsEmptyClause(int[][]formula, int n){
290         int i = 0;
291         boolean val= false;
292
293 /*@ loop_invariant
294     @ validformula(formula,n) && 0 <= i && i <= formula.length &&
295     @ (\forall int j; 0 <= j && j < i; !isEmptyClause(formula[j],n)) &&
296     @ (val == false || (val == true && i < formula.length && isEmptyClause(formula[i],n)));
297
298     @ decreases val == false ? formula.length-i : 0;
299
300     @ assignable val, i;
301 @*/
302         while( i<formula.length && val==false){
303             if(isEmptyClause(formula[i],n)) val=true;
304             else i++;
305         }
306         return val;
307     }
308
309
310 /*@ public normal_behavior
311     @ requires formula != null;
312
313     @ assignable \nothing;
314
315     @ ensures (\result <==> (formula.length == 0));
316 @*/
317     public /*@ pure @*/ static boolean isEmptyFormula(int[][]formula, int n){
318         return formula.length==0;
319     }
320
321
322 /*@ public normal_behavior
323     @ requires validclause(clause,n);
324
325     @ assignable \nothing;
326
327     @ ensures (\result <==> (clause.length==0));
328 @*/
329     public /*@ pure @*/ static boolean isEmptyClause(int[] clause, int n){
330         return clause.length==0;
331     }
332
333
334 // Input condition
335 /*@
```

```
336    @ ensures \result <==> ((1<=l && l<=n) || (-n<=l && l<=-1));
337    @ public static pure model boolean validliteral(int l, int n);
338
339    @ ensures \result <==> !(LitInClause(l,c) && LitInClause(-l,c));
340    @ public static pure model boolean consistent(int l, int[] c);
341
342    @ ensures \result <==>
343    @       (\forall int i; 0<=i && i<c.length; validliteral(c[i],n)) &&
344    @       (\forall int j; 0<=j && j<=n; consistent(j,c)) &&
345    @       c!=null;
346    @ public static pure model boolean validclause(int[] c,int n);
347
348    @ ensures \result <==> (\forall int i; 0<=i && i<f.length; validclause(f[i],n)) && f!=null;
349    @ public static pure model boolean validformula(int[][] f,int n);
350
351    @ ensures \result <==> (\forall int i; 0<=i && i<j; validclause(f[i],n)) && f!=null;
352    @ public static pure model boolean validFormulaUpToJ(int[][] f,int n, int j);
353  @*/
354
355  // Output condition
356  /*@
357    @ ensures \result <==>
358    @      (\forall int i; 0<=i && i<v.length; validliteral(v[i],n)) &&
359    @      (\forall int j; 0<=j && j<=n; consistent(j,v));
360    @ public static pure model boolean valuation(int []v, int n);
361
362    @ ensures \result <==> (valuation(v,n) && LitInClause(l,v));
363    @ public static pure model boolean ValuationSatLit(int l, int[]v, int n);
364
365    @ ensures \result <==>
366    @      (\exists int l; validliteral(l,n)&&LitInClause(l,c); ValuationSatLit(l,v,n));
367    @ public static pure model boolean ValuationSatClause(int[] c, int[]v, int n);
368
369    @ ensures \result <==>
370    @      (\forall int[] c; validclause(c,n)&&ClauseInFormula(c,f); ValuationSatClause(c,v,n));
371    @ public static pure model boolean ValuationSatFormula(int[][] f, int[]v, int n);
372
373    @ ensures \result <==>
374    @      (\exists int[] v; valuation(v,n); ValuationSatFormula(f,v,n));
375    @ public static pure model boolean satisfiable(int[][] f, int n);
376  @*/
377
378  // Helper functions
379  /*@
380      @ ensures \result <==> (\exists int i; 0<=i && i<c.length; c[i]==l);
381      @ public static pure model boolean LitInClause(int l, int[] c);
382
383      @ ensures \result <==> (\exists int i; 0<=i && i<f.length; LitInClause(l,f[i]));
384      @ public static pure model boolean LitInFormula(int l, int[][] f);
385
386      @ ensures \result <==> (\exists int i; 0<=i && i<f.length;
387      @   (\forall int j; 0<=j && j<c.length; LitInClause(c[j],f[i])) && c.length == f[i].length);
388      @ public static pure model boolean ClauseInFormula(int [] c, int[][] f);
389
390      @ ensures (\forall int l; validliteral(l,n);
391      @   LitInClause(l,\result) <==> (LitInClause(l,c) &&
392      @       ((val==false && l!=lit)||(val==true && l!=-lit))));
393      @ public static pure model int[] remove(boolean val, int lit, int[]c, int n);
394
395      @ ensures \result == (\num_of int l; 1<=l && l<=n;
396      @   LitInFormula(l,formula)||LitInFormula(-l,formula));
397      @ public static pure model int numVar(int [][] formula, int n);
398
399      @ axiom power2(0) == 1 && (\forall int i; 0<i; power2(i)==2*power2(i-1));
400      @ public static pure model int power2(int j);
401  @*/
402
```

```
}
```

A.3. TLA

In this Appendix, the model of the DPLL algorithm in TLA from Section 3.6 is given.

```
------------ MODULE DPLL --------------

EXTENDS Integers, TLC, FiniteSets

CONSTANT RangeLit, (* Maximal integer value for a literal *)
         ClauseLength, (* Number of literals in a clause *)
         ClauseNumber (* Number of clauses in a formula *)

(* PlusCal options (-termination) *)
(*
--algorithm Dpll{


variables
    Literals = {l \in -RangeLit..RangeLit: (l > 0 \/ l<0)};
    Clause = {c \in SUBSET(Literals): Cardinality(c)<=ClauseLength
              /\ \A l \in 1..RangeLit: \neg(l \in c /\ -l \in c)};

    FormulaSet = {f \in SUBSET(Clause): Cardinality(f)<=ClauseNumber};

    Formula \in FormulaSet;
    CFormula = Formula;
    OFormula = Formula;

    emptyformula = FALSE;
    emptyclause = FALSE;
    sat = 0;
    lit = 0;

    inStack = 0;
    stack = [l \in 1..RangeLit |-> {}];


macro substitute(F,n){
    F:= {c\{-n}: c \in F\{cl \in F: n \in cl}}
}

{
   while(sat = 0){
        emptyformula:= Formula={};
        emptyclause := \E c \in Formula: c={};

        if(emptyformula = TRUE) {sat := 1;}

        else if(emptyclause = TRUE){
            if(inStack = 0) {sat := -1;}

            else{
            Formula:=stack[inStack];
            inStack:=inStack-1;}
        }

        else{
            lit := CHOOSE l \in Literals: \E c \in Formula : l \in c;
            CFormula := Formula;

            substitute(Formula,lit);
```

```
            substitute(CFormula,-lit);

            inStack:= inStack+1;
            stack[inStack]:=CFormula;
        };
    };
}
}
*)

\* BEGIN TRANSLATION
VARIABLES Literals, Clause, FormulaSet, Formula, CFormula, OFormula,
          emptyformula, emptyclause, sat, lit, inStack, stack, pc

vars == << Literals, Clause, FormulaSet, Formula, CFormula, OFormula,
           emptyformula, emptyclause, sat, lit, inStack, stack, pc >>

Init == (* Global variables *)
        /\ Literals = {l \in -RangeLit..RangeLit: (l > 0 \/ l<0)}
        /\ Clause = {c \in SUBSET(Literals): Cardinality(c)<=ClauseLength
               /\ \A l \in 1..RangeLit: \neg(l \in c /\ -l \in c)}
        /\ FormulaSet = {f \in SUBSET(Clause):
               Cardinality(f)<=ClauseNumber}
        /\ Formula \in FormulaSet
        /\ CFormula = Formula
        /\ OFormula = Formula
        /\ emptyformula = FALSE
        /\ emptyclause = FALSE
        /\ sat = 0
        /\ lit = 0
        /\ inStack = 0
        /\ stack = [l \in 1..RangeLit |-> {}]
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ IF sat = 0
              THEN /\ emptyformula' = (F={})
                   /\ emptyclause' = (\E c \in F: c={})
                   /\ IF emptyformula' = TRUE
                        THEN /\ sat' = 1
                             /\ pc' = "Lbl_1"
                             /\ UNCHANGED << Formula, CFormula, lit, inStack >>
                        ELSE /\ IF emptyclause' = TRUE
                                  THEN /\ IF inStack = 0
                                            THEN /\ sat' = -1
                                                 /\ UNCHANGED << Formula,
                                                                 inStack >>
                                            ELSE /\ Formula' = stack[inStack]
                                                 /\ inStack' = inStack-1
                                                 /\ sat' = sat
                                       /\ pc' = "Lbl_1"
                                       /\ UNCHANGED << CFormula, lit >>
                                  ELSE /\ lit' = (CHOOSE l \in Literals: \E c \in Formula : l \in c)
                                       /\ CFormula' = Formula
                                       /\ Formula' = {c\{-lit'}: c \in Formula\{cl \in Formula:
                                                                                  lit' \in cl}}
                                       /\ pc' = "Lbl_2"
                                       /\ UNCHANGED << sat, inStack >>
              ELSE /\ pc' = "Done"
                   /\ UNCHANGED << Formula, CFormula, emptyformula,
                                   emptyclause, sat, lit, inStack >>
         /\ UNCHANGED << Literals, Clause, FormulaSet, OFormula, stack >>

Lbl_2 == /\ pc = "Lbl_2"
         /\ CFormula' = {c\{-(-lit)}: c \in CFormula\{cl \in CFormula: (-lit) \in cl}}
         /\ inStack' = inStack+1
         /\ stack' = [stack EXCEPT ![inStack'] = CFormula']
```

```
125         /\ pc' = "Lbl_1"
126         /\ UNCHANGED << Literals, Clause, FormulaSet, Formula, OFormula,
127                         emptyformula, emptyclause, sat, lit >>
128
129 Next == Lbl_1 \/ Lbl_2
130           \/ (* Disjunct to prevent deadlock on termination *)
131              (pc = "Done" /\ UNCHANGED vars)
132
133 Spec == /\ Init /\ [][Next]_vars
134        /\ WF_vars(Next)
135
136
137 \* END TRANSLATION
138 ----------------------------------------------------------------------------
139 \* WHAT TO CHECK
140
141 \* Algorithm terminates
142 Termination == <>(pc = "Done")
143
144 \* Algorithm satisfies output condition
145 Valuation == {c \in SUBSET(Literals): Cardinality(c)<=ClauseLength /\
146               \A l \in 1..RangeLit: \neg(l \in c /\ -l \in c)}
147
148 ValuationSatLit(v,l) == v \in Valuation /\ l \in v
149
150 ValuationSatClause(v,c) == \E l \in Literals: l \in c /\ ValuationSatLit(v,l)
151
152 ValuationSatFormula(v,f) == \A c \in f: ValuationSatClause(v,c)
153
154 Satisfiable(f) == \E v \in Valuation: ValuationSatFormula(v,f)
155
156 IsCorrect == [](pc = "Done" => (sat = 1 <=> Satisfiable(OFormula)))
157 ============================================================================
```

A.4. Alloy

A.4.1. Model 1

In this Appendix the Alloy specification of the DPLL Algorithm from Section 3.7.1 is given.

```
1  module runSatisfiable
2
3  //Literals
4  abstract sig literal {}
5  one sig A, nA, B, nB extends literal {}
6
7  //Clause
8  sig clause {LitInClause: set literal}{
9      not A + nA in LitInClause and
10     not B + nB in LitInClause
11 }
12
13 //Formula
14 sig formula{ ClauseInFormula: set clause}{}
15
16 //Facts about clauses
17 fact allClausesBelongToAFormula {
18         all c:clause | some f:formula | c in f.ClauseInFormula
19 }
20
21 fact clausesAreUnique {
22   all c1:clause | all c2: clause |
```

```
23          c1.LitInClause = c2.LitInClause => c1 = c2
24  }
25
26  //Valuation
27  sig valuation {LitInValuation: set literal}{
28      not A + nA in LitInValuation and
29      (A in LitInValuation or
30      nA in LitInValuation) and
31      not B + nB in LitInValuation and
32      (B in LitInValuation or
33      nB in LitInValuation)
34   }
35
36  fact valuationsAreUnique {
37    all v1,v2: valuation | v1.LitInValuation = v2.LitInValuation => v1 = v2
38  }
39
40  //satisfiable
41  pred satisfiable(f:formula){
42     some v:valuation | all c : f.ClauseInFormula | some l:c.LitInClause | l in v.LitInValuation
43  }
44
45  run satisfiable for 4 but 1 formula, 1 valuation
```

A.4.2. Model 2

The second version of the Alloy specification of the DPLL Algorithm from Section 3.7.2 is shown.

```
1   module runDpll
2
3   open util/ordering[state] as ord
4
5   //Literals
6   abstract sig literal {negate: one literal}
7   one sig A  extends literal {}{negate = nA}
8   one sig nA extends literal {}{negate = A}
9   one sig B  extends literal {}{negate = nB}
10  one sig nB extends literal {}{negate = B}
11  one sig C  extends literal {}{negate = nC}
12  one sig nC extends literal {}{negate = C}
13
14  //Clause
15  sig clause {LitInClause: set literal}{
16      not A + nA in LitInClause and
17      not B + nB in LitInClause and
18      not C + nC in LitInClause
19  }
20
21  //Formula
22  sig formula{ ClauseInFormula: set clause}{}
23
24  //States
25  sig state{currentFormula: one formula}
26
27  //Facts about clauses
28  fact allClausesBelongToAFormula {
29          all c:clause | some f:formula | c in f.ClauseInFormula
30  }
31
32  fact clausesAreUnique {
33    all c1:clause | all c2: clause | c1.LitInClause = c2.LitInClause => c1 = c2
34  }
35
```

```
36 //Substitute
37 pred substitute[f, f': one formula]{
38     one l: f.ClauseInFormula.LitInClause | {
39          let cs0 = { c: f.ClauseInFormula | not l in c.LitInClause } |
40          let cs1 = cs0.{c1:clause, c2:clause | c2.LitInClause = c1.LitInClause-l.negate} |
41            f'.ClauseInFormula = cs1
42     }
43 }
44
45 //Initial State
46 fact initialState {
47    let s0 = ord/first | #s0.currentFormula.ClauseInFormula <5
48 }
49
50 //Algorithm
51 fact DPLL{
52     all s: state, s': ord/next[s] {
53         (#s.currentFormula.ClauseInFormula = 0 or
54         some c: s.currentFormula.ClauseInFormula | #c.LitInClause = 0) => s' = s
55      else
56         substitute[s.currentFormula ,s'.currentFormula]
57     }
58 }
59
60 //run Algorithm
61 pred solve{
62     #ord/last.currentFormula.ClauseInFormula = 0
63     or some c: ord/last.currentFormula.ClauseInFormula | #c.LitInClause = 0
64 }
65
66 run solve for 4 but 10 clause
```

A.4.3. Model 3

In this Appendix the third version of the Alloy specification for the DPLL method from Section 3.7.3 is listed.

```
1  module checkDpll
2
3  open util/ordering[state] as ord
4
5  //Literal
6  abstract sig literal {negate: one literal}
7  one sig A  extends literal {}{negate = nA}
8  one sig nA extends literal {}{negate = A}
9  one sig B  extends literal {}{negate = nB}
10 one sig nB extends literal {}{negate = B}
11 one sig C  extends literal {}{negate = nC}
12 one sig nC extends literal {}{negate = C}
13
14 //Clause
15 sig clause {LitInClause: set literal}{
16     not A + nA in LitInClause and
17     not B + nB in LitInClause and
18     not C + nC in LitInClause
19 }
20
21 //Formula
22 sig formula{ ClauseInFormula: set clause}{}
23
24 //States
25 sig state{currentFormula: one formula}
26
27 //Facts about clauses
```

```
28 fact allClausesBelongToAFormula {
29   all c:clause | some f:formula | c in f.ClauseInFormula
30 }
31
32 fact clausesAreUnique {
33   all c1:clause | all c2: clause | c1.LitInClause = c2.LitInClause => c1 = c2
34 }
35
36 //Substitute
37 pred substitute[f, f': one formula]{
38     one l: f.ClauseInFormula.LitInClause | {
39          let cs0 = { c: f.ClauseInFormula | not l in c.LitInClause } |
40          let cs1 = cs0.{c1:clause, c2:clause | c2.LitInClause = c1.LitInClause-l.negate} |
41            f'.ClauseInFormula = cs1
42     }
43 }
44
45 //Initial State
46 fact initialState {
47    let s0 = ord/first | #s0.currentFormula.ClauseInFormula <5
48 }
49
50 //Algorithm
51 fact DPLL{
52     all s: state, s': ord/next[s] {
53         (#s.currentFormula.ClauseInFormula = 0 or
54         some c: s.currentFormula.ClauseInFormula | #c.LitInClause = 0) => s' = s
55      else
56          substitute[s.currentFormula ,s'.currentFormula]
57     }
58 }
59
60 //Valuation and Satisfiable
61 sig valuation {LitInValuation: set literal}{
62     not A + nA in LitInValuation and
63     (A in LitInValuation or
64     nA in LitInValuation) and
65     not B + nB in LitInValuation and
66     (B in LitInValuation or
67     nB in LitInValuation) and
68     not C + nC in LitInValuation and
69     (C in LitInValuation or
70     nC in LitInValuation)
71 }
72
73 fact valuationsAreUnique {
74   all v1:valuation | all v2: valuation | v1.LitInValuation = v2.LitInValuation => v1 = v2
75 }
76
77 pred satisfiable(f:formula){
78    some v:valuation | all c : f.ClauseInFormula | some l:c.LitInClause | l in v.LitInValuation
79 }
80
81 assert solve{
82   (#ord/last.currentFormula.ClauseInFormula = 0 <=> satisfiable[ord/first.currentFormula])
83    and
84   (#ord/last.currentFormula.ClauseInFormula = 0 or
85    some c: ord/last.currentFormula.ClauseInFormula | #c.LitInClause = 0)
86 }
87
88 check solve for 8 but 5 state, 10 clause
```

A.5. VDM

A.5.1. Recursive Model

In this Appendix the specification of the recursive DPLL Algorithm from Section 3.8.1 is listed.

```
class DPLLrec

values
  public n: int = 2;
  public numOfClause: int = 4;

types

public Clause = set of Literal
inv c == forall l in set c & not (l in set c and -l in set c);

public Literal = int
inv l == l <= n and l>=-n and l <> 0;

public Valuation = set of Literal
inv v == (forall l in set {1,...,n} &
            (l in set v or -l in set v) and (not (-l in set v and l in set v)));

functions

/*substitutes a literal in the formula */
public substitute: Literal * set of Clause -> set of Clause
substitute(l,f) ==
   def f1 = {c | c in set f & not l in set c} in {c\{-l}|c in set f1};

/* runs the recursive algorithm*/
public Dpllrec: set of Clause -> bool
Dpllrec(f) ==
    if f = {}
        then true
    elseif exists c in set f & c = {}
        then false
    else
         let l in set {-n,...,n}\{0} be st (exists c in set f & l in set c) in (
            if Dpllrec(substitute(l,f)) = true
                   then true
            else Dpllrec(substitute(-l,f)))
pre validFormula(f)
post RESULT = satisfiable(f)
measure varInFormula;

/* returns the number of variables in the Formula */
public varInFormula: set of Clause -> nat
varInFormula(f) ==
 let lit ={l |l in set {1,...,n} & (exists c in set f & (l in set c or -l in set c))}
       in card(lit) ;

/* input condition */
public validFormula: set of Clause -> bool
validFormula(f)==
   forall c in set f & validClause(c);

public validClause: Clause -> bool
validClause(c) ==
   forall l in set c & ( not(l in set c and -l in set c) and l <= n and l>=-n and l <> 0);

/* output condition */
public satisfiable: set of Clause -> bool
satisfiable(f) ==
```

```
   let pcl = power({1,...,n} union {-n,...,-1}) in (
       let cl = {c | c in set pcl & (forall l in set {1,...,n} &
           (l in set c or -l in set c) and (not (-l in set c and l in set c)))}
       in (exists val in set cl & valuationSatFormula(f,val)));

public valuationSatFormula: set of Clause * Valuation -> bool
valuationSatFormula(f,v)==
   (forall c in set f & valuationSatClause(c,v));

public valuationSatClause: Clause * Valuation -> bool
valuationSatClause(c,v)==
    exists l in set c & valuationSatLit(l,v);

public valuationSatLit: Literal * Valuation -> bool
valuationSatLit(l,v)==
     l in set v;


traces
S1: let pcl = power({1,...,n} union {-n,...,-1}) in (
      let cl = {c | c in set pcl & (forall val in set c & (not (-val in set c and val in set c)))}
      in (
       let cl1 = power(cl) in (
          let f in set cl1 be st card(f)<=numOfClause in Dpllrec(f))))


end DPLLrec
```

A.5.2. Iterative Model

In this Appendix the specification of the iterative DPLL Algorithm from Section 3.8.2 is listed.

```
class DPLLiter

values
  public n: int = 2;
  public numOfClause: int = 4;

types

public Clause = set of Literal
inv c == forall l in set c & not (l in set c and -l in set c);

public Literal = int
inv l == l <= n and l>=-n and l <> 0;

public Valuation = set of Literal
inv v == (forall l in set {1,...,n} &
            (l in set v or -l in set v) and (not (-l in set v and l in set v)));

instance variables
  public Formula : set of Clause := {};
  public OldFormula: set of Clause := {};
  public stack : seq of (set of Clause) := [];


inv  validFormula(Formula) and validFormula(OldFormula) and
     (forall f in set elems stack & validFormula(f)) and
     n>=0 and numOfClause >=0 and len(stack)<=n and
     (forall i in set {1,...,len(stack)-1} &
         varInFormula(stack(i+1)) >= varInFormula(stack(i))) and
     card(Formula)<=card(OldFormula) and
     OldFormula <> {} => (satisfiable(OldFormula) <=>
```

```
32          (satisfiable(Formula) or (exists f in set elems stack & satisfiable(f))))
33
34  operations
35
36  /* substitutes a literal in a formula */
37  public substitute: Literal * set of Clause ==> set of Clause
38  substitute(l, f) ==
39     def f1 = {c | c in set f & not l in set c} in
40        (def f2 = {c\{-l}|c in set f1} in return f2);
41
42
43  /* runs the algorithm iteratively */
44   public Dplliter: () ==> bool
45  Dplliter() ==
46  (
47     while true do (
48        if Formula = {}
49            then return true
50        elseif exists c in set Formula & c = {}
51            then (
52              if stack = [] then
53                  return false
54              else (
55                  Formula := hd stack;
56                  stack := tl stack)
57             )
58        else
59           let l in set {-n,...,n}\{0} be st
60             (exists c in set Formula & l in set c) in
61                  (stack := [substitute(l, Formula)]^stack;
62                  Formula := substitute(-l, Formula); )
63     )
64  )
65  pre validFormula(Formula)
66  post RESULT = satisfiable(Formula~);
67
68  /* allocates the Formula */
69  public setFormula: set of Clause ==> ()
70  setFormula(c) ==
71   (Formula:= c;
72   OldFormula := c);
73
74  /* returns the number of variables in the Formula */
75  public varInFormula: set of Clause ==> nat
76  varInFormula(f) ==
77   let lit ={l |l in set {1,...,n} & (exists c in set f & (l in set c or -l in set c))}
78         in return card(lit) ;
79
80  functions
81
82  /* input condition */
83  public validFormula: set of Clause -> bool
84  validFormula(f)==
85     forall c in set f & validClause(c);
86
87  public validClause: Clause -> bool
88  validClause(c) ==
89     forall l in set c & ( not(l in set c and -l in set c) and l <= n and l>=-n and l <> 0);
90
91  /* output condition */
92  public satisfiable: set of Clause -> bool
93  satisfiable(f) ==
94     let pcl = power({1,...,n} union {-n,...,-1}) in (
95         let cl = {c | c in set pcl & (forall l in set {1,...,n} &
96             (l in set c or -l in set c) and (not (-l in set c and l in set c)))}
97         in (exists val in set cl & valuationSatFormula(f,val)));
98
```

```
public valuationSatFormula: set of Clause * Valuation -> bool
valuationSatFormula(f,v)==
   (forall c in set f & valuationSatClause(c,v));

public valuationSatClause: Clause * Valuation -> bool
valuationSatClause(c,v)==
    exists l in set c & valuationSatLit(l,v);

public valuationSatLit: Literal * Valuation -> bool
valuationSatLit(l,v)==
      l in set v;


traces
 S1: let pcl = power({1,...,n} union {-n,...,-1}) in (
       let cl = {c | c in set pcl & (forall val in set c & (not (-val in set c and val in set c)))}
        in (
          let cl1 = power(cl) in (
            let y in set cl1 be st card(y)<=numOfClause in setFormula(y);
            Dplliter())))


end DPLLiter
```

A.6. Event-B

A.6.1. Context: VarDefinition

This Appendix shows the `context` from Section 3.9.1, where the data types for the DPLL method are defined.

```
CONTEXT
    VarDefinition
CONSTANTS
    n
    Literal
    Clause
    Formula
    SetOfFormulas
    SetOfClause
    Valuation
AXIOMS
    defConstant:  n = 3
    TypeLit:  Literal ⊆ ℤ
    validLiteral: ∀ l · (l ∈ Literal ⇒( l ≤ n ∧ l ≥ -n ∧ l ≠ 0))
    TypeClause: Clause ⊆ Literal
    consistent: ∀ l · (l ∈ Clause ⇒( ¬(l ∈ Clause ∧ -l ∈ Clause)))
    TypeSetOfClause: SetOfClause = ℙ(Clause)
    TypeFormula: Formula ⊆ SetOfClause
    TypeSetOfFormulas: SetOfFormulas = ℙ(Formula)
    TypeVal:  Valuation = {v | v ⊆ Literal ∧
      ∀ l · (l ∈ Literal ⇒ ((l ∈ v ∨ -l ∈v) ∧¬ (l ∈ v ∧ -l ∈ v)))}
    axm1: finite(Formula)
 END
```

A.6.2. Context: ProgramCounter

This Appendix shows the `context` from Section 3.9.1, where the program counter for the DPLL method is defined.

```
CONTEXT
    ProgramCounter
SETS
    PC
CONSTANTS
    init
    done
    lbl1
AXIOMS
    axm1:  partition(PC, {init},{done},{lbl1})
END
```

A.6.3. Machine: Algorithm

This Appendix shows the `machine` from Section 3.9.1, where the DPLL method is specified.

```
MACHINE
    Algorithm

SEES
    VarDefinition
    ProgramCounter

VARIABLES
    f
    stack
    inStack
    sat
    pc
    Oformula

INVARIANTS
    inv1:  f ∈ SetOfFormulas
    inv2:  stack ∈ 1..n → SetOfFormulas
    inv3:  inStack ≥ 0 ∧ inStack ≤ n
    inv4:  sat ∈ {−1, 0, 1}
    inv5:  pc ∈ PC
    inv6:  Oformula ∈ SetOfFormulas
    inv7:  n≥0
    inv8:  card(f)≤card(Oformula)
    inv9:  ∀ i · (i ∈ 1..inStack ⇒ stack(i) ∈ SetOfFormulas)
    inv10: ∀ i · (i ∈ 1..inStack-1 ⇒ card(stack(i+1)) ≤ card(stack(i)))
   inv11: pc ≠ init ⇒
          (( ∃ v· ( v ∈ Valuation ∧ (∀ c · (c ∈ Oformula ⇒ (∃ l ·(l ∈ c ∧ l ∈ v)))))) ⇔
            ((∃ v· ( v ∈ Valuation ∧ (∀ c · (c ∈ f ⇒ (∃ l ·(l ∈ c ∧ l ∈ v)))))) ∨
            (∃ i · (i ∈ 1..inStack ∧ (∃ v· ( v ∈ Valuation ∧
                (∀ c · (c ∈ stack(i) ⇒ (∃ l ·(l ∈ c ∧ l ∈ v))))))))))
    inv12: ∀ i · (i ∈ 1..inStack ⇒ card(stack(i))≤card(Oformula))
    goal:  pc = done ⇒ (sat = 1 ⇔ (∃ v· ( v ∈ Valuation ∧ (∀c · (c ∈ Oformula ⇒ (∃ l ·(l ∈ c ∧
          l ∈ v)))))))

EVENTS
    INITIALISATION:
      THEN
        act1:  Oformula :∈ SetOfFormulas
        act2:  stack := λi·i ∈ 1..n |∅
        act3:  inStack := 0
        act4:  sat := 0
        act5:  pc := init
        act6:  f := ∅
     END

     SetOldFormula:
```

```
47        WHERE
48          grd1:  pc = init
49        THEN
50          act1:  f := Oformula
51          act2:  pc := lbl1
52        END
53
54      EmptyFormula:
55       WHERE
56          grd1:  f = ∅
57          grd2:  pc = lbl1
58       THEN
59          act1:  sat := 1
60          act2:  pc := done
61       END
62
63      EmptyClauseAndEmptyStack:
64       WHERE
65          grd1:  pc = lbl1 n
66          grd2:  ∃c·(c ∈ f ∧ c = ∅)
67          grd3:  inStack = 0
68       THEN
69          act1:  sat := -1
70          act2:  pc := done
71       END
72
73      EmptyClauseAndNonemptyStack:
74       WHERE
75          grd1:  pc = lbl1
76          grd2:  ∃c·(c ∈ f ∧ c = ∅)
77          grd3:  inStack ≠ 0
78       THEN
79          act1:  f := stack(inStack)
80          act2:  inStack:= inStack -1
81       END
82
83      substitute:
84       ANY
85          lit
86          cF
87       WHERE
88          grd1:  f ≠ ∅
89          grd2:  pc = lbl1
90          grd3:  ∀ c ·(c ∈ f ⇒ c ≠ ∅)
91          grd4:  lit ∈ Literal ∧ (∃ c ·(c ∈ f ∧ lit ∈ c))
92          grd5:  cF  ∈ SetOfFormulas ∧ cF = f
93          grd6:  inStack < n
94       THEN
95          act1:  f:= {c· c ∈ f\{cl· cl ∈ f ∧ lit ∈ cl|cl}|c\{-lit}}
96          act2:  inStack := inStack +1
97          act3:  stack(inStack+1) := {c ·c ∈ cF\{cl ·cl ∈ cF ∧-lit ∈ cl|cl}| c\{lit}}
98       END
99  END
```

A.6.4. Context: VarDefRefined

This Appendix shows the `context` from Section 3.9.2, where the data types for the refined machine of the DPLL method are defined.

```
1  CONTEXT
2      VarDefRefined
3  EXTENDS
4      VarDefinition
5  CONSTANTS
```

```
    m
    FormulaRef
    SetOfFormulaRef
AXIOMS
    axm1:  m = 5
    axm2:  FormulaRef = 1..m → {Clause ∪ {0}}
    axm3:  SetOfFormulaRef = {f | f ∈ FormulaRef}
END
```

A.6.5. Machine: AlgorithmRef

This Appendix shows the `machine` from Section 3.9.2, where the specification of the DPLL method is refined.

```
MACHINE
   AlgorithmRef
REFINES
   Algorithm

SEES
   VarDefRefined
   ProgramCounter

VARIABLES
   f1
   stack1
   inStack
   sat
   pc
   Oformula1

INVARIANTS
   inv1:  f1  ∈ SetOfFormulaRef
   inv2:  Oformula1 ∈ SetOfFormulaRef
   inv3:  stack1 ∈ 1..n → SetOfFormulaRef
   glu1:  ∀ c · (c ∈ SetOfClause ⇒ (c ∈ f ⇔ (∃ i ·(i ∈ 1..m ∧ f1(i) = c))))
   glu2:  ∀ c · (c ∈ SetOfClause ⇒ (c ∈ Oformula ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1(i) = c))))
   glu3:  ∀ c · (c ∈ SetOfClause ⇒ (∀ i · (i ∈ 1..n ⇒ ( c ∈ stack(i) ⇔ (∃ j · (j ∈ 1..m ∧
        stack1(i)(j) = c))))))
   glu4:  f = ∅ ⇔ (∀ i · (i ∈ 1..m ⇒ f1(i) = {0}))
   glu5:  Oformula = ∅ ⇔ (∀ i · (i ∈ 1..m ⇒ Oformula1(i) = {0}))
   glu6:  ∀ i · (i ∈ 1..n ⇒ ( stack(i) = ∅ ⇔ (∀ j · (j ∈ 1..m ∧ stack1(i)(j) = {0}))))
   glu7:  ∅ ∈ f ⇔ (∃ i ·(i ∈ 1..m ∧ f1(i) = ∅))
   glu8:  ∅ ∈ Oformula ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1(i) = ∅))
   glu9:  ∀ i · (i ∈ 1..n ⇒ ( ∅ ∈ stack(i) ⇔ (∃ j · (j ∈ 1..m ∧ stack1(i)(j) = ∅))))

EVENTS
  INITIALISATION:
     WITH
      Oformula': ∀ c · (c ∈ SetOfClause ⇒ (c ∈ Oformula' ⇔ (∃ i ·(i ∈ 1..m ∧ Oformula1'(i)=c))))
     THEN
       act1:  Oformula1 :∈ SetOfFormulaRef
       act2:  stack1 := λi·i ∈ 1..n | (λj · j ∈ 1..m | {0})
       act3:  inStack := 0
       act4:  sat := 0
       act5:  pc := init
       act6:  f1 := λi · i ∈1..m | {0}
     END

  SetFormula:
     REFINES
       SetOldFormula
     WHERE
       grd1:  pc = init
```

```
50      THEN
51        act1:  f1 := Oformula1
52        act2:  pc := lbl1
53      END
54
55    EmptyFormula:
56      REFINES
57        EmptyFormula
58      WHERE
59        grd1:  ∀i · (i ∈ 1..m ⇒ f1(i) = {0})
60        grd2:  pc = lbl1
61      THEN
62        act1:  sat := 1
63        act2:  pc := done
64      END
65
66    EmptyClauseAndEmptyStack:
67      REFINES
68        EmptyClauseAndEmptyStack
69      WHERE
70        grd1:  pc = lbl1
71        grd2:  ∃i·(i ∈ 1..m ∧ f1(i) = ∅)
72        grd3:  inStack = 0
73      THEN
74        act1:  sat := -1
75        act2:  pc := done
76      END
77
78    EmptyClauseAndNonemptyStack:
79      REFINES
80        EmptyClauseAndNonemptyStack
81      WHERE
82        grd1:  pc = lbl1
83        grd2:  ∃i·(i ∈ 1..m ∧ f1(i) = ∅)
84        grd3:  inStack ≠ 0
85      THEN
86        act1:  f1 := stack1(inStack)
87        act2:  inStack:= inStack -1
88      END
89
90    substitute:
91      REFINES
92         substitute
93      ANY
94         lit
95         cF1
96         subsF
97      WHERE
98         grd1:  ∀ i · (i ∈ 1..m ⇒ f1(i) ≠ {0})
99         grd2:  pc = lbl1
100        grd3:  ∀ i ·(i ∈ 1..m ⇒ f1(i) ≠ ∅)
101        grd4:  lit ∈ Literal ∧ (∃ i ·(i ∈ 1..m ∧ lit ∈ f1(i)))
102        grd5:  cF1 ∈ SetOfFormulaRef
103        grd7:  ∀ i · (i ∈ 1..m ⇒ (lit ∈ f1(i) ⇒ cF1(i) = {0}))
104        grd8:  ∀ i · (i ∈ 1..m ⇒ cF1(i) = f1(i)\{-lit})
105        grd6:  inStack < n
106        grd9:  subsF ∈ SetOfFormulaRef
107        grd10: ∀ i · (i ∈ 1..m ⇒ (-lit ∈ f1(i) ⇒ subsF(i) = {0}))
108        grd11: ∀ i · (i ∈ 1..m ⇒ subsF(i) = f1(i)\{lit})
109     WITH
110        cF: ∀ c · (c ∈ SetOfClause ⇒ (c ∈ cF ⇔ (∃ i ·(i ∈ 1..m ∧cF1(i) = c))))
111     THEN
112        act1:  f1 := subsF
113        act2:  inStack := inStack +1
114        act3:  stack1(inStack):=cF1
115     END
116 END
```

B. Model Listings for Dijkstra's Algorithm

B.1. Java

In this Appendix the Java prototype for Dijkstra's Algorithm from Section 4.3 is listed.

```
1  public class Dijkstra {
2
3      public static int[][] DijkstraAlg(int weight[][], int s, int n){
4
5          // marks connected vertices
6          int [] C = new int[n];
7          initArray(C,0);
8          C[s] = 1;
9
10         // marks unvisited vertices
11         int [] Q = new int[n];
12         initArray(Q,1);
13
14         // stores the predecessors of each vertex
15         int [] pre = new int[n];
16         initArray(pre,-1);
17         pre[s]=s;
18
19         // stores the shortest distances
20         int [] dist = new int[n];
21         initArray(dist,-1);
22         dist[s]=0;
23
24         int u = -1;
25         int newdist = -1;
26
27         while(existsMin(Q,C)){
28             u = returnMinDist(Q,C,dist);
29             Q[u]=0;
30
31             for(int v =0; v<n; v++){
32                 if (Q[v]==1 && weight[u][v]>0){
33                     newdist = dist[u]+weight[u][v];
34
35                     if(C[v]==1){
36                         if(newdist<dist[v]){
37                             dist[v]=newdist;
38                             pre[v] = u;
39                         }
40                     }
41                     else{
42                         dist[v]=newdist;
43                         pre[v] = u;
44                         C[v]=1;
45                     }
46                 }
47             }
48         }
49
50         int [][] result = new int[3][n];
51         result[0]=C;
52         result[1]=dist;
53         result[2]=pre;
```

```
54
55          return result;
56      }
57
58
59      public static void initArray(int[] arr, int value){
60          for(int i=0; i< arr.length; i++){
61              arr[i]=value;
62          }
63      }
64
65
66      public static boolean existsMin(int []Q, int[]C){
67          boolean result = false;
68          for(int i=0;i<Q.length;i++){
69              if(Q[i]==1 && C[i]==1){
70                  result = true;
71              }
72          }
73          return result;
74      }
75
76
77      public static int returnMinDist(int []Q, int[]C, int[]d){
78          int result = -1;
79          for(int i=0;i<Q.length;i++){
80              if(Q[i]==1 && C[i]==1){
81                  if(result==-1)
82                      result=i;
83                  else if(d[i]<d[result])
84                      result=i;
85              }
86          }
87          return result;
88      }
89  }
```

B.2. JML

B.2.1. Version 1

JML Specification

This Appendix shows the first version of the JML specification for Dijkstra's Algorithm from Section 4.4.1.

```
1   public class Dijkstra {
2
3   /*@ public normal_behavior
4   @ requires weight != null;
5   @ requires DefWeight(weight, n);
6
7   @ requires n > 0;
8   @ requires 0 <= s && s < n;
9
10  @ assignable \nothing;
11
12  @ ensures \result.length == 3;
13  @ ensures (\forall int i; 0<=i && i< 3; \result[i]!=null && \result[i].length==n);
14
15  @ ensures SetOfLinkedVertices(\result[0], s, weight, n);
16  @ ensures DefinesShortestDist(\result[1], \result[2], \result[0], s, weight, n);
17  @*/
```

```
18      public static int[][] DijkstraAlg(int weight[][], int s, int n){
19
20          // marks connected vertices
21          int [] C = new int[n];
22          initArray(C,0);
23          C[s]=1;
24
25          // marks unvisited vertices
26          int [] Q = new int[n];
27          initArray(Q,1);
28
29          // stores the predecessors of each vertex
30          int [] pre = new int[n];
31          initArray(pre,-1);
32          pre[s]=s;
33
34          // stores the shortest distances
35          int [] dist = new int[n];
36          initArray(dist,-1);
37          dist[s]=0;
38
39          int u = -1;
40          int newdist = -1;
41
42          while(existsMin(Q,C)){
43              u = returnMinDist(Q,C,dist);
44              Q[u]=0;
45
46              for(int v =0; v<n; v++){
47                  if (Q[v]==1 && weight[u][v]>0){
48                      newdist = dist[u]+weight[u][v];
49
50                      if(C[v]==1){
51                          if(newdist<dist[v]){
52                              dist[v]=newdist;
53                              pre[v] = u;
54                          }
55                      }
56                      else{
57                          dist[v]=newdist;
58                          pre[v] = u;
59                          C[v]=1;
60                      }
61
62                  }
63              }
64          }
65
66
67          int [][] result = new int[3][n];
68          result[0]=C;
69          result[1]=dist;
70          result[2]=pre;
71
72          return result;
73      }
74
75
76  /*@ public normal_behavior
77  @ requires arr != null;
78  @ requires arr.length > 0;
79
80  @ assignable arr[*];
81
82  @ ensures (\forall int i; 0<=i && i< arr.length; arr[i]==value);
83  @*/
84      public static void initArray(int[] arr, int value){
```

```
85
86          for(int i=0; i< arr.length; i++){
87              arr[i]=value;
88          }
89      }
90
91  /*@ public normal_behavior
92  @ requires Q != null;
93  @ requires Q.length > 0;
94  @ requires C != null;
95  @ requires C.length > 0;
96  @ requires C.length == Q.length;
97
98  @ assignable \nothing;
99
100 @ ensures \result == true <==> (\exists int i; 0<=i && i< Q.length; Q[i]==1 && C[i]==1);
101 @*/
102     public static boolean existsMin(int []Q, int[]C){
103     boolean result = false;
104     for(int i=0;i<Q.length;i++){
105         if(Q[i]==1 && C[i]==1){
106             result = true;
107         }
108     }
109     return result;
110 }
111
112
113 /*@ public normal_behavior
114 @ requires Q != null;
115 @ requires Q.length > 0;
116 @ requires C != null;
117 @ requires C.length > 0;
118 @ requires d != null;
119 @ requires d.length > 0;
120 @ requires C.length == Q.length && C.length == d.length;
121
122 @ assignable \nothing;
123
124 @ ensures -1<= \result && \result < Q.length;
125 @ ensures \result == -1 <==> (\forall int i; 0<=i && i< Q.length; Q[i]!=1 || C[i]!=1);
126 @ ensures \result != -1 ==> (\forall int i;
127 @    0<=i && i< \result && Q[i]==1 && C[i]==1; d[\result] < d[i]);
128 @ ensures \result != -1 ==> (\forall int i; \result<=i && i<Q.length
129 @    && Q[i]==1 && C[i]==1; d[\result] <= d[i]);
130 @*/
131     public static int returnMinDist(int []Q, int[]C, int[]d){
132     int result = -1;
133     for(int i=0;i<Q.length;i++){
134         if(Q[i]==1 && C[i]==1){
135             if(result==-1)
136                 result=i;
137             else if(d[i]<d[result])
138                 result=i;
139         }
140     }
141     return result;
142 }
143
144
145 /*@
146 /* Input condition */
147 @ ensures \result <==>
148 @   weight.length ==n &&
149 @   (\forall int i; 0<=i && i< weight.length; weight[i]!=null && weight[i].length==n) &&
150 @   (\forall int i; 0<=i && i<n; (\forall int j; 0<=j && j<n;
151 @        weight[i][j] == weight[j][i]&& weight[i][j]>=0))&&
```

```
@   (\forall int i; 0<=i && i<n;weight[i][i] == 0);
@ public static pure model boolean DefWeight(int[][]weight, int n);


/* Output condition */
@ ensures \result <==> (w[u][v]!=0);
@ public static pure model boolean neighbour(int [][]w, int u, int v);

@ ensures \result <==>
@    C[s] == 1 &&
@    (\forall int i; 0<=i && i<n && C[i]==1;
@        (\forall int j; 0<=j && j<n; neighbour(w,i,j) ==> C[j]==1)) &&
@    (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
@        (\exists int j; 0<=j && j<n && C[j]==1; neighbour(w,i,j)));
@ public static pure model boolean SetOfLinkedVertices(int[]C, int s,
@   int [][]w, int n);

@ ensures \result <==>
@    dist[s] == 0 &&
@    (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
@        (\exists int j; 0<=j && j<n && C[j]==1; pre[i]==j &&
@           neighbour(w,i,j) && dist[i]==dist[j]+w[i][j])) &&
@    (\forall int i; 0<=i && i<n && C[i]==1;
@       (\forall int j; 0<=j && j<n && C[j]==1;
@          neighbour(w,i,j) ==> dist[i]<= dist[j]+w[i][j]));
@ public static pure model boolean DefinesShortestDist(int[]dist,
@   int[]pre, int[]C, int s, int [][]w, int n);
@*/
}
```

Output of ESC/Java2

This Appendix shows the outout of ESC/Java2 for the first version of the JML specification for Dijkstra's Algorithm from Section 4.4.1.

```
guest@debian:~/Dijkstra$ escjava2 -NoCautions Dijkstra.java
ESC/Java version ESCJava-2.0.5
    [0.066 s 8475488 bytes]

Dijkstra ...
  Prover started:0.0070 s 11191320 bytes
    [0.467 s 10506608 bytes]

Dijkstra: DijkstraAlg(int[][], int, int) ...
    [0.859 s 12220056 bytes] passed

Dijkstra: initArray(int[], int) ...
    [0.012 s 11411936 bytes] passed

Dijkstra: existsMin(int[], int[]) ...
    [0.0080 s 11848528 bytes] passed

Dijkstra: returnMinDist(int[], int[], int[]) ...
    [0.021 s 11450296 bytes] passed

Dijkstra: Dijkstra() ...
    [0.0030 s 11612576 bytes] passed

Dijkstra: neighbour(int[][], int, int) ...
    [0.0010 s 11615784 bytes] passed immediately
  [1.372 s 11616664 bytes total]
```

B.2.2. Version 2

JML Specification

This Appendix shows the second version of the JML specification for Dijkstra's Algorithm from Section 4.4.2.

```
1  public class Dijkstra {
2
3  /*@ public normal_behavior
4  @ requires weight != null;
5  @ requires weight.length ==n;
6  @ requires (\forall int i; 0<=i && i< weight.length; weight[i]!=null && weight[i].length==n);
7  @ requires (\forall int i; 0<=i && i<n; (\forall int j; 0<=j && j<n;
8  @     weight[i][j] == weight[j][i]&& weight[i][j]>=0));
9  @ requires (\forall int i; 0<=i && i<n;weight[i][i] == 0);
10
11 @ requires n >0;
12 @ requires 0 <= s && s < weight.length;
13
14 @ assignable \nothing;
15
16 @ ensures \result.length == 3;
17 @ ensures (\forall int i; 0<=i && i< 3; \result[i]!=null && \result[i].length==n);
18 @ ensures \result[0][s] == 1;
19 @ ensures (\forall int i; 0<=i && i<n && \result[0][i]==1;
20 @     (\forall int j; 0<=j && j<n; neighbour(weight,i,j) ==> \result[0][j]==1));
21 @ ensures (\forall int i; 0<=i && i<n && \result[0][i]==1 && i!=s;
22 @   (\exists int j; 0<=j && j<n && \result[0][j]==1; neighbour(weight,i,j)));
23
24 @ ensures \result[1][s] == 0;
25 @ ensures (\forall int i; 0<=i && i<n && \result[0][i]==1 && i!=s;
26 @    (\exists int j; 0<=j && j<n && \result[0][j]==1;
27 @         \result[2][i]==j && neighbour(weight,i,j) &&
28 @         \result[1][i]==\result[1][j]+weight[i][j]));
29 @ ensures (\forall int i; 0<=i && i<n && \result[0][i]==1;
30 @    (\forall int j; 0<=j && j<n && \result[0][j]==1;
31 @         neighbour(weight,i,j) ==> \result[2][i]<=\result[2][j] +weight[i][j]));
32 @*/
33     public static int[][] DijkstraAlg(int weight[][], int s, int n){
34
35         // marks connected vertices
36         int [] C = new int[n];
37         initArray(C,0);
38         C[s]=1;
39
40         // marks unvisited vertices
41         int [] Q = new int[n];
42         initArray(Q,1);
43
44         // stores the predecessors of each vertex
45         int [] pre = new int[n];
46         initArray(pre,-1);
47         pre[s]=s;
48
49         // stores the shortest distances
50         int [] dist = new int[n];
51         initArray(dist,-1);
52         dist[s]=0;
53
54         int u = -1;
55         int newdist = -1;
56 /*@
57 /* Precondition of the function */
58 @ loop_invariant weight != null;
59 @ loop_invariant weight.length == n;
```

```
60  @ loop_invariant (\forall int i; 0<=i && i< weight.length;
61  @      weight[i]!=null && weight[i].length==n);
62  @ loop_invariant (\forall int i; 0<=i && i<n;
63  @      (\forall int j; 0<=j && j<n; weight[i][j] == weight[j][i] && weight[i][j]>=0));
64  @ loop_invariant (\forall int i; 0<=i && i<n;weight[i][i] == 0);
65  @ loop_invariant n>0 && 0 <= s && s < weight.length;
66
67  /* Range of local variables */
68  @ loop_invariant C!= null && C.length == n;
69  @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==0 || C[i]==1);
70
71  @ loop_invariant Q!= null && Q.length == n;
72  @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 || Q[i]==1);
73
74  @ loop_invariant dist!= null && dist.length == n;
75  @ loop_invariant (\forall int i; 0<=i && i<n; dist[i]>=-1);
76
77  @ loop_invariant pre!= null && pre.length == n;
78  @ loop_invariant (\forall int i; 0<=i && i<n; -1<=pre[i] && pre[i]<n);
79
80  @ loop_invariant C[s]==1 && dist[s]==0;
81  @ loop_invariant -1<=u && u<n;
82  @ loop_invariant newdist>=-1;
83
84  /* Set Of Visited Linked Vertices*/
85  @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0;
86  @     (\forall int j; 0<=j && j<n; neighbour(weight,i,j)==> C[j]==1 ));
87  @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
88  @     (\exists int j; 0<=j && j<n && C[j]==1; neighbour(weight,i,j)));
89
90  /* Defines Shortest Dist Of Visited Nodes*/
91  @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
92  @     (\exists int j; 0<=j && j<n && Q[j]==0; pre[i]==j &&
93  @        neighbour(weight,i,j) && dist[i]==dist[i]+weight[i][j]));
94  @ loop_invariant(\forall int i; 0<=i && i<n && C[i]==1;
95  @     (\forall int j; 0<=j && j<n && C[j]==1;
96  @        neighbour(weight,i,j) ==> dist[j]<=dist[i]+weight[i][j]));
97
98  /* Visited implies Connected */
99  @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 ==> C[i]==1);
100
101 /* Connected implies defined predecessor and distance */
102 @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==1 ==> (dist[i]>=0 && pre[i]!=-1));
103
104 /* Distance of visited nodes is shorter than the distance of unvisited but connected nodes */
105 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0; (\forall int j;
106 @      0<=j && j<n && Q[j]==1 && C[j]==1; dist[i]<=dist[j]));
107
108 @ decreases (\num_of int i; 0<=i && i<n; Q[i]==1);
109 @*/
110         while(existsMin(Q,C))
111         while(existsMin(Q,C)){
112             u = returnMinDist(Q,C,dist);
113             Q[u]=0;
114
115 /*@ loop_invariant weight != null;
116 @ loop_invariant weight.length ==n;
117 @ loop_invariant (\forall int i; 0<=i && i< weight.length; weight[i]!=null &&
        weight[i].length==n);
118 @ loop_invariant (\forall int i; 0<=i && i<n;
119 @     (\forall int j; 0<=j && j<n; weight[i][j] == weight[j][i] && weight[i][j]>=0));
120 @ loop_invariant (\forall int i; 0<=i && i<n;weight[i][i] == 0);
121 @ loop_invariant n>0 && 0 <= s && s < weight.length;
122
123 @ loop_invariant C!= null && C.length == n;
124 @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==0 || C[i]==1);
125
```

```
126 @ loop_invariant Q!= null && Q.length == n;
127 @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 || Q[i]==1);
128
129 @ loop_invariant dist!= null && dist.length == n;
130 @ loop_invariant (\forall int i; 0<=i && i<n; dist[i]>=-1);
131
132 @ loop_invariant pre!= null && pre.length == n;
133 @ loop_invariant (\forall int i; 0<=i && i<n; -1<=pre[i] && pre[i]<n);
134
135 @ loop_invariant C[s]==1 && dist[s]==0;
136 @ loop_invariant newdist>=-1;
137 @ loop_invariant 0<=u && u<n && Q[u]==0;
138
139 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0 && i!=u;
140 @    (\forall int j; 0<=j && j<n; neighbour(weight,i,j)==> C[j]==1));
141 @ loop_invariant (\forall int i; 0<=i && i<v; neighbour(weight,u,i)==> C[i]==1 );
142 @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
143 @   (\exists int j; 0<=j && j<n && C[j]==1; neighbour(weight,i,j)));
144
145 @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i!=s;
146 @   (\exists int j; 0<=j && j<n && Q[j]==0; pre[i]==j &&
147 @       neighbour(weight,i,j) && dist[i]==dist[i]+weight[i][j]));
148 @ loop_invariant(\forall int i; 0<=i && i<n && C[i]==1 && i!=u;
149 @    (\forall int j; 0<=j && j<n && C[j]==1;
150 @        neighbour(weight,i,j) ==> dist[j]<=dist[i]+weight[i][j]));
151 @ loop_invariant (\forall int j; 0<=j && j<v && C[j]==1;
152 @     neighbour(weight,u,j) ==> dist[j]<=dist[u]+weight[u][j]);
153
154 @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 ==> C[i]==1);
155
156 @ loop_invariant (\forall int i; 0<=i && i<n; C[i]==1 ==> (dist[i]>=0 && pre[i]!=-1));
157
158 @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 ==> dist[i]<=dist[u]);
159 @ loop_invariant (\forall int i; 0<=i && i<n && Q[i]==0;
160 @    (\forall int j; 0<=j && j<n && Q[j]==1 && C[j]==1; dist[i]<=dist[j]));
161
162 @ decreases n-v;
163 @*/
164           for(int v =0; v<n; v++){
165               if (Q[v]==1 && weight[u][v]>0){
166                   newdist = dist[u]+weight[u][v];
167
168                   if(C[v]==1){
169                       if(newdist<dist[v]){
170                           dist[v]=newdist;
171                           pre[v] = u;
172                       }
173                   }
174                   else{
175                       dist[v]=newdist;
176                       pre[v] = u;
177                       C[v]=1;
178                   }
179
180               }
181           }
182       }
183
184       int [][] result = new int[3][n];
185       result[0]=C;
186       result[1]=dist;
187       result[2]=pre;
188
189       return result;
190    }
191
192
```

```
193 /*@ public normal_behavior
194 @ requires arr != null;
195 @ requires arr.length > 0;
196
197 @ assignable arr[*];
198
199 @ ensures (\forall int i; 0<=i && i< arr.length; arr[i]==value);
200 @*/
201     public static void initArray(int[] arr, int value){
202 /*@ loop_invariant arr!= null && arr.length > 0;
203 @ loop_invariant 0 <= i && i <= arr.length;
204 @ loop_invariant (\forall int j; 0<=j && j<i; arr[j] == value);
205
206 @ decreases arr.length-i;
207 @*/
208         for(int i=0; i< arr.length; i++){
209             arr[i]=value;
210         }
211     }
212
213
214 /*@ public normal_behavior
215 @ requires Q != null;
216 @ requires Q.length > 0;
217 @ requires C != null;
218 @ requires C.length > 0;
219 @ requires C.length == Q.length;
220
221 @ assignable \nothing;
222
223 @ ensures \result == true <==> (\exists int i; 0<=i && i< Q.length; Q[i]==1 && C[i]==1);
224 @*/
225     public static boolean existsMin(int []Q, int[]C){
226     boolean result = false;
227 /*@ loop_invariant Q!= null && Q.length > 0;
228 @ loop_invariant C!= null && C.length > 0;
229 @ loop_invariant C.length == Q.length;
230 @ loop_invariant 0 <= i && i <= Q.length;
231 @ loop_invariant result==false <==>(\forall int j; 0<=j && j<i; Q[j]!=1 || C[j]!=1);
232 @ decreases Q.length-i;
233 @*/
234     for(int i=0;i<Q.length;i++){
235         if(Q[i]==1 && C[i]==1){
236             result = true;
237         }
238     }
239     return result;
240 }
241
242
243 /*@ public normal_behavior
244  @ requires Q != null;
245  @ requires Q.length > 0;
246  @ requires C != null;
247  @ requires C.length > 0;
248  @ requires d != null;
249  @ requires d.length > 0;
250  @ requires C.length == Q.length && C.length == d.length;
251
252  @ assignable \nothing;
253
254  @ ensures -1<= \result && \result < Q.length;
255  @ ensures \result == -1 <==> (\forall int i; 0<=i && i< Q.length; Q[i]!=1 || C[i]!=1);
256  @ ensures \result != -1 ==> (\forall int i; 0<=i && i< \result &&
257  @      Q[i]==1 && C[i]==1; d[\result] < d[i]);
258  @ ensures \result != -1 ==> (\forall int i; \result<=i && i< Q.length &&
259  @      Q[i]==1 && C[i]==1; d[\result] <= d[i]);
```

```
@*/
    public static int returnMinDist(int []Q, int[]C, int[]d){
    int result = -1;
/*@ loop_invariant Q!= null && Q.length > 0;
@ loop_invariant C!= null && C.length > 0;
@ loop_invariant d!= null && d.length > 0;
@ loop_invariant C.length == Q.length && d.length == Q.length;
@ loop_invariant 0 <= i && i <= Q.length;
@ loop_invariant result==-1 <==>(\forall int j; 0<=j && j<i; Q[j]!=1 || C[j]!=1);
@ loop_invariant result!=-1 ==>(\forall int j; 0<=j && j< result &&
@       Q[j]==1 && C[j]==1; d[result]<d[j]);
@ loop_invariant result!=-1 ==>(\forall int j; result<=j && j< i &&
@       Q[j]==1 && C[j]==1; d[result]<=d[j]);

@ decreases Q.length-i;
@*/
    for(int i=0;i<Q.length;i++){
        if(Q[i]==1 && C[i]==1){
            if(result==-1)
                result=i;
            else if(d[i]<d[result])
                result=i;
        }
    }
    return result;
}


/*@
 @ ensures \result <==> (w[u][v]>0);
 @ public static pure model boolean neighbour(int [][]w, int u, int v);
@*/
}
```

Output of ESC/Java2

The output of ESC/Java2 for the second version of the JML specification for Dijkstra's Algorithm from Section 4.4.2 is listed.

```
ESC/Java version ESCJava-2.0.5
    [0.068 s 8591288 bytes]

Dijkstra ...
  Prover started:0.0080 s 10793400 bytes
    [0.496 s 11021856 bytes]

Dijkstra: DijkstraAlg(int[][], int, int) ...
------------------------------------------------------------------------
Dijkstrav3.java:92: Warning: Loop invariant possibly does not hold (LoopInv)
                while(existsMin(Q,C)){
                    ^
Associated declaration is "Dijkstrav3.java", line 56, col 5:
         @ loop_invariant (\forall int i; 0<=i && i<n; (\forall int j ...
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.

------------------------------------------------------------------------
Dijkstrav3.java:136: Warning: Loop invariant possibly does not hold (LoopInv)
                      for(int v =0; v<n; v++){
                                    ^
Associated declaration is "Dijkstrav3.java", line 98, col 5:
         @ loop_invariant (\forall int i; 0<=i && i<n; (\forall int j ...
           ^
Execution trace information:
```

```
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.

------------------------------------------------------------------------
Dijkstrav3.java:136: Warning: Loop invariant possibly does not hold (LoopInv)
                     for(int v =0; v<n; v++){
                                 ^
Associated declaration is "Dijkstrav3.java", line 123, col 5:
         @ loop_invariant (\forall int i; 0<=i && i<n; Q[i]==0 ==> C ...
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.

------------------------------------------------------------------------
Dijkstrav3.java:136: Warning: Loop invariant possibly does not hold (LoopInv)
                     for(int v =0; v<n; v++){
                                 ^
Associated declaration is "Dijkstrav3.java", line 122, col 12:
         @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i ...
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.
    Executed then branch in "Dijkstrav3.java", line 137, col 34.
    Executed else branch in "Dijkstrav3.java", line 146, col 9.
    Reached top of loop after 1 iteration in "Dijkstrav3.java", line 136, col 3.

------------------------------------------------------------------------
Dijkstrav3.java:136: Warning: Loop invariant possibly does not hold (LoopInv)
                     for(int v =0; v<n; v++){
                                 ^
Associated declaration is "Dijkstrav3.java", line 125, col 5:
         @ loop_invariant (\forall int i; 0<=i && i<n && C[i]==1 && i ...
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.
    Executed then branch in "Dijkstrav3.java", line 137, col 34.
    Executed else branch in "Dijkstrav3.java", line 146, col 9.
    Reached top of loop after 1 iteration in "Dijkstrav3.java", line 136, col 3.

------------------------------------------------------------------------
Dijkstrav3.java:136: Warning: Loop invariant possibly does not hold (LoopInv)
                     for(int v =0; v<n; v++){
                                 ^
Associated declaration is "Dijkstrav3.java", line 127, col 5:
         @ loop_invariant(\forall int i; 0<=i && i<n && C[i]==1 && i! ...
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.
    Executed then branch in "Dijkstrav3.java", line 137, col 34.
    Executed else branch in "Dijkstrav3.java", line 146, col 9.
    Reached top of loop after 1 iteration in "Dijkstrav3.java", line 136, col 3.

------------------------------------------------------------------------
Dijkstrav3.java:92: Warning: Negative loop variant function may not lead to loop exit
     (DecreasesBound)
               while(existsMin(Q,C)){
                     ^
Associated declaration is "Dijkstrav3.java", line 89, col 5:
         @ decreases (\num_of int i; 0<=i && i<n && Q[i]==1);
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.
```

```
    Short circuited boolean operation in "Dijkstrav3.java", line 137, col 16.
    Executed else branch in "Dijkstrav3.java", line 137, col 4.
    Reached top of loop after 1 iteration in "Dijkstrav3.java", line 136, col 3.

----------------------------------------------------------------------
Dijkstrav3.java:92: Warning: Loop variant function possible not decreased (Decreases)
              while(existsMin(Q,C)){
                   ^
Associated declaration is "Dijkstrav3.java", line 89, col 5:
         @ decreases (\num_of int i; 0<=i && i<n && Q[i]==1);
           ^
Execution trace information:
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 92, col 2.
    Reached top of loop after 0 iterations in "Dijkstrav3.java", line 136, col 3.
    Short circuited boolean operation in "Dijkstrav3.java", line 137, col 16.
    Executed else branch in "Dijkstrav3.java", line 137, col 4.
    Reached top of loop after 1 iteration in "Dijkstrav3.java", line 136, col 3.

----------------------------------------------------------------------
    [12.808 s 13197360 bytes] failed

Dijkstra: initArray(int[], int) ...
    [0.0090 s 13654096 bytes] passed

Dijkstra: existsMin(int[], int[]) ...
    [0.011 s 14268856 bytes] passed

Dijkstra: returnMinDist(int[], int[], int[]) ...
    [0.026 s 14117512 bytes] passed

Dijkstra: Dijkstra() ...
    [0.0020 s 14279208 bytes] passed

Dijkstra: neighbour(int[][], int, int) ...
    [0.0 s 14281944 bytes] passed immediately
  [13.353 s 14282824 bytes total]
8 warnings
```

B.3. TLA

The TLA specification of Dijkstra's Algorithm from Section 4.5 is listed.

```
1  ============================================================================
2
3  ------------ MODULE Dijkstra --------------
4
5  EXTENDS Naturals, Integers, TLC, FiniteSets
6
7  CONSTANT n, (* Maximal integer value for a vertex *)
8           maxWeight
9
10 (* PlusCal options (-termination) *)
11 (*
12 --algorithm Dijkstra{
13
14
15 variables
16     Vertex = {l \in 1..n : TRUE};
17     Edge = {e \in SUBSET(Vertex): Cardinality(e)=2};
18
19     weights = [e : Edge , w : 0..maxWeight];
20     Wgraph = {wg \in SUBSET(weights) : Cardinality(wg)= ((n*(n-1))\div 2) /\
21                   \A edge \in Edge: \E tup \in wg : tup.e =edge};
22     wgraph \in Wgraph;
```

```
23
24      s \in Vertex;
25
26      C = {s};
27      Q = Vertex;
28
29      dist = [ver \in Vertex |-> 0];
30      pre = [ver \in Vertex |-> -1];
31
32      u = -1;
33      vtup = <<{0,0},0>>;
34      v = 0;
35      length = -1;
36
37  {
38     while(Q \cap C # {}){
39        u := CHOOSE v1 \in Q \cap C: \A v2 \in Q \cap C : dist[v1]<=dist[v2];
40        Q := Q\{u};
41
42        v := 1;
43        while(v<=n){
44          if( v \in Q /\ \E tup \in wgraph : tup.e ={u,v} /\ tup.w >0){
45              vtup := CHOOSE tup \in wgraph: tup.e ={u,v};
46              length := dist[u] + vtup.w;
47
48              if(v \in C){
49                  if(length<dist[v]){
50                      dist[v]:= length;
51                      pre[v]:=u;
52                  }
53              }
54              else{
55                  dist[v]:= length;
56                  pre[v]:=u;
57                  C := C \cup {v};
58              };
59          };
60          v := v+1;
61        };
62      };
63  }
64  }
65  *)
66  \* BEGIN TRANSLATION
67  VARIABLES Vertex, Edge, weights, Weights, Wgraph, wgraph, s, C, Q, dist, pre, u, vtup, v, length,
        pc
68
69  vars == << Vertex, Edge, weights, Weights, Wgraph, wgraph, s, C, Q, dist, pre, u, vtup, v,
        length, pc >>
70
71  Init == (* Global variables *)
72          /\ Vertex = {l \in 1..n : TRUE}
73          /\ Edge = {e \in SUBSET(Vertex): Cardinality(e)=2}
74          /\ weights = [e : Edge , w : 0..maxWeight]
75          /\ Weights = {wg \in SUBSET(weights) : Cardinality(wg)= ((n*(n-1))\div 2) }
76          /\ Wgraph = {wg \in SUBSET(weights) : Cardinality(wg)= ((n*(n-1))\div 2) /\ \A edge \in
                Edge: \E tup \in wg : tup.e =edge}
77          /\ wgraph \in Wgraph
78          /\ s \in Vertex
79          /\ C = {s}
80          /\ Q = Vertex
81          /\ dist = [ver \in Vertex |-> 0]
82          /\ pre = [ver \in Vertex |-> -1]
83          /\ u = -1
84          /\ vtup = <<{0,0},0>>
85          /\ v = 0
86          /\ length = -1
```

```
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ IF Q \cap C # {}
               THEN /\ u' = (CHOOSE v1 \in Q \cap C: \A v2 \in Q \cap C : dist[v1]<=dist[v2])
                    /\ Q' = (Q\{u'})
                    /\ v' = 1
                    /\ pc' = "Lbl_2"
               ELSE /\ pc' = "Done"
                    /\ UNCHANGED << Q, u, v >>
         /\ UNCHANGED << Vertex, Edge, weights, Weights, Wgraph, wgraph, s, C,
                         dist, pre, vtup, length >>

Lbl_2 == /\ pc = "Lbl_2"
         /\ IF v<=n
               THEN /\ IF v \in Q /\ \E tup \in wgraph : tup.e ={u,v} /\ tup.w >0
                          THEN /\ vtup' = (CHOOSE tup \in wgraph: tup.e ={u,v})
                               /\ length' = dist[u] + vtup'.w
                               /\ IF v \in C
                                     THEN /\ IF length'<dist[v]
                                                THEN /\ dist' = [dist EXCEPT ![v] = length']
                                                     /\ pre' = [pre EXCEPT ![v] = u]
                                                ELSE /\ TRUE
                                                     /\ UNCHANGED << dist, pre >>
                                          /\ C' = C
                                     ELSE /\ dist' = [dist EXCEPT ![v] = length']
                                          /\ pre' = [pre EXCEPT ![v] = u]
                                          /\ C' = (C \cup {v})
                          ELSE /\ TRUE
                               /\ UNCHANGED << C, dist, pre, vtup, length >>
                    /\ v' = v+1
                    /\ pc' = "Lbl_2"
               ELSE /\ pc' = "Lbl_1"
                    /\ UNCHANGED << C, dist, pre, vtup, v, length >>
         /\ UNCHANGED << Vertex, Edge, weights, Weights, Wgraph, wgraph, s, Q,
                         u >>

Next == Lbl_1 \/ Lbl_2
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)


\* END TRANSLATION
------------------------------------------------------------------------------
\* WHAT TO CHECK

\* Algorithm terminates
Termination == <>(pc = "Done")

\* Algorithm satisfies output condition
neighbour(w,u1,v1) == \E tup \in w: tup.e ={u1,v1} /\ tup.w >0

SetOfLinkedVertices(C1,s1,w,n1) ==
    C1 \in SUBSET(Vertex) /\ s1 \in C1 /\
    (\A v1 \in C1: \A v2 \in Vertex: neighbour(w, v1,v2) => v2 \in C1) /\
    (\A v1 \in C1\{s1}: \E v2 \in C1: neighbour(w, v1,v2))

DefinesShortestDistances(dist1,pre1,C1,s1,w) ==
    dist[s1]=0 /\
    (\A v1 \in C1\{s1}: LET v2==pre1[v1] IN (neighbour(w, v1,v2) /\
        (LET vtup1==CHOOSE tup \in w: tup.e ={v1,v2} IN dist[v1]=dist[v2]+vtup1.w))) /\
    (\A v1,v2 \in C1: ( (LET vtup1==CHOOSE tup \in w: tup.e ={v1,v2} IN
        neighbour(w, v1,v2) =>dist[v1]<=dist[v2]+vtup1.w)))

```

```
IsShortestPath(C1,dist1,pre1,w,s1,n1) ==
    SetOfLinkedVertices(C1,s1,w,n1) /\
    DefinesShortestDistances(dist1,pre1,C1,s1,w)

IsCorrect == [](pc = "Done" => (IsShortestPath(C,dist,pre,wgraph,s,n)))
=============================================================
```

B.4. Alloy

B.4.1. Model 1

The first Alloy model of Dijkstra's Algorithm from Section 4.6.1 is shown.

```
module ShortestPath

open util/graph[Node] as graph
open util/ordering[State] as so
open util/integer as integer

sig Node { w: Node -> lone Int}
one sig Root extends Node {}

fact wSymm { all x,y: Node | w[x,y] = w[y,x] }
fact wDef { all x,y: Node | w[x,y]!=Int[0] and gt[w[x,y], Int[-1]] and lt[w[x,y], Int[5]]}

fun edge: Node -> set Node {
   { x,y: Node | some w[x,y] }
}

fact Graph {
     noSelfLoops[edge]
     undirected[edge]
     weaklyConnected[edge]
}

sig State {
  visited: set Node,
  unvisited: set Node,
  d: Node -> lone Int,
  prev: Node -> lone Node
}

pred init {
  let fs = so/first | {
      fs.visited = Root
      fs.unvisited = Node - Root
      fs.d[Root] = Int[0]
      no fs.prev[Root]
      all x: (Node - Root) | {x in edge[Root] =>{ fs.d[x] = w[Root,x] and fs.prev[x]=Root}
                              else {fs.d[x] = Int[-1] and no fs.prev[x]}}
      }
}

pred Dijkstra [pre, post: State] {
    some u: pre.unvisited |
       {pos[pre.d[u]] and
           all x: pre.unvisited - u | {pos[pre.d[x]] => { integer/lte[pre.d[u],pre.d[x]]}}}
     and all v: Node | {
        {
         v in edge[u] and v in pre.unvisited
         and (neg[pre.d[v]] or (pos[pre.d[v]] and add[pre.d[u],w[u,v]] < Int[pre.d[v]]))}
           =>{ post.d[v] = add[pre.d[u],w[v,u]] and post.prev[v]=u}
        else {post.d[v] = pre.d[v] and post.prev[v]= pre.prev[v]} }
      and post.visited = pre.visited + u
```

```
52       and post.unvisited = pre.unvisited - u
53  }
54
55  fact DijkstraAlg {
56       init
57       all s: State - so/last | let s' = so/next[s] |Dijkstra[s,s']
58  }
59
60  pred allVisited{ so/last.visited = Node}
61
62  run allVisited for 4 Node, 4 State, 5 Int
```

B.4.2. Model 2

In this Appendix the second Alloy model of Dijkstra's Algorithm from Section 4.6.2 is shown.

```
1   module ShortestPath_check
2
3   open util/graph[Node] as graph
4   open util/ordering[State] as so
5   open util/integer as integer
6
7
8   // graphs with weighted edges
9   sig Node { w: Node -> lone Int}
10  one sig Root extends Node {}
11
12  fact wSymm { all x,y: Node | w[x,y] = w[y,x] }
13  fact wDef { all x,y: Node | w[x,y]!=Int[0] and gt[w[x,y], Int[-1]] and lt[w[x,y], Int[5]]}
14
15  //define edge in terms of weighted edges
16  fun edge: Node -> set Node {
17     { x,y: Node | some w[x,y]}
18  }
19
20  fact niceGraph {
21       noSelfLoops[edge]
22       undirected[edge]
23       weaklyConnected[edge]
24  }
25
26  sig State {
27    known: set Node,
28    unknown: set Node,
29    d: Node -> lone Int,
30    prev: Node -> lone Node
31  }
32
33  pred init {
34    let fs = so/first | {
35        fs.visited = Root
36        fs.unvisited = Node - Root
37        fs.d[Root] = Int[0]
38        no fs.prev[Root]
39        all x: (Node - Root) | {x in edge[Root] =>{ fs.d[x] = w[Root,x] and fs.prev[x]=Root}
40                                else {fs.d[x] = Int[-1] and no fs.prev[x]}}
41        }
42  }
43
44  pred Dijkstra [pre, post: State] {
45      some u: pre.unvisited |
46          {pos[pre.d[u]] and
47              all x: pre.unvisited - u |{pos[pre.d[x]] => { integer/lte[pre.d[u],pre.d[x]]}}}
```

```
48      and all v: Node | {
49         {
50          v in edge[u] and v in pre.unvisited
51          and (neg[pre.d[v]] or ( pos[pre.d[v]] and add[pre.d[u],w[u,v]] < Int[pre.d[v]]))}
52            =>{ post.d[v] = add[pre.d[u],w[v,u]] and post.prev[v]=u}
53         else {post.d[v] = pre.d[v] and post.prev[v]= pre.prev[v]} }
54       and post.visited = pre.visited + u
55       and post.unvisited = pre.unvisited - u
56  }
57
58  fact DijkstraAlg {
59       init
60       all s: State - so/last | let s' = so/next[s] |Dijkstra[s,s']
61  }
62
63  sig Path { firstStep: Step }{firstStep.from=Root and firstStep.sumdist=Int[0]}
64
65  sig Step {
66      from, to: Node,
67      goal: one Node,
68      currdist: one Int,
69      sumdist: one Int,
70      nextStep: lone Step}
71     {to in edge[from] and currdist=w[from,to] and from=goal => no nextStep}
72
73
74  fact RelationshipBetweenSteps {
75      all curr: Step, next: curr.nextStep |{
76          next.from = curr.to and
77          next.goal = curr.goal and
78          next.sumdist = add[curr.currdist, curr.sumdist]}}
79
80  fact PathNotCyclic {
81      no s:Step| s.from in s.^nextStep.to}
82
83  fact allNodesBelongToSomeQueue {
84      all s:Step | one p:Path| s in p.firstStep.^nextStep}
85
86  assert NoShorterPath{
87      so/last.known = Node and
88      all n: Node| no p:Path |{
89          p.firstStep.goal=n and
90          some s:Step| {s in p.firstStep.*nextStep and s.from = s.goal and
                s.sumdist<so/last.d[s.goal]}}
91  }
92
93  check NoShorterPath for 49 Step, 16 Path, 4 Node, 4 State, 5 Int
```

B.5. VDM

In this Appendix the VDM specification of Dijkstra's Algorithm from Section 4.7 is listed.

```
1   class Dijkstra
2
3   values
4     public n: int = 3;
5     public maxWeight: int = 2;
6
7
8   types
9     public Vertex = int
10    inv v == v >= 1 and v <= n;
11
```

```
12    public Edge = set of Vertex
13    inv e == card(e)=2;
14
15    public Weight = int
16    inv w == w >= 0 and w <= maxWeight ;
17
18    public Wgraph :: e : Edge
19                     w : Weight;
20
21
22  instance variables
23    public wgraph : set of Wgraph :={};
24    public s : int :=0;
25    public C: set of Vertex :={};
26    public Q: set of Vertex := {1,...,n};
27    public dist: map Vertex to nat :={|->};
28    public pred: map Vertex to Vertex:={|->};
29
30
31  inv DefinesShortestDistofVisited(dist, pred, C, Q, s, wgraph) and VisitedImpliesConnected(C, Q)
        and LengthOfDist(C, Q, dist)
32
33  operations
34
35  public DijkstraAlg: set of Wgraph * Vertex ==> (set of Vertex * map Vertex to int * map Vertex to
        Vertex)
36  DijkstraAlg(g,startV) ==
37  (
38    s:=startV;
39    wgraph:= g;
40    C := C union {s};
41    dist := dist ++ {s|->0};
42    pred := pred ++ {s|->s};
43
44    while (C inter Q <> {}) do
45      (let u in set C inter Q be st (forall u1 in set C inter Q & dist(u)<=dist(u1)) in (
46      Q := Q\{u};
47
48      for all v in set Q do
49      (
50          let uv in set wgraph be st uv.e={u,v} in(
51          if uv.w>0 then
52           (
53             let length=dist(u)+uv.w in (
54               if v in set C then (
55                 if length < dist(v) then (
56                      dist := dist ++ {v|->length};
57                      pred := pred ++ {v|-> u};
58                 )
59               )
60               else(
61                    dist := dist ++ {v|->length};
62                      pred := pred ++ {v|-> u};
63                      C := C union {v};
64               )
65           )
66       ))
67      );
68    ));
69   return mk_(C,dist,pred);
70  )
71  pre validGraph(g) and validStart(startV)
72  post IsShortestPath(dist, pred, C, s, wgraph);
73
74
75  functions
76  /*Input condition*/
```

```
77  public validGraph: set of Wgraph -> bool
78  validGraph(wg)==
79  (let vert = power({1,...,n}) in (
80     let edges = {e | e in set vert & card(e)=2} in (
81        card(wg)=n*(n-1)/2 and (forall ed in set edges & (exists tup in set wg & tup.e =ed)) and
82        (forall tup in set wg & ((exists ed in set edges & tup.e =ed) and (exists weight in set
              {0,...,maxWeight} & tup.w =weight)))
83         )
84      )
85  );
86
87  public validStart: Vertex -> bool
88  validStart(v)==
89   v >= 1 and v <= n;
90
91
92  /* Output condition*/
93  public IsShortestPath: map Vertex to nat * map Vertex to Vertex * set of Vertex * Vertex * set of
        Wgraph -> bool
94  IsShortestPath(dist, pred, C, s, wgraph) ==
95      DefinesShortestDist(dist, pred, C, s, wgraph) and SetOfLinkedVertices(C,s,wgraph);
96
97  public DefinesShortestDist: map Vertex to nat * map Vertex to Vertex * set of Vertex * Vertex *
        set of Wgraph -> bool
98  DefinesShortestDist(dist, pred, C, s, wgraph) ==
99     (dist(s)=0 and
100    forall u in set C\{s} & (exists v in set C & (
101        pred(u)=v and neighbour(wgraph, u, v) and
102        let tup in set wgraph be st tup.e={u,v} in (dist(u)=dist(v)+tup.w)))
103    and
104    forall u1, v in set C & (neighbour(wgraph, u1, v) =>
105         let tup in set wgraph be st tup.e={u1,v} in (dist(u1)<=dist(v)+tup.w))
106 );
107
108 public SetOfLinkedVertices: set of Vertex * Vertex * set of Wgraph -> bool
109 SetOfLinkedVertices(C,s,wgraph) ==
110   (forall u in set C & (forall v in set {1,...,n} & neighbour(wgraph, u, v) => v in set C) and
111   forall u1 in set C\{s} & (exists v in set C & neighbour(wgraph, u1, v))
112 );
113
114 public neighbour: set of Wgraph * Vertex * Vertex -> bool
115 neighbour(wg, i, j) ==
116     exists tup in set wg & (tup.e={i,j} and tup.w>0);
117
118
119 /* Invariant */
120 public DefinesShortestDistofVisited: map Vertex to nat * map Vertex to Vertex *set of Vertex *
        set of Vertex * Vertex * set of Wgraph -> bool
121 DefinesShortestDistofVisited(dist, pred, C, Q, s, wgraph) ==
122    (let V={1,...,n}\Q in (
123    forall u in set C\{s} & (exists v in set V & (
124       pred(u)=v and neighbour(wgraph, u, v) and
125       let tup in set wgraph be st tup.e={u,v} in (dist(u)=dist(v)+tup.w)))
126    and
127    forall u1, v in set C & (neighbour(wgraph, u1, v) =>
128       let tup in set wgraph be st tup.e={u1,v} in (dist(u1)<=dist(v)+tup.w)))
129    );
130
131 public VisitedImpliesConnected: set of Vertex * set of Vertex -> bool
132 VisitedImpliesConnected(C, Q) ==
133    (let V={1,...,n}\Q in (forall u in set V & u in set C)
134 );
135
136 public LengthOfDist: set of Vertex * set of Vertex * map Vertex to nat -> bool
137 LengthOfDist(C, Q, dist)==
138    (let V={1,...,n}\Q in (
139        forall u in set V & (forall v in set C inter Q & dist(u)<=dist(v)))
```

```
);


traces
 S1:
 let vert = power({1,...,n}) in (
    let edges = {e | e in set vert & card(e)=2} in (
        let weights = {0,...,maxWeight} in (
           let graphTuple = {mk_Wgraph(e,w) | e in set edges , w in set weights} in (
              let graph={wg| wg in set power(graphTuple) & (card(wg)=n*(n-1)/2 and (forall ed in
                   set edges & (exists tup in set wg & tup.e =ed)))} in (
                  let g in set graph in(
                     let startV in set {1,...,n} in (
                        DijkstraAlg(g,startV)))))))

end Dijkstra
```

B.6. Event - B

B.6.1. Context: VarDefinition

This Appendix shows the `context` from Section 4.8, where the data types for Dijkstra's Algorithm are defined.

```
CONTEXT
   VarDefinition

CONSTANTS
   n
   maxWeight
   Vertex
   Edge
   Weight
   Wgraph

AXIOMS
   axm1:  n = 5
   axm2:  maxWeight = 3
   axm3:  Vertex = 1..n
   axm4:  Edge = {e | e ∈ ℙ(Vertex) ∧ card(e)=2}
   axm5:  Weight = 0..maxWeight
   axm6:  Wgraph = Edge → Weight
END
```

B.6.2. Context: ProgramCounter

This Appendix shows the `context` from Section 4.8, where the program counter for Dijkstra's Algorithm is defined.

```
CONTEXT
   ProgramCounter

SETS
   PC

CONSTANTS
   setS
   chooseU
   chooseV
   selectNeighbour
```

```
12    updateV
13    Done
14
15  AXIOMS
16    axm1: partition(PC, {setS}, {chooseU}, {chooseV}, {selectNeighbour}, {updateV}, {Done})
17  END
```

B.6.3. Machine: Algorithm

This Appendix shows the `machine` from Section 4.8, where Dijkstra's Algorithm is specified.

```
1   MACHINE
2     Algorithm
3
4   SEES
5     ProgramCounter
6     VarDefinition
7
8   VARIABLES
9     wgraph
10    s
11    C
12    Q
13    dist
14    pre
15    pc
16    u
17    v
18    neighbour
19    length
20
21  INVARIANTS
22    inv1:  wgraph ∈ Wgraph
23    inv2:  s ∈ Vertex
24    inv3:  C ⊆ Vertex
25    inv4:  Q ⊆ Vertex
26    inv5:  dist ∈ Vertex → ℕ
27    inv6:  pre ∈ Vertex → Vertex ∪ {0}
28    inv7:  pc ∈ PC
29    inv8:  u ∈ Vertex ∪ {0}
30    inv9:  v ∈ Vertex ∪ {0}
31    inv10: neighbour ⊆ Vertex
32    inv11: length ∈ ℕ
33    inv12: pc ≠ setS ⇒ (s ∈ C)
34    inv13: pc ≠ setS ⇒(dist(s)=0 ∧ pre(s)=s)
35    inv14: pc = setS ⇒ (u = 0 ∧ v = 0 ∧ C = ∅ ∧ Q = Vertex ∧ neighbour = ∅ ∧ length = 0)
36    inv15: ∀ u1 · (u1 ∈ Vertex ⇒ (u1 ∉Q
37    ⇒ u1 ∈ C)) chooseU_1: pc = chooseU ⇒(∀ u1 · (u1 ∈ Vertex\Q ⇒
38          ∀ u2 · (u2 ∈ Vertex ⇒(wgraph({u1, u2})>0 ⇒ u2 ∈ C))))
39    chooseU_2: pc = chooseU ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
40    chooseU_3: pc = chooseU ⇒(∀ u1 · (u1 ∈ C\{s} ⇒
41        (∃ u2 · (u2 ∈ Vertex\Q ∧ pre(u1)=u2 ∧
42           wgraph({u1,u2})>0 ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
43    chooseU_4: pc = chooseU ⇒ (∀ u1 · (u1 ∈ Vertex\Q ⇒
44        (∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒ dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
45    chooseU_5: pc = chooseU ⇒ v = 0 ∧ u = 0 ∧ neighbour = ∅
46    selectN_1: pc = selectNeighbour ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
47        ∀ u2 · (u2 ∈ Vertex ⇒ (wgraph({u1, u2})>0 ⇒ u2 ∈ C))))
48    selectN_2: pc = selectNeighbour ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
49    selectN_3: pc = selectNeighbour ⇒(∀ u1 · (u1 ∈ C\{s} ⇒
50        (∃ u2 · (u2 ∈ Vertex\Q ∧ pre(u1)=u2 ∧
51            wgraph({u1,u2})>0 ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
52    selectN_4: pc = selectNeighbour ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
```

```
53          (∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒ dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
54      selectN_5: pc = selectNeighbour ⇒ (u ≠ 0 ∧ u ∉ Q ∧ neighbour = ∅ ∧ v = 0 ∧ u ∈ C)
55      chooseV_1: pc = chooseV ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
56          ∀ u2 · (u2 ∈ Vertex ⇒ (wgraph({u1, u2})>0 ⇒ u2 ∈ C))))
57      chooseV_1b: pc = chooseV ⇒ (∀ v1 · ((v1 ∉ neighbour ∧ v1∈ Q ∧ wgraph({u, v1})>0 ) ⇒ v1 ∈ C))
58      chooseV_2: pc = chooseV ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
59      chooseV_3: pc = chooseV ⇒(∀ u1 · (u1 ∈ C\{s} ⇒
60          (∃ u2 · (u2 ∈ Vertex\Q ∧ pre(u1)=u2 ∧
61              wgraph({u1,u2})>0 ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
62      chooseV_4: pc = chooseV ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
63          (∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒ dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
64      chooseV_4b: pc = chooseV ⇒ (∀ v1 · ((v1 ∉ neighbour ∧ v1 ∈ Q ∧ wgraph({u, v1})>0 ) ⇒
            dist(v1)≤dist(u)+wgraph({u,v1})))
65      chooseV_5: pc = chooseV ⇒ u ≠ 0 ∧ u ∉ Q ∧ u ∈ C
66      updateV_1: pc = updateV ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
67          ∀ u2 · (u2 ∈ Vertex ⇒ (wgraph({u1, u2})>0 ⇒ u2 ∈ C))))
68      updateV_1b: pc = updateV ⇒ (∀ v1 · ((v ≠ v1 ∧ v1 ∈ Q ∧ v1 ∉ neighbour ∧ wgraph({u, v1})>0 )
            ⇒ v1 ∈ C))
69      updateV_2: pc = updateV ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
70      updateV_3: pc = updateV ⇒(∀ u1 · (u1 ∈ C\{s} ⇒
71          (∃ u2 · (u2 ∈ Vertex\Q ∧ pre(u1)=u2 ∧
72              wgraph({u1,u2})>0 ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
73      updateV_4: pc = updateV ⇒ (∀ u1 · (u1 ∈ Vertex\(Q ∪ {u}) ⇒
74          (∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒ dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
75      updateV_4b: pc = updateV ⇒ (∀ v1 · ((v ≠ v1 ∧ v1∈ Q ∧ v1 ∉ neighbour ∧ wgraph({u, v1})>0 )
            ⇒ dist(v1)≤dist(u)+wgraph({u,v1})))
76      updateV_5: pc = updateV ⇒ u≠ 0 ∧ v ≠ 0 ∧ u ∉ Q ∧ u ∈ C ∧ v ∉ neighbour ∧ length=dist(u) +
            wgraph({u,v})
77      goal1: pc = Done ⇒(∀ u1 · (u1 ∈ C ⇒
78          ∀ u2 · (u2 ∈ Vertex ⇒ (wgraph({u1, u2})>0 ⇒ u2 ∈ C))))
79      goal2: pc = Done ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ ∃ u2 · (u2 ∈ C ∧ wgraph({u1, u2})>0)))
80      goal3: pc = Done ⇒(∀ u1 · (u1 ∈ C\{s} ⇒ (∃ u2 · (u2 ∈ C ∧ pre(u1)=u2 ∧ wgraph({u1,u2})>0
81              ∧ dist(u1) = dist(u2)+wgraph({u1,u2})))))
82      goal4: pc = Done ⇒ (∀ u1 · (u1 ∈ C ⇒ (
83          ∀ u2 · (u2 ∈ C ⇒ (wgraph({u1,u2})>0 ⇒ dist(u2)≤dist(u1)+wgraph({u1,u2}))))))
84
85
86  EVENTS
87    INITIALISATION:
88      THEN
89        act1:  wgraph:∈ Wgraph
90        act2:  s :∈ Vertex
91        act3:  C :=∅
92        act4:  Q := Vertex
93        act5:  dist := λi · i ∈ Vertex | 0
94        act6:  pre := λi · i ∈ Vertex | 0
95        act7:  pc := setS
96        act8:  u := 0
97        act9:  v := 0
98        act10: neighbour := ∅
99        act11: length := 0
100     END
101
102   SetStart:
103     WHERE
104       grd1:  pc = setS
105       grd2:  Q = Vertex
106     THEN
107       act1:  C := {s}
108       act2:  pre(s):=s
109       act3:  dist(s):=0
110       act4:  pc := chooseU
111     END
112
113   Termination:
114     WHERE
115       grd1:  pc = chooseU
```

```
      grd2:  Q ∩ C = ∅
    THEN
      act1:  pc := Done
    END

ChooseNextU:
    ANY
      u1
    WHERE
      grd1:  pc = chooseU
      grd2:  Q ∩ C ≠ ∅
      grd3:  u1 ∈ Q ∩ C ∧ (∀ u2 · (u2 ∈ Q ∩ C ⇒ dist(u1)≤dist(u2)))
    THEN
      act1:  u := u1
      act2:  Q := Q\{u1}
      act3:  neighbour := ∅
      act4:  pc := selectNeighbour
    END

SelectNeighboursFromU:
    ANY
      neighbours1
    WHERE
      grd1:  pc = selectNeighbour
      grd2:  neighbours1 = { u1 | u1 ∈ Vertex ∧ u1 ∈ Q ∧ wgraph({u,u1})>0}
      grd3:  u ≠ 0
    THEN
      act1:  neighbour := neighbours1
      act2:  pc := chooseV
    END

ChooseVfromNeighbours:
    ANY
      v1
    WHERE
      grd1:  pc = chooseV
      grd2:  neighbour ≠ ∅
      grd3:  v1 ∈ neighbour
      grd4:  u≠0
    THEN
      act1:  v := v1
      act2:  neighbour := neighbour\{v1}
      act3:  length := dist(u)+wgraph({u,v1})
      act4:  pc := updateV
    END

TerminationOnV:
    WHERE
      grd1:  pc = chooseV
      grd2:  neighbour = ∅
    THEN
      act1:  u:=0
      act2:  v:=0
      act3:  pc := chooseU
    END

VinC_LengthUpdate:
    WHERE
      grd1:  pc = updateV
      grd2:  v ∈ C ∧ v ∈ Q
      grd3:  length < dist(v) ∧ wgraph({u,v})>0 ∧
             length =dist(u)+wgraph({u,v})
      grd4:  v≠0
    THEN
      act1:  dist(v) := length
      act2:  pre(v) := u
      act3:  length := 0
```

```
183         act4:  pc := chooseV
184       END
185
186    VinC_NolengthUpdate:
187       WHERE
188         grd1:  pc = updateV
189         grd2:  v ∈ C ∧ v ∈ Q
190         grd3:  length ≥ dist(v) ∧ wgraph({u,v})>0
191         grd4:  v≠0
192       THEN
193         act1:  length := 0
194         act2:  pc := chooseV
195       END
196
197    VnotinC:
198       WHERE
199         grd1:  pc = updateV
200         grd2:  v ∉ C ∧ v ∈ Q ∧ wgraph({u,v})>0
201         grd3:  v ≠ 0
202       THEN
203         act1:  dist(v) := length
204         act2:  pre(v) := u
205         act3:  C := C ∪ {v}
206         act4:  length := 0
207         act5:  pc := chooseV
208       END
209
210 END
```

B.6.4. Generated Proof Obligations

The following Figures show the generated proof obligations for Dijkstra's Algorithm from Section 4.8

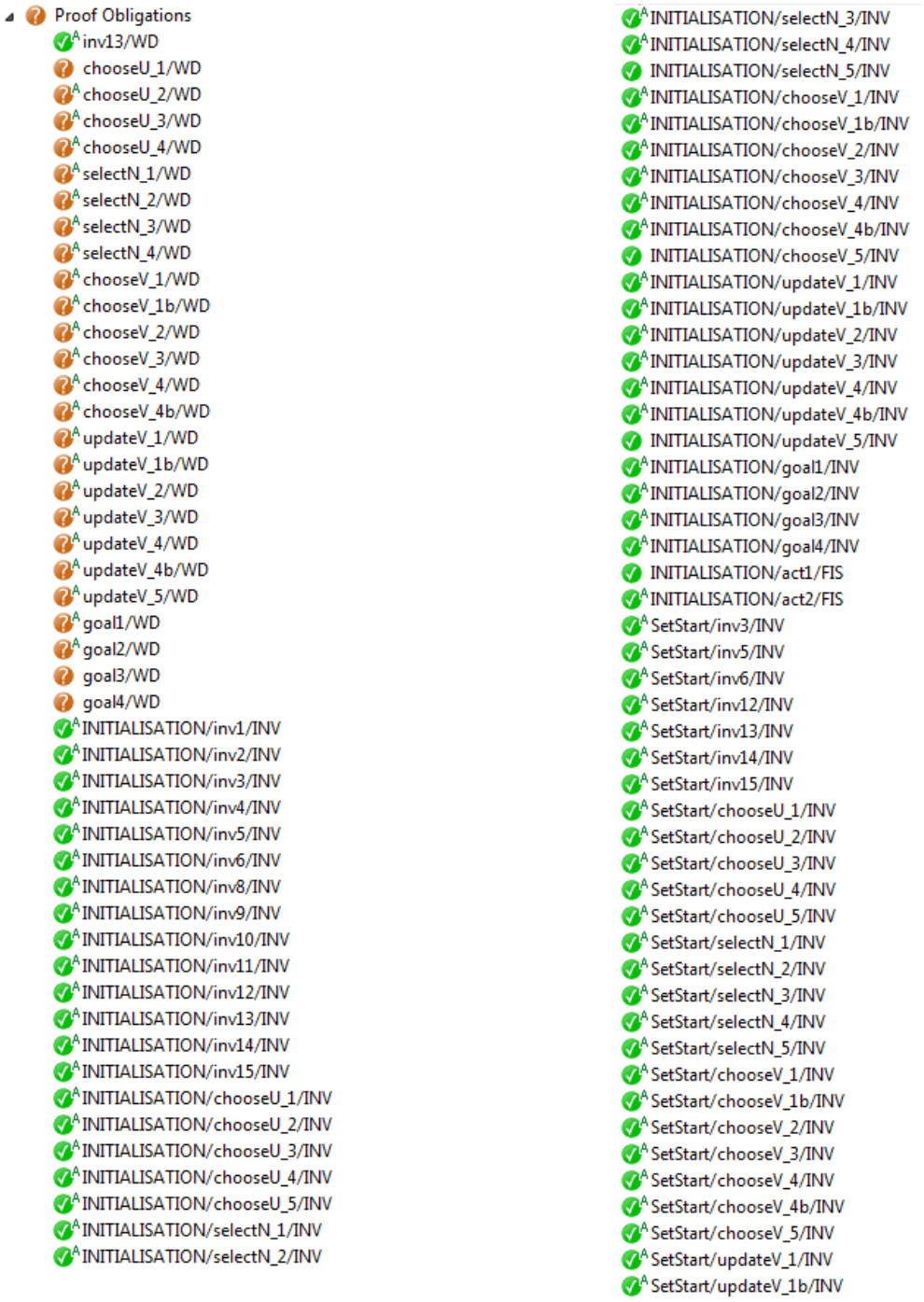


Figure B.1.: Generated proof obligations

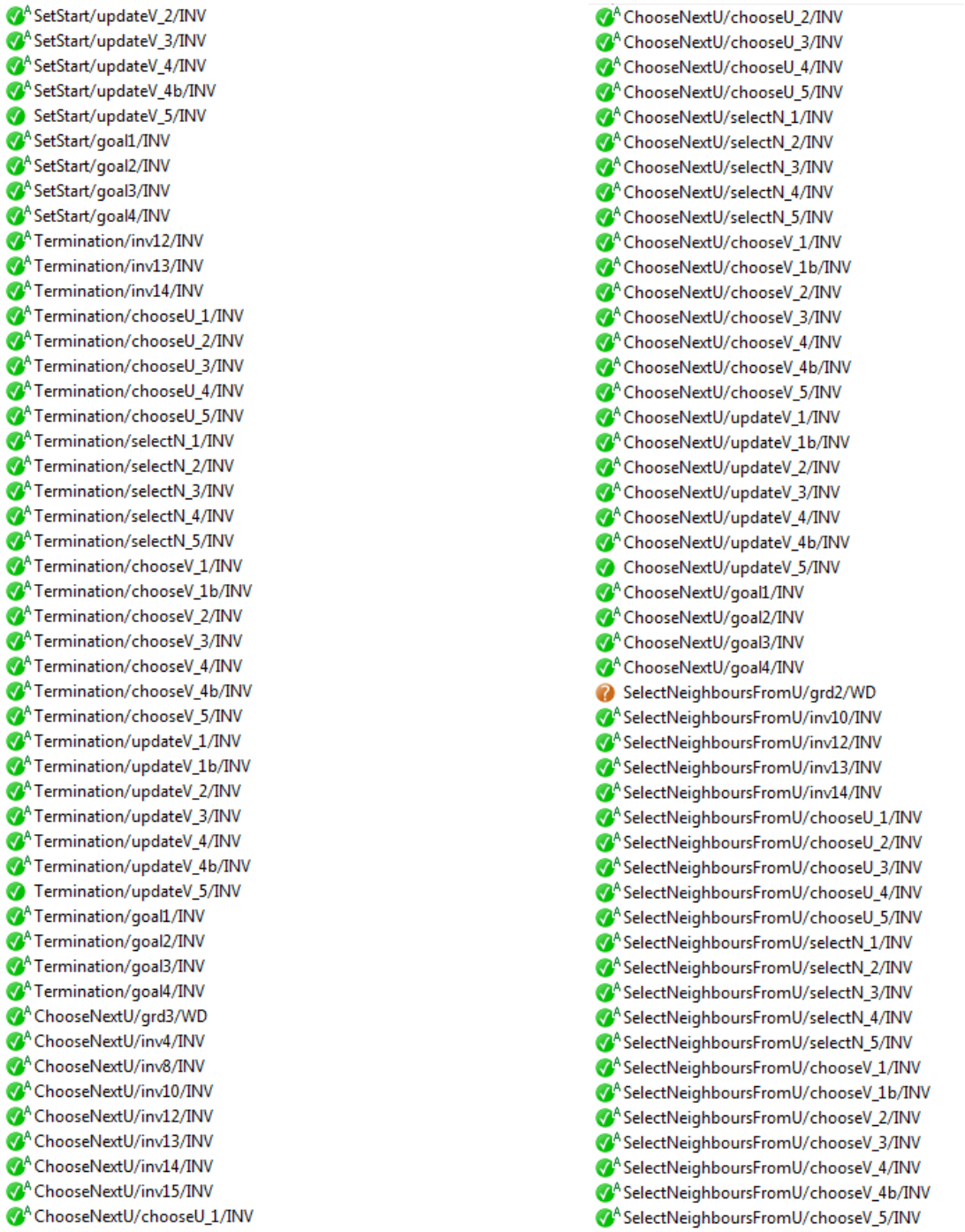


Figure B.2.: Generated proof obligations - continued

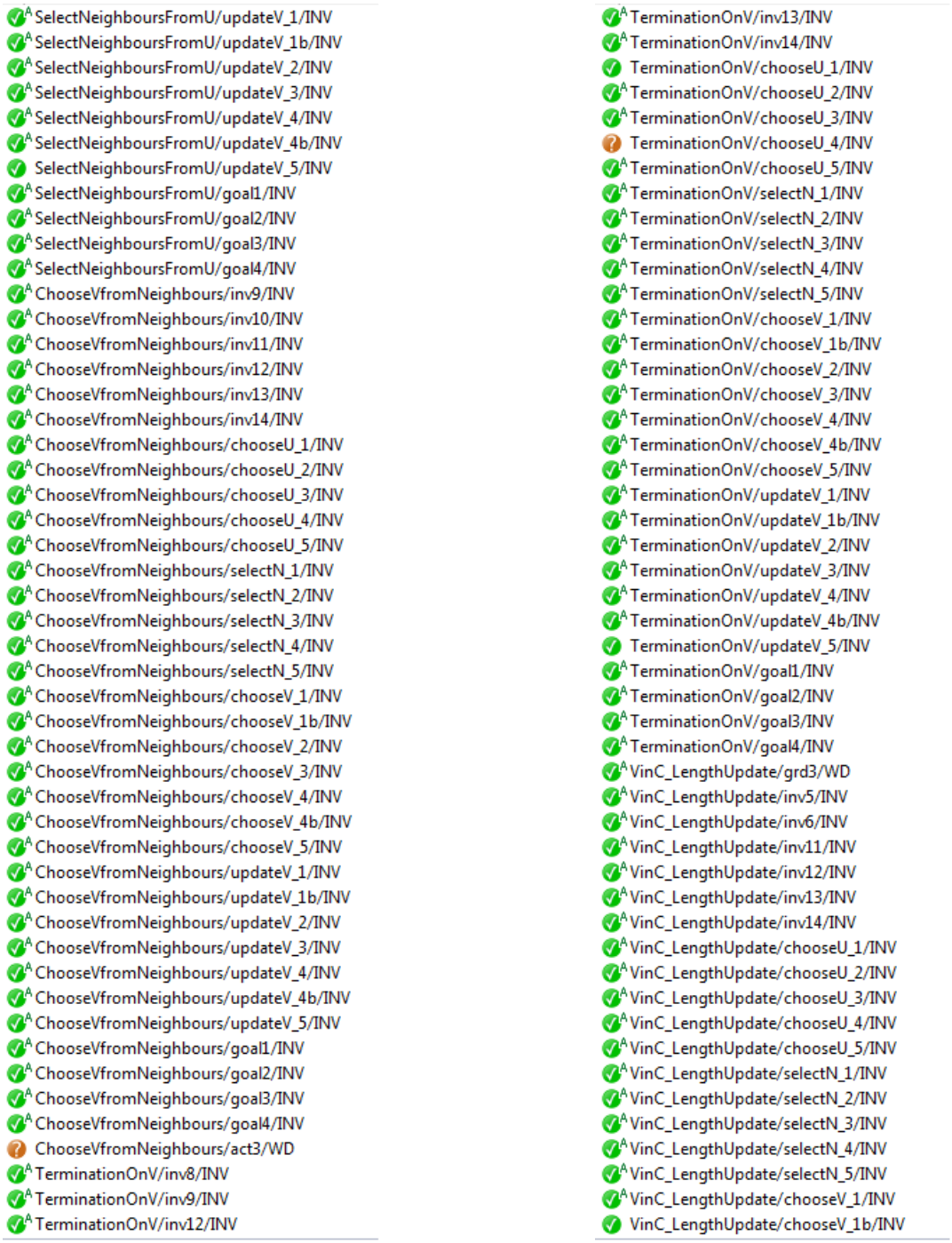


Figure B.3.: Generated proof obligations - continued

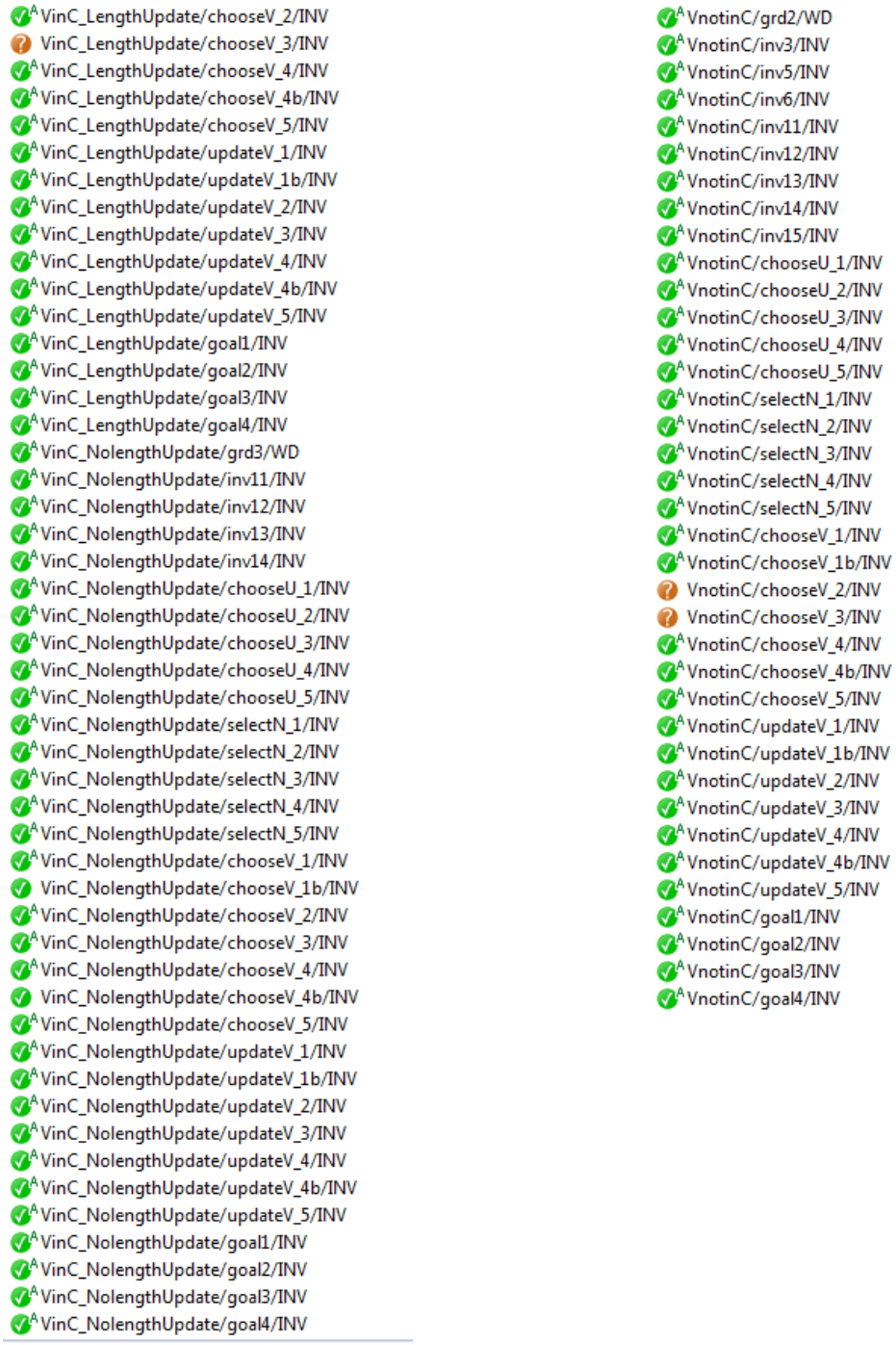


Figure B.4.: Generated proof obligations - continued