JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Alexander Brunhuemer

Submitted at
RISC
Research Institute for
Symbolic Computation

Supervisor
Univ.-Prof. DI Dr.
Franz Winkler

Co-Supervisor
A.Univ.-Prof. DI Dr.
Wolfgang Schreiner

September 2017

Validating the Formalization of Theories and Algorithms of Discrete Mathematics by the Computer-Supported Checking of Finite Models

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Technische Mathematik

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

Abstract

The goal of this Bachelor's thesis is the formal specification and implementation of central theories and algorithms in the field of discrete mathematics by using the RISC Algorithm Language (RISCAL), developed at the Research Institute for Symbolic Computation (RISC). This specification language and associated software system allow the verification of specifications, by using the concept of finite model checking. Validation on finite models is intended to serve as a foundation layer for further research on the corresponding generalized theories on infinite models.

This thesis results in a collection of specifications of exemplarily chosen formalized algorithms of set theory, relation and function theory and graph theory. The algorithms are specified in different ways (implicit, recursive and procedural), to emphasize the corresponding connections between them.

The evaluation and validation of implemented theories is demonstrated on Dijkstra's algorithm for finding a shortest path between vertices in a graph.

Kurzfassung

Das Ziel dieser Bachelorarbeit ist die formale Spezifikation und Implementierung von zentralen Theorien und Algorithmen im Bereich der diskreten Mathematik, mithilfe der RISC Algorithm Language (RISCAL), die am Research Institute for Symbolic Computation (RISC) entwickelt wurde. Diese Spezifikationssprache und das dazugehörige Software-System erlauben die Verifizierung von Spezifikationen mittels dem Konzept des Model-Checkings auf endlichen Bereichen. Die Validierung auf endlichen Modellen soll als Grundstein zur weiteren Untersuchung auf verallgemeinerten Theorien auf unendlichen Domänen dienen.

Diese Arbeit resultiert in einer Sammlung von Spezifikationen über beispielhaft ausgewählte, formalisierte Algorithmen der Mengenlehre, Relationen- und Funktionentheorie, sowie Graphentheorie. Die Algorithmen sind auf verschiedene Weisen spezifiziert (implizit, rekursiv und prozedural), um die entsprechenden Zusammenhänge zwischen diesen hervorzuheben.

Die Auswertung und Validierung der implementierten Theorien wird anhand von Dijkstras Algorithmus zum Finden eines kürzesten Pfades zwischen Knoten in einem Graphen durchgeführt.

Contents

1. Introduction and Background

More and more systems of our society are dependent on software components. Therefore it is without doubt of crucial importance that these components are free of errors or possible difficulties. However, problems can be easily overlooked when tested manually, especially with growing complexity of systems. Consequently it is just natural, to look out for ways to enable automated testing and reasoning. To accomplish this, it is absolutely necessary to formalize increasingly large parts of these systems. Hence it comes without surprise that these fields of research are trending and knowledge in this area is in great demand. Mathematicians and computer scientists are commissioned to model the important components and put them into theories and algorithms.

1.1. Formalization Leads to Automation

The huge technological progress in the last centuries provides us with many useful tools with opportunities to accelerate processes, which took lots of time before their invention. Computers allow us to calculate solutions for mathematical problems in a blink of an eye, which would have taken years to determine by hand. But to use this big advantages, it is a must-have to formalize the mathematical theories and put them into algorithms, because computers do not understand informal human intentions. They just execute what the programmer or user commands them to. The formal specifications have to be absolutely accurate, because even little mistakes lead to frustrating meaninglessness and the verification will fail for sure.

1.2. Verification of Formalization

Since it can be a really challenging task to formalize mathematical theories or algorithms, developers created (and still create) more and more tools to facilitate this process. However, if one wants to verify the formalization of a computer program, which operates on an unbounded domain of values, the only way to ensure correctness is via the generation of verification conditions. These are logical formulas whose validity warrants the correctness of the program with respect to its specification.

The usual way to generate these verification conditions, and in further steps proof the correctness of an algorithm, is as follows: At first the algorithm is specified formally and attached with some annotations, to guide the verification process. These two steps are typically done by human interaction. From the specification and additional annotations the program automatically generates the according verification conditions, from which the proof of correctness should follow with help of e.g. *theorem provers* or *model checkers*. Typically this requires some guidance of a human (tools supporting this step are called *interactive proving assistants*). However, where humans interact, mistakes can happen and typically most effort in the verification process is spent in proving wrong verification conditions (arising from too strong or weak loop invariants). Therefore it would be great to have a way to be safe that implemented conditions are correct.

Indeed it is a problem to make fully automatic verification possible. One possibility is to restrict the domain of values to a finite number instead of operating on an unbounded domain. To achieve that, one can apply model checkers that checks all possible executions of the program and consequently it is decidable. RISCAL [21] is a specification language and associated software system built exactly on this principle of decidability. RISCAL combines a mathematical modelling language with an algorithmic descriptive language, and has the purpose to support students and researchers in finding problems in specifications as quickly as possible. RISCAL operates on finite models; as a consequence all propositions in RISCAL are decidable.

1.3. Purpose and Results

The goal of this thesis is the formalization of theories from discrete mathematics [19] in the specification language RISCAL. This includes the specification and assignment of according meta-information of both the mathematical theories and the resulting algorithms. Furthermore the concepts are validated on small finite domains, which should work as a ground layer for further research and propositions on infinite models. This approach should not only give confidence that one is on the correct path, but also save much time to find errors in considerations, because in most cases errors in the specifications and annotations from the generalized concepts also appear in the finite domains.

The paper results in a collection of formalized mathematical theories and algorithms from discrete mathematics, including the specifications and according annotations Appendix A. A big focus in the elaborations lies on drawing the connections between different ways of describing an algorithm, which leads to a deeper understanding of the underlying theories. For most functions or algorithms we provide an implicit, a recursive and a procedural specification.

Finally the validation process is shown on Dijkstra's algorithm.

In Chapter 2 we start out with an overview on formal specifications, verifications and different approaches, how these can be performed. Also some tools for supporting this process are highlighted, especially the RISCAL environment is described more precise. Chosen results from the collection are presented more detailed in Chapter 3, following the strategy demonstrated in Section 3.1. The process of validation by means of model checking is illustrated on Dijkstra's algorithm in Chapter 4. Chapter 5 concludes and summarizes our results.

2. State of the Art

We start with an overview on formal specifications and verifications and describe concrete methods to accomplish these (semi-) automatically. Furthermore we learn about some tools and languages for specification; especially we take a closer look on the RISCAL environment.

2.1. Formal Specification and Verification

According to the Oxford Dictionary [24], specification is „an act of identifying something precisely or of stating a precise requirement“. In particular, formal specifications are specifications, expressed in a notation (syntax) with a semantics that is formally defined in the language of logic on the basis of well understood mathematical concepts. The mathematical ground layer uses theories from discrete mathematics, logic and algebra, which allows us to take advantage of techniques to check compliance to the rules of our language. So what is included in a specification? Alagar and Periyasamy [2] list the following items (even though they refer to software specifications, this can easily be generalised):

- *Properties of Objects:* Objects (simple or structured) associated with a defined type.
- *Correctness Condition:* A system should maintain some global correctness condition. This condition can be verified at any stage of the process. If it cannot be verified, then either the condition is too strong/weak or the stage does not suit your specification.
- *Observable Behavior:* A system's interaction with its environment. This also includes pre- or post-conditions of functions/procedures as well as invariants maintained by loops.

The main goal of formally specifying is to assure the possibility of validation and verification. There is a subtle difference between these two processes [14]:

- *Validation:* Are we trying to make the right thing? (i.e. is the specified product what the user needs?)
- *Verification:* Are we trying to make the thing right? (i.e. is our product conform with our specifications?)

Consequently, when we talk about verification, this always depends on a certain specification. Formal verification uses certain techniques to ensure correctness of the system with regard to the formal specification which fall into one of the categories of *model checking* or *automated reasoning*.

2.2. Model Checking

One approach to verify the correctness of a system is called *model checking* [5], where exhaustive checking over all possible states of a system is carried out. The big advantage of model checking is that this verification often can be done fully automatically. However, this method is only applicable to a bounded problem domain, since the model has to be finite (or at least one has to be able to represent all states finitely). If this is not the case, the model checking would not terminate. Therefore, when using model checking, one usually has to make some cutbacks, like restricting to a finite model (and check if the specification holds for this set of states) or renounce to check the complete system but instead only a critical core part, where model checking is possible.

A special form of model checking is the so-called *runtime assertion checking* [6], which allows to check the correctness of individually selected executions. In many specification or programming languages assertions are statements, which allow to test assumptions about the specified system or the program. E.g. in Java (1.4 and higher) a runtime assertion check can be implemented as follows:

```java
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // we know i % 3 == 2
    assert i % 3 == 2 : i;
    ...
}
```

Code 2.1: Java example to runtime assertion checking

If the assertion was wrong (in Java this could happen if the variable i was negative, since the remainder can happen to be negative in this case and the statement would yield false), an `AssertionException` would be raised and the error could be traced directly.

2.3. Automated Reasoning

Another, more general, approach to verification is automated reasoning [9], i.e. the automatic (or semi-automatic) construction of a mathematical proof of the correctness of a system. In [15] we find:

> „A problem being presented to an automated reasoning program consists of two main items, namely a statement expressing the particular question being asked called the problem's conclusion, and a collection of statements expressing all the relevant information available to the program — the problem's assumptions. Solving a problem means proving the conclusion from the given assumptions by the systematic application of rules of deduction embedded within the reasoning program. The problem solving process ends when one such proof is found, when the program is able to detect the non-existence of a proof, or when it simply runs out of resources."

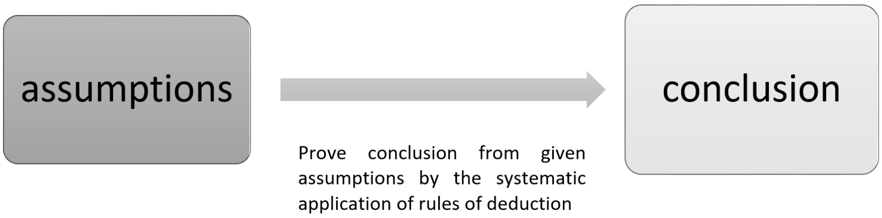

Figure 2.1.: The process of automated reasoning

To use this way of verification, one has to derive mathematical correctness obligations from the system and its specifications, the truth of which imply conformity of the system to the specification. To dismantle these obligations, automated theorem provers or interactive theorem provers are used. The difference between these two is, that interactive theorem provers need at least a little guidance in the proving process, whilst automated theorem provers work completely automatically.

RISC has developed several tools to support the process of theorem proving:

- *Theorema*: Theorema is a Mathematica package for computer supported mathematical theorem proving and theory exploration [4, 25]. It mainly concentrates on automated theorem proving, but also includes an interactive mode, in which the user is asked to provide minimalistic inputs in the proving process.

- *RISC ProofNavigator*: The RISC ProofNavigator is an interactive proof assistant for supporting formal reasoning about computer programs and computing systems. It is the core reasoning component of the *RISC ProgramExplorer* [17, 23], a computer-supported program reasoning environment, which was developed for supporting students in the process of learning the techniques of program verification [16, 20].

The focus of this thesis, however, will lie on RISCAL, a tool developed at RISC that is (currently) based on model-checking, and will be described in section 2.4.1

2.4. Software Specification Languages and Tools

In this section we will take a glance on some software specification languages and tools. Therefore I will mainly stick to results of the master's thesis of Daniela Ritirc, which compares some of these, and demonstrates their behaviour on specific examples [18].

Alloy Alloy [10, 11] is a language which is completely based on relations and allows to describe structures and their relationships. As described in [18], it is pretty complicated to define mathematical algorithms with Alloy (e.g. a loop is specified in Alloy by describing the changes of the variables during an iteration of the loop).

JML The Java Modeling Language (JML) [13] is an extension for the formal specifications of Java programs, which also allows the introduction of loop invariants and other annotations. However, as described in [18], it struggles with the complex semantics of Java, when it comes to expressive specifications.

TLA/PlusCal The Temporal Logic of Actions (TLA) [12] with the extension PlusCal allows to define mathematical algorithms in a very convenient way. Additionally it includes a model-checker, which yields an error and the complete path, when it violates properties of the algorithm. One essential disadvantage is the lack of the missing possibility to implement recursive algorithms.

VDM The Vienna Development Method (VDM) [3] includes mathematical objects like sets and functions and is therefore very helpful in defining mathematical algorithms. Moreover it allows to define recursive functions. Still it has its deficiencies in defining verification conditions, since it is only possible to specify system conditions and not e.g. for individual loops.

Event-B In specifications with Event-B [1], changes in variables are described with events, where one can restrict which event can be executed at which state of the algorithm. The language allows mathematical expressions like sets and functions, as well as invariants for each state. However when invariants are too complex, the provers cannot finish proofs, although the specification is correct.

Summarised, we can say, that each tool has its advantages, but still lack some important feature for our purposes. As a result the RISCAL was developed, which shall combine the useful aspects of the languages, to provide a powerful gadget.

2.4.1. RISCAL

RISCAL [21, 22] is aimed to support the verification of mathematical algorithms. Therefore it allows the developer to formulate the underlying mathematical theories (in the form of functions, predicates, and theorems) and, on the basis of these theories, high-level algorithms as they can be found in textbooks. To guarantee decidability, the language is based on a type system which ensures that all variable domains are finite at any time. However, the types may depend on unspecified numerical constants, which will be instantiated when starting the program (and further become decidable). In summary RISCAL validates the meaningfulness of definitions, the truthfulness of propositions and correctness of programs automatically, by evaluation of terms and formulas and executing programs over all possible inputs.

In addition to the support of verification, RISCAL provides a very intuitive way to describe the mathematical theories and algorithms. It supports most of the common special Unicode-characters, which are used in mathematics. Consequently the specification is much easier to read and somehow intuitive for the users, to understand the meaning behind the code.

The description of the mathematical and algorithmic theories consists of several parts:

- *Types:* With types, we introduce the mathematical objects we are working on in our further specifications. They build the base for our further definitions.

- *Predicates:* Predicates are boolean-valued functions which describe, if a given property is either true or false for given inputs of selected types.

- *Functions:* Functions are mappings from a given set of inputs to an according set of outputs. Functions can be specified in two ways:
  - *Implicit:* Implicit functions declare which predicates a result shall fulfil, but they do not give a way how to compute such a result. It is a descriptive approach to the desired solution.

  - *Explicit:* Explicit functions describe a constructive way to find such a result. Explicit functions may be recursively defined, provided that a termination measure ensures the well-definedness of the definition.

- *Theorems:* Theorems are special forms of predicates, for which all applications are expected to yield „true“ (if this is not the case, the evaluation will abort with an error message).

- *Procedures:* A procedure returns a value for a given input, after executing commands in sequence that update the values of variables. Like functions, procedures may be defined recursively.

The definition of a function, predicate, theorem or procedure may also include given preconditions (*requires*), postconditions (*ensures*) and termination measures (*decreases*) in form of annotations. Types, predicates and theorems shape the description of the mathematical theories, whilst functions and procedures form the algorithmic part. With all these points listed above, RISCAL also aims to give an understanding of the connections between the mathematical theories and algorithmic approaches. Detailed examples of RISCAL theories and algorithms are given in Appendix A.

3. Formal Specifications in Discrete Mathematics

In the following chapter we will discuss some specifications in the RISCAL environment of exemplarily chosen theories from discrete mathematics, specifically set theory, relation and function theory, as well as graph theory. The algorithms in this chapter are for the most part formulated in various versions (implicitly, recursively and procedural) to gain deeper understanding of the connections between these different ways.

3.1. General Strategy

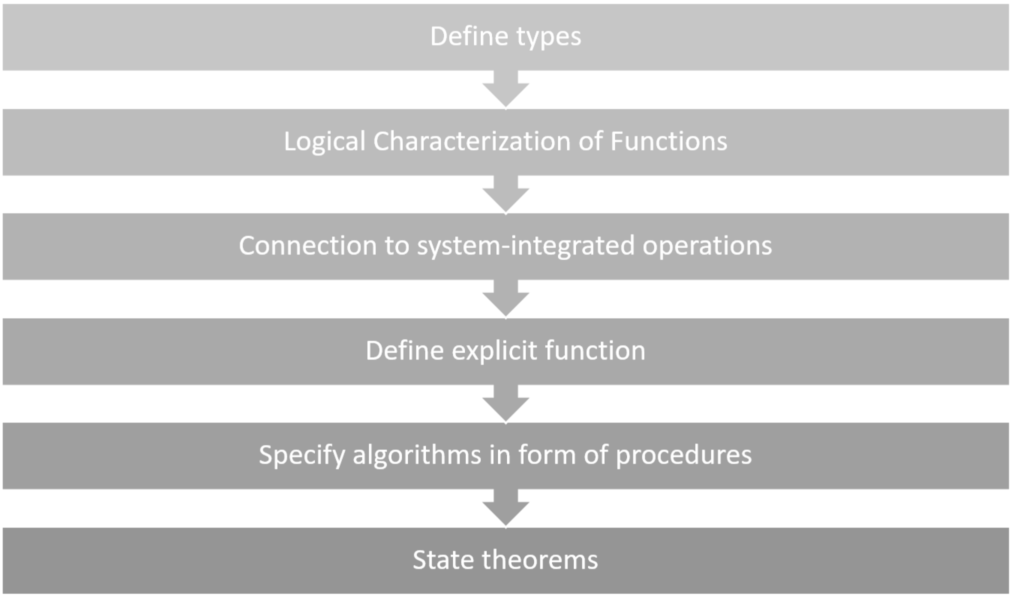

Figure 3.1.: General specification strategy

We apply a general strategy for the specification, which is demonstrated in this section on an introductory example, namely the union of two sets from the numbers between 0 and N.

For $a, b \subseteq \{0, \ldots, N\}$, we specify the equation:

$$a \cup b = \{x | x \in a \vee x \in b\} \tag{3.1}$$

After defining the type structures, we give a first definition of the operation implicitly, or by means of set quantifiers, if the function result is a set. Further we show the connections between built-in operators and our implemented version, if possible. Finally, we specify the functions concretely in form of recursions and procedures, and state theorems based on our implemented specifications. The complete strategy is illustrated in Figure 3.1.

3.1.1. Step 1: Type Definition

First we specify the *types* needed for our theories. In our case there are two kind of types, the elements included in a set as natural numbers from 0 to N, as well as the sets of these elements. The simplest form of a type definition would be `type id = T;` which introduces a name `id` for type `T`. Therefore for (3.1) we have:

```
val N:ℕ;
type elem = ℕ[N];
type set = Set[elem];
```

Specification 3.1: Type definition for $a \cup b$

The first statement creates a value `N` from the natural numbers ℕ, which will be instantiated (chosen by the user) when the evaluation of the specification is executed. `elem` is the type representation of one element from ℕ between 0 and `N` and type `set` is a set of `elem`, where `Set` is a language specific keyword, to introduce a set.

3.1.2. Step 2: Logical Characterization of Function

After building the fundamentals of our specifications we now define the predicates or functions. As described in section 2.4.1, there are different ways of defining mathematical theories and algorithms. The first way we are choosing is as an *implicit function*:

```
fun unionI(a:set,b:set):set =
  choose c:set with (∀ x:elem. x ∈ c ⇔ x ∈ a ∨ x ∈ b);
```

Specification 3.2: $a \cup b$ as an implicit function

The function `unionI` ensures to choose a set „`c`“, which fulfills the property, that any element „`x`“ of „`c`“ is either in set „`a`“ or „`b`“.

In case of set operations it is also possible to provide a definition of this operation explicitly by means of set quantifiers, without changing the underlying logical structure of the property:

```
fun unionS(a:set,b:set):set =
  { x | x:elem with (x ∈ a ∨ x ∈ b) };
```

Specification 3.3: $a \cup b$ by using set quantifiers

In the following sections we will apply the choose-notation, if the function result is not a set, otherwise the set quantifier notation is used.

`unionS` is the name of our function with parameters `a` and `b` of type `set`. The return value of the function is again of type `set`. In the second line we give the description of the desired result by set quantification and make use of the convenience to use many standard elements from the mathematical language. If necessary, it would also be possible to state some preconditions by using the keyword `requires`, for restriction of the input parameters, which is not needed here.

3.1.3. Step 3: Connections to Operations Provided by the Language

For functions, which are also implemented by built-in RISCAL operations, we show that both functions (the built-in and the user-defined function) provide the same result. If we can point this out, we are allowed to use the function provided by the system, which leads to massive performance improvements in further evaluations. To accomplish this, we state a theorem on equivalence of the two outcomes:

```
theorem unionT(a:set,b:set) ⇔
  unionS(a,b) = a ∪ b;
```

Specification 3.4: Check, if implemented ∪-operator equals our specification

For theorems we expect all applications to yield *true*, so if our function `unionS` would be different from the implemented ∪-operator for any input, the evaluation would be aborted with an according error message.

3.1.4. Step 4: Explicit Predicate or Function Definition

Next we define the same function in an *explicit* way, i.e. as a recursively defined function:

```
multiple fun unionR(a:set,b:set):set
  decreases |a|;
  ensures result = a ∪ b;
```

```
= choose x:elem with x ∈ a
    in ({x} ∪ unionR(a\{x},b))
  else b;
```

Specification 3.5: $a \cup b$ as an explicit function

The `multiple` keyword is required for recursively defined functions or predicates with non-deterministic semantics, such as the `choose` operator in this sample. With `decreases` we can define termination measures, to make sure our function terminates. In our illustration above we state, that with every call of our function the number of elements in set „`a`“ decreases, if this is not the case, the execution is aborted.

In the definition of our recursive functions, we can make sure that our developed recursive function equals the implicit function defined in Specification 3.2 (or in this case the system operator), by creating a postcondition (with keyword `ensures`) to verify, that both yield the same result. By use of these postconditions we can derive the connection between the different ways of describing a function, which helps in the process of understanding the theories.

3.1.5. Step 5: Specific Algorithms in Form of Procedures

Another option to specify the mathematical theory is as a specific step-by-step algorithm. By defining a sequence of commands we can give a clear recipe to find our desired solution. This is exactly what procedures in RISCAL are used for:

```
proc unionP(a:set,b:set):set
  ensures result = a ∪ b;
{
  var res:set := a;
  for x ∈ b do
    invariant res = (a ∪ forSet);
  {
      res := res ∪ {x};
  }
  return res;
}
```

Specification 3.6: $a \cup b$ as a procedure

Additionally to the options we already used for the explicit function, we now used an `invariant`, which states the crucial property for the correctness of the algorithm. Before and after every iteration of the loop, the union of set „`a`“ and `forSet` (which is equal to the set of all elements chosen in this loop so far) equals our result set at that moment. These loop invariants are not strictly necessary for successful execution of the algorithm, but support us in deeper understanding of the algorithms, provide help in finding errors in our specifications, and support subsequent proof-based verifications.

The postcondition (`ensures...`) again helps us to make sure that our result fits the previous specifications.

3.1.6. Step 6: Stating Theorems

Finally, after having formalized our theory in different ways, we use the definitions to formulate and check theorems, e.g.:

```
theorem unionSubsetT(a:set,b:set) ⇔
   a ⊆ (a ∪ b) ∧ b ⊆ (a ∪ b);
```

Specification 3.7: Verify that $a \subseteq (a \cup b) \wedge b \subseteq (a \cup b)$ holds

The result of the evaluation of this specification can be used as a confirmation, that the theorem is correct, at least on some finite domains. The strategy behind this activity is, that if errors occur, they often also occur on small bounded domains, thus we can find over checking the theorems on such domains.

3.2. Set Theory

The goal of this section is the specification of elementary parts of set theory in the RISCAL environment. We adhere to the strategy described in Section 3.1 and start with the type definition.

3.2.1. Type Definition

In [19, Chapter 2] we find: A set is defined as an unordered collection of objects. These objects are called *elements* of the set, and it is said, the set *contains* an element (if set A contains element a we write $a \in A$).

Like in the definition, we presuppose the existence of an element-operator and the structure of a set for containing elements itself in our RISCAL specifications. This approach is called *naive set theory* and can be looked up in [8]. More exact approaches would not be purposeful and beyond the scope of this paper.

As a first step, we only need to define of which type our elements are, for the other types we use implemented data structures:

```
val N:ℕ;
val Universe = 0..N;
type elem = ℕ[N];
type set = Set[elem];
```

Specification 3.8: Basic type definition for specifications of set theory

The definition of `N`, `elem` and `set` is the same es in section 3.1.1. With `Universe` we indicate the set which consists of all possible elements of type `elem`, which corresponds to `Universe` $= \{0, \ldots, N\}$.

3.2.2. Basic Set Operations

After defining the underlying type structure, we start with a basic relation on sets, namely the subset relation ($\subseteq$). For two sets A and B in $\{0, \ldots, N\}$ we have

$$A \subseteq B \Leftrightarrow \forall x \in A : x \in B \tag{3.2}$$

Which leads to the description:

```
pred isSubsetEq(a:set,b:set) ⇔
∀x ∈ a. x ∈ b;
```

Specification 3.9: $a \subseteq b$ as a predicate

RISCAL implements a subset operator, which allows us to verify the correctness of our specification, by means of equality over all possible inputs. This can be achieved by stating the corresponding theorem:

```
theorem subsetEqT(a:set,b:set) ⇔ isSubsetEq(a,b) ⇔ a ⊆ b;
```

Specification 3.10: Verify, if $a \subseteq b$ equals our implementation

The implemented operators are much faster, than our developed functions. Hence we now can use this equivalence in further specifications, e.g. in the postcondition, to accelerate the process of evaluation.

This can be directly observed in the next step, the explicit function definition:

```
multiple fun isSubsetEqR(a:set,b:set):Bool
   decreases |a|;
   ensures result = (a ⊆ b);
= choose x:elem with x ∈ a
```

```
    in (x ∈ b ∧ isSubsetEqR(a\{x},b))
  else true;
```

Specification 3.11: $a \subseteq b$ as an explicit function

If „`a`“ is not empty (else we return `true`), we choose an element of it and check, if it is in „`b`“. If this is not the case, the function returns `false`, else it calls the same function without the chosen element in set „`a`“. Further, our annotation „`decreases |a|;`“ (where $|\cdot|$ is the cardinality) verifies termination of our function.

For the procedural approach we proceed as follows:

```
proc isSubsetEqP(a:set,b:set):Bool
  ensures result = (a ⊆ b);
{
  var res:Bool := true;
  for x ∈ a do
    invariant res = (forSet ⊆ b);
  {
    res := res ∧ (x ∈ b);
  }
  return res;
}
```

Specification 3.12: $a \subseteq b$ as a procedure

The `invariant` ensures that after/before any loop iteration our interim result equals the truth content of `forSet ⊆ b`, with `forSet` corresponding to the set of all visited elements inside this loop.

We will not provide further specifications on basic set operations in this context and instead refer to Section 3.1 (where we specified $a \cup b$) as well as Appendix A, since these operations are all pretty similar and their specifications are following the same procedure. We presuppose the existence of the other operators in the listing below.

Based on our specifications of basic set operations and the connection to the system operators, we can define some known laws of set theory as theorems:

```
theorem commutativeUnionT(a:set,b:set) ⇔
  (a ∪ b = b ∪ a);
theorem associativeUnionT(a:set,b:set,c:set) ⇔
  (a ∪ (b ∪ c) = (a ∪ b) ∪ c);
theorem distributiveUnionT(a:set,b:set,c:set) ⇔
  (a ∪ (b ∩ c) = ((a ∪ b) ∩ (a ∪ c)));
```

```
theorem deMorganUnionT(a:set,b:set) ⇔
  complementS(a ∪ b) = complementS(a) ∩ complementS(b);
```

Specification 3.13: Stated theorems, on some basic laws of set operations

3.2.3. Cartesian Product

The Cartesian product of two sets A and B is a mathematical operation, which yields a set of pairs where the first element of the pair is part of set A and the second element of set B. Formally we have:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\} \tag{3.3}$$

Since we need the structure `pair` for the implementation of the Cartesian product, the following type definition is required:

```
type pair = Tuple[elem,elem];
```

Specification 3.14: Define a tuple of two elements of type `elem` as a `pair`

From (3.3) we directly gain the specification with set quantifiers:

```
fun cartesianProductS(a:set,b:set):Set[pair] =
  { p | p:pair with (p.1 ∈ a ∧ p.2 ∈ b) };
```

Specification 3.15: $a \times b$ as a predicate

With `p.1` and `p.2` we can access the first respectively the second entry of „`p`“ from type `pair` (we introduced this type in section 3.2.1). Again we can find an implemented operator for the Cartesian product, therefore we state the according theorem:

```
theorem cartesianProductT(a:set,b:set) ⇔
  cartesianProductS(a,b) = a × b;
```

Specification 3.16: Verify that $a \times b$ equals our specification

For a recursive implementation of the Cartesian product we choose an arbitrary element „`x`“ of set „`a`“ (or return an empty set if „`a`“ is empty). Further, we add all possible pairs with „`x`“ at position one and an element of set „`b`“ at position two into a set. The union of this set and the result of the same function without element „`x`“ in set „`a`“ gives the desired solution.

```
multiple fun cartesianProductR(a:set,b:set):Set[pair]
   decreases |a|;
   ensures result = (a × b);
= choose x:elem with x ∈ a
   in ({p | p:pair with p.1 = x ∧ p.2 ∈ b}
      ∪ cartesianProductR(a\{x},b))
   else ∅[pair];
```

Specification 3.17: $a \times b$ as an explicit function

The algorithmic description of the operation uses the same basic idea. Again, we choose an arbitrary element and couple it with all elements of the other set. Then we choose another element, et cetera. As before, we provide a loop invariant for verification of our interim results.

```
proc cartesianProductP(a:set,b:set):Set[pair]
   ensures result = (a × b);
{
   var res:Set[pair] := ∅[pair];
   for x ∈ a do
      invariant res = (forSet × b);
   {
      res := res ∪
         {p | p:pair with p.1 = x ∧ p.2 ∈ b};
   }
   return res;
}
```

Specification 3.18: $a \times b$ as a procedure

3.2.4. Cardinality

The cardinality of a set is defined as the number of its elements and can be determined by use of bijective functions. Generally a set has cardinality S, if and only if there exists a bijection from set $\{0, \ldots, S-1\}$ (or any other set with S elements) to this set. A function $f : A \to B$ is a bijection, if for any element of A exactly one element of B is associated by function f, and additionally all elements of B are in the image of f. We could also say f is bijective, if (and only if) it is injective as well as surjective. The cardinality of a set is usually denoted by $|\cdot|$.

Consequently, the introduction of some type for the definition of bijective functions and subsequently the definiton of cardinality are necessary.

```
val S = N+1;
type size = ℕ[S];
type map = Map[size,elem];
```

Specification 3.19: Type definitions for cardinality

Additionally, we first need some properties in form of predicates, before specifying the cardinality operation. As explained, we need a bijective mapping $f : \{0, \dots, S - 1\} \to A$, with $A \subseteq \{0, \dots, N\}$. For this reason there are two properties required:

- The function $f(x)$ is exclusively a mapping to set A for $x \in \{0, \dots, N\}$ (for all further specifications on cardinality this predicate will be our precondition):

```
pred isMapToA(s:size, f:map, a:set) ⇔
   (∀i:size with i >= s. f[i] = 0) ∧
   (∀i:size with i < s. f[i] ∈ a);
```

Specification 3.20: Check, if `f` is mapping to set `a`

- f is a bijective function

```
pred isInjective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ ∀x:size,y:size with x < s ∧ y < s.
      (f[x] = f[y]) ⇒ (x = y);

pred isSurjective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ ∀x:elem with x ∈ a. ∃y:size with y < s. f[y] = x;

pred isBijective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ isInjective(s,f,a) ∧ isSurjective(s,f,a);
```

Specification 3.21: Injectivity, surjectivity and bijectivity of mapping `f`

With these helper functions the implicit definition comes straight forward:

```
fun cardinalityS(a:set):size =
  choose s:size with ∃f:map
    with isMapToA(s,f,a). isBijective(s, f, a);
```

Specification 3.22: $|a|$ as an implicit function

Since $| \cdot |$ is an system-integrated cardinality operator, we can again determine if our implementation matches the system operator:

```
theorem cardinalityT(a:set) ⇔ cardinalityS(a) = |a|;
```

Specification 3.23: Verify that $|a|$ equals our implemented function

A much more intuitive way to describe cardinality are the recursive as well as the algorithmic implementation. We only have to choose an arbitrary element of the set, remove it and count how often this can be performed, before the set is empty.

```
multiple fun cardinalityR(a:set):size
  ensures result = |a|;
  decreases |a|;
=  choose x:elem with x ∈ a
    in (1 + cardinalityR(a\{x}))
  else 0;
```

Specification 3.24: $|a|$ as an explicit function

```
proc cardinalityP(a:set):size
  ensures result = |a|;
{
  var res:size := 0;
  for x ∈ a do
    invariant res = |forSet|;
  {
    res := res + 1;
  }
  return res;
}
```

Specification 3.25: $|a|$ as a procedure

3.3. Relation and Function Theory

In this section we deal with basic specifications of relation and function theory (particularly we will deal with binary relations) and adhere to the strategy described in Section 3.1. Detailled theories and descriptions on relation theory are provided in [19, Chapter 9].

3.3.1. Type Definition

Let `N`, `elem`, `set` and `pair` be defined as described in section 3.2.1, additionally let $A, B \subseteq \{0, \ldots, N\}$ be two sets. Then r is called a relation between A and B, if and only if r is a set of `pairs` (a, b), where $a \in A$ and $b \in B$. In fact, r is a relation, if and only if it is a subset of $A \times B$. This implies the type definitions and the additional predicate:

```
val N:ℕ;
val Universe = 0..N;
type elem = ℕ[N];
type set = Set[elem];
type pair = Tuple[elem,elem];
type relation = Set[pair];

pred isRelation(r:relation,a:set,b:set)
  ⇔ r ⊆ a × b;
```

Specification 3.26: Basic type definitions for relation and function theory

3.3.2. Composition of Relations

Let A, B, $C \subseteq \{0, \ldots, N\}$ be three sets. Let r be a relation between A and B, and s a relation between B and C. Then the composition $s \circ r$ is a relation between A and C and is defined as:

$$s \circ r = \{(a, c) \mid \exists b \in B : (a, b) \in r \wedge (b, c) \in s\} \tag{3.4}$$

The corresponding RISCAL specification can be defined as:

```
fun composeS(r:relation, s:relation, a:set, b:set, c:set):relation
  requires isRelation(r,a,b) ∧ isRelation(s,b,c);
= {p | p:pair with (p.1 ∈ a ∧ p.2 ∈ c
  ∧ (∃x ∈ b. (⟨p.1,x⟩ ∈ r ∧ ⟨x,p.2⟩ ∈ s )))};
```

Specification 3.27: $s \circ r$ as a predicate

In the preconditions we first check if both, „`r`“ and „`s`“, fulfill our `isRelation`-predicate and only then form the accordingly composed relation. A strategy to gain an explicit function as a recursion or a procedural algorithm can be found by taking an arbitrary pair $x \in r$ and create the set of pairs $\{p\}$, which are contained in $s \circ r$ with `x.1` at position `p.1`.

```
multiple fun composeR(r:relation,s:relation,a:set,b:set,c:set):relation
   requires isRelation(r,a,b) ∧ isRelation(s,b,c);
   ensures result = composeS(r,s,a,b,c);
   decreases |r|;
= choose x:pair with x ∈ r
      in ({p | p:pair with (p.1 = x.1
         ∧ p.2 ∈ {e | e:elem with ⟨x.2,e⟩ ∈ s})}
         ∪ composeR(r\{x},s,a,b,c))
  else ∅[pair];
```

Specification 3.28: $s \circ r$ as an explicit function

```
proc composeP(r:relation,s:relation,a:set,b:set,c:set):relation
   requires isRelation(r,a,b) ∧ isRelation(s,b,c);
   ensures result = composeS(r,s,a,b,c);
{
   var res:relation := ∅[pair];
   for x ∈ r do
      invariant res = composeS(forSet,s,a,b,c);
   {
      res := res ∪
         {p | p:pair with (p.1 = x.1
         ∧ p.2 ∈ {e | e:elem with ⟨x.2,e⟩ ∈ s})};
   }
   return res;
}
```

Specification 3.29: $s \circ r$ as a procedure

In the beginning of this section we claimed that the achieved result is again a relation. For verification of this assumption on our finite model, the following theorem is stated:

```
theorem compositionFromAtoC(r:relation, s:relation, a:set, b:set, c:set)
    requires isRelation(r,a,b) ∧ isRelation(s,b,c);
```

```
⇔ isRelation(composeS(r,s,a,b,c),a,c);
```

Specification 3.30: Verify that $s \circ r$ is a relation between `a` and `c`

3.3.3. Inverse Relations

From any relation r it is also possible to create another relation r^{-1} by simply switching the arguments. For $r = \{\langle a, b\rangle\}$:

$$r^{-1} = \{\langle b, a\rangle | \langle a, b\rangle \in r\} \tag{3.5}$$

Consequently this leads to the description with set quantifiers:

```
fun inverseS(r:relation, a:set, b:set):relation
   requires isRelation(r,a,b);
= {p | p:pair with ⟨p.2,p.1⟩ ∈ r};
```

Specification 3.31: r^{-1} as a predicate

We will not give the explicit function and the algorithmic description in this context, since this is going all along with the previous specifications, and instead refer to Appendix A.

If r is a subset of $A \times B$, r^{-1} is a subset of $B \times A$, called the *inverse* relation of r. However, $r \circ r^{-1}$ does not necessarily equal the identity relation. On the other hand the inverse satisfies:

$$(r^{-1})^{-1} = r \text{ and } (s \circ r)^{-1} = r^{-1} \circ s^{-1} \tag{3.6}$$

Finally, these propositions can again be checked in the theorems:

```
theorem isInverseARelationT(r:relation, a:set, b:set)
   requires isRelation(r,a,b);
⇔ isRelation(inverseS(r,a,b),b,a);
```

Specification 3.32: Verify that r^{-1} is a relation from `b` to `a`

```
theorem inverseOfInverseT(r:relation, a:set, b:set)
   requires isRelation(r,a,b);
⇔ inverseS(inverseS(r,a,b),b,a) = r;
```

Specification 3.33: Verify that $(r^{-1})^{-1} = r$

```
theorem composeInverseT(r:relation,s:relation,a:set,b:set,c:set)
  requires isRelation(r,a,b) ∧ isRelation(s,b,c);
⇔ inverseS(composeS(r,s,a,b,c),a,c)
  = composeS(inverseS(s,b,c),inverseS(r,a,b),c,b,a);
```

Specification 3.34: Verify that $(s \circ r)^{-1} = r^{-1} \circ s^{-1}$

Endorelations as Monoids

It is also possible to verify that endorelations on a set (a relation r between A and B is an endorelation $\Leftrightarrow A = B$) fulfill the properties of a monoid structure on the specified finite domain in RISCAL. For this we have to show:

1. *Associativity:* Let r, s, t be endorelations on the same set, then:

$$(r \circ s) \circ t = r \circ (s \circ t)$$

2. *Neutral element:* Let e be the relation on set A with $e = \{\langle a, a\rangle \mid a \in A\}$ and let r be any endorelation on set A. Then

$$e \circ r = r \circ e = r$$

Or the same in RISCAL as theorems:

```
theorem associativityEndoT(r:relation,s:relation,t:relation,a:set)
  requires isRelation(r,a,a) ∧ isRelation(s,a,a)
      ∧ isRelation(t,a,a);
⇔ composeS(composeS(r,s,a,a,a),t,a,a,a) =
    composeS(r,composeS(s,t,a,a,a),a,a,a);
```

Specification 3.35: Verify that endorelations are associative

```
fun identity(a:set):relation
= { ⟨ x,x ⟩ | x:elem with x ∈ a};

theorem composeIdentityT(r:relation,a:set, b:set)
  requires isRelation(r,a,b);
⇔ composeS(r,identity(b),a,b,b) = r
```

```
  ∧ composeS(identity(a),r,a,a,b) = r;
```

Specification 3.36: The identity relation is the neutral element in the monoid of endorelations

3.3.4. Transitive Closure on Endorelations

Before defining the transitive closure of an endorelation, we need some preliminary work to be provided:

Transitivity

An endorelation r on set A is called *transitive*, if whenever $\langle a, b\rangle \in r$ and $\langle b, c\rangle \in r$, then $\langle a, c\rangle \in r$ for all $a, b, c \in A$. Hence, this property is required as a predicate:

```
pred isTransitiveS(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ ∀ x ∈ r, y ∈ r. (x.2 = y.1) ⇒ ⟨ x.1,y.2 ⟩ ∈ r;
```

Specification 3.37: Transitivity of an endorelation

Transitive Closure

Let r be an endorelation on set A. A transitive relation s containing r such that s is a subset of every other transitive relation containing r, is called the *transitive closure* of r. So the transitive closure is the smallest (with respect to $\subset$) transitive set containing r. This leads to the following RISCAL specification:

```
pred isRelationSubsetAndTransitive(s:relation,r:relation,a:set)
⇔ isRelation(s,a,a) ∧ r ⊆ s ∧ isTransitiveS(s,a);
```

Specification 3.38: Check, if s is transitive and contains r

```
fun transitiveClosureS(r:relation,a:set):relation
  requires isRelation(r,a,a);
= choose s:relation with (isRelationSubsetAndTransitive(s,r,a) ∧
  (∀t:relation.
  isRelationSubsetAndTransitive(t,r,a) ⇒ s ⊆ t));
```

Specification 3.39: The transitive closure of r as an implicit function

Again we can find a recursive and algorithmic implementation of the transitive closure. With the postconditions and termination measures we confirm correctness and decidability of the functions.

```
val RelationUniverse = Universe × Universe;

multiple fun transitiveClosureR(r:relation,a:set):relation
   requires isRelation(r,a,a);
   ensures result = transitiveClosureS(r,a);
   decreases |RelationUniverse\r|;
= if isTransitiveS(r,a) then r
  else transitiveClosureR(
   r ∪ { ⟨x,y⟩ | x:elem,y:elem with
      (∃p∈r,q∈r. (x = p.1 ∧ y = q.2 ∧ p.2 = q.1)) }
      , a);
```

Specification 3.40: The transitive closure of r as an explicit function

```
proc transitiveClosureP(r:relation,a:set):relation
   requires isRelation(r,a,a);
   ensures result = transitiveClosureS(r,a);
{
   var res:relation := ∅[pair];
   var toCheck:relation := r;
   choose x ∈ toCheck do
   {
      for y ∈ res do
      {
         if x.1 = y.2 ∧ ¬(⟨y.1, x.2⟩ ∈ res) then
         {
            toCheck := toCheck ∪ { ⟨y.1, x.2⟩ };
         }

         if x.2 = y.1 ∧ ¬(⟨x.1, y.2⟩ ∈ res) then
         {
            toCheck := toCheck ∪ { ⟨x.1, y.2⟩ };
         }
      }

      res := res ∪ { x };
```

```
    toCheck := toCheck \ { x };
  }
  return res;
}
```

Specification 3.41: The transitive closure of r as a procedure

3.3.5. Functions as Specific Relations

A relation is named *function* if it suffices some additional conditions. There are two basic types of functions, *partial* functions and *total* functions.

Partial Function and Total Function

Let R be a relation between A and B. R is a total function, if and only if each element of set A is related to exactly one element of set B, with respect to R. On the other hand a function is called partial, if R is a total function on $A' \subseteq A$ and all elements of $A \setminus A'$ are not related to any elements of B.

Figure 3.2.: An example of a total function (left) and a partial function (right)

```
pred isPartialFunctionS(r:relation,a:set,b:set)
⇔ isRelation(r,a,b) ∧ ∀x ∈ r, y ∈ r. (x.1 = y.1) ⇒ x.2 = y.2;
```

Specification 3.42: Check, if r is a partial function between a and b

```
pred isFunctionS(r:relation,a:set,b:set)
⇔ isRelation(r,a,b) ∧ ∀z ∈ a. ∃f ∈ r.
    (f.1 = z) ∧ ∀g ∈ r. (g.1 = f.1) ⇒ g.2 = f.2;
```

Specification 3.43: Check, if r is a total function between a and b

Connection to Implemented Type „Map“

In RISCAL we can introduce a type `map`, which describes a mapping between two sets. Our goal is to develop the connections between our functions defined by relations and this type `map`. For this some preliminary specifications are necessary:

```
type map = Map[elem,elem];

pred isFunctionM(m:map, a:set, b:set)
⇔ ∀x ∈ a. m[x] ∈ b;

pred equal(r:relation, m:map, a:set, b:set)
   requires isFunctionS(r,a,b) ∧ isFunctionM(m,a,b);
⇔ ∀x ∈ a. ∃y ∈ r. (y.1 = x ∧ m[x] = y.2);
```

Specification 3.44: Compare our specified functions with the implemented type `Map`

First we defined the type as a `Map` between two elements of type `elem`. The predicate `isFunctionM` is required for verification, if our map „`m`“ is a mapping into set „`b`“ for any element in set „`a`“. Predicate `equal` checks if „`m`“ maps all elements of „`a`“ to the same element as our relation „`r`“ does.

With this specified, we can show that each relation with the function property induces a map and vice versa.

```
fun relToMap(r:relation, a:set, b:set):map
  requires isFunctionS(r,a,b);
  ensures isFunctionM(result,a,b) ∧
          equal(r,result,a,b);
= choose m:map with ∀x ∈ a. ∃y ∈ r. (x = y.1 ∧ m[x] = y.2) ;
```

Specification 3.45: Get the induced map of relation r

```
fun mapToRelation(m:map, a:set, b:set):relation
  requires isFunctionM(m,a,b);
  ensures isFunctionS(result,a,b) ∧ equal(result,m,a,b);
= {p | p:pair with (p.1 ∈ a ∧ p.2 = m[p.1])};
```

Specification 3.46: Get the induced relation of map m

3.4. Graph Theory

Our third big topic of discrete mathematics concerns with graph theory, in particular we will concentrate on Dijkstra's algorithm, for finding the shortest path between vertices. When we are talking about graphs in this section, we deal with undirected, unweighted and simple graphs. This means:

- there is only one edge allowed between each vertex
- no loops (edges from a vertex to itself) are allowed
- the distance between any pair of nodes is 1
- an edge is always bidirectional

The basic type definition and some additional functions and properties for directed graphs can be found in the specifications in Appendix A.

3.4.1. Type Definition and Required Predicates

What we need in the first place, are the basic types to define what a graph or a path indeed is. A (undirected and unweighted) graph consists of vertices and edges, which connect these vertices. In undirected simple graphs edges are generally described over a set of edges, where each edge is a set of two vertices. In our definition we choose our set of vertices as a subset of the set $\{0, \ldots, N\}$. We have to specify our edges as a general set of vertices, the restriction to two-element sets comes with the predicate `isUndirectedGraph`.

```
val N:ℕ;
type vertex = ℕ[N];
type vertices = Set[vertex];
type undirEdge = Set[vertex];
type undirEdges = Set[undirEdge];
type undirGraph = Tuple[vertices, undirEdges];
```

Specification 3.47: Basic type definitions for graph theory

```
pred isUndirectedGraph(g:undirGraph)
⇔ g.1 ≠ ∅[vertex] ∧ g.2 ⊆ Set(g.1,2);
```

Specification 3.48: Check, if set of vertices is not empty and set of edges only contains sets with two elements

Paths are another fundamental structure in graph theory, which are necessary for our algorithm. As described in [19, Chapter 10], „paths are sequences of edges, that begin at a certain vertex of a graph and travels from vertex to vertex along edges of the graph.“ Again we only consider simple paths, which means that it does not contain the same edge more than once, moreover we only allow that every vertex only occures once in the path, which is sufficient, since we are looking for the shortest one. Therefore we can use an array of edges with length N for storage.

```
type undirPath = Array[N,undirEdge];
```

Specification 3.49: Define the type path as array of edges

```
pred isPathInGraph(p:undirPath, g:undirGraph)
   requires isUndirectedGraph(g);
⇔ ∀m ∈ 0..N-1. (p[m] ∈ g.2) ∨ p[m] = ∅[vertex];
```

Specification 3.50: Check, if path is in graph g

To make sure that our array of edges fulfills the path properties we define predicates to check, if each vertex only occures once, and that the sequence of edges are adjacent. I.e. for any edge e_i follows, that e_{i+1} is connected with e_i.

```
// get number of edges within path, which include v
fun numberOfEdgesWithVertex(p:undirPath, v:vertex):ℕ[N]
= |{e| e:undirEdge with (∃n ∈ 0..N-1. (p[n] = e)) ∧ v ∈ e}|;

// check if vertices are at most once in the path
// start- and end-vertex have to be checked extra
pred isVertexOnceInPath(p:undirPath, start:vertex, end:vertex,
     v:vertices)
⇔ numberOfEdgesWithVertex(p,start) = 1
  ∧ numberOfEdgesWithVertex(p,end) = 1
  ∧ ∀v1 ∈ (v\{start,end}). numberOfEdgesWithVertex(p,v1) <= 2;

// check if the edges are adjacent (neighboured)
pred isEdgeAdjacent(e1:undirEdge, e2:undirEdge)
⇔ e1 ∩ e2 ≠ ∅[vertex] ∧ e1 ≠ e2;
```

Specification 3.51: Check, if path only contains each vertex once and successive edges are adjacent

Additionally it is required to verify if there are no gaps in our array, that all non-empty entries are unique and to get the length of a path. All these additional properties specified as predicates will not be provided here, instead they can be found specified in Appendix A.

Finally, after stating these restricting predicates, it is possible to check if a specific path is connecting certain start- and end-vertices in a given graph, or if it is even possible to connect these two vertices in it.

```
pred isPathBetweenVertices( p:undirPath, g:undirGraph,
                start:vertex, end:vertex)
  requires isUndirectedGraph(g)
     ∧ isVertexInSetOfVertices(start,g.1)
     ∧ isVertexInSetOfVertices(end,g.1)
     ∧ isPathRequirementsFulfilled(p)
     ∧ isPathInGraph(p,g);
⇔ (start = end ∧ isArrayEmpty(p)) ∨
  (start ≠ end ∧ (∃n:ℕ[N-1]. isArrayFilledToIndex(p,n)
 ∧ isVertexOnceInPath(p, start, end, g.1)
 ∧ ∀m ∈ 1..n. isEdgeAdjacent(p[m-1], p[m])));
```

Specification 3.52: Check if p is a path between start- and endvertex in graph g

```
pred isPathBetweenVerticesExisting(g:undirGraph, start:vertex,
                end:vertex)
  requires isUndirectedGraph(g)
     ∧ isVertexInSetOfVertices(start,g.1)
     ∧ isVertexInSetOfVertices(end,g.1);
⇔ ∃p:undirPath. isPathRequirementsFulfilled(p) ∧
     isPathInGraph(p,g) ∧
     isPathBetweenVertices(p, g, start, end);
```

Specification 3.53: Check if a path between start- and end-vertex is existing in graph g

3.4.2. Shortest Path

Let p be a path, which suffices the requirements from above. p is called a *shortest path* between start- and end-node, if for all paths q (which again suffice the requirement) between the same vertices applies, that the length of p is smaller or equal to the length of q. Note that a shortest path is not necessarily unique, since it can happen, that two different paths from *start* to *end* have the same length.

```
pred isShortestPath(g:undirGraph, start:vertex,
             end:vertex, p:undirPath)
  requires isUndirectedGraph(g)
    ∧ start ∈ g.1 ∧ end ∈ g.1
    ∧ isPathRequirementsFulfilled(p)
    ∧ isPathInGraph(p,g);
⇔ isPathBetweenVertices(p,g,start,end) ∧
  ∀q:undirPath with isPathRequirementsFulfilled(q)
    ∧ isPathInGraph(q,g)
    ∧ isPathBetweenVertices(q,g,start,end)
    . getLengthOfPath(p) <= getLengthOfPath(q);
```

Specification 3.54: Check if p is a *shortest path* between start- and endvertex in graph g

With this property, we can easily give an implicit version to find the shortest path between given start- and end-vertices in a certain graph. The function returns a tuple with a Boolean value and a path. The Boolean indicates, if a path was found, the path describes a shortest path, if one was found.

```
fun getShortestPath(g:undirGraph, start:vertex,
     end:vertex):Tuple[Bool,undirPath]
  requires isUndirectedGraph(g)
    ∧ isVertexInSetOfVertices(start,g.1)
    ∧ isVertexInSetOfVertices(end,g.1);
  ensures
    result.1 = isPathBetweenVerticesExisting(g,start,end)
    ∧ ((¬result.1) ∨
    (isPathBetweenVertices(result.2,g,start,end)
    ∧ isShortestPath(g,start,end,result.2)));
= choose p:undirPath with (isPathRequirementsFulfilled(p)
             ∧ isPathInGraph(p,g)
             ∧ isPathBetweenVertices(p,g,start,end)
             ∧ isShortestPath(g,start,end,p))
  in ⟨true,p⟩
  else ⟨false,Array[N,undirEdge](∅[vertex])⟩;
```

Specification 3.55: Check if p is a path between start- and endvertex in graph g

Dijkstra's Algorithm

Dijkstra's algorithm, published in 1959 by Edsger W. Dijkstra [7], is an algorithm conceived to find the shortest path between vertices in an arbitrary graph. Dijkstra's algorithm begins with setting the distance of the source code to zero, the distance to all other nodes is set to infinity (or in our implementation $N + 1$, since the maximum size of our path array is N). The algorithm repeatedly chooses the vertex, which is connected to the start vertex and not visited yet, with the least distance. The distance to the neighbours of the chosen vertex is compared with the stored distances, and if the new distance is smaller than before, the distance and predecessor of the neighbour is updated. The neighbours are now marked as connected, and are potential candidates for the next iteration. A detailled description of the algorithm can be found in [18, 19].

The algorithm terminates, since no vertex is visited twice and we are working with a finite number of vertices. The same return values as in the previous function are used. In the first place I will provide a version of the algorithm, without included invariants, for space and readability reasons. The invariants will be treated extra in Section 3.4.2.

A validation of the algorithm, can be found in Chapter 4.

```
proc dijkstra(g:undirGraph, start:vertex,
        end:vertex):Tuple[Bool,undirPath]
   requires isUndirectedGraph(g)
     ∧ start ∈ g.1 ∧ end ∈ g.1;
   ensures
     (result.1 = isPathBetweenVerticesExisting(g,start,end))
     ∧ ((¬result.1) ∨
     (isPathBetweenVertices(result.2,g,start,end)
     ∧ isShortestPath(g,start,end,result.2)));
{
   var res:undirPath := Array[N,undirEdge](∅[vertex]);
   var found:Bool := false;

   // initialize
   var dist:Map[vertex,ℕ[N+1]] := Map[vertex,ℕ[N+1]](N+1);
   var prev:Map[vertex,ℕ[N+1]] := Map[vertex,ℕ[N+1]](N+1);
   var conn:vertices := {start};
   dist[start] := 0;
   prev[start] := start;
   var Q:vertices := g.1;
   var visited:vertices := ∅[vertex];
```

```

  // loop over all unvisited vertices and choose the
  // one with the least distance
  choose q ∈ (Q ∩ conn) with
        (∀v ∈ (Q ∩ conn). dist[q] <= dist[v]) do
    decreases |Q|;
  {
    // if q = end we have found the path and can stop
    if(q = end) then
    {
      Q := ∅[vertex];
    } else {
      visited := visited ∪ {q};
      Q := Q\{q};
      // check unvisited neighborhood of chosen vertex
      var V:vertices := getNeighborhood(q,g);
      for n ∈ (V ∩ Q) do
      {
        var alt:ℕ[N+1];
        // if distance is already N+1, don't raise it
        if dist[q] = N+1 then alt := N+1;
        // save alternativ distance
        else alt := dist[q] + 1;
        // if distance is smaller, then save new path
        if n ∈ conn then
        {
          if alt < dist[n] then
          {
            dist[n] := alt;
            prev[n] := q;
          }
        }
        else
        {
          dist[n] := alt;
          prev[n] := q;
          conn := conn ∪ {n};
        }
      }
    }
```

```
    }
    // if path found, then create path array
    if dist[end] ≠ N+1 then
    {
      found := true;
      var index:ℕ[N];
      var u:vertex := end;
      for index := dist[end]; index > 0; index := index - 1
      do {
        res[index - 1] := {prev[u],u};
        u := prev[u];
      }
    }
    return ⟨ found, res ⟩;
}
```

Specification 3.56: Dijkstra's algorithm (without invariants)

Invariants

There are two different invariants needed, first the invariants for the outer `choose`-loop, second for the nested `for`-loop. These invariants (amongst other things) confirms, that before/after any iteration the distance to any visited node, is the shortest distance possible in the set of visited nodes. The detailed specifications can be seen below:

```
  ⋮
  choose q ∈ (Q ∩ conn) with
      (∀v ∈ (Q ∩ conn). dist[q] <= dist[v]) do
    decreases |Q|;
    // all neighbours of visited nodes are connected
    invariant ∀v ∈ visited.
          ∀neigh ∈ getNeighborhood(v,g).
          neigh ∈ conn;
    // all connected vertices (except start) have a
    // connected neighbor
    invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
          ∃v2:vertex with (v2 ∈ conn).
          v2 ∈ getNeighborhood(v,g);
    // defines shortest dist of visited nodes
    invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
```

```
            ∃v2 ∈ visited. (prev[v] = v2
                ∧ v2 ∈ getNeighborhood(v,g)
                ∧ dist[v] = dist[v2] + 1);
    invariant ∀v:vertex with v ∈ conn.
            (∀v2:vertex with v2 ∈ conn.
            (v2 ∈ getNeighborhood(v,g) ⇒
             dist[v] <= dist[v2] + 1));
    // visited implies connected
    invariant ∀v ∈ visited. (v ∈ conn);
    // connected implies defined predecessor and distance
    invariant ∀v ∈ conn. (prev[v] ≠ N+1 ∧ dist[v] ≠ N+1);
    // Distance of visited nodes is shorter than the
    // distance of unvisited but connected nodes
    invariant ∀v ∈ visited. (∀v2 ∈ (Q ∩ conn).
            (dist[v] <= dist[v2]));
  ⋮
```

Specification 3.57: Invariants for outer loop in Dijkstra's algorithm

The invariants for the inner for-loop are basically the same as for the outer loop. Only for the check, if all neighbours of the visited nodes are connected, an exception has to be implemented. This statement does not hold for q, the vertex, which is checked in this iteration:

```
  ⋮
       for n ∈ (V ∩ Q) do
         ⋮
         // all neighbours of visited nodes are connected
         invariant ∀v ∈ visited with v ≠ q.
               ∀neigh ∈ getNeighborhood(v,g).
               neigh ∈ conn;
         ⋮
  ⋮
```

Specification 3.58: Invariants for inner loop in Dijkstra's algorithm

4. Evaluation and Validation

The RISCAL environment is a powerful tool, when it comes to validation of specifications. By stating suitable pre- and postconditions, termination measures and loop invariants in form of annotations, the system provides big support in verification of the correctness of algorithms and specifications. When errors occur in the definitions of the annotations, they often can be revealed, by running the model checks on the specification. This really saves one's nerves, since finding errors in wrong declared conditions is without doubt absolutely frustrating.

In this chapter the validation process in the RISCAL environment is demonstrated on a concrete problem, which was formally specified in Chapter 3. For this purpose Dijkstra's algorithm will hold as an example.

The validation and evaluation process splits into two parts:

- *Concrete representatives:* In this verification process, test cases are created manually and with these, the according functions/predicates/procedures/theorems are executed and outputs are compared with the expected results. This method is applied for definitions without post-conditions, which appears often for predicates, which are also used as preconditions for other language constructs or inside of functions.
- *Model checking:* Because every type introduced in the RISCAL environment is finite and therefore all RISCAL specifications (including predicates, functions, theorems, procedure) are executable and can be evaluated at any time, model checking can be applied on the implemented theories, after the user set the unspecified constants, declared as values, over the control panel. However, we are restricted to small values, otherwise the domain of the possible input would grow into dimensions, where evaluation would consume too much time with today's computing performance.

4.1. Validating Dijkstra's Algorithm

We start out with validating the required predicates and functions in Dijkstra's algorithm on concrete graphs. The formal specification of Dijkstra's algorithm and the required predicates can be found both in Section 3.4 and Appendix A.

4.1.1. Concrete Representatives

Following predicates/functions are used in the algorithm, and need to be verified:

- `isUndirectedGraph`
- `isPathBetweenVerticesExisting`
- `isPathBetweenVertices`
- `isShortestPath`
- `getNeighborhood`

For this purpose we define a few graphs for testing purposes.

```
val testGraph:undirGraph = ⟨ {0,1,2,3,4}
              , {{0,1},{0,2},{0,3},{1,3},{2,3}} ⟩;
val testGraph2:undirGraph = ⟨ {0,1,2,3,4}
              , {{0,1},{1,4},{3,4},{2,3}} ⟩;
val testGraph3:undirGraph = ⟨ {0,1,2,3}
              , {{0,1},{1,2},{2,3},{3,0}} ⟩;
```

Code 4.1: Concrete graphs for validating predicates/functions for Dijkstra's algorithm

This figure illustrates the test-graph definitions:

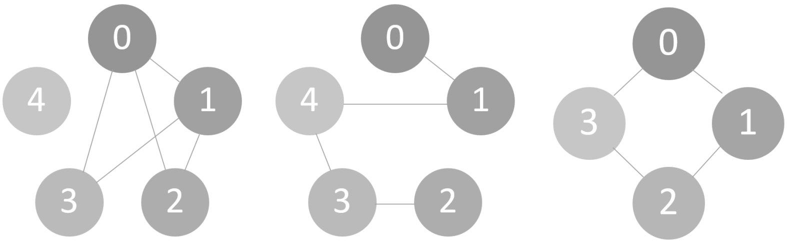

Figure 4.1.: testGraph (left), testGraph2 (middle), testGraph3 (right)

isUndirectedGraph

The predicate `isUndirectedGraph(g)` verifies, if the set of vertices in graph `g` is not empty, and the set of edges only contains sets with 2 elements.

```
  proc testIsUndirectedGraph():()
{
   print "Is testGraph an undirected graph? ";
   print isUndirectedGraph(testGraph);
   print "Is testGraph3 an undirected graph? ";
   print isUndirectedGraph(testGraph3);
   val noGraph1:undirGraph := ⟨ {}[vertex]
              , {{0,1},{1,2},{2,3},{3,0}} ⟩;
   val noGraph2:undirGraph := ⟨ {0,1,2,3}
              , {{0,1,2}} ⟩;
   print "Is noGraph1 an undirected graph? ";
   print isUndirectedGraph(noGraph1);
   print "Is noGraph2 an undirected graph? ";
   print isUndirectedGraph(noGraph2);
}
```

Code 4.2: Test-procedure for validating `isUndirectedGraph`

Expected result: The first two function calls are performed with valid test-graphs and should yield „true". `noGraph1` contains an empty set of vertices and `noGraph2` contains an edge with 3 vertices in it, therefore both calls should yield „false".

Actual output:

```
Executing testIsUndirectedGraph().
Is testGraph an undirected graph?
true
Is testGraph3 an undirected graph?
true
Is noGraph1 an undirected graph?
false
Is noGraph2 an undirected graph?
false
```

getNeighborhood

The function `getNeighborhood(v,g)` determines the set of vertices, which are adjacent (neighbors) to vertex `v` in graph `g`.

```
proc testGetNeighborhood():()
{
  print "Testgraph: ";
  print "Neighbors vertex 0 :";
  print getNeighborhood(0,testGraph);
  print "Neighbors vertex 4:";
  print getNeighborhood(4,testGraph);

  print "";
  print "Testgraph 2: ";
  print "Neighbors vertex 3:";
  print getNeighborhood(3,testGraph2);
}
```

Code 4.3: Test-procedure for validating `getNeighborhood`

Expected result:

- Vertex 0 in „testGraph“: {1,2,3}
- Vertex 4 in „testGraph“: {}
- Vertex 3 in „testGraph2“: {2,4}

Actual output:

```
 Executing testGetNeighborhood().
Testgraph:
Neighbors vertex 0 :
{1,2,3}
Neighbors vertex 4:
{}

Testgraph 2:
Neighbors vertex 3:
{2,4}
```

isPathBetweenVerticesExisting

The predicate `isPathBetweenVerticesExisting(g,v1,v2)` verifies, if a path is existing between vertex `v1` and `v2` in graph `g`.

```
proc testIsPathBetweenVerticesExisting():()
{
  print "Is path between vertices existing in testGraph?";
  print isPathBetweenVerticesExisting(testGraph, 1, 2);

  print "";
  print "Is path between vertices existing in testGraph?";
  print isPathBetweenVerticesExisting(testGraph, 1, 4);
}
```

Code 4.4: Test-procedure for validating `isPathBetweenVerticesExisting`

Expected result: The first call of the predicate is expected to yield „true“, since vertices 1 and 2 are direct neighbors. On the other hand, the second test should yield „false“, since there is no path between vertices 1 and 4 in `testGraph`.

Actual output:

```
Executing testIsPathBetweenVerticesExisting().
Is path between vertices existing in testGraph?
true

Is path between vertices existing in testGraph2?
false
```

isPathBetweenVertices

The predicate `isPathBetweenVertices(p,g,start,end)` verifies, if `p` is a path from vertex `start` to `end` in graph `g`.

```
proc testIsPathBetweenVertices():()
{
  var p:undirPath := Array[N,undirEdge](∅[vertex]);
  p[0] := {0,1}; p[1] := {1,3}; p[2] := {3,2};

  print "is path between vertices? Testgraph, start:0, end:2";
  print isPathBetweenVertices(p,testGraph,0,2);

  print "";
  print "is path between vertices? Testgraph, start:1, end:2";
  print isPathBetweenVertices(p,testGraph,1,2);
```

```

  print "";
  print "is path between vertices? Testgraph, start:0, end:3";
  print isPathBetweenVertices(p,testGraph,0,3);

}
```

Code 4.5: Test-procedure for validating `isPathBetweenVertices`

Expected result: The first call of the predicate is expected to yield „true“, since the path connects vertices 0 and 2 in `testGraph`. On the other hand, the other tests should yield „false“, since `p` is not a path between the given vertices in `testGraph`.

Actual output:

```
Executing testIsPathBetweenVertices().
is path between vertices? Testgraph, start:0, end:2
true

is path between vertices? Testgraph, start:1, end:2
false

is path between vertices? Testgraph, start:0, end:3
false
```

isShortestPath

The predicate `isShortestPath(g,start,end,p)` verifies, if `p` is a shortest path from vertex `start` to `end` in graph `g`.

```
proc testIsShortestPath():()
{
  var p:undirPath := Array[N,undirEdge](∅[vertex]);
  p[0] := {0,1}; p[1] := {1,3}; p[2] := {3,2};

  print "";
  print "is shortest path between vertices? Testgraph, start:0, end:2";
  print isShortestPath(testGraph,0,2,p);

  var q:undirPath := Array[N,undirEdge](∅[vertex]);
  q[0] := {0,2};

```

```
  print "";
  print "is shortest path between vertices? Testgraph, start:0, end:2";
  print isShortestPath(testGraph,0,2,q);

  var p2:undirPath := Array[N,undirEdge](∅[vertex]);
  p2[0] := {0,1}; p2[1] := {1,2};
  print "";
  print "is shortest path between vertices? Testgraph3, start:0, end:2";
  print isShortestPath(testGraph3,0,2,p2);

  var p3:undirPath := Array[N,undirEdge](∅[vertex]);
  p3[0] := {0,3}; p3[1] := {3,2};
  print "";
  print "is shortest path between vertices? Testgraph3, start:0, end:2";
  print isShortestPath(testGraph3,0,2,p3);
}
```

Code 4.6: Test-procedure for validating `isShortestPath`

Expected result: The first call of the predicate is expected to yield „false“, since `p` is a path, that connects vertices 0 and 2, but not the shortest one. The second call should yield „true“, since `q` is a shortest path between 0 and 2 in testGraph. Test three and four should both yield „true“, since they both connect vertices 0 and 2 in testGraph3 and have the same length.

Actual output:

```
Executing testIsShortestPath().

is shortest path between vertices? Testgraph, start:0, end:2
false

is shortest path between vertices? Testgraph, start:0, end:2
true

is shortest path between vertices? Testgraph3, start:0, end:2
true

is shortest path between vertices? Testgraph3, start:0, end:2
true
```

Outcome

All function and predicate calls delivered the expected result and passed our tests with our concrete graphs.

4.1.2. Model Checking

Before applying the model check on Dijkstra's algorithm, we have to decide, which value we are choosing for our yet unspecified constant `N`. The value should neither be too small (to create useful test-cases), nor too big (to avoid unending evaluation). For $N = 2$ we would have 18432 different input values, which seems a bit too small. For $N = 4$ it grows to about $3.436 * 10^{12}$ different input values, which takes too long to evaluate. So $N = 3$, with 16777216 different input values, would be a good choice to start our model check on Dijkstra's algorithm, since the number of input values is manageable, but still representative.

When running the model check, the RISCAL environment fulfills a number of validations:

1. The system chooses one possible input after the other, and …
2. …checks, if the chosen input value fulfills the stated preconditions, if not the input is dismissed
3. …performs the defined command sequence in the procedure
4. …if loop invariants or termination measures are defined, before and after each loop iteration these are validated
5. …after all commands in the sequence are performed, the returned result is compared with the given postconditions
6. If any of the validations between steps 4-5 failed, the execution aborts, and an error message is shown with the corresponding failure in execution

As a result, if our execution is successful, we can be sure, that our result matches all postconditions, as well as all invariants and termination measures. When running the model check in *silent mode* (only errors are shown), we get the following output:

```
Using N=3.
Type checking and translation completed.
Executing dijkstra(Tuple[Set[ℤ],Set[Set[ℤ]]],ℤ,ℤ) with all 16777216 inputs.
```

```
PARALLEL execution with 4 threads (output disabled).
⋮
Execution completed for ALL inputs (122061 ms, 1364 checked, 16775852 inadmissible).
```

The check ran through without errors, so for our specification, this implies:

- *Postconditions*
  - The Boolean return value matches the result of `isPathBetweenVerticesExisting`, and
  - if a path was found, the result ensures, that it suffices `isPathBetweenVertices`, and
  - the found result is actually a shortest path, since `isShortestPath` yields `true`, for all i.
- *Termination measures*
  - Because `decreases |Q|` confirms, that with every loop iteration the set of unvisited nodes becomes smaller, we can be sure that our algorithm terminates.
- *Invariants*: Before/after any iteration we know:
  - all `visited` vertices are in set `conn`
  - all vertices in `conn` have an initialized predecessor and distance to start-vertex
  - the calculated distances, define shortest distances in the set of visited nodes (proofs on this fact can be found in [18])

These properties all hold for at least $N = 3$.

5. Conclusions and Summarization

The various formalizations and implementations of theories and algorithms on different topics of discrete mathematics, provided in this thesis, showed, that the RISCAL environment is a powerful tool to support the process of specification. Not only verifying the specified algorithm itself by using the verification conditions, but also checking, if the specified annotations are neither too strong nor to weak, has become a manageable task.

Additionally, it is very easy to implement different types of the same function in RISCAL. The system can cope with implicit definitions, as well as recursive or procedural algorithms, which leads to a deeper understanding of the connections in between. Consequently, RISCAL has the potential to function as a supportive tool in teaching students, when it comes to lectures on topics like algorithms or data structures.

Sure, there are limits to the theories, one can specify and verify with the RISCAL environment. E.g.: For theories, which mainly depend on models over infinite domains, the results in the verification by RISCAL only have limited meaningfulness. It can always occur, that an algorithm, which works on a bounded domain, does not determine the correct result on unbounded domains. However the results can still be used as a hint, if one's considerations lead into the right direction.

Another thing to keep in mind is the fast growing number of possible inputs, when using the model checking. One really has to be aware of the number and complexity of parameters used in the specifications, since even little values could multiply to a huge domain, which would be too time-consuming, if verified.

In Appendix A, a complete collection of the specified theories described in this thesis is attached.

Acronyms

JML Java Modeling Language

RISC Research Institute for Symbolic Computation

RISCAL RISC Algorithm Language

TLA Temporal Logic of Actions

VDM Vienna Development Method

List of Figures

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] V. S. Alagar and K. Periyasamy. *Specification of Software Systems*. 2nd Edition. Springer-Verlag London Limited, 2011.

[3] Dines Bjorner and Martin C. Henson. *Logics of Specification Languages*. Springer-Verlag Berlin Heidelberg, 2008.

[4] Bruno Buchberger et al. „Theorema 2.0: Computer-Assisted Natural Style Mathematics“. In: *Journal of Formalized Reasoning* (2016). URL: `https://jfr.unibo.it/article/view/4568`.

[5] Edmund M. Clarke et al. *Handbook of Model Checking*. Springer International Publishing, 2016.

[6] Lori A. Clarke and David S. Rosenblum. „A Historical Perspective on Runtime Assertion Checking in Software Development“. In: *SIGSOFT Softw. Eng. Notes* 31.3 (May 2006), pp. 25–37.

[7] Edsger W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* (1959), pp. 269–271.

[8] Paul R. Halmos. *Naive Set Theory*. Undergraduate texts in mathematics. Repr. of the ed. publ. by Van Nostrand, Princeton, NJ, i.d.R.: The University series in undergraduate mathematics. New York, NY [u.a.]: Springer, 2001.

[9] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[10] Daniel Jackson. *Alloy: A Language and Tool for Relational Models*. `http://alloy.mit.edu/alloy/`.

[11] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2011.

[12] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. „Preliminary Design of JML: A Behavioral Interface Specification Language for Java“. In: *ACM SIGSOFT Software Engineering Notes* 31 (2006), pp. 1–38.

[14] PMI. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. 4th ed. Project Management Institute, 2008.

[15] Frederic Portoraro. *Automated Reasoning, in The Stanford Encyclopedia of Philosophy (Winter 2014 Edition)*. https://plato.stanford.edu/archives/win2014/entries/reasoning-automated/. 2014.

[16] *RISC Program Explorer*. https://www.risc.jku.at/research/formal/software/ProgramExplorer/.

[17] *RISC ProofNavigator*. https://www.risc.jku.at/research/formal/software/ProofNavigator/.

[18] Daniela Ritirc. „Formally Modeling and Analyzing Mathematical Algorithms with Software Specification Languages and Tools“. MA thesis. Research Institute for Symbolic Computation (RISC) at Johannes Kepler University, Linz, Austria, 2016.

[19] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. 7th ed. McGraw-Hill, 2012.

[20] Wolfgang Schreiner. „Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs“. In: *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 79 (Feb. 2012), pp. 124–142.

[21] Wolfgang Schreiner. *RISCAL*. http://www.risc.jku.at/research/formal/software/RISCAL/.

[22] Wolfgang Schreiner. *The RISC Algorithmic Language (RISCAL): Tutorial and Reference Manual*. http://www.risc.jku.at/research/formal/software/RISCAL/manual/main.pdf.

[23] Wolfgang Schreiner. „The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom“. In: *Formal Aspects of Computing* 21 (May 2009), pp. 277–291.

[24] Angus [Hrsg.] Stevenson and 1962-[Begr.] Pearsall Judy. *Oxford dictionary of English*. 3. ed. Oxford [u.a.]: Oxford Univ. Press, 2010.

[25] *The Theorema System*. https://www.risc.jku.at/research/theorema/software/.

A. Appendix

A.1. Set Theory

```
val N:ℕ;
val Universe = 0..N;
type elem = ℕ[N];
type set = Set[elem];
type pair = Tuple[elem,elem];
//-----------------------------
// a ⊆ b
pred isSubsetEq(a:set,b:set) ⇔
∀x ∈ a. x ∈ b;

// compare isSubsetEq with the implemented operator ⊆
theorem subsetEqT(a:set,b:set) ⇔ isSubsetEq(a,b) = (a ⊆ b);

// a ⊆ b defined as a procedure
proc isSubsetEqP(a:set,b:set):Bool
   ensures result = (a ⊆ b);
{
   var res:Bool := true;
   for x ∈ a do
      invariant res = (forSet ⊆ b);
   {
      res := res ∧ (x ∈ b);
   }
   return res;
}

// a ⊆ b recursively defined
multiple fun isSubsetEqR(a:set,b:set):Bool
   decreases |a|;
   ensures result = (a ⊆ b);
```

```
= choose x:elem with x ∈ a
     in (x ∈ b ∧ isSubsetEqR(a\{x},b))
  else true;

//------------------------------
// is a real subset of b?
pred isSubset(a:set,b:set) ⇔
a ⊆ b ∧ (∃x ∈ b. ¬(x ∈ a));

// are a and b equal?
pred isEqual(a:set,b:set) ⇔
(a ⊆ b) ∧ (b ⊆ a);

// compare defined equal predicate with the implemented operator
theorem equalT(a:set,b:set) ⇔ isEqual(a,b) = (a = b);

//-----------------------------
fun powersetS(a:set):Set[set] =
   { x | x:set with (x ⊆ a) };

theorem powersetT(a:set) ⇔ powersetS(a) = Set(a);

theorem powersetProperties(a:set) ⇔
   (∅[elem] ∈ powersetS(a) ∧ ∅[set] ⊆ powersetS(a));

//-----------------------------
// a ∪ b implicitly defined
fun unionI(a:set,b:set):set =
   choose c:set with (∀ x:elem. x ∈ c ⇔ x ∈ a ∨ x ∈ b);

fun unionS(a:set,b:set):set =
   { x | x:elem with (x ∈ a ∨ x ∈ b) };

// is unionS(a,b) = (a ∪ b)?
theorem unionT(a:set,b:set) ⇔
   unionS(a,b) = (a ∪ b);

// a ∪ b defined as a procedure
proc unionP(a:set,b:set):set
   ensures result = a ∪ b;
```

```
{
   var res:set := a;
   for x ∈ b do
      invariant res = (a ∪ forSet);
   {
      res := res ∪ {x};
   }
   return res;
}

// a ∪ b recursively defined
multiple fun unionR(a:set,b:set):set
   decreases |a|;
   ensures result = a ∪ b;
= choose x:elem with x ∈ a
      in ({x} ∪ unionR(a\{x},b))
  else b;


//------------------------------
// a ∩ b defined with set quantifiers
fun intersectS(a:set,b:set):set =
   { x | x:elem with (x ∈ a ∧ x ∈ b) };

// is intersectS(a,b) equal to a ∩ b
theorem intersectT(a:set,b:set) ⇔ intersectS(a,b) = (a ∩ b);

// a ∩ b defined as a procedure
proc intersectP(a:set,b:set):set
   ensures result = a ∩ b;
{
   var res:set := ∅[elem];
   for x ∈ a do
      invariant res = (b ∩ forSet);
   {
      if x ∈ b then res := res ∪ {x};
   }
   return res;
}
```

```
// a ∩ b recursively defined
multiple fun intersectR(a:set,b:set):set
   decreases |a|;
   ensures result = a ∩ b;
= choose x:elem with x ∈ a
      in (if x ∈ b then {x} ∪ intersectR(a\{x},b)
            else intersectR(a\{x},b))
  else ∅[elem];

//------------------------------
// a\b defined with set quantifiers
fun differenceS(a:set,b:set):set =
   { x | x:elem with (x ∈ a ∧ ¬(x ∈ b)) };

// is differenceS(a,b) equal to a\b ?
theorem differenceT(a:set,b:set) ⇔ differenceS(a,b) = (a \ b);

// a\b defined as a procedure
proc differenceP(a:set,b:set):set
   ensures result = (a\b);
{
   var res:set := ∅[elem];
   for x ∈ a do
      invariant res = (forSet\b);
   {
      if ¬(x ∈ b) then res := res ∪ {x};
   }
   return res;
}

// a\b recursively defined
multiple fun differenceR(a:set,b:set):set
   decreases |b|;
   ensures result = (a\b);
= choose x:elem with x ∈ b
      in differenceR(a\{x},b\{x})
  else a;

//------------------------------
fun complementS(a:set):set =
```

```
   { x | x:elem with ¬(x ∈ a) };

theorem complementT(a:set) ⇔ complementS(a) = (Universe\a);

proc complementP(a:set):set
   ensures result = (Universe\a);
{
   var res:set := 0..N;
   for x ∈ a do
      invariant res = (0..N\forSet);
   {
      res := res \ {x};
   }
   return res;
}

multiple fun complementR(a:set):set
   decreases |Universe\a|;
   ensures result = (0..N\a);
= choose x:elem with ¬(x ∈ a)
      in ({x} ∪ complementR(a ∪ {x}))
  else ∅[elem];

//-------------------------------
// a,b disjunct:
pred disjunct(a:set,b:set) ⇔ (a ∩ b = ∅[elem]);

//-------------------------------
// laws for operations on sets:
theorem commutativeUnionT(a:set,b:set) ⇔ (a ∪ b = b ∪ a);
theorem commutativeIntersectT(a:set,b:set) ⇔ (a ∩ b = b ∩ a);

theorem associativeUnionT(a:set,b:set,c:set) ⇔ (a ∪ (b ∪ c) = (a ∪ b) ∪ c);
theorem associativeIntersectT(a:set,b:set,c:set) ⇔ (a ∩ (b ∩ c) = (a ∩ b) ∩ c);

theorem distributiveUnionT(a:set,b:set,c:set) ⇔ (a ∪ (b ∩ c) = ((a ∪ b) ∩ (a ∪ c)));
theorem distributiveIntersectT(a:set,b:set,c:set) ⇔ (a ∩ (b ∪ c) = ((a ∩ b) ∪ (a ∩ c)));
```

```
theorem deMorganUnionT(a:set,b:set) ⇔ complementS(a ∪ b) = complementS(a) ∩
    complementS(b);
theorem deMorganIntersectT(a:set,b:set) ⇔ complementS(a ∩ b) = complementS(a) ∪
    complementS(b);


//-----------------------------
// Cartesian product defined with set quantifiers
fun cartesianProductS(a:set,b:set):Set[pair] =
   { p | p:pair with (p.1 ∈ a ∧ p.2 ∈ b) };

// is cartesianProductS(a,b) equal to a × b
theorem cartesianProductT(a:set,b:set) ⇔ cartesianProductS(a,b) = a × b;

// Cartesian product defined as a procedure
proc cartesianProductP(a:set,b:set):Set[pair]
   ensures result = (a × b);
{
   var res:Set[pair] := ∅[pair];
   for x ∈ a do
      invariant res = (forSet × b);
   {
      res := res ∪
         {p | p:pair with p.1 = x ∧ p.2 ∈ b};
   }
   return res;
}

// Cartesian product recursively defined
multiple fun cartesianProductR(a:set,b:set):Set[pair]
   decreases |a|;
   ensures result = (a × b);
= choose x:elem with x ∈ a
      in ({p | p:pair with p.1 = x ∧ p.2 ∈ b}
         ∪ cartesianProductR(a\{x},b))
  else ∅[pair];

//-------------------------------------
// cardinality
val S = N+1;
```

```
type size = ℕ[S];
type map = Map[size,elem];

// m is a map of the first s natural numbers into a
pred isMapToA(s:size, f:map, a:set) ⇔
   (∀i:size with i >= s. f[i] = 0) ∧
   (∀i:size with i < s. f[i] ∈ a);

// is f injective on a at the first s entries?
pred isInjective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ ∀x:size,y:size with (x < s ∧ y < s). ((f[x] = f[y]) ⇒ (x = y));

// is every element of a in the image of f?
pred isSurjective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ ∀x:elem with x ∈ a. ∃y:size with y < s. f[y] = x;

// is f bijective on set a?
pred isBijective(s:size, f:map, a:set)
   requires isMapToA(s,f,a);
⇔ isInjective(s,f,a) ∧ isSurjective(s,f,a);

// the cardinality of set a
fun cardinalityS(a:set):size =
   choose s:size with ∃f:map with isMapToA(s,f,a). isBijective(s, f, a);

// is cardinalityS(a) equal to |a|?
theorem cardinalityT(a:set) ⇔ cardinalityS(a) = |a|;

// the cardinality defined recursively
multiple fun cardinalityR(a:set):size
   ensures result = |a|;
   decreases |a|;
=  choose x:elem with x ∈ a
      in (1 + cardinalityR(a\{x}))
   else 0;


proc cardinalityP(a:set):size
```

```
  ensures result = |a|;
{
  var res:size := 0;
  for x ∈ a do
    invariant res = |forSet|;
  {
    res := res + 1;
  }
  return res;
}
```

A.2. Relation and Function Theory

```
val N:ℕ;
val Universe = 0..N;
type elem = ℕ[N];
type set = Set[elem];
type pair = Tuple[elem,elem];
type relation = Set[pair];

// r is relation between sets a and b ⇔ r ⊆ a × b
pred isRelation(r:relation,a:set,b:set)
⇔ r ⊆ a × b;

// is r relation between a and b as a procedure
proc isRelationP(r:relation,a:set,b:set):Bool
   ensures result = isRelation(r,a,b);
{
   var res:Bool := true;
   for x ∈ r do
      invariant res = isRelation(forSet,a,b);
   {
      res := res ∧ (x.1 ∈ a ∧ x.2 ∈ b);
   }
   return res;
}

// is r relation between a and b as a recursion
multiple fun isRelationR(r:relation,a:set,b:set):Bool
   decreases |r|;
   ensures result = isRelation(r,a,b);
= choose x:pair with x ∈ r
      in (x.1 ∈ a ∧ x.2 ∈ b ∧ isRelationR(r\{x},a,b))
  else true;

//----------------------------------------------------
// composition of relation r:a->b and s:b->c (s∘r)
fun composeS(r:relation, s:relation, a:set, b:set, c:set):relation
   requires isRelation(r,a,b) ∧ isRelation(s,b,c);
= {p | p:pair with (p.1 ∈ a ∧ p.2 ∈ c
   ∧ (∃x ∈ b. (⟨p.1,x⟩ ∈ r ∧ ⟨x,p.2⟩ ∈ s )))};
```

```

// composition is relation from a to c
theorem compositionFromAtoC(r:relation, s:relation, a:set, b:set, c:set)
    requires isRelation(r,a,b) ∧ isRelation(s,b,c);
⇔ isRelation(composeS(r,s,a,b,c),a,c);

// composition s∘r as a procedure
proc composeP(r:relation,s:relation,a:set,b:set,c:set):relation
   requires isRelation(r,a,b) ∧ isRelation(s,b,c);
   ensures result = composeS(r,s,a,b,c);
{
   var res:relation := ∅[pair];
   for x ∈ r do
      invariant res = composeS(forSet,s,a,b,c);
   {
      res := res ∪ {p | p:pair with (p.1 = x.1
           ∧ p.2 ∈ {e | e:elem with ⟨x.2,e⟩ ∈ s})};
   }
   return res;
}


// composition s∘r as a recurrence
multiple fun composeR(r:relation,s:relation,a:set,b:set,c:set):relation
   requires isRelation(r,a,b) ∧ isRelation(s,b,c);
   ensures result = composeS(r,s,a,b,c);
   decreases |r|;
= choose x:pair with x ∈ r
      in ({p | p:pair with (p.1 = x.1
        ∧ p.2 ∈ {e | e:elem with ⟨x.2,e⟩ ∈ s})}
        ∪ composeR(r\{x},s,a,b,c))
  else ∅[pair];

//-----------------------------------------------------
// inverse relations
fun inverseS(r:relation, a:set, b:set):relation
   requires isRelation(r,a,b);
= {p | p:pair with ⟨p.2,p.1⟩ ∈ r};

// the inverse relation of r:a->b is a relation from b->a
```

```
theorem isInverseARelationT(r:relation, a:set, b:set)
  requires isRelation(r,a,b);
⇔ isRelation(inverseS(r,a,b),b,a);

// (r⁻¹)⁻¹ = r
theorem inverseOfInverseT(r:relation, a:set, b:set)
  requires isRelation(r,a,b);
⇔ inverseS(inverseS(r,a,b),b,a) = r;

// r⁻¹ as a procedure
proc inverseP(r:relation,a:set,b:set):relation
  requires isRelation(r,a,b);
  ensures result = inverseS(r,a,b);
{
  var res:relation := ∅[pair];
  for x ∈ r do
    invariant res = inverseS(forSet,a,b);
  {
    res := res ∪ { ⟨x.2,x.1⟩ };
  }
  return res;
}

// r⁻¹ as a recursion
multiple fun inverseR(r:relation,a:set,b:set):relation
  requires isRelation(r,a,b);
  ensures result = inverseS(r,a,b);
  decreases |r|;
= choose x:pair with x ∈ r
    in ({ ⟨x.2,x.1⟩ } ∪ inverseR(r\{x},a,b))
  else ∅[pair];

// (s∘r)⁻¹ = r⁻¹∘s⁻¹
theorem composeInverseT(r:relation,s:relation,a:set,b:set,c:set)
  requires isRelation(r,a,b) ∧ isRelation(s,b,c);
⇔ inverseS(composeS(r,s,a,b,c),a,c) =
    composeS(inverseS(s,b,c),inverseS(r,a,b),c,b,a);

//---------------------------------------------
// identity relation on set a
```

```
fun identity(a:set):relation
= { ⟨x,x⟩ | x:elem with x ∈ a};

// r∘I=I∘r=r
theorem neutralT(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ composeS(r,identity(a),a,a,a) = r
       ∧ composeS(identity(a),r,a,a,a) = r;


//---------------------------------------------
// monoid axioms for endorelations
// associativity: t∘(s∘r) = (t∘s)∘r
theorem associativityEndoT(r:relation,s:relation,t:relation,
             a:set)
  requires isRelation(r,a,a) ∧ isRelation(s,a,a)
         ∧ isRelation(t,a,a);
⇔ composeS(composeS(r,s,a,a,a),t,a,a,a) =
     composeS(r,composeS(s,t,a,a,a),a,a,a);

//---------------------------------------------
// is relation r:a->a reflexive?
pred isReflexiveS(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ ∀x ∈ a. ⟨x,x⟩ ∈ r;

// is relation r:a->a symmetric?
pred isSymmetricS(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ ∀x ∈ r. ⟨x.2,x.1⟩ ∈ r;

// is relation r:a->a anti-symmetric?
pred isAntiSymmetricS(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ ∀x ∈ r. (⟨x.2,x.1⟩ ∈ r ⇒ x.2 = x.1);

// is relation r:a->a transitive?
pred isTransitiveS(r:relation,a:set)
  requires isRelation(r,a,a);
⇔ ∀x ∈ r, y ∈ r. ((x.2 = y.1) ⇒ ⟨x.1,y.2⟩ ∈ r);
```

```

//---------------------------------------------
// is s a relation on a->a ∧ r ⊆ s ∧ is s transitive?
pred isRelationSubsetAndTransitive(s:relation,r:relation,a:set)
⇔ isRelation(s,a,a) ∧ r ⊆ s ∧ isTransitiveS(s,a);

// transitive closure of relation r:a->a
// transitive closure t is the smallest relation with r ⊆ t and
// t is transitive
fun transitiveClosureS(r:relation,a:set):relation
   requires isRelation(r,a,a);
= choose s:relation with (isRelationSubsetAndTransitive(s,r,a) ∧
   (∀t:relation.
   isRelationSubsetAndTransitive(t,r,a) ⇒ s ⊆ t));

// transitive closure of r as a procedure
proc transitiveClosureP(r:relation,a:set):relation
   requires isRelation(r,a,a);
   ensures result = transitiveClosureS(r,a);
{
   var res:relation := ∅[pair];
   var toCheck:relation := r;
   choose x ∈ toCheck do
      //invariant res = transitiveClosureS(forSet,a);
   {
      for y ∈ res do
      {
         if x.1 = y.2 ∧ ¬(⟨y.1, x.2⟩ ∈ res) then
         {
            toCheck := toCheck ∪ { ⟨y.1, x.2⟩ };
         }
         if x.2 = y.1 ∧ ¬(⟨x.1, y.2⟩ ∈ res) then
         {
            toCheck := toCheck ∪ { ⟨x.1, y.2⟩ };
         }
      }
      res := res ∪ { x };
      toCheck := toCheck \ { x };
   }
   return res;
```

```
}

val RelationUniverse = Universe × Universe;
// transitive closure of r as a recursion
multiple fun transitiveClosureR(r:relation,a:set):relation
   requires isRelation(r,a,a);
   ensures result = transitiveClosureS(r,a);
   decreases |RelationUniverse\r|;
= if isTransitiveS(r,a) then r
  else transitiveClosureR(
     r ∪ { ⟨x,y⟩ | x:elem,y:elem with
           (∃p∈r,q∈r. (x = p.1 ∧ y = q.2 ∧ p.2 = q.1)) }
           , a);

//----------------------------------------------
// is relation r:a->b a partial function?
pred isPartialFunctionS(r:relation,a:set,b:set)
⇔ isRelation(r,a,b) ∧ ∀x ∈ r, y ∈ r. (x.1 = y.1 ⇒ x.2 = y.2);

// is relation r:a->b a total function?
pred isFunctionS(r:relation,a:set,b:set)
⇔ isRelation(r,a,b) ∧
   ∀z ∈ a. (∃f ∈ r.
      (f.1 = z ∧ ∀g ∈ r. (g.1 = f.1 ⇒ g.2 = f.2))) ;

//--------------------------------------
// Connection to type 'Map'
type map = Map[elem,elem];

pred isFunctionM(m:map, a:set, b:set)
⇔ (∀x ∈ a. m[x] ∈ b) ;

pred equal(r:relation, m:map, a:set, b:set)
   requires isFunctionS(r,a,b) ∧ isFunctionM(m,a,b);
⇔ ∀x ∈ a. ∃y ∈ r. (y.1 = x ∧ m[x] = y.2);

fun relToMap(r:relation, a:set, b:set):map
  requires isFunctionS(r,a,b);
  ensures isFunctionM(result,a,b) ∧
         equal(r,result,a,b);
```

```
= choose m:map with ((∀x ∈ a. ∃y ∈ r. (x = y.1 ∧ m[x] = y.2)));

fun mapToRelation(m:map, a:set, b:set):relation
  requires isFunctionM(m,a,b);
  ensures isFunctionS(result,a,b) ∧ equal(result,m,a,b);
= {p | p:pair with (p.1 ∈ a ∧ p.2 = m[p.1])};

// a relation is equal to its induced
// relation of the induced map
theorem relationT1(r:relation, a:set, b:set)
   requires isFunctionS(r,a,b);
⇔ r = mapToRelation(relToMap(r,a,b),a,b);

// a map is equal to its induced map of the induced relation
theorem mapT1(m:map, a:set, b:set)
   requires isFunctionM(m,a,b);
⇔ ∀x ∈ a. (m[x] = relToMap(mapToRelation(m,a,b),a,b)[x]);

// composition of functions f2∘f1 is again a function
theorem compositionOfFunctions(f1:relation, f2:relation,
              a:set,b:set,c:set)
   requires isRelation(f1,a,b) ∧ isRelation(f2,b,c);
⇔ (isFunctionS(f1,a,b) ∧ isFunctionS(f2,b,c)) ⇒
         isFunctionS(composeS(f1,f2,a,b,c),a,c);

// is the function surjective?
pred isSurjectiveFunction(f:relation, a:set, b:set)
   requires isFunctionS(f,a,b);
⇔ ∀x ∈ b. (∃y ∈ a. (⟨y,x⟩ ∈ f));

// is the function injective?
pred isInjectiveFunction(f:relation, a:set, b:set)
   requires isFunctionS(f,a,b);
⇔ ∀x ∈ f, y ∈ f. ((x.2 = y.2) ⇒ (x.1 = y.1));

// is the function bijective?
pred isBijectiveFunction(f:relation, a:set, b:set)
   requires isFunctionS(f,a,b);
⇔ isSurjectiveFunction(f,a,b) ∧ isInjectiveFunction(f,a,b);

```

```
// inverse of bijective function is bijective
theorem inverseBijectiveT(f:relation,a:set,b:set)
  requires isFunctionS(f,a,b);
⇔ isBijectiveFunction(f,a,b) ⇒
     isBijectiveFunction(inverseS(f,a,b),b,a);
```

A.3. Graph Theory

```
type vertex = ℕ[N];
type vertices = Set[vertex];
type dirEdge = Tuple[vertex,vertex];
type dirEdges = Set[dirEdge];
type undirEdge = Set[vertex];
type undirEdges = Set[undirEdge];
type dirGraph = Tuple[vertices, dirEdges];
type undirGraph = Tuple[vertices, undirEdges];
type undirPath = Array[N,undirEdge];

// disable empty set of vertices and
// only allow sets with 2 elements in set of edges
pred isUndirectedGraph(g:undirGraph)
⇔ g.1 ≠ ∅[vertex] ∧ (g.2 ⊆ Set(g.1,2));

// don't allow empty set of vertices and check if all
// elements in the edges are in the set of vertices
pred isDirectedGraph(g:dirGraph)
⇔ g.1 ≠ ∅[vertex] ∧ (g.2 ⊆ { e | e:dirEdge with
                   e.1 ∈ g.1 ∧ e.2 ∈ g.1});

// check if a certain vertex v1 is in a set of vertices v
pred isVertexInSetOfVertices(v1:vertex, v:vertices)
⇔ v1 ∈ v;

// are the vertices v1 and v2 adjacent in graph g?
pred areVerticesAdjacent(g:undirGraph, v1:vertex, v2:vertex)
   requires isVertexInSetOfVertices(v1,g.1)
         ∧ isVertexInSetOfVertices(v2,g.1)
         ∧ isUndirectedGraph(g);
⇔ {v1,v2} ∈ g.2;

// get the complete undirected graph to a set of vertices
fun getCompleteUndirectedGraph(v:vertices):undirGraph
   requires v ≠ ∅[vertex];
   ensures isUndirectedGraph(result);
= ⟨v, { {x,y} | x:vertex,y:vertex with
      x ∈ v ∧ y ∈ v ∧ x ~= y } ⟩;
```

```

// get undirected graph from directed graph
fun getUndirectedGraph(g:dirGraph):undirGraph
   requires isDirectedGraph(g);
   ensures isUndirectedGraph(result);
= ⟨g.1, {{x,y} | x:vertex,y:vertex with x ≠ y ∧
              (⟨x,y⟩ ∈ g.2 ∨ ⟨y,x⟩ ∈ g.2)}⟩;

// get the neighborhood of a vertex v in graph g
fun getNeighborhood(v:vertex, g:undirGraph):vertices
   requires isUndirectedGraph(g)
         ∧ isVertexInSetOfVertices(v,g.1);
= {v2 | v2:vertex with (v2 ∈ g.1 ∧ {v,v2} ∈ g.2)};

// get the degree of a vertex v in graph g
fun getDegree(v:vertex, g:undirGraph):ℕ[N]
   requires isUndirectedGraph(g)
     ∧ isVertexInSetOfVertices(v,g.1);
= | getNeighborhood(v,g) |;

// theorem: 2 times the number of edges equals the
// sum over all degrees of the vertices
theorem handshakingTheorem(g:undirGraph)
   requires isUndirectedGraph(g);
⇔ 2*|g.2| = ∑v ∈ g.1 . getDegree(v,g);

// theorem: number of vertices of odd degree is even
theorem numberOfVerticesOfOddDegree(g:undirGraph)
   requires isUndirectedGraph(g);
⇔ (|{v | v:vertex with (v ∈ g.1)
     ∧ (getDegree(v,g) % 2) = 1}| % 2 ) = 0;

// are 2 sets of vertices disjoint?
pred isDisjoint(v1:vertices, v2:vertices)
⇔ (v1 ∩ v2) = ∅[vertex];

// is graph g bipartite?
pred isGraphBipartite(g:undirGraph)
   requires isUndirectedGraph(g);
⇔ ∃v1:vertices,v2:vertices. (∀x ∈ g.1, y ∈ g.1 with {x,y} ∈ g.2 .
```

```
    (isDisjoint(v1,v2) ∧
    ((x ∈ v1 ∧ y ∈ v2) ∨ (x ∈ v2 ∧ y ∈ v1))));

// is h subgraph of g?
pred isSubGraph(g:undirGraph,h:undirGraph)
  requires isUndirectedGraph(g) ∧ isUndirectedGraph(h);
⇔ h.1 ⊆ g.1 ∧ h.2 ⊆ g.2;

// returns the induced graph of graph g and set of vertices v
fun inducedSubGraph(g:undirGraph, v:vertices):undirGraph
  requires isUndirectedGraph(g) ∧ v ⊆ g.1 ∧ v ≠ ∅[vertex];
  ensures isSubGraph(g,result);
= ⟨v, { {x,y} | x ∈ v,y ∈ v with {x,y} ∈ g.2 } ⟩;

//------------------------------------------------------
// paths

// check if array is filled till index n and if array
// is empty after this index
pred isArrayFilledToIndex(a:Array[N,undirEdge], n:ℕ[N-1])
⇔ (∀m ∈ n+1..N-1. (a[m] = ∅[vertex])) ∧
 (∀l ∈ 0..n. (a[l] ≠ ∅[vertex]));

// check if array is empty
pred isArrayEmpty(a:Array[N,undirEdge])
⇔ ∀m ∈ 0..N-1. (a[m] = ∅[vertex]);

// check if all entries are empty, after the first entry
// is empty
pred isArrayEmptyFromFirstEmptyEntry(a:Array[N,undirEdge])
⇔ ∀m ∈ 0..N-1. (((a[m] = ∅[vertex]) ⇒
          (∀n ∈ 0..N-1 with n > m. (a[n] = ∅[vertex]))));

// check if path is in graph g
pred isPathInGraph(p:undirPath, g:undirGraph)
  requires isUndirectedGraph(g);
⇔ ∀m ∈ 0..N-1. (p[m] ∈ g.2 ∨ p[m] = ∅[vertex]);

// get number of edges within path, which include v
fun numberOfEdgesWithVertex(p:undirPath, v:vertex):ℕ[N]
```

```
= |{e| e:undirEdge with ((∃n ∈ 0..N-1. (p[n] = e)) ∧ v ∈ e)}|;

// check if vertices are at most once in the path
// start- and end-vertice have to be checked extra
pred isVertexOnceInPath(p:undirPath, start:vertex, end:vertex,
     v:vertices)
⇔ numberOfEdgesWithVertex(p,start) = 1
  ∧ numberOfEdgesWithVertex(p,end) = 1
  ∧ ∀v1 ∈ (v\{start,end}). numberOfEdgesWithVertex(p,v1) <= 2;

// check if the edges are adjacent
pred isEdgeAdjacent(e1:undirEdge, e2:undirEdge)
⇔ (e1 ∩ e2) ≠ ∅[vertex] ∧ e1 ≠ e2;

pred areNonEmptyEntriesUnique(a:Array[N,undirEdge])
⇔ ∀m ∈0..N-1. ((a[m] = ∅[vertex]) ∨
     (∀n ∈ (0..N-1)\{m}. a[n] ≠ a[m]));

pred isPathRequirementsFulfilled(p:undirPath)
⇔ isArrayEmptyFromFirstEmptyEntry(p)
     ∧ areNonEmptyEntriesUnique(p);

// check if p is a path between start- and endvertice in graph g
pred isPathBetweenVertices( p:undirPath, g:undirGraph,
                start:vertex, end:vertex)
   requires isUndirectedGraph(g)
     ∧ isVertexInSetOfVertices(start,g.1)
     ∧ isVertexInSetOfVertices(end,g.1)
     ∧ isPathRequirementsFulfilled(p)
     ∧ isPathInGraph(p,g);
⇔ ((start = end) ∧ isArrayEmpty(p)) ∨
   (start ≠ end ∧ (∃n:ℕ[N-1]. (isArrayFilledToIndex(p,n)
  ∧ isVertexOnceInPath(p, start, end, g.1)
  ∧ ∀m ∈ 1..n. (isEdgeAdjacent(p[m-1], p[m])))));

// is graph g connected? that means, is every vertex connected
// by a path to every other vertex in graph g?
pred isGraphConnected(g:undirGraph)
   requires isUndirectedGraph(g);
⇔ ∀v1 ∈ g.1, v2 ∈ g.1.
```

```
   (∃p:undirPath. isPathRequirementsFulfilled(p) ∧
      isPathInGraph(p,g) ∧
      isPathBetweenVertices(p, g, v1, v2));

pred isPathBetweenVerticesExisting(g:undirGraph, start:vertex,
                end:vertex)
   requires isUndirectedGraph(g)
      ∧ isVertexInSetOfVertices(start,g.1)
      ∧ isVertexInSetOfVertices(end,g.1);
⇔ ∃p:undirPath. (isPathRequirementsFulfilled(p) ∧
      isPathInGraph(p,g) ∧
      isPathBetweenVertices(p, g, start, end));
// ---------------------------------------------------
// get all paths between 2 vertices
fun getPathsBetweenVertices(g:undirGraph, start:vertex,
                end:vertex):Set[undirPath]
   requires isUndirectedGraph(g)
      ∧ isVertexInSetOfVertices(start,g.1)
      ∧ isVertexInSetOfVertices(end,g.1);
   ensures ¬(∃q:undirPath with (isPathRequirementsFulfilled(q)
             ∧ isPathInGraph(q,g)).
             isPathBetweenVertices(q,g,start,end)
             ∧ ¬(q ∈ result));
= {p | p:undirPath with isPathRequirementsFulfilled(p)
             ∧ isPathInGraph(p,g)
             ∧ isPathBetweenVertices(p,g,start,end)};

// find any path between start- and end-vertex
fun getPathBetweenVertices(g:undirGraph, start:vertex,
                end:vertex):Tuple[Bool,undirPath]
   requires isUndirectedGraph(g)
      ∧ isVertexInSetOfVertices(start,g.1)
      ∧ isVertexInSetOfVertices(end,g.1);
   ensures
      (result.1 = isPathBetweenVerticesExisting(g,start,end))
      ∧ ((¬result.1) ∨
      isPathBetweenVertices(result.2,g,start,end));
= choose p:undirPath with (isPathRequirementsFulfilled(p)
             ∧ isPathInGraph(p,g)
             ∧ isPathBetweenVertices(p,g,start,end))
```

```
  in ⟨true,p⟩
  else ⟨false,Array[N,undirEdge](∅[vertex])⟩;

// find any path between start- and end-vertex as a procedure
proc getPathBetweenVerticesP(g:undirGraph, start:vertex,
                end:vertex):Tuple[Bool,undirPath]
   requires isUndirectedGraph(g)
     ∧ isVertexInSetOfVertices(start,g.1)
     ∧ isVertexInSetOfVertices(end,g.1);
   ensures
     (result.1 = isPathBetweenVerticesExisting(g,start,end))
     ∧ ((¬result.1) ∨
     isPathBetweenVertices(result.2,g,start,end));
{
   var res:undirPath := Array[N,undirEdge](∅[vertex]);
   var found:Bool := false;
   if start ≠ end then
   {
     var lastVertex:vertex := start;
     var visited:vertices := {start};
     var i:ℕ[N+1];
     for i := 0; i < N ∧ ¬found; i := i+1 do
     {
        if {lastVertex,end} ∈ g.2 then
        {
          res[i] := {lastVertex,end};
          found := true;
        } else
        {
          var v1:ℕ[N+1];
          choose v ∈ getNeighborhood(lastVertex, g)\visited
            with isPathBetweenVerticesExisting(
                  inducedSubGraph(g,g.1\visited),
                  v, end)
            then v1 := v;
            else v1 := N+1;
          if(v1 ≠ N+1) then {
             res[i] := {lastVertex,v1};
             visited := visited ∪ {v1};
             lastVertex := v1;
```

```
            } else {
               // no viable vertex found
               // -> return found = false
               i = N;
            }
         }
      }
   } else {
      found := true;
   }
   return ⟨found,res⟩;
}

// --------------------------------------------------
// length of a path
fun getLengthOfPath(p:undirPath):ℕ[N]
   requires isPathRequirementsFulfilled(p);
= choose i:ℕ[N-1] with (p[i] = ∅[vertex] ∧
         ∀j ∈ 0..(i-1). p[j] ≠ ∅[vertex])
     in i
   else N;

// --------------------------------------------------
// shortest path
pred isShortestPath(g:undirGraph, start:vertex,
               end:vertex, p:undirPath)
   requires isUndirectedGraph(g)
      ∧ isVertexInSetOfVertices(start,g.1)
      ∧ isVertexInSetOfVertices(end,g.1)
      ∧ isPathRequirementsFulfilled(p)
      ∧ isPathInGraph(p,g);
⇔ isPathBetweenVertices(p,g,start,end) ∧
   ∀q:undirPath with (isPathRequirementsFulfilled(q)
      ∧ isPathInGraph(q,g)
      ∧ isPathBetweenVertices(q,g,start,end))
      . (getLengthOfPath(p) ≤ getLengthOfPath(q));

fun getShortestPath(g:undirGraph, start:vertex,
      end:vertex):Tuple[Bool,undirPath]
   requires isUndirectedGraph(g)
```

```
      ∧ isVertexInSetOfVertices(start,g.1)
      ∧ isVertexInSetOfVertices(end,g.1);
   ensures
      (result.1 = isPathBetweenVerticesExisting(g,start,end))
      ∧ ((¬result.1) ∨
      (isPathBetweenVertices(result.2,g,start,end)
      ∧ isShortestPath(g,start,end,result.2)));
= choose p:undirPath with (isPathRequirementsFulfilled(p)
               ∧ isPathInGraph(p,g)
               ∧ isPathBetweenVertices(p,g,start,end)
               ∧ isShortestPath(g,start,end,p))
  in ⟨true,p⟩
  else ⟨false,Array[N,undirEdge](∅[vertex])⟩;

proc dijkstra(g:undirGraph, start:vertex,
        end:vertex):Tuple[Bool,undirPath]
   requires isUndirectedGraph(g)
      ∧ start ∈ g.1
      ∧ end ∈ g.1;
   ensures
      (result.1 = isPathBetweenVerticesExisting(g,start,end))
      ∧ ((¬result.1) ∨
      (isPathBetweenVertices(result.2,g,start,end)
      ∧ isShortestPath(g,start,end,result.2)));
{
   var res:undirPath := Array[N,undirEdge](∅[vertex]);
   var found:Bool := false;

   // initialize
   var dist:Map[vertex,ℕ[N+1]] := Map[vertex,ℕ[N+1]](N+1);
   var prev:Map[vertex,ℕ[N+1]] := Map[vertex,ℕ[N+1]](N+1);
   var conn:vertices := {start};
   dist[start] := 0;
   prev[start] := start;
   var Q:vertices := g.1;
   var visited:vertices := ∅[vertex];

   // loop over all unvisited vertices and choose the
   // one with the least distance
   choose q ∈ (Q ∩ conn) with
```

```
      (∀v ∈ (Q ∩ conn). dist[q] ≤ dist[v]) do
    decreases |Q|;
    // all neighbours of visited nodes are connected
    invariant ∀v ∈ visited.
          ∀neigh ∈ getNeighborhood(v,g).
          neigh ∈ conn;
    // all connected vertices (except start) have a
    // connected neighbor
    invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
          ∃v2:vertex with (v2 ∈ conn).
          v2 ∈ getNeighborhood(v,g);
    // defines shortest dist of visited nodes
    invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
          ∃v2 ∈ visited. (prev[v] = v2
             ∧ v2 ∈ getNeighborhood(v,g)
             ∧ dist[v] = dist[v2] + 1);
    invariant ∀v:vertex with v ∈ conn.
          (∀v2:vertex with v2 ∈ conn.
          (v2 ∈ getNeighborhood(v,g) ⇒
           dist[v] <= dist[v2] + 1));
    // visited implies connected
    invariant ∀v ∈ visited. (v ∈ conn);
    // connected implies defined predecessor and distance
    invariant ∀v ∈ conn. (prev[v] ≠ N+1 ∧ dist[v] ≠ N+1);
    // Distance of visited nodes is shorter than the
    // distance of unvisited but connected nodes
    invariant ∀v ∈ visited. (∀v2 ∈ (Q ∩ conn).
          (dist[v] <= dist[v2]));
  {
    // if q = end we have found the path and can stop
    if(q = end) then
    {
      Q := ∅[vertex];
    } else {
      visited := visited ∪ {q};
      Q := Q\{q};
      // check unvisited neighborhood of chosen vertex
      var V:vertices := getNeighborhood(q,g);
      for n ∈ (V ∩ Q) do
        // all neighbours of visited nodes are connected
```

```
         invariant ∀v ∈ visited with v ≠ q.
               ∀neigh ∈ getNeighborhood(v,g).
               neigh ∈ conn;
         // all connected vertices (except start) have a
         // connected neighbor
         invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
                ∃v2:vertex with (v2 ∈ conn).
                v2 ∈ getNeighborhood(v,g);
         // defines shortest dist of visited nodes
         invariant ∀v:vertex with (v ∈ conn ∧ v ≠ start).
                ∃v2 ∈ visited. (prev[v] = v2
                    ∧ v2 ∈ getNeighborhood(v,g)
                    ∧ dist[v] = dist[v2] + 1);
         invariant ∀v:vertex with v ∈ conn.
                (∀v2:vertex with v2 ∈ conn.
                (v2 ∈ getNeighborhood(v,g) ⇒
            dist[v] <= dist[v2] + 1));
         // visited implies connected
         invariant ∀v ∈ visited. (v ∈ conn);
         // connected implies defined predecessor and distance
         invariant ∀v ∈ conn. (prev[v] ≠ N+1 ∧ dist[v] ≠ N+1);
         // Distance of visited nodes is shorter than the
         // distance of unvisited but connected nodes
         invariant ∀v ∈ visited. (∀v2 ∈ (Q ∩ conn).
                (dist[v] <= dist[v2]));
      {
         var alt:ℕ[N+1];
         // if distance is already N+1, don't raise it
         if dist[q] = N+1 then alt := N+1;
         // save alternativ distance
         else alt := dist[q] + 1;
         // if distance is smaller, then save new path
         if n ∈ conn then
         {
            if alt < dist[n] then
            {
               dist[n] := alt;
               prev[n] := q;
            }
         }
```

```
          else
          {
             dist[n] := alt;
             prev[n] := q;
             conn := conn ∪ {n};
          }
       }
    }
  }

  // if path found, then create path array
  if dist[end] ≠ N+1 then
  {
    found := true;
    var index:ℕ[N];
    var u:vertex := end;
    for index := dist[end]; index > 0; index := index - 1
    do {
       res[index - 1] := {prev[u],u};
       u := prev[u];
    }
  }

  return ⟨ found, res ⟩;
}

// ------------------------------------------------
// tests for unchecked predicates with concrete graphs
// set N ≥ 4

val testGraph:undirGraph = ⟨ {0,1,2,3,4}
              , {{0,1},{0,2},{0,3},{1,3},{2,3}} ⟩;
val testGraph2:undirGraph = ⟨ {0,1,2,3,4}
              , {{0,1},{1,4},{3,4},{2,3}} ⟩;
val testGraph3:undirGraph = ⟨ {0,1,2,3}
              , {{0,1},{1,2},{2,3},{3,0}} ⟩;

proc testIsUndirectedGraph():()
{
  print "Is testGraph an undirected graph? ";
```

```
  print isUndirectedGraph(testGraph);
  print "Is testGraph3 an undirected graph? ";
  print isUndirectedGraph(testGraph3);
  val noGraph1:undirGraph := ⟨ {}[vertex]
            , {{0,1},{1,2},{2,3},{3,0}} ⟩;
  val noGraph2:undirGraph := ⟨ {0,1,2,3}
            , {{0,1,2}} ⟩;
  print "Is noGraph1 an undirected graph? ";
  print isUndirectedGraph(noGraph1);
  print "Is noGraph2 an undirected graph? ";
  print isUndirectedGraph(noGraph2);
}

proc testGetNeighborhood():()
{
  print "Testgraph: ";
  print "Neighbors vertex 0 :";
  print getNeighborhood(0,testGraph);
  print getDegree(0,testGraph);
  print "Neighbors vertex 4:";
  print getNeighborhood(4,testGraph);
  print getDegree(4,testGraph);

  print "";
  print "Testgraph 2: ";
  print "Neighbors vertex 3:";
  print getNeighborhood(3,testGraph2);
  print getDegree(3,testGraph2);
}

proc testCompleteUndirectedGraph():()
{
  print "CompleteGraph of {0,1,2}";
  print getCompleteUndirectedGraph({0,1,2});

  print "";
  print "CompleteGraph of {0,1,3,4}";
  print getCompleteUndirectedGraph({0,1,3,4});
}

```

```
proc testIsGraphBipartite():()
{
   print "Testgraph:";
   print "Bipartite:";
   print isGraphBipartite(testGraph);

   print "";
   print "Testgraph2:";
   print "Bipartite:";
   print isGraphBipartite(testGraph2);
}

proc testInducedSubgraph():()
{
   print "Testgraph:";
   print "Induced Subgraph:";
   print inducedSubGraph(testGraph, {0,1,2,4});

   print "";
   print "Testgraph2:";
   print "Induced Subgraph:";
   print inducedSubGraph(testGraph2, {0,1});
}

proc testIsPathBetweenVertices():()
{
   var p:undirPath := Array[N,undirEdge](∅[vertex]);
   p[0] := {0,1}; p[1] := {1,3}; p[2] := {3,2};

   print "is path between vertices? Testgraph, start:0, end:2";
   print isPathBetweenVertices(p,testGraph,0,2);

   print "";
   print "is path between vertices? Testgraph, start:1, end:2";
   print isPathBetweenVertices(p,testGraph,1,2);

   print "";
   print "is path between vertices? Testgraph, start:0, end:3";
   print isPathBetweenVertices(p,testGraph,0,3);
}
```

```

proc testIsShortestPath():()
{
  // is shortest path?
  var p:undirPath := Array[N,undirEdge](∅[vertex]);
  p[0] := {0,1}; p[1] := {1,3}; p[2] := {3,2};
  print "";
  print "is shortest path between vertices? Testgraph, start:0, end:2";
  print isShortestPath(testGraph,0,2,p);

  var q:undirPath := Array[N,undirEdge](∅[vertex]);
  q[0] := {0,2};

  print "";
  print "is shortest path between vertices? Testgraph, start:0, end:2";
  print isShortestPath(testGraph,0,2,q);

  var p2:undirPath := Array[N,undirEdge](∅[vertex]);
  p2[0] := {0,1}; p2[1] := {1,2};
  print "";
  print "is shortest path between vertices? Testgraph3, start:0, end:2";
  print isShortestPath(testGraph3,0,2,p2);

  var p3:undirPath := Array[N,undirEdge](∅[vertex]);
  p3[0] := {0,3}; p3[1] := {3,2};
  print "";
  print "is shortest path between vertices? Testgraph3, start:0, end:2";
  print isShortestPath(testGraph3,0,2,p3);
}

proc testIsGraphConnected():()
{
  print "Is testGraph connected?";
  print isGraphConnected(testGraph);

  print "Is testGraph2 connected?";
  print isGraphConnected(testGraph2);
}

proc testIsPathBetweenVerticesExisting():()
```

```
{
   print "Is path between vertices existing in testGraph?";
   print isPathBetweenVerticesExisting(testGraph, 1, 2);

   print "";
   print "Is path between vertices existing in testGraph2?";
   print isPathBetweenVerticesExisting(testGraph, 1, 4);
}

proc testGetLengthOfPath():()
{
   var p:undirPath := Array[N,undirEdge](∅[vertex]);
   p[0] := {0,1}; p[1] := {1,3}; p[2] := {3,2};
   print "length of path:";
   print getLengthOfPath(p);

   var q:undirPath := Array[N,undirEdge](∅[vertex]);
   print "length of path:";
   print getLengthOfPath(q);

   var p1:undirPath := Array[N,undirEdge](∅[vertex]);
   p1[0] := {0,1}; p1[1] := {1,3}; p1[2] := {3,2}; p1[3] := {2,4};
   print "length of path:";
   print getLengthOfPath(p1);
}
```

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit 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 Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

....

Alexander Brunhuemer