Using a Program Verification Calculus for Constructing Specifications from Implementations

André Platzer

5th June 2004

Minor Thesis

University of Karlsruhe
Department of Computer Science
Institute for Logic, Complexity and Deduction Systems

Responsible Supervisor: Prof. Dr. Peter H. Schmitt
Supervisor: Dr. Bernhard Beckert

Using a Program Verification Calculus for Constructing Specifications from Implementations

André Platzer

5th June 2004

Studienarbeit

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Bernhard Beckert

Abstract

In this paper we examine the possibility of automatically constructing the program specification from an implementation, both from a theoretical perspective and as a practical approach with a sequent calculus. As a setting for program specifications we choose dynamic logic for the JAVA programming language. We show that – despite the undecidable nature of program analysis – the strongest specification of any program can always be constructed algorithmically. Further we outline a practical approach embedded into a sequent calculus for dynamic logic and with a higher focus on readability. Therefor, the central aspect of describing unbounded state changes incorporates the concept of modifies lists for expressing the modifiable portion of the state space. The underlying deductions are carried out by the theorem prover of the KeY System.

Contents

Chapter 1

Introduction

1.1 Intention

The goal of this project is to provide a means for computing the specification of a program from its implementation, whenever possible. So given a (part of a) computer program, the task is to find a specification that this program satisfies, and also to determine the conditions when this construction is feasible. Since every single program satisfies several specifications this construction should favour specifications that are in some sense superior to others. Quite clearly, for example, it should usually prefer stronger specifications to weaker specifications that are a consequence of the stronger ones. However, a second criterion for specifications should be readability, and the true problems arise when both criteria conflict. In this project, we generally focus on stronger specifications first, attaining maximum readability whenever possible. Notice that the converse priorisation would, of course, be undecidable (except for trivial specifications with no real content).

Given this setting, the results of our approach will be integrated as a module into the KeY System [Ahrendt et al., 2004]. This also necessitates that the resulting specifications are formulated in dynamic logic [Harel et al., 2001, Harel et al., 2000, Harel, 1984] for the JAVA programming language [Gosling et al., 1996]. The construction process for computed specifications will be based on the sequent calculus theorem prover of the KeY specification and verification system including the available program transformations of the underlying deduction and rewrite system.

Possible application scenarios of automatic specification construction include reverse-engineering of pre-existing code into a formal model, partial specification completion, and computer-generated descriptions of JAVA methods that use complicated statements, in single-step terminology. Whilst the latter scenario is intended for supporting the user, the former two would in principle profit from user interaction as well, but are not limited to that.

In the reverse-engineering case, our approach may help to further bridge the gap between the theoretical requirement of a completely specified formal model, and the usual practice of a mixture of non-specified code, code annotated with formal specifications, and unimplemented pure specifications. Also an automatic generation of specifications can be realised for simple cases where a user did not consider that any explicit specifications were necessary. For a full integration of partially specified code and partially implemented specifications in a two-way tool, also the inverse possibility of generating the source code according to a simple specification should be possible. However, source code generation is beyond the scope of this project. In addition to the mere engineering advantages of mixing implementation and specification, there can also be a computational advancement for proof generation. Computing the specification of a non-specified method enables bottom-up generation of proofs in a way very similar to lemma generation. Prior to proving conjectures about a piece of code, an automatic specification of all the methods it calls may speed up the proof procedure. Whenever a method call targets on a method without specification, the proof system has to step into its implementation source code. As invocations of the same method may happen more than once throughout the source code, and they even multiply on different proof branches, the same source code will have to be examined by the calculus over and over again. On the other hand if a method specification is available or can be constructed automatically then all these inspections of the method body can be replaced by the specification. In the case of a complex piece of code achieving a result that is simple to express, this could have a worthwhile effect on the overall proof length.

In the scenario of partial specifications, instead, the goal is not one of enriching completely non-specified code automatically with formal specifications. Rather, the user manually specifies some part of the program's effect and leaves its completion to the proof system.

1.2 Context

A device for automatic specification construction following our practical approach is integrated into the KeY System as a module. The KeY System [Ahrendt et al., 2004] is a semi-automatic interactive proof system based on sequent calculus for dynamic logic for Java (called JavaDL). KeY adds formal specification and verification facilities to UML[1]-based software models

[1]UML = Unified Modeling Language [Rumbaugh et al., 1998, Rumbaugh et al., 1999]

by means of theorem proving. Due to the close integration into a familiar modelling environment and a real-world programming language, KeY has surpassing prospects of accomplishing its goal of bridging the gap between what academic case-studies demonstrate is possible and industrial practice. In order to facilitate a smooth integration into the present development process, and a gradual involvement of formal methods, KeY combines with usual case-tools. Currently, KeY is available as a stand-alone prover and as a plug-in for Together [TogetherSoft, 2003].

1.3 Related Work

In the past, there have been some approaches related to automatic specification construction. First of all, immediate transformations to strongest specifications similar to the well known weakest precondition [Dijkstra, 1976] calculus. Basically, the process relies on direct transformation functions from programs to formulas, inductively defined over the syntactical constructs of the programming language. Contrary to the weakest precondition calculus they do not start from a postcondition and derive the weakest precondition required to let program runs satisfy the postcondition. Rather they take a pre-specified precondition and derive the strongest postcondition fulfilled after the general program run. See [Gannod and Cheng, 1995, Gannod and Cheng, 1997, Gannod et al., 1998] for more details.

Then there is the approach Houdini [Flanagan and Leino, 2001], intended as an add-on to the Extended Static Checker for Java (see [Detlefs et al., 1998] for ESC/Java). Houdini is an annotation assistant that guesses a large number of possible specification candidates by heuristics and then uses ESC/Java to verify or refute the individual candidates. Also refer to [Flanagan, 2002].

Some prototype experiences have been made using techniques of inductionless induction [Comon, 2001] and rippling [Bundy et al., 1993, Bundy and Lombart, 1995], for guessing induction invariants and using them for the treatment of recursion in the verification of (usually) functional programming languages.

Finally, our practical sequent calculus approach displays some similarities to program normalisation [Ammarguellat, 1992], since in one particular interpretation it can be used to construct semantically equivalent programs in some normal form.

1.4 Notation

In this section we briefly summarise the notation used in this paper. In general we adopt the logical language and notions like term, formula, model, state, satisfiable, and valid from [Beckert and Schmitt, 2003, Schmitt, 2000], with the following notational exceptions. Quantifiers and modalities bind strong instead of extending far to the right. The precedence order is thus $\leftrightarrow, \supset, \vee, \wedge, \neg, \exists, \forall, [], \langle\rangle, \doteq, \neq, <, \leq, >, \geq$. Those signs represent respectively the logical operators equivalent, implication, or, and, not, existence quantifier, universal quantifier, box operator of dynamic logic, diamond operator, term equality, term inequality, less relation, less or equal relation, greater relation, greater or equal relation. $\top$, and $\bot$ denote the formula true, resp. false. In the meta-language we use $\vDash, \equiv, \Rightarrow, \iff$ to denote the local consequence relation[2], local equivalence relation (which is a congruence), meta-level implication, or meta-level equivalence. Especially note that our binding preferences imply

$$\begin{array}{lcccc} (\forall i\,\phi) \supset \psi & = & \forall i\,\phi \supset \psi & \neq & \forall i\,(\phi \supset \psi) \\ (\langle\alpha\rangle\phi) \supset \psi & = & \langle\alpha\rangle\phi \supset \psi & \neq & \langle\alpha\rangle(\phi \supset \psi) \end{array}$$

Further, $[x \mapsto t]$ denotes the uniform substitution replacing x by t. Similarly, $\phi[x \mapsto t]$ denotes the result of replacing simultaneously every occurrence of x in ϕ by t, i.e. the result of applying the substitution $[x \mapsto t]$ to ϕ. As suggestive notation, for the semantic modification on x to a of an interpretation I (or its variable assignment part β, or its program variable state part s) we write $I[x \mapsto a]$. Then $I[x \mapsto a]$ differs from I only at x where it assigns the value a. In the same context, whenever appropriate, we prefer $\mathrm{val}_I(s, \phi)$ for the valuation of the expression ϕ under the interpretation I in the state s. That there is a possible transition from state s to t when running the program α is denoted with $s\rho(\alpha)t$. This gives the modal accessibility relation for the program α as occurring in the definition of the semantics of $\langle\alpha\rangle$ and $[\alpha]$. If the state s is known from the context and no ambiguities arise, then "there is $s\rho(\alpha)t$ " is a short notation for "there is t with $s\rho(\alpha)t$", and similar when t is known from the context.

As a matter of convention, program variables are written in typewriter `x`, logical variables in italics x. Additionally, in this paper we will use the concept of generic variables (written as $\mathbf{x}$) as will be introduced in Sect. 2.2 for simplicity of notation. For a formula ϕ, $FV(\phi)$ denotes the set of free variables occurring in ϕ.

[2]The local consequence relation with local premises Ψ and consequence ϕ is defined by $\Psi \vDash \phi \;:\iff\;$ for each interpretation (G, ρ, ℓ) for each state $s \in G$ $(s \vDash \Psi \Rightarrow s \vDash \phi)$

Chapter 2

Formalisation

In this chapter we formalise the problem of constructing a specification from an implementation, and impose restrictions on which specifications to prefer, when several are possible. After illuminating the motivations for choosing our particular formalisation with a few examples, we introduce some further terminology.

2.1 Programming Language

Even though we apply the theoretical results of the following sections to JAVA [Gosling et al., 1996] (more precisely: J2ME [J2ME, 2003] and JAVACARD [JavaCard, 1999]), they are not limited to deterministic languages but have a broader applicability[1]. In order to reflect this we first start with a hypothetical programming language called WHILE for the more formal part of this paper. We imagine it is a subset of JAVA with value semantics and primitive atomic types only, the permitted statements being exclusively `while`, `if`, assignments, and (static) function calls, perhaps of nondeterministic effect. Despite the language restrictions, WHILE still is Turing-complete. Our abstraction serves two purposes: First, it simplifies the notation in this paper. And second, it defers the treatment of peculiarities of references, aliasing and compound objects until a thorough overview of the fundamental principles has been given, hopefully to the advantage of the reader. In a later section, we then demonstrate which minor extensions facilitate the step to full J2ME. After all, the programming language does not affect things much in the following, anyway. More or less the only true restriction on the language is to have a semantics formulated in a dynamic logic.

[1] which can be of advantage, since nondeterministic effects on a macro level sometimes play a role even in languages of purely deterministic semantics like JAVA.

2.2 Specifications

For now we assume to know the modifies list[2] $MV(\alpha) = \{\mathtt{x_1}, \dots, \mathtt{x_n}\}$, which forms the maximum set of program variables or function symbols[3] modified by the program α. We formalise the general goal of this project as follows: For every program α with modifies list $MV(\alpha) = \{\mathtt{x_1}, \dots, \mathtt{x_n}\}$, find a "strongest" formula ψ of dynamic logic such that

$$\models \quad \forall {x_1}^{\mathrm{pre}} \ \dots \ {x_n}^{\mathrm{pre}} \ \big(\underbrace{\mathtt{x_1} \doteq {x_1}^{\mathrm{pre}} \land \dots \land \mathtt{x_n} \doteq {x_n}^{\mathrm{pre}}}_{\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}} \supset [\alpha]\psi\big) \tag{2.1}$$

Where the notion of a "strongest" such formula means that ψ is a maximum with respect to some specific preference order '$\preceq$' on formulas. Also, for every term[4] x_i in the modifies list $MV(\alpha)$ we provide a new logical variable symbol ${x_i}^{\mathrm{pre}}$ such that we can refer to the initial state prior to executing α.[5] Terms not in $MV(\alpha)$ of which we thus know that α will never change their value do not require such a special treatment, since referring to their prestate value is still possible in the poststate by the same name.

The intuition behind the above defining condition is that we want ψ to specify *every* possible transition (run) of α under *every* possible assignment of input variables, not just one run or assignment. The $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$ condition on the formula serves the purpose of remembering the state prior to executing α such that the specification ψ may later refer to it for state transitions depending on the initial state. Here, we refrain from using a built-in $\cdot^{\mathrm{pre}}$-operator but rather confine ourselves to basic logic. If we just see x^{pre} as an ordinary logical variable of a special name, the $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$ condition permits us to achieve a similar result without adding a special built-in mechanism.

By-reference arguments and thus instance variables necessitate an explicit treatment of state transitions via a construct like $\cdot^{\mathrm{pre}}$ variables.

Definition 2.2.1 *Any formula ψ that satisfies condition 2.1 is a* specification *of the program α.*

Also note the equivalent formulations in Res. A.4.1.

[2] We will show how to avoid the need to know the modifies list in advance, both from a theoretical and practical perspective, later on, and defer this to Chapt. 4

[3] A function symbol f occurs in the modifies list if α changes an array f at any index. This intuition will be explained more precisely in Sect. 4.1.

[4] In our case, only program variables or function symbols may occur in $MV(\alpha)$.

[5] Note that the important information about ${x_i}^{\mathrm{pre}}$ is that it is rigid and new, not that it is a logical variable. Instead, distinct rigid constants would do as well, saving universal quantifiers.

In order to simplify notation and make the affected formulas more concise, we prefer to use abbreviations in this paper. From now on, instead of explicitly referencing and managing a list of variables like $\mathtt{x_1}, \dots, \mathtt{x_n}$, we use a single vectorial or generic variable $\mathbf{x}$ to express the same circumstance. Even though we only provide this intuitive notion, the meaning of the abbreviation will always be immediate and well-defined. For example, $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$ abbreviates $\mathtt{x_1} \doteq {x_1}^{\mathrm{pre}} \wedge \dots \wedge \mathtt{x_n} \doteq {x_n}^{\mathrm{pre}}$. Similarly, as a dual subformula for remembering the poststate, $\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}$ will then abbreviate $\mathtt{x_1} \doteq {x_1}^{\mathrm{post}} \wedge \dots \wedge \mathtt{x_n} \doteq {x_n}^{\mathrm{post}}$. With this notation, condition 2.1 simplifies to

$$\models \quad \forall \mathbf{x}^{\mathrm{pre}} \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ [\alpha]\psi \big) \tag{2.2}$$

Example 2.2.1 As a plausibility check, consider the simple program α

```
x = x + 1;
```

It has the following obvious specification

$$\models \quad \forall x^{\mathrm{pre}} \big(\mathtt{x} \doteq x^{\mathrm{pre}} \ \supset \ [\mathtt{x} = \mathtt{x} + 1] \underbrace{\mathtt{x} \doteq x^{\mathrm{pre}} + 1}_{\psi} \big) \tag{2.3}$$

Because the program is deterministic and terminates, we can sharpen[6] the statement to

$$\models \quad \forall x^{\mathrm{pre}} \big(\mathtt{x} \doteq x^{\mathrm{pre}} \ \supset \ \langle \mathtt{x} = \mathtt{x} + 1 \rangle \underbrace{\mathtt{x} \doteq x^{\mathrm{pre}} + 1}_{\psi} \big)$$

$\psi_2 \ := \ \mathtt{x} > x^{\mathrm{pre}}$ is also a specification of α, though "less precise", and $\psi_3 \ := \ \mathtt{x} \neq x^{\mathrm{pre}}$ is even worse. While $\psi_4 \ := \ \mathtt{x} > 0$ or $\psi_5 \ := \ \mathtt{x} \doteq 5$ are not even specifications of α. The true formula $\top$ is admissible as a specification for every program but does not contain any exciting information about α. $\square$

Usually, it is natural to assume that a specification of a program only talks about the variables that occur in the program. As far as output variables (like $\mathbf{x}^{\mathrm{post}}$) are concerned, one would even assume that only those variables that the program modifies somehow should occur in the specification. However, in programming languages containing pointers or references (like JAVA does) which are thus subject to the aliasing problem, it can be difficult to determine precisely which variables are not modified. Moreover, even though it appears natural to restrict variables to occurring ones, it may sometimes be inappropriate. If, for example, a programmer knows that his algorithm will run at least all day long then he should be allowed to specify that, after its termination, the days count variable will have increased. Most probably, the program

[6] with knowledge of determinism

will not even talk about days or time, but nevertheless such a specification should be valid. Though, of course, an automatic specification generator cannot produce such specifications without having access to platform-specific information. Finally[7], runtime strongly depends on the specific system architecture whose peculiarities are not known to the proof system in the form of specifications in the first place. But if a programmer knows or if runtime information is available, it should still be allowed to specify runtime properties. Even though we do not officially restrict specifications to variables occurring in the program we usually guarantee that most specifications ψ satisfy the $FV(\psi) \subseteq \{\mathbf{x}^{\text{pre}}, \mathbf{x}^{\text{post}}, \mathbf{x}\} = \{{x_i}^{\text{pre}}, {x_i}^{\text{post}}, x_i \;:\; i = 1, \ldots, k\}$. Then specifications usually only talk about components of the state that modalities could directly assign values to.

At first sight, an alternative choice for the defining condition of specifications could have been

$$\vDash \quad \forall \mathbf{x}^{\text{pre}} \; \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle\psi\big) \tag{2.4}$$

Even though this variant appears to be stronger because it includes a termination assertion, it also has several drawbacks. Now we will examine the relationship of condition 2.1 and condition 2.4, and briefly mention the consequences of the two choices.

According to condition 2.1, a specification ψ contains information about what is true after all completions of the program run, if there are some. However, unlike $\langle\rangle$, the modal operator $[]$ does not talk about termination at all. To some degree, knowledge about termination results from the assertion $\vDash \langle\alpha\rangle\top$. Indeed, such an additional assertion would only guarantee that a program *can* sometimes terminate, but does not always need to terminate. For purely deterministic programs, this already is all we need, because if the single possible run terminates then the program always terminates. However, nondeterministic and probabilistic languages require a finer treatment of termination conditions, since there are programs that have one terminating run, but also several other nonterminating runs under the same input. And then $\vDash \langle\alpha\rangle\phi$ only says that we can be lucky to experience one particular program run after which ϕ holds, but leaves unspecified whether there are other unpleasant cases of nonterminating program runs, or terminating program runs after which ϕ does not hold, in spite of the same input. Those cases will be examined further in example 2.2.2 and example 2.2.3 below.

Generally, the modality $\langle\rangle$ can be inferred by $[\alpha]\phi, \langle\alpha\rangle\top \vDash \langle\alpha\rangle\phi$. So given the additional termination assertion $\vDash \langle\alpha\rangle\top$, – possibly even predicated to specific input argument values – condition 2.1 is more general than

[7]Apart from the fact that runtime analysis would be undecidable anyway.

condition 2.4. Since we want to include the treatment of nondeterministic languages, we avoid condition 2.4 and prefer condition 2.1 in the general case. Additionally, programs that do not terminate, or do not terminate under some inputs, cannot be described with a formula of the form of condition 2.4, but only of condition 2.1. Especially, if α does not terminate at all, then it still has a specification

$$\models \ \forall \mathbf{x}^{\mathrm{pre}}\big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ [\alpha]\bot\big)$$

An analogous condition would not hold for the $\langle\rangle$ version. But we explicitly want to include the treatment of such nonterminating programs in our terminology. At least programs that do not terminate under *some*, though not all, inputs occur quite frequently in practical applications. Particularly, uncaught exceptions give rise to nonterminating – more precisely abruptly completing – programs. These coherences will be illuminated more explicitly in the examples below.

Example 2.2.2 The integer division program

```
z = x / y
```

respectively the more explicit form

```
z = 0;
while (x >= y) {
    x = x - y;
    z = z + 1;
}
```

only terminates (or only terminates without error) for $\mathtt{y} \neq 0$ which can neither be expressed with $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \langle\alpha\rangle\psi$ nor with $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset ([\alpha]\psi \land \langle\alpha\rangle\top)$, but only with $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset ([\alpha]\psi \land (\langle\alpha\rangle\top \leftrightarrow \mathtt{y} \neq 0))$ □

Example 2.2.3 Regarding pseudo-random number generators as nondeterministic – for example when abstracting from their implementation – the following program is essentially nondeterministic.

```
i = 0;
while (random() < 0.5) {
    i = i + 1;
}
x = 17;
```

It satisfies $\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle(\mathtt{x} \doteq 17)$ and even satisfies $\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset ([\alpha](\mathtt{x} \doteq 17) \wedge \langle\alpha\rangle\top)$, but still does not always terminate (independent from the input). Its termination is very likely but never sure, regardless of the input. So even though after each successful termination $\mathtt{x} \doteq 17$ holds, and even though there are at least *some* termination cases, termination does not always happen. Even worse, it also is difficult to specify precisely what the termination conditions are.[8] Unexpectedly subtle cases like these contribute to the reasons why we have not stipulated termination assertions as a fixed part of our notion of specifications. □

2.3 Choosing a Preference Ordering

What we want the strongest specification to achieve is to describe precisely the input-output behaviour of a program, without concealing changes, and without inventing untrue changes. If, in a first attempt, we ignore readability and restrict ourselves to constructing the strongest specifications, the preference order '$\preceq$' can be chosen as follows.

Definition 2.3.1

$$\phi \preceq \psi \quad :\Longleftrightarrow \quad C(\psi) \supseteq C(\phi) \quad \Longleftrightarrow^{9} \quad \psi \vDash \phi \tag{2.5}$$

Where $\vDash$ denotes the local consequence relation (with respect to logical and program variables).

This directly gives us a partial quasi-order, and thus also a partial order by passing to the quotient. Unfortunately, this does *not* result in a total order on all formulas. Therefore, when comparing two formulas, like specifications, it is not at all obvious whether there always is the possibility of saying which one is stronger or even whether there is a joint strongest formula apart from $\perp$. Even though there still is no total order on specifications, and there still is no exciting top element of all formulas, we only need a greatest element of the specifications. Indeed, it will turn out that there is a computable maximal ψ_α within all specifications of α, which is a strongest specification of α: for example the arithmetic representation of α (which we will use in Res. 3.2.5).

[8] In real world applications, the distinction between probabilistic Monte Carlo and Las Vegas algorithms causes similar difficulties with termination conditions like: can terminate, always terminates, never terminates.

[9] This follows by reflexive and monotonic. Further $C(\phi) := \{\psi : \phi \vDash \psi\}$ is the set of consequences of ϕ.

Intuitively, the above choice of a preference order has two related aspects which may serve as a justification for this particular approach: First, if one specification implies another one, then we will certainly prefer the former one. Second, if a specification generally has more consequences, then we would prefer it as well. However, when comparing general sets of consequences it is easy to say which set contains more only if one set comprises the other.

Also notice that we use local consequence, since formulas that describe post states seldom hold for *every* assignment of the logical variables anyway, but rather constrain the values to some extent. This is because most programs will not permit every possible outcome. So global consequence would probably collapse to the void requirement and would not distinguish good specifications from bad ones.

Example 2.3.1 $\psi \;:=\; \mathtt{x} \doteq x^{\mathrm{pre}} + 1$ does not hold in every state on every variable assignment. So there are no interpretations and states that globally satisfy ψ regardless of the variable assignments, because ψ only holds locally in some states under some particular assignments. Still ψ holds after every execution of program α from example 2.2.1 which is what specifications are about. □

We illustrate this aspect and the surrounding concepts a little more in the following example.

Example 2.3.2 In continuation of example 2.2.1, consider again our simple program α, and its specification condition 2.3 which we repeat here

$$\vDash \;\; \forall x^{\mathrm{pre}}\big(\mathtt{x} \doteq x^{\mathrm{pre}} \;\supset\; [\mathtt{x} = \mathtt{x} + \mathtt{1}]\,\underbrace{\mathtt{x} \doteq x^{\mathrm{pre}} + 1}_{\psi}\big)$$

This specification is stronger than the following four, which are nevertheless valid specifications of α.

$$\begin{array}{rcl}
\psi_1 & = & \mathtt{x} > x^{\mathrm{pre}} \\
\psi_2 & = & \mathtt{x} \geq x^{\mathrm{pre}} + 1 \\
\psi_3 & = & \mathtt{x} > x^{\mathrm{pre}} \wedge \mathtt{x} \in \mathbf{Z} \wedge x^{\mathrm{pre}} \in \mathbf{Z} \\
\psi_4 & = & \mathtt{x} \neq x^{\mathrm{pre}} \\
\text{then} & & \psi_1 \preceq \psi\,,\ \psi_2 \preceq \psi\,,\ \psi_3 \preceq \psi\,,\ \psi_4 \preceq \psi\,, \\
 & & \psi_1 \preceq \psi_2\,,\ \psi \not\preceq \psi_2\,,\ \psi_4 \preceq \psi_1\,,\ \psi_4 \preceq \psi_2\,,\ \psi_2 \not\preceq \psi_1
\end{array}$$

In fact, ψ is the maximal specification w.r.t. $\preceq$ of α (as expected). If, instead, we had chosen global consequence as the underlying notion for stronger specifications, then every formula above would be equally strong.

We pick, for example, ψ_1 and ψ. The condition of global consequence is that in any interpretation where the one formula is true under all assignments of logical variables, the other is true under all assignments as well. But if we choose an interpretation where $>$ and $\doteq$ take their standard meaning neither $\mathtt{x} \doteq x^{\mathrm{pre}} + 1$ nor $\mathtt{x} > x^{\mathrm{pre}}$ are true under all assignments of x^{pre}. So there is no proper requirement to satisfy. □

Example 2.3.3 In the last example, the maximal specification did contain neither more nor less information than the program itself. However, in general, specifications contain less, although they only lack irrelevant internal details. What specifications do not usually talk about, for example, is the precise sequential order in which the individual operations are executed. Consider the simple program α_2

```
x = x + 1;
y = y - 1;
```

It has the following obvious (maximal) specification

$$\begin{aligned} \models \quad & \forall x^{\mathrm{pre}}\, y^{\mathrm{pre}} \big(\mathtt{x} \doteq x^{\mathrm{pre}} \wedge \mathtt{y} \doteq y^{\mathrm{pre}} \supset \\ & \quad [\mathtt{x} = \mathtt{x}+1; \mathtt{y} = \mathtt{y}-1] \underbrace{(\mathtt{x} \doteq x^{\mathrm{pre}} + 1 \wedge \mathtt{y} \doteq y^{\mathrm{pre}} - 1)}_{\psi} \big) \end{aligned} \tag{2.6}$$

Our notion of specification only characterises the input-output behaviour of a program, not its implementation. For example condition 2.6 also is a maximal specification of

```
y = y - 1;
x = x + 1;
```

and of

```
y = y + 1;
x = x + 1;
y = y - 2;
```

and if $\mathtt{x}, \mathtt{y} \in \mathbf{Z}$ even of

```
int y0 = y;
if (y > x) {
    y = y + 2;
} else {
    y = y - 1;
}
x = x + 1;
```

```
if (y0 >= x) {
    y = y - 3;
}
```

This example also demonstrates incomparable specifications. The following two specifications are weaker than ψ but incomparable to each other.

$$\begin{aligned}
\psi_1 &= \mathtt{x} = x^{\mathrm{pre}} + 1 \\
\psi_2 &= \mathtt{y} = y^{\mathrm{pre}} - 1 \\
&\Rightarrow \psi_1 \preceq \psi \,,\, \psi_2 \preceq \psi \,,\, \psi_1 \npreceq \psi_2 \,,\, \psi_2 \npreceq \psi_1
\end{aligned}$$

□

Example 2.3.4 Even more than the last example, this one shows what is hidden from the specification of a program in comparison to the full detail of the implementation. The following three programs all have equivalent maximal specifications.

```
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

Reversible computing version:

```
x = x + y;
y = y - x;
x = x + y;
y = -y;
```

Geometrically inspired version:

```
{
    int d = Math.abs(x - y);
    if (x < y) {
        x = x + d;
        y = y - d;
    } else {
        x = x - d;
        y = y + d;
    }
}
```

which all satisfy

$$\vDash \quad \forall x^{\text{pre}}\ y^{\text{pre}}\ \big(\mathtt{x} \doteq x^{\text{pre}} \land \mathtt{y} \doteq y^{\text{pre}} \supset \\ [\alpha]\mathtt{x} = y^{\text{pre}} \land \mathtt{y} = x^{\text{pre}}\big)$$

□

2.4 Terminology

In this section we will introduce some basic terminological notions which will be needed in the following.

Definition 2.4.1 *(Semantically relatively rigid / Invariant) An expression*[10] ϕ *is called* rigid for the program α *or invariant under* α *if it has the same value prior to and after executing* α*, i.e.*

$$\textit{for each state } s \ \ \textit{for each state } t \ \ \big(s\rho(\alpha)t \Rightarrow \mathrm{val}(s,\phi) = \mathrm{val}(t,\phi)\big)$$

Remark 2.4.2 *In particular, an expression is rigid for* α *if it is only build of components that are rigid for* α*.*

Definition 2.4.3 *(Semantically rigid) An expression* ϕ *is called* semantically rigid *if it has the same value in all worlds, i.e.*

$$\textit{for each state } s \ \ \textit{for each state } t \ \ \mathrm{val}(s,\phi) = \mathrm{val}(t,\phi)$$

Remark 2.4.4 *Especially, an expression is rigid if it is only build of rigid constituents.*

Definition 2.4.5 *(Syntactically rigid) A (function) symbol that the syntactic vocabulary classifies* a priori *as having a value that is independent from the particular state is* (syntactically) rigid*. In analogy to the semantical definitions, expressions that are only composed of syntactically rigid atomic terms are called syntactically rigid. Contrary to program variables which are non-rigid constants, logical variables are rigid by definition.*

The relationship between the various notions of rigidity is as follows.

[10] We use expressions as a unifying notion of elements of a suitable term-algebra like formulas or terms of dynamic logic.

Remark 2.4.6 *Let ϕ be an expression and α an arbitrary program.*

$$\begin{array}{ll} & \phi \text{ syntactically rigid} \\ \Rightarrow & \phi \text{ semantically rigid} \\ \Rightarrow & \phi \text{ rigid for } \alpha \end{array}$$

Proof: Directly by definition. ■

Note that most of our results only require formulas that are rigid for the specific program being analysed. However, unlike semantical rigidity, syntactical rigidity bears no decidability problems because it emanates from the declaration of symbols.

During the proofs in this work, there is a need for specifications of a slightly different defining condition. While the initial notion of specifications in Res. 2.2.1 has a strong intuitive justification, there is an alternative characterisation, which permits a simpler generalisation to characterisations of maximal specifications. This alternative characterisation of program specifications will constitute an important ingredient, both in the theoretical result Res. 3.2.5 and in the practical result Res. 5.5.1. Therefore, we introduce special names for these concepts and collect some properties in the following sections.

Definition 2.4.7 *(Exterior form specification) Any formula ψ that satisfies the following condition is an* exterior specification *of the program α.*

$$\models \quad \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \Big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ \big(\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}) \ \supset \ \psi\big)\Big) \tag{2.7}$$

The antecedent placement of the $\langle\rangle$ modality in condition 2.7 – in combination with the state export via $\mathbf{x}^{\mathrm{post}}$ – has the effect of a *generic* program run. The state described by $\mathbf{x}^{\mathrm{post}}$ is existentially quantified, with this quantification placed in the antecedent of an implication. ψ only knows about $\mathbf{x}^{\mathrm{post}}$ that it is some unknown or anonymous state that can be reached by executing α in some way. So whatever ψ knows about some such anonymous state, it effectively knows about all such states, since it has no way to influence which particular state the $\langle\rangle$ operator reaches.

With this being said, the intuition of the above condition is that if ψ can say something about the generic program run, i.e. just with knowledge of the fact that there *can* be some program run leading to some anonymous $\mathbf{x}^{\mathrm{post}}$, then it eventually holds for any program run. And things that hold of all program runs are what is called a specification of the program.

Example 2.4.1 In continuation of example 2.2.1, we remember that the simple increment program has the following obvious specification

$$\models \quad \forall x^{\mathrm{pre}} \big(\mathtt{x} \doteq x^{\mathrm{pre}} \ \supset \ [\mathtt{x} = \mathtt{x} + 1] \underbrace{\mathtt{x} \doteq x^{\mathrm{pre}} + 1}_{\psi}\big)$$

Furthermore, α has the following exterior specification

$$\models \ \forall x^{\mathrm{pre}} \forall x^{\mathrm{post}} \Big(\mathtt{x} \doteq x^{\mathrm{pre}} \ \supset \ \big(\langle \mathtt{x} = \mathtt{x} + 1 \rangle (\mathtt{x} \doteq x^{\mathrm{post}}) \ \supset \ \underbrace{x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 1}_{\psi'} \big) \Big)$$

We observe that the difference between ψ and ψ' is just the way of referring to the poststate. This difference results from the different levels of modality. $\square$

Example 2.4.2 As an example to illustrate the concept of generic program runs, let us examine a possible translation into an ordinary first-order analogon. This "translation" only leads to informal language. Assume that ψ is a formula that only contains $\mathbf{x}^{\mathrm{post}}$ as free variables.

$$\begin{array}{ll} & \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}) \ \supset \ \psi \\ \hat{=} & (\text{there is } \ t \text{ with } s\rho(\alpha)t \ \ t \models \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}) \ \ \Rightarrow \ \ s \models \psi \\ \hat{=} & \text{for each } \ \ t \text{ with } s\rho(\alpha)t \ \ (t \models \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \ \Rightarrow \ s \models \psi) \\ \hat{=} & \text{for each } \ \ t \text{ with } s\rho(\alpha)t \ \ (t \models \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \ \Rightarrow \ t \models \psi) \\ \hat{=} & \text{for each } \ \ t \text{ with } s\rho(\alpha)t \ \ (t \models \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \supset \psi) \end{array}$$

Thereby we can see how the existential statement in the antecedent of a condition corresponds to a universal statement on the top-level of positive polarity. The $\mathbf{x}^{\mathrm{post}}$ variables achieve an export of the state information within the scope of the existential statement to the succedent ψ. $\square$

Be aware that exterior specifications forecast a subtle change in the use of modalities. While in Res. 2.2.1 the prevailing modality has been [] and a considerable amount of motivation has been spent to justify its preference over $\langle\rangle$ from condition 2.4, now Res. 2.4.7 re-establishes the use of $\langle\rangle$. But the important difference is that $\langle\rangle$ occurs with *negative* polarity in condition 2.7, thereby raising hope that there could be a connection with the occurrence of [] with positive polarity in condition 2.1. At least for rigid specifications, this hope will not be shattered.

Some specifications only describe the upcoming state change when evaluated prior to executing α, while other specifications only portray the state change in retrospect when evaluated after executing α. Particularly good specifications, however, ensure that the specification formula is not restricted to holding at a particular point in time, but describes the state change performed by α universally and independently from the current context. Those specifications characterise the state change equally precise in the prestate as well as in the poststate. This is important, because some proofs in this paper take advantage of such a uniform systematic treatment of pre- and poststate. We will see that specifications can always be cast into a form which respects a homogeneous treatment of pre- and poststate values, which we call standard form.

Definition 2.4.8 *(Standard form) Any formula ψ that satisfies the following two equivalent conditions is a* standard form specification

$$\begin{aligned} &\vDash \ \forall \mathbf{x}^{\mathrm{pre}} \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \\ &\qquad [\alpha]\forall \mathbf{x}^{\mathrm{post}} \ (\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \supset \psi)\big) \\ &\vDash \ \forall \mathbf{x}^{\mathrm{pre}} \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \\ &\qquad [\alpha]\exists \mathbf{x}^{\mathrm{post}} \ (\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \wedge \psi)\big) \end{aligned}$$

Standard forms are a minor variation of ordinary specifications. Furthermore, both variants, defining condition 2.1 and condition 2.7 can be united and (in the case of rigidity for α) expressed equivalently by standard form specifications. In Res. 3.1.6 we will finally explain more precisely that all the variants are essentially equivalent, anyhow.

Example 2.4.3 In continuation of example 2.4.1, the specification of the simple increment program can be transformed to the following standard form specification

$$\begin{aligned} &\vDash \ \forall x^{\mathrm{pre}} \big(\mathtt{x} \doteq x^{\mathrm{pre}} \ \supset \\ &\qquad [\mathtt{x} = \mathtt{x} + \mathtt{1}]\forall x^{\mathrm{post}} \, (\mathtt{x} \doteq x^{\mathrm{post}} \supset \underbrace{x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 1}_{\psi''})\big) \end{aligned}$$

Here, we find that the exterior specification ψ' and the standard form specification ψ'' are equal, despite their distinct defining conditions. Even though this is the normal "reasonable" case, exterior and standard form specifications do not necessarily coincide. For example, $\mathtt{x} \doteq x^{\mathrm{pre}} + 1$ will still be an (unusual) standard form specification but not an exterior specification of α. This is because of the different levels of modality where the program variable $\mathtt{x}$ occurs in both defining conditions.

Standard form specification are more systematic though less readable than ordinary specifications. Furthermore, they avoid the asymmetric naming convention of calling prestate values $\mathbf{x}$ in prestates, but decorating prestate values with $\mathbf{x}^{\mathrm{pre}}$ in poststates, while calling poststate values again $\mathbf{x}$ in poststates. Instead the proper specification part will refer to the pre- and poststate values equally explicitly. Thus such standard form specifications can be removed from their context more easily without disturbing their reference to the surroundings, which is why we prefer to use standard form specifications in our proofs for simplicity. $\square$

By now, the essential notions have been presented. Unfortunately, though, there still is a need for some technical remarks concerning the translation of the different kinds of specifications. Those transformations rely on the presence of rigidity.

Remark 2.4.9 *For standard form specifications, there is no need to contain* $\mathbf{x}$ *as a (free) variable, since previous and post state values of* $\mathbf{x}$ *can be addressed in rigid variables* $\mathbf{x}^{\mathrm{pre}}$ *and* $\mathbf{x}^{\mathrm{post}}$ *where they have been memorised. And occurrences of* $\mathbf{x}$ *in between (i.e. within modalities of the specification* ψ*) could have been called* $\mathbf{x}'$ *just as well.*

Some of our results implicitly assume this purification has been performed, because they would require technical acrobatics otherwise.

Remark 2.4.10 *We implicitly assume that standard form specifications do not contain* $\mathbf{x}$*, which is no loss of generality by Res. 2.4.9.*

There is one more terminologically simplifying concept that we need for the concise formulation of results. Since in some situations, like Res. A.1.2, the possible variations of formulas have the same justification, we do not want to treat them disparately or take one as original and the other as derived.[11] Instead, we write $\phi(z)$, and $\phi(t)$ respectively for both variations of ϕ having z or t, respectively at corresponding positions. Formally, this intuitive notion can be made precise by introducing an original variable λ_1 in ϕ, which plays the role of a formal parameter of the meta-language, and by defining $\phi(z) \ := \ \phi[\lambda_1 \mapsto z]$, resp. $\phi(t) \ := \ \phi[\lambda_1 \mapsto t]$.

[11] In fact, in the case of Res. A.1.2, both, z and t can be transformed to one another with generous uniform substitutions like $[z \mapsto t]$ and $[t \mapsto z]$

Chapter 3

Computability Analysis

In this chapter, we analyse the problem of specification construction from a theoretical perspective. We will prove that (up to local equivalence) the strongest specification of an arbitrary program exists and can be constructed algorithmically. For that proof and the following course of this paper, we first introduce some key coherences, which correlate statements about different levels of modality.

3.1 Hop States

Talking about states other than the current requires a way of rescuing information from states over modal operator boundaries. For that purpose, the state-dependent non-rigid information (for example program variables) has to be emulated with rigid logical variables. In doing so replacements will be carried out in the formulas. Those, indeed, should not perform semantical free flight, but exhibit the same meaning as the original formulas. On that account, a relationship between the evaluation of formulas with replacements and corresponding evaluations of the unchanged formulas has to be established. Such a kind of substitution procedure must not deploy surrogates beyond arbitrary modal operators, because occurrences of the same $\mathtt{x}$ on different levels of quantification may denote different values and thus different rigid correspondents. From a syntactical point of view such descriptions need a concept of substitutions that surrender in the face of modalities. However, instead of pursuing this line, we prefer a less technical approach and rely on Res. 2.4.9 to rename conflicting occurrences of $\mathtt{x}$.

Example 3.1.1 $\phi \ := \ \mathtt{x} \doteq 0 \supset [\mathtt{x} = \mathtt{x} + 1]\mathtt{x} \doteq 1$ is true in all states, but $\phi(i) \ := \ i \doteq 0 \supset [\mathtt{x} = \mathtt{x} + 1]i \doteq 1$ is unsatisfiable with i being a (rigid) logical variable. $\square$

Remark 3.1.1 (Rigidify) *With the help of Res. 2.4.9 the following statements are true*

$$\begin{aligned} \Big(\forall i\left(i \doteq \mathtt{x} \supset \phi(i)\right)\Big) &\quad\equiv\quad \phi(\mathtt{x}) \\ \vDash \quad \forall i\left(i \doteq \mathtt{x} \supset (\phi(i) \leftrightarrow \phi(\mathtt{x}))\right) \end{aligned}$$

Proof: The first is a direct consequence of Res. A.1.2, and the second then follows. $\blacksquare$

For examining the relationship of propositions on different levels of modality, we now show an auxiliary result that correlates preconditions in Hoare-logic style with the corresponding conditions on the previous state in post-conditions. It enables us to talk about *the* previous state in the posterior state.

Lemma 3.1.2 (Pushforward) *If $\phi(i)$ is rigid for α (and i is fresh), and t an arbitrary term, then for $\star \in \{\vee, \supset\}$*

$$\phi(\mathbf{x}) \star [\alpha]\psi \;\equiv\; \forall i\left(i \doteq \mathbf{x} \supset [\alpha](\phi(i) \star \psi)\right)$$

Proof: By Res. 3.1.1 it is

$$\begin{aligned}
& \vDash \; \forall i\Big(i \doteq \mathbf{x} \supset \left(\phi(\mathbf{x}) \leftrightarrow \phi(i)\right)\Big) \\
\overset{\text{congruence}}{\Rightarrow} & \vDash \; \forall i\Big(i \doteq \mathbf{x} \supset \left((\phi(\mathbf{x}) \star [\alpha]\psi) \leftrightarrow (\phi(i) \star [\alpha]\psi)\right)\Big) \\
\overset{\text{Res. A.2.1}}{\Longleftrightarrow} & \vDash \; \forall i\Big(i \doteq \mathbf{x} \supset \left((\phi(\mathbf{x}) \star [\alpha]\psi) \leftrightarrow [\alpha](\phi(i) \star \psi)\right)\Big) \\
\Rightarrow & \vDash \; \forall i\Big(\left(i \doteq \mathbf{x} \supset (\phi(\mathbf{x}) \star [\alpha]\psi)\right) \leftrightarrow \left(i \doteq \mathbf{x} \supset [\alpha](\phi(i) \star \psi)\right)\Big) \\
\overset{i \notin FV(\{\phi(\mathbf{x}),\alpha,\psi\})}{\Rightarrow} & \vDash \; \forall i\Big((\phi(\mathbf{x}) \star [\alpha]\psi) \leftrightarrow \left(i \doteq \mathbf{x} \supset [\alpha](\phi(i) \star \psi)\right)\Big) \\
\overset{\text{Res. A.3.1}}{\Rightarrow} & \vDash \; (\phi(\mathbf{x}) \star [\alpha]\psi) \leftrightarrow \forall i\left(i \doteq \mathbf{x} \supset [\alpha](\phi(i) \star \psi)\right)
\end{aligned}$$

$\blacksquare$

Corollary 3.1.3 (Pushforward) *If $\phi(\mathbf{x}^{\text{pre}})$ is rigid for α, then for $\star \in \{\vee, \supset\}$*

$$\begin{aligned} \vDash \; \forall \mathbf{x}^{\text{pre}} \forall \mathbf{x}^{\text{post}} \Big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \;\supset \\ \left((\phi(\mathbf{x}) \star [\alpha]\psi) \;\leftrightarrow\; [\alpha](\phi(\mathbf{x}^{\text{pre}}) \star \psi)\right)\Big) \end{aligned}$$

Proof: By congruence, this is a direct consequence of Res. 3.1.2, subject to the additional relativising condition $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$. Thereby note that quantifier transposition is possible for $\mathbf{x}^{\mathrm{pre}}$. ■

This observation shows that there is no restriction due to our decision not to consider explicit separate preconditions as part of the specification (no pairs of precondition and postcondition, apart from $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$), since the postcondition ψ under the box still can catch up.

From this result we can further conclude that, in principle, there is no need to distinguish sets of several program specifications[1] from a single program specification, since both can be translated into one another. In particular, this means that we do not lose generality when restricting attention to a single specification.

Corollary 3.1.4 *(Finite) sets of specifications are equivalent to a single specification by*

$$\begin{aligned}\vDash\ \ \forall \mathbf{x}^{\mathrm{pre}} \Big(&\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \\ &\big(\textstyle\bigwedge_k (\phi_k(\mathbf{x}) \supset [\alpha]\psi_k) \ \leftrightarrow\ [\alpha]\big(\textstyle\bigwedge_k (\phi_k(\mathbf{x}^{\mathrm{pre}}) \supset \psi_k)\big)\big)\Big)\end{aligned}$$

Proof: by using

$$(A \leftrightarrow B) \land (C \leftrightarrow D) \vDash (A \land C) \leftrightarrow (B \land D)$$

$$[\alpha]A \land [\alpha]B \iff [\alpha](A \land B)$$

Provided determinism, the same result holds true for $\langle\rangle$ operators, when using

$$\langle\alpha\rangle A \land \langle\alpha\rangle B \iff \langle\alpha\rangle(A \land B)$$

■

Now we state and prove a very useful method of referring to the state within the scope of a modal operator from outside, thereby trespassing the state transition barrier. In a certain way we export a state from under a modal operator. This result, which is the converse of Res. 3.1.3, then allows us to talk about *a* generic post state already in the previous state. Talking about that generic poststate from under the $\langle\rangle$ modality condition still is equivalent to speaking about all states in a [] modality, according to the next result. This constitutes the key observation. It heralds a creeping transition from [] to $\langle\rangle$ modalities, in the spirit of exterior specifications.

[1]A set of program specifications for a single program describes multiple cases of different state transitions, depending on the particular precondition satisfied. Thus they are implicitly conjunctively connected.

Lemma 3.1.5 (Pullback) *If ψ is rigid for α (and i does not occur within the formula ϕ, or α), then*

$$[\alpha]\big(\phi \supset \forall i\,(i \doteq \mathtt{x} \supset \psi)\big) \quad\equiv\quad \forall i\,\big(\langle\alpha\rangle(\phi \wedge i \doteq \mathtt{x}) \supset \psi\big)$$

Proof:

$$\begin{array}{rl}
 & [\alpha]\big(\phi \supset \forall i\,(i \doteq \mathtt{x} \supset \psi)\big) \\
\overset{i \notin FV(\phi)}{\equiv} & [\alpha]\forall i\,\big(\phi \wedge i \doteq \mathtt{x} \supset \psi\big) \\
\overset{i \notin \alpha,\text{constant domain}}{\equiv} & \forall i\,[\alpha]\big(\phi \wedge i \doteq \mathtt{x} \supset \psi\big) \\
\equiv & \forall i\,[\alpha]\big(\neg(\phi \wedge i \doteq \mathtt{x}) \vee \psi\big) \\
\overset{\text{Res. A.2.1}}{\equiv} & \forall i\,\big([\alpha]\neg(\phi \wedge i \doteq \mathtt{x}) \vee \psi\big) \\
\equiv & \forall i\,\big(\neg\langle\alpha\rangle(\phi \wedge i \doteq \mathtt{x}) \vee \psi\big) \\
\equiv & \forall i\,\big(\langle\alpha\rangle(\phi \wedge i \doteq \mathtt{x}) \supset \psi\big)
\end{array}$$

■

From this fact, we can establish a translation between standard form specifications of Res. 2.4.8 and the exterior specifications from Res. 2.4.7. Standard form specifications are just a more systematic variant of the initial notion of specifications from Res. 2.2.1 with their intuitive motivation. And since theoretical and practical approaches presented later will be able to produce exterior specifications, it is important that the following result finally reunites the distinct variants of defining conditions for specifications.

Corollary 3.1.6 *If ψ is rigid for α, then ψ is a standard form specification of α (c.f. Res. 2.4.8) $\iff$ ψ is an exterior specification of α (c.f. Res. 2.4.7).*

Proof: Using Res. 3.1.5 with $\phi := \top, i := \mathbf{x}^{\text{post}}$ applied to the marked part we can conclude

$$\begin{array}{l}
\vDash \ \forall \mathbf{x}^{\text{pre}}\,\forall \mathbf{x}^{\text{post}} \\
\quad \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \ \supset\ \underbrace{(\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \ \supset\ \psi(\mathbf{x}^{\text{post}}))}_{\equiv[\alpha]\psi(\mathbf{x})}\big)
\end{array}$$

Note that $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ does not contain $\mathbf{x}^{\text{post}}$. Therefore the quantifier $\forall \mathbf{x}^{\text{post}}$ can be rearranged correspondingly. ■

3.2 Maximal Specifications

Having defined what we actually aim to find as program specifications, we will now examine whether there is a way of constructing such specifications. It will be shown that specifications can be derived from the logical representation of a (usually imperative) program. Whereas the logical representations in turn can be constructed algorithmically from the program's source code – or, in particular in the case of JAVA, just as well from the compiled binary representation. We start by translating programs to logic, which is only one possible way of achieving this.

Claim 3.2.1 (Translating programs to logic) *For every program α there is a (standard form) specification ψ_α of α that is rigid for α[2], and even[3]*

$$\vDash \ \forall \mathbf{x}^{\mathrm{pre}} \ \forall \mathbf{x}^{\mathrm{post}} \ \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}) \leftrightarrow \psi_\alpha)\big) \tag{3.1}$$

Proof: Either prove by a general result shown for example in [Schlager, 2000], or by explicit argument: Let $f : \mathbf{N}^n \to \mathbf{N}^m$ be the function computed by α. Then f has for example an arithmetic representation, i.e. an arithmetic formula

$$\psi_\alpha = \psi_\alpha(\mathbf{x}^{\mathrm{pre}}, \mathbf{x}^{\mathrm{post}})$$

over $\{\neg, \wedge, \vee, \forall, \exists, \doteq, +, \cdot\} \cup \mathbf{N}$ such that

$$\begin{aligned} & \vDash \ \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \, (\psi_\alpha \ \leftrightarrow \ f(\mathbf{x}^{\mathrm{pre}}) \doteq \mathbf{x}^{\mathrm{post}}) \\ \Rightarrow \ & \vDash \ \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \, \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ (f(\mathbf{x}) \doteq (\mathbf{x}^{\mathrm{post}}) \leftrightarrow \psi_\alpha)\big) \\ \Rightarrow \ & \vDash \ \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \, \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \ \supset \ (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}}) \leftrightarrow \psi_\alpha)\big) \end{aligned}$$

This proves the conjecture, and therefore ψ_α is a specification by Res. 3.1.6.[4]

■

Note that this translation results in a first-order arithmetic formula over natural numbers, but anything comparably strong (like for example dynamic logic) would have been suitable as well. So there are no miracles here. We cannot translate programs into pure first-order logic without "built-in" integers. One particular construction trick, again proving the above result, plays a role in the practical approach as well, which is why it comes as a separate claim.

[2]The usual case will even be $FV(\psi_\alpha) \subseteq \{\mathbf{x}^{\mathrm{pre}}, \mathbf{x}^{\mathrm{post}}\}$, but this is not a necessary consequence.

[3]Note the similarity with condition 2.7 of exterior specifications.

[4]For ψ_α we could just as well choose the (perhaps second-order) formula resulting from a transformation of α into a functional specification and further on to logic.

Claim 3.2.2 *In α replace every program variable $\mathbf{x}$ by $\mathbf{x}'$ and call the resulting program α'. Then Res. 3.2.1 is true with*

$$\psi_\alpha \quad := \quad \langle \mathbf{x}' = \mathbf{x}^{\text{pre}} \rangle (\langle \alpha' \rangle (\mathbf{x}^{\text{post}} \doteq \mathbf{x}'))$$

Proof: This choice for ψ_α says that states described by $\mathbf{x}^{\text{post}}$ are reachable by α' thus (apart from variable renaming) by α, when starting in a state described by $\mathbf{x}^{\text{pre}}$. In the end, this is all one can say about the general program run, namely that we have finally reached any one of the states that we could possibly reach by executing the program (starting in the very same particular state under consideration). By construction, it is immediate (Res. A.4.2) that this ψ_α is rigid for α because it does not share anything modifiable with α. Now to the proof of condition 3.1. Let s be any state with $s \vDash \mathbf{x} \doteq \mathbf{x}^{\text{pre}}$.

1. If $s \vDash \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \Rightarrow$ there is $s\rho(\alpha)t \;\; t \vDash \mathbf{x} \doteq \mathbf{x}^{\text{post}}$. Then after $\langle \mathbf{x}' = \mathbf{x}^{\text{pre}} \rangle$[5], s is "the same" starting point for both, α and α'. More precisely: because of the mere variable renaming $[\mathbf{x} \mapsto \mathbf{x}']$, $s\rho(\alpha)t$ implies with $t' := t[\mathbf{x} \mapsto t(\mathbf{x}')][\mathbf{x}' \mapsto t(\mathbf{x})]$ that $s\rho(\alpha')t'$ and $t' \vDash \mathbf{x}^{\text{post}} \doteq \mathbf{x}'$.

2. If $s \nvDash \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \Rightarrow$ for each $s\rho(\alpha)t \;\; t \nvDash \mathbf{x} \doteq \mathbf{x}^{\text{post}}$. Then let $s\rho(\mathbf{x}' = \mathbf{x}^{\text{pre}})s'$ and $s'\rho(\alpha')t$. Because of the variable renaming then with $t := t'[\mathbf{x} \mapsto t'(\mathbf{x}')][\mathbf{x}' \mapsto t'(\mathbf{x})]$ it is $s\rho(\alpha)t$ and thus $t \nvDash \mathbf{x}^{\text{post}} \doteq \mathbf{x}$, but then $t' \nvDash \mathbf{x}^{\text{post}} \doteq \mathbf{x}'$ by construction of t from t'.

■

Corollary 3.2.3 *The condition 3.1 is equivalent to*

$$\begin{aligned} \vDash \;& \forall \mathbf{x}^{\text{pre}} \; \forall \mathbf{x}^{\text{post}} \\ & \Big(\big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \supset \psi_\alpha)\big) \\ & \wedge \big(\psi_\alpha \supset (\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}))\big)\Big) \end{aligned}$$

When ψ_α is rigid for α, then condition 3.1 is equivalent to

$$\begin{aligned} \vDash \;& \forall \mathbf{x}^{\text{pre}} \; \forall \mathbf{x}^{\text{post}} \\ & \big(\langle \mathbf{x} = \mathbf{x}^{\text{pre}} \rangle \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \leftrightarrow \psi_\alpha\big) \end{aligned} \tag{3.2}$$

while further equivalent formulations of condition 3.1 then result from

$$\begin{aligned} & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \leftrightarrow \psi_\alpha) \\ \equiv \;& \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}} \leftrightarrow \psi_\alpha) \\ \equiv \;& \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha](\mathbf{x} \doteq \mathbf{x}^{\text{post}} \supset \psi_\alpha)\big) \wedge \big(\psi_\alpha \supset (\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle \alpha \rangle (\mathbf{x} \doteq \mathbf{x}^{\text{post}}))\big) \end{aligned}$$

[5]We still call the state after this modality s, for simplicity

Proof:

$$
\begin{array}{cl}
 & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \leftrightarrow \psi_\alpha) \\
\equiv & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \big((\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \supset \psi_\alpha) \wedge (\psi_\alpha \supset \langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}))\big) \\
\equiv & \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \supset \psi_\alpha)\big) \\
 & \wedge\big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\psi_\alpha \supset \langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}))\big) \\
\equiv & \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \supset \psi_\alpha)\big) \\
 & \wedge\big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \wedge \psi_\alpha \supset \langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}})\big) \\
\equiv & \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}) \supset \psi_\alpha)\big) \\
 & \wedge\big(\psi_\alpha \supset (\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}}))\big) \\
\overset{\text{Res. 3.1.6}}{\equiv} & \big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha](\mathbf{x} \doteq \mathbf{x}^{\text{post}} \supset \psi_\alpha)\big) \\
 & \wedge\big(\psi_\alpha \supset (\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle\mathbf{x} \doteq \mathbf{x}^{\text{post}})\big) \\
\text{And} & \\
 & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset (\langle\alpha\rangle\mathbf{x} \doteq \mathbf{x}^{\text{post}} \leftrightarrow \psi_\alpha) \\
\overset{\text{Res. A.2.1}}{\equiv} & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle(\mathbf{x} \doteq \mathbf{x}^{\text{post}} \leftrightarrow \psi_\alpha)
\end{array}
$$

The equivalences marked with propositions rely on ψ_α being rigid. The remaining equivalence with condition 3.2 follows from Res. A.1.3 when keeping in mind that ψ_α is rigid for α. Thus the particular value of $\mathbf{x} \in MV(\alpha)$ cannot make a difference for ψ_α, so moving it in and out of the scope of $\langle\mathbf{x} = \mathbf{x}^{\text{pre}}\rangle$ does not change anything[6]. ∎

The part $\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha](\mathbf{x} \doteq \mathbf{x}^{\text{post}} \supset \psi_\alpha)$ of the last formula from Res. 3.2.3 intuitively says that when running α from the state memorised by $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$, then the specification formula is true after all possible program runs – assuming the (universally quantified) final state variables have been assigned appropriately.

The other part $\psi_\alpha \supset (\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset \langle\alpha\rangle\mathbf{x} \doteq \mathbf{x}^{\text{post}})$ from Res. 3.2.3 says that whenever the specification formula is satisfied there is – given a corresponding assignment of the input variables – a program run of the expected result.

Now that we have a computable translation of a program, we can show that such a translation really is "the" strongest specification for a program.

Remark 3.2.4 *In the following, for a program α, ψ_α will always denote any formula satisfying condition 3.1 including the additional assertion that it is rigid for α.*

[6]Because if ψ_α was sensitive to its placement in relation to $\langle\mathbf{x} = \mathbf{x}^{\text{pre}}\rangle$, it would obviously notice changes of $\mathbf{x}$, and thus could never have been rigid for α

Proposition 3.2.5 (Strongest Specifications) *ψ_α is a maximum with respect to '$\preceq$' of all (standard form) specifications of α.*

Proof: Let ϕ be a (standard form) specification of α. We have to show that $\phi \preceq \psi_\alpha$, i.e.

$$\models \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \big((\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \psi_\alpha) \ \supset \ (\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \phi) \big)$$

This is equivalent to proving that

$$\models \forall \mathbf{x}^{\mathrm{pre}} \forall \mathbf{x}^{\mathrm{post}} \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \ (\psi_\alpha \supset \phi) \big)$$

Since ϕ is a (standard form) specification,

$$\begin{array}{rl} & \models \ \forall \mathbf{x}^{\mathrm{pre}} \ \forall \mathbf{x}^{\mathrm{post}} \ \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \ [\alpha](\mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \supset \phi) \big) \\ \overset{\text{Res. 3.1.6}}{\Longleftrightarrow} & \models \ \forall \mathbf{x}^{\mathrm{pre}} \ \forall \mathbf{x}^{\mathrm{post}} \ \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \ (\langle \alpha \rangle \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \supset \phi) \big) \end{array}$$

By premises, ψ_α satisfies condition 3.1

$$\models \ \forall \mathbf{x}^{\mathrm{pre}} \ \forall \mathbf{x}^{\mathrm{post}} \ \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \ (\langle \alpha \rangle \mathbf{x} \doteq \mathbf{x}^{\mathrm{post}} \ \leftrightarrow \ \psi_\alpha) \big)$$

Therefore when combining the above formulas by congruence we get

$$\models \ \forall \mathbf{x}^{\mathrm{pre}} \ \forall \mathbf{x}^{\mathrm{post}} \ \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \ (\psi_\alpha \ \supset \ \phi) \big)$$

■

At this stage we have shown that there is an effective procedure for computing the strongest specification of any program. From a theoretical perspective, the specification computation problem is solved. However the construction in proof Res. 3.2.1 reveals that the specification thus produced is only a recoding of the source code and as such does not really offer more insight than the source code itself, nor necessarily attains better readability. For this reason it seems adequate to look out for practical approaches with a stronger focus on readability and beneficial use than on generality. In Chapt. 5 we present one approach based on a sequent calculus.

Chapter 4

The Modifies List

In this chapter, we investigate the problem of describing the previous state in object-oriented languages. Unlike with the WHILE programming language, JAVA programs can modify an unbounded number of memory locations by following references in dynamic data structures. Strongest specifications have to describe the change for all memory locations, where the new values generally depend on the old values. How the previous state values of the unlimited set of possibly modified locations can be memorised for the specification will be examined next.

4.1 Generic Modifies List

Having provided the feasibility analysis and theoretical solutions for automatic specification construction, we relieve of the burden of the WHILE programming language imagined so far, and float back to the real world of object-oriented programming languages. In this section we address the central problem of the modifies list $MV(\alpha)$. Throughout our analysis we have made use of this maximum list of terms (or locations) that the program under consideration can possibly modify. Now we examine methods to avoid the need for this *a priori* knowledge. Of course, determining the smallest modifies list is undecidable, but reasonable approximations will suffice for our purpose.

Somewhat contrary to applications of the modifies list in proof techniques (like [Beckert and Schmitt, 2003]), isolating the modified memory locations as narrowly as possible is of less impact here. What's rather more important, is a concise description of every state element that can possibly be modified by the program, such that no information about the previous state disappears when moving to the poststate.

In case of a programming language like WHILE with value semantics on atomic types and without pointers, a conservative estimation of the modifies list can be computed very easily. The set of all program variables occurring in the program's source – including all method calls per open embedding[1] – can be used as the modifies list. It is finite because the program is, and it satisfies our needs as nothing else can change. Because of the absence of difficulties like aliasing or reference navigation, we could even restrict this set to variables really occurring on the left hand side of an assignment. However, programming languages with pointers or compound types like record types or arrays etc. complicate things.

Example 4.1.1 The following code fragment performs a list traversion and modifies more than just the occurring variables `root`, `n`, `n.x`, `n.next` but also `root.next.x`, `root.next.next.x`, etc. Here we use `empty` as an end of list marker.

```
Node n = root;
while (n != empty) {
    n.x = n.x + 1;
    n = n.next;
}
```

□

Example 4.1.2 To illustrate peculiarities of aliasing, we examine a second example. The following program modifies more than just the object `o` and `o.x` occurring in the source code, but also `y.x`. Nevertheless, mentioning both, `o` and `y`, often may not be convenient, especially in the presence of the inferred knowledge that $\mathtt{o} \doteq \mathtt{y}$ holds in the poststate.

```
o = y;
o.x = 7;
```

□

With the above difficulties in mind, we start a systematic treatment of the modifies list in the presence of (most) features of modern object-oriented programming languages like JAVA. From the example 4.1.1 it is immediate that the set of memory locations modified by a program is generally unbounded. Even though finite programs will only alter a finite amount of space in finite time, the termination time and thus the list of modified memory locations is

[1] recursion is no problem for mere variable occurrence analysis

generally unknown *a priori*, and could well exceed every finite limit. Specifications targeting at a description of *every* possible program run under all inputs therefore have to deal with an unbounded list of modified memory locations. But finite formulas only can attempt to achieve this with finite means. So we have to make sure there still is a finite description of the unbounded changes. And there must be one because the program is a finite description, including a (hidden complex) description of the state elements.

Since every program has an equivalent in the WHILE programming language, we could – in principle – even compute a surrogate of the finite descriptions from this translation.[2] But apart from the fact that the inverse translation back to, say, JAVA has to be achieved as well, we would certainly lose the programmer's intuition of what happens in between and what the result says about the original program. Furthermore, unlike pure WHILE programs, programs implemented in JAVA may produce an unbounded output by following object reference cascades in dynamic data structures. These unbounded data structures cannot even be described with (finite) arrays without pointers, but rather require computable encoding into arbitrary sized natural numbers. Thus those reduction attempts would deviate unnecessarily far from the actual program and its terminology. Therefore we prefer to take a more direct approach. Then we will not have to leave the programming language in favour of another representation in a totally different programming model, just because describing modifies lists finitely would be easier.

Basic dynamic logic works with purely functional signatures, on the logic layer. Thus we decide to embed features of object-oriented programming languages into a functional signature for simplicity of concepts. An extension of dynamic logic avoiding this reformulation is straightforward but promises no further insight on the essence of the subject. The programming language itself (i.e. the layer of programs in modal operators) undergoes no change, of course, but remains object-oriented as expected. The relation between advanced concepts of the programming language and the logical signature is as follows: Classes give rise to types, program variables occur as 0-ary (constant) function symbols, object attributes `o.v` occur as unary function symbols $\mathtt{v}(o)$, and n-dimensional arrays occur as n-ary function symbols $\mathtt{a}(i_1, \ldots, i_n)$ for $\mathtt{a}[i_1, \ldots, i_n]$. See [Beckert and Schmitt, 2003] for more details on the program

[2] We can even deduce that – depending on the number of the dedicated input and output variables – there will be a constant bound for the size of the modifies list regardless of the specific program at hand and its (original) local variables. This is true because every program has a translation for the Turing machine whose mechanism can in turn be simulated with a finite and constant number of variables in the WHILE programming language (with a single proper loop). But the modifies list will still describe a possibly unbounded change by the program.

Table 4.1: Equality translation and quantification order for different types of program variables.

Form	Type	Quantification	Equality
`x`	primitive type	$\forall y$	$\mathtt{x} \doteq y$
`a`	array	$\forall Y : function$	$\forall i : \mathbf{N}\ \mathtt{a}(i) \doteq Y(i)$ [3]
`o.v`	object attribute	$\forall Y : function$	$\forall o : object\ \mathtt{v}(o) \doteq Y(o)$
`o`	reference	$\forall y$	$\mathtt{o} \doteq y$

logic.

Now, when we boldly include a function symbol f resulting from an object attribute or array into the modifies list $MV(\alpha)$ just like ordinary local program variables, there is a semantical problem. What would the formula $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}}$ resulting from the modifies list mean then? It would contain a term $f \doteq f^{\mathrm{pre}}$ involving a (still undefined) equality on functions. What we want to achieve in such a situation, is that we have a term remembering the previous state which – in the case of an array or object attribute – consists of the values at each single index. Therefore we define equality pointwise.

Definition 4.1.1 *For function symbols f and g of the same arity and type, we define equality on functions as*

$$f \doteq g \quad := \quad \forall i\, f(i) \doteq g(i)$$

and similarly for higher arities.

Table 4.1 gives an overview of the resulting equality translations for all types of program variables. Notice that now the quantifier[4] prefix of the form $\forall \mathbf{x}^{\mathrm{pre}}$ occurring in the defining condition 2.1 of specifications can still contain higher-order quantifiers[5], if $MV(\alpha)$ contains a function symbol as resulting from an array or object attribute. We will address this problem in the next section.

With this translation of equality on functions we have a finite description of the (maximum) set of state elements modified. We could simply gather each

[3] Especially arrays allow another natural approach using memory(a, i) instead of $a(i)$ as a translation. Then Java array reference copy semantics can be modelled directly. But since for specification computation concerns this choice is of minor impact, we do not pursuit this line of thought.

[4] Remember that $\forall \mathbf{x}^{\mathrm{pre}}$ is an abbreviation for $\forall {x_1}^{\mathrm{pre}} \ldots {x_n}^{\mathrm{pre}}$

[5] In our notation, higher-order quantifiers are hinted at with capital letter variables. In Sect. 4.2 we will see how these particular quantifiers can be reduced to first-order quantifiers.

object attribute, array, or general program variable[6] referenced in the whole program, into the modifies list $MV(\alpha)$. Then a formula like $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ resulting from $MV(\alpha)$ gives rise to a finite description of the possible modifications. More precisely, $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ then remembers the full modifiable state prior to the execution of α. So we also have a finite description of a very conservative estimation of the memory locations modifiable by α. The specification has the additional task of describing in a finite way not only what, but also how precisely, it changes. That this is possible is evident from Res. 3.2.1 when complete reference to the previous state (expressed in $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$) is available.

Example 4.1.3 Continuing example 4.1.1 we get

$$\begin{aligned}
MV(\alpha) &= \{\mathtt{x}\}\big(\cup\{\mathtt{root},\mathtt{next}\}\big)\\
\mathbf{x} \doteq \mathbf{x}^{\text{pre}} &:= \mathtt{root} \doteq root^{\text{pre}} \wedge \mathtt{x} \doteq x^{\text{pre}} \wedge \mathtt{next} \doteq next^{\text{pre}}\\
&= \mathtt{root} \doteq root^{\text{pre}} \wedge \forall o\, \mathtt{x}(o) \doteq x^{\text{pre}}(o) \wedge \forall o\, \mathtt{next}(o) \doteq next^{\text{pre}}(o)
\end{aligned}$$

So the specification condition will be

$$\begin{aligned}
\vDash\ & \forall root^{\text{pre}}\ x^{\text{pre}}\ next^{\text{pre}}\\
& \big(\mathtt{root} \doteq root^{\text{pre}} \wedge \forall o\, \mathtt{x}(o) \doteq x^{\text{pre}}(o) \wedge \forall o\, \mathtt{next}(o) \doteq next^{\text{pre}}(o) \supset [\alpha]\psi\big)
\end{aligned}$$

and the strongest specification is

$$\begin{aligned}
\psi &:= \forall o\,\big(\text{reachable}(\mathtt{root}, o) \supset \mathtt{x}(o) \doteq x^{\text{pre}}(o) + 1\big) \wedge \text{nonfin}\\
\text{reachable}(x,y) &:= \langle \mathtt{while(x} \neq \mathtt{y)x} = \mathtt{x.next}\rangle \top\\
\text{nonfin} &:= \text{reachable}(\mathtt{root}, \mathtt{empty})
\end{aligned}$$

The condition nonfin especially guarantees that the specification is false for infinite lists or cyclic lists, in which case the list traversion does not terminate such that the strongest specification should then be $\bot$. □

4.2 Lowering Higher-Order Logic

In the last section we have seen that there is a finite description of the (maximally) modified state elements and corresponding terms for remembering the previous state, in second-order logic. Now we examine methods for getting rid of those second-order parts and thus show that they do not impose further limitations on decidability, completeness, etc. We will see that our

[6] of which there is only a finite number

second-order formula ingredients are simple enough to possess an equivalent first-order variant without loss of generality.

For this purpose we first note that we only introduce higher-order quantifiers in the universal quantifier prefix of the formula on top-level. By permutation of these universal quantifiers the higher-order quantifiers can be moved to the top-level in front of the formula[7]. Then, by definition of entailment, we can dispose of the higher-order quantifiers by constantification resp. Skolemisation:

$$\models \forall F\!:\!function\ \phi \quad \Longleftrightarrow \quad \models \phi[F \mapsto f]$$

where f denotes a new constant function symbol for the higher-order function variable F. So even though $\forall F : function\ \phi$ and $\phi[F \mapsto f]$ are not locally equivalent, their property of being a logical truth is.[8] And that one is a logical truth if and only if the other is, justifies the transformation for the central aspects Res. 2.2.1 and Res. 3.2.1, which are based on logical truth.

4.3 Practical Treatment

For practical applications several approaches for dealing with the modifies list are conceivable. These are

- to make an explicit separate pre-processing step to compute a conservative estimation of the modifies list.
- to use every existing program variable, object attribute, etc.
- to start a special proof attempt with the empty modifies list[9], and to restart the process with a modifies list accumulated from the left hand sides of the affected state change assignments occurring in the resulting specification of the first run.
- to prefer a more implicit treatment of the modifies list by using a built-in $\cdot^{\mathrm{pre}}$-operator whenever necessary.

[7]Then the formula is in Π^1_1 form.

[8]In other words, the transition from $\forall F\!:\!function\ \phi$ to $\phi[F \mapsto f]$ is validity-preserving.

[9]More precisely: to introduce a constant symbol $\mathcal{M}$ of type formula, which a proof system can handle as a purely formal modifies list. Later on $\mathcal{M}$ can be instantiated to something more specific than the trivial modifies list, which only says everything could have changed. Although we will not investigate this possibility directly, we will showin Sect. 5.2 how a similar treatment for a constant symbol $\mathcal{C}$ of type formula can be achieved.

Perhaps the last variant is the most promising in the long run. More practical aspects of the modifies list following this last variant will be covered in Sect. 5.5, when the basic specification construction process has been introduced.

Chapter 5

Sequent Calculus Approach

In this chapter, we present the practical approach of automatic specification construction by theorem proving, and prove that it produces maximal specifications. Further, we introduce the concept of state change accumulators for built-in dealing with unknown modifies lists.

5.1 Overview

So far we have presented a problem formalisation for strongest specification construction and a constructive existence proof. Thereby we have proven that an effective construction procedure for strongest specifications exists. Now we want to examine how a more algorithmic realisation of this procedure could look like, and to what extent readability concerns can be taken into account.

Having achieved a formalisation of the specification construction problem in dynamic logic, it is an apparent idea to employ means of "automatic logic", namely theorem proving using a sequent calculus, for an automatic construction process. This follows a general principle: when realising a description of any problem in logic, using solution methods of the computational branch of logic as well should at least be worth a closer look. In fact, in our case it will turn out that the results of this are pretty good, and that the corresponding process is very flexible.

In this section, we explain how the specification of a program can be computed by means of a theorem prover on the basis of sequent calculus[1]. We achieve this computation by starting a proof from a specially prepared proof obligation and perform as many proof steps as possible until we reach a dead end. Looking at a proof situation that has no more applicable inference

[1]For the moment, imagine the KeY sequent calculus for dynamic logic for Java of [Ahrendt et al., 2004].

rules from the right perspective will reveal the program's specification. In our implementation we actually employ such failed proof attempts directly. However, more formally, we could just as well include special closing rules or axioms whose applicability unveils the program specification.

Example 5.1.1 For a motivation of the theorem proving approach to specification construction, consider the program α.

```
if (x > 0) {
    x = x + 1;
}
x = x + 2;
```

Assuming that some specification ψ of α appears from out of nowhere. Then we can establish the following conjecture that ψ is a formal specification of α.

$$\vDash \ \forall x^{\text{pre}}\big(\mathtt{x} \doteq x^{\text{pre}} \ \supset \ [\alpha]\psi\big)$$

Further suppose that for some reason we prefer to claim the following, instead.

$$\vDash \ \forall x^{\text{pre}}\big(\mathtt{x} \doteq x^{\text{pre}} \ \supset \ \langle\alpha\rangle\psi\big) \tag{5.1}$$

Then regardless of the fact that we have not yet revealed which particular formula ψ actually is, we can start a proof of condition 5.1, nevertheless. It continues as follows.

$$\dfrac{\dfrac{\dfrac{\mathtt{x} \doteq x^{\text{pre}}, \mathtt{x} > 0 \ \vdash \langle \mathtt{x} = x^{\text{pre}} + 3\rangle\psi}{\mathtt{x} \doteq x^{\text{pre}}, \mathtt{x} > 0 \ \vdash \langle \mathtt{x} = x^{\text{pre}} + 1 + 2\rangle\psi}}{\mathtt{x} \doteq x^{\text{pre}}, \mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 1\rangle\langle \mathtt{x} = \mathtt{x} + 2\rangle\psi} \qquad \dfrac{\mathtt{x} \doteq x^{\text{pre}}, \mathtt{x} \leq 0 \ \vdash \langle \mathtt{x} = x^{\text{pre}} + 2\rangle\psi}{\mathtt{x} \doteq x^{\text{pre}}, \neg\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 2\rangle\psi}}{\mathtt{x} \doteq x^{\text{pre}} \ \vdash \langle\alpha\rangle\psi}$$

In this situation, the proof attempt cannot continue any further without knowledge about the real structure of ψ. As anticipated, our attempt to proof that ψ is a specification (in the sense of condition 5.1) has failed. Nonetheless, we have come to know several properties of the program α during the proof. Now the idea is that a thorough inspection of the above proof attempt could reveal in retrospect, which formula ψ we would have to pick for completing the proof. From the individual branches, we can see the effect of α, and combine the logical descriptions of those effects into a single formula.

$$\begin{aligned} \psi' \quad := \quad & (x^{\text{pre}} > 0 \supset \langle \mathtt{x} = x^{\text{pre}} + 3\rangle\psi) \\ & \wedge(x^{\text{pre}} \leq 0 \supset \langle \mathtt{x} = x^{\text{pre}} + 2\rangle\psi) \end{aligned}$$

In fact, the above proof attempt closes when choosing ψ along these lines, as follows.

$$\begin{array}{rcl}\psi & := & (x^{\text{pre}} > 0 \supset \mathtt{x} \doteq x^{\text{pre}} + 3) \\ & & \wedge(x^{\text{pre}} \leq 0 \supset \mathtt{x} \doteq x^{\text{pre}} + 2)\end{array}$$

□

As a brief sketch of the theorem proving approach to specification construction, we summarise what has been unveiled in example 5.1.1 in a more systematic way. The basic idea is to start a proof attempt like that of condition 5.1 with an unknown formula ψ, perform some (limited) inferences, and then extract a specification from the open goals of the proof. Hopefully, the extracted specification allows to close the proof in retrospect, and thereby turns out to be a specification of the original program.

5.2 Constructing Special Proof Obligations

Part of the secret of our approach lies in the proper preparation of the proof obligation for computing specifications. In fact, there are several possibilities, depending on a trade-off choice between excessive pre-processing and non-invasive rule modifications[2]. If a (reasonably small) upper bound for the set of variables modified by the program is known in advance or can be computed in a pre-processing step, then the computation is straightforward and close to the theory.

Assuming that we know that the program α modifies at most the variables $MV(\alpha) = \{x_1, \ldots, x_n\}$ – for simplicity this set is assumed to contain all input variables already – then we start proving the following proof obligation.[3]

$$\models \ \forall \mathbf{x}^{\text{pre}}\big(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \ \supset \ \langle\alpha\rangle\mathcal{C}\big) \tag{5.2}$$

In preparation of later generalisations this conjecture already includes the formula $\mathcal{C}$ which – for now – we specialise as

$$\mathcal{C} \ := \ \mathbf{x} \doteq \mathbf{x}^{\text{post}} \ := \ \mathtt{x_1} \doteq {x_1}^{\text{post}} \wedge \ldots \wedge \mathtt{x_n} \doteq {x_n}^{\text{post}} \tag{5.3}$$

Contrary to the definition of specifications we prefer $\langle\alpha\rangle$ instead of $[\alpha]$ here, which has its deeper reason in Res. 3.1.6 and will become clearer when we

[2]As we will see later, calculi used for specification computation have to fulfil some conditions. Thus smooth rule adaptation and a high degree of compatibility between existing rules and any new rules introduced for specification construction is of advantage.

[3]Remember that the formula $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ with generic names abbreviates $\mathtt{x_1} \doteq {x_1}^{\text{pre}} \wedge \ldots \wedge \mathtt{x_n} \doteq {x_n}^{\text{pre}}$.

investigate the overall statement in Sect. 5.5. For now we will confine ourselves to noting that we intend to apply transformations of an effect related to the antecedent placement of the modal operator in condition 2.7.

5.3 State Change Accumulation

An alternative and perhaps more applicable approach does not depend on previous knowledge of the modifies list. The clue is to build support for accumulating intermediate state changes into the proof system. This is quite straightforward in principle, but gets obfuscated by the technical details. These complications have to do with inference rules that discard all those state changes that do not affect the formulas of the current goal. Of course, such rules are necessary for proving statements that ignore some information about a piece of code, but here they interfere with our intent not to overlook *any* effects of the code. Nevertheless, we still need them for closing many subgoals during the proof, so switching them off completely would not solve the problem, either.

Example 5.3.1 In the KeY System, the rules that have to be blocked selectively for specification computation with state change accumulation are called update simplifications. Since introducing updates is not necessary for understanding the particular consequences for specification construction, we retain ordinary assignment modalities of quantified dynamic logic. In the usual notation without updates, a reduction like the following would be legal but abolish information.

$$\langle \mathtt{x} = \mathtt{t} \rangle y > 1 \;\mapsto\; y > 1 \qquad \text{(provided that x does not occur free in y)}$$

Intuitively, what we have to make sure is that these simplifications do not happen to every part of the formulas in the proof attempt. Otherwise a state change by the program would have gone by unnoticed . If – for now – you imagine the above formula $y > 1$ to be a representative case for the formula $\mathcal{C}$ of the proof obligation condition 5.2, and the assignment $\mathtt{x} = \mathtt{t}$ to occur within the program, then the above reduction would annihilate the statement completely without recording its effect. So whatever "specification" we would be able to reconstruct at the end of the proof attempt does not depend on the statement $\mathtt{x} = \mathtt{t}$. And this independence is, of course, untrue in the general case, since most programs have a different effect when removing one particular statement in between. Instead, we should make sure that not a single effect of the program slips our attention. □

Thus, in order to guarantee coming up with the strongest specification, we have to remember every kind of information about the program somewhere. In essence, due to the imperative nature of our programming language, this requires that state change information accumulates instead of disappears from the formulas. And for intermediate state change accumulation, we introduce a constant symbol of type formula called *state (change) accumulator* $\mathcal{C}$ into the postcondition. It is a surrogate for an unknown formula and serves the purpose of blocking all simplifications or reductions that would require knowledge about variables not occurring in the formula. An inference rule that simplifies away (or forgets) a state change – just because its affected variable has not yet been mentioned within the formula and thus changes to it seem irrelevant – is no longer possible for $\mathcal{C}$. This is due to the fact that $\mathcal{C}$'s symbolic nature entails that it could have been replaced by any formula, especially one referring to the particular variable that is subject to the state change and would thus notice the change. So in a way, $\mathcal{C}$ has an eidetic (cumulative) memory for state changes. [4]

5.4 Specification Extraction

From a proof attempt of a proof obligation for a specification computation (condition 5.2) that allows no more applicable inference rules, we can excerpt some information to construct the computed specification of the underlying program. When a sequent proof runs out of applicable inference rules – apart perhaps from non-helpful "cyclic" ones, like commuting formulas –, it ends up with a set of sequents forming the still open goals and the closed goals. The closed goals can safely be ignored because they only contain logical truths and thus no longer contribute to the specification of the particular program at hand. The open goals, however, represent valuable information. From a positive view, they constitute formulas that have to be true in order to validate the tautological nature of the original conjecture postulated as the proof obligation. So for the proof obligation – which, in our case, is a statement about what is true of the state change performed by the program – to be valid it is necessary that all remaining formulas of the open goals are valid. Turned upside down, with some care this reads as: whatever is true about the state change is equivalent to the conjunction of the open goals. So

[4]From a systematic point of view, using the constant symbol $\mathcal{C}$ of type formula is a variation of the conceptual ideas behind the automatic construction of characterising formulas for correspondence theory by [Gabbay and Ohlbach, 1992]. Here, $\mathcal{C}$ plays a role comparable to the schematic propositional atom p in the characterising formula $\Box p \supset p$ of the system T of modal logic (refer to [van Benthem, 2001] for more information on correspondence theory).

let us extract the specification $\psi_{\mathcal{C}}$ as a conjunct of the open goals. And there it appears, the specification of the piece of program, constructed by at least semi-automatic means of sequent calculus theorem proving. What still has to be taken care of are the precise conditions under which the proof procedure does not disturb the intermediate equivalences. We examine them next.

Definition 5.4.1 *(Equivalence Transformation) An inference rule*

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \dots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

is called equivalent transformation *if its premise and its conclusion are locally equivalent, i.e. if for each interpretation I for each state s*

$$\begin{array}{l} I, s \models (\Gamma \vdash \Delta) \\ \Longleftrightarrow \\ I, s \models (\Gamma_1 \vdash \Delta_1) \wedge \dots \wedge (\Gamma_n \vdash \Delta_n) \end{array}$$

Note that this definition also applies for axioms without premises ($n = 0$) where it collapses to that of tautologies.[5] *The sequents have their usual translation*

$$(\gamma_1, \dots, \gamma_m \vdash \delta_1, \dots, \delta_l) := (\bigwedge_{i=1}^{m} \gamma_i) \supset (\bigvee_{j=1}^{l} \delta_j)$$

Remark 5.4.2 *An inference rule of the above form is an equivalent transformation $\Longleftrightarrow$*

$$\models (\Gamma \vdash \Delta) \leftrightarrow (\Gamma_1 \vdash \Delta_1) \wedge \dots \wedge (\Gamma_n \vdash \Delta_n)$$

Remark 5.4.3 *Local equivalence is a congruence relation.*

Actually this choice for the notion of equivalence transformations is not necessarily the only possibility. It is quite convincing that the condition is sufficient, but not at all apparent whether it has been chosen too strong. This criterion for inference rules could require re-weakening if too many practical programs cannot be specified automatically with this approach. On the other hand, our theory (especially Res. 3.2.1 in conjunction with Res. 3.2.5) suggests that local equivalence is what can be obtained in the large. Therefore demanding local equivalence for each intermediate step suggests itself instantly. Of course, we have to keep in mind that what holds for the large

[5] Also note that the notion is not inherent to sequent calculus inference rules of the above for, but can be extended directly to other calculi.

does not always apply to the small as well. However intuitive such a conclusion may appeal, local equivalence in the large does not necessarily require exclusively locally equivalent transformations at every single argument in between.

Anyway, the intuition behind the local equivalence approach as opposed to a global equivalence approach is natural. We do not only want to say something about those structures in which the formulas are valid in every state, but also about every single state which validates the formula in an arbitrary structure. Even though specifications are usually intended to hold for every state, we also want the transformed formulas to retain each property that is only true for some but not all initial states.

The following results justify simplified reasoning about equivalent transformations and the investigation of individual inference rules.

Claim 5.4.4 ("Context-free equivalent transformations")

$$\frac{\Gamma, \phi_1 \ \vdash \psi_1, \Delta \quad ... \quad \Gamma, \phi_n \ \vdash \psi_n, \Delta}{\Gamma, \phi \ \vdash \psi, \Delta}$$

is an equivalence transformation if and only if the following is

$$\frac{\phi_1 \ \vdash \psi_1 \quad ... \quad \phi_n \ \vdash \psi_n}{\phi \ \vdash \psi}$$

Proof: Since $\supset$ left-distributes over $\supset, \land, \lor, \leftrightarrow$ so does $\vdash$, and since $\Gamma, \phi \ \vdash \psi$ is equivalent to $\Gamma \ \vdash \phi \supset \psi$, the following equivalences hold.

$$\begin{array}{rl} & (\Gamma, \phi \ \vdash \psi) \leftrightarrow \big((\Gamma, \phi_1 \ \vdash \psi_1) \land \cdots \land (\Gamma, \phi_n \ \vdash \psi_n)\big) \\ \iff & (\Gamma \ \vdash \phi \supset \psi) \leftrightarrow \big((\Gamma \ \vdash \phi_1 \supset \psi_1) \land \cdots \land (\Gamma \ \vdash \phi_n \supset \psi_n)\big) \\ \iff & \Gamma \ \vdash \Big((\phi \supset \psi) \leftrightarrow \big((\phi_1 \supset \psi_1) \land \cdots \land (\phi_n \supset \psi_n)\big)\Big) \end{array}$$

and thus (use $\Gamma, \neg\Delta$ for Γ) also

$$\begin{array}{rl} & (\Gamma, \phi \ \vdash \psi, \Delta) \leftrightarrow \big((\Gamma, \phi_1 \ \vdash \psi_1, \Delta) \land \cdots \land (\Gamma, \phi_n \ \vdash \psi_n, \Delta)\big) \\ \iff & \Gamma \ \vdash \Big((\phi \supset \psi) \leftrightarrow \big((\phi_1 \supset \psi_1) \land \cdots \land (\phi_n \supset \psi_n)\big)\Big), \Delta \end{array}$$

If the latter is a tautology for every interpretation and state, independent of the context Γ, this is especially true for $\Gamma = \top$ and $\Gamma = \bot$, and conversely so. Likewise arguments for Δ now conclude the proof. ∎

Remark 5.4.5 ("omnipresent equivalent transformations") *An inference rule*

$$\frac{....\psi....}{....\phi....}$$

is an equivalence transformation if and only if its premise and conclusion are locally equivalent.

Proof: This is because local equivalence is a congruence relation ■

Example 5.4.1 Examples ("+") and counter examples ("−") for equivalent transformations. See Res. 5.4.6 for those proofs that involve more than just a few obvious arguments:

"+" and-right

$$\frac{\Gamma \ \vdash \phi, \Delta \quad \Gamma \ \vdash \psi, \Delta}{\Gamma \ \vdash \phi \wedge \psi, \Delta}$$

"−" weakening

$$\frac{\Gamma \ \vdash \Delta}{\Gamma, \phi \ \vdash \Delta} \qquad \frac{\Gamma \ \vdash \Delta}{\Gamma \ \vdash \Delta, \phi}$$

That this rule is precluded because it is no equivalence transformation is what we wanted since its application would omit information, rendering the specification useless, as we will see in example 6.3.1.

"−" Skolemisation

$$\frac{\Gamma, \phi(X_1, \ldots, X_n, s(X_1, \ldots, X_n)) \ \vdash \Delta}{\Gamma, \exists y\, \phi(X_1, \ldots, X_n, y) \ \vdash \Delta}$$

where $\{X_1, \ldots, X_n\}$ are the free variables occurring in $\exists y\, \phi(X_1, \ldots, X_n, y)$ and s is a new Skolem-function constant.

"+" Hilbert ϵ-rule, sometimes called "critical axiom"

$$\frac{\Gamma, \phi[y \mapsto \epsilon y\, \phi] \ \vdash \Delta}{\Gamma, \exists y\, \phi \ \vdash \Delta}$$

This rules is a locally equivalent replacement for Skolemisation.[6]

[6] During the time of writing there has been a change from the ϵ-rule to Skolemisation throughout the KeY system for other reasons.

"+" universal quantifier

$$\frac{\Gamma, \phi[x \mapsto t], \forall x\, \phi \;\vdash \Delta}{\Gamma, \forall x\, \phi \;\vdash \Delta}$$

"+" branch

$$\frac{\Gamma, b \;\vdash \langle \mathtt{s} \rangle \phi, \Delta \quad \Gamma, \neg b \;\vdash \langle \mathtt{t} \rangle \phi, \Delta}{\Gamma \;\vdash \langle \mathtt{if(b)\ s\ else\ t} \rangle \phi, \Delta}$$

"−" "weakening" single-side branch

$$\frac{\Gamma \;\vdash b, \Delta \quad \Gamma, b \;\vdash \langle \mathtt{s} \rangle \phi, \Delta}{\Gamma \;\vdash \langle \mathtt{if(b)\ s\ else\ t} \rangle \phi, \Delta}$$

"−" cut with weakening

$$\frac{\Gamma \;\vdash \phi \quad \Gamma, \phi \;\vdash \Delta}{\Gamma \;\vdash \Delta}$$

"+" cut

$$\frac{\Gamma \;\vdash \phi, \Delta \quad \Gamma, \phi \;\vdash \Delta}{\Gamma \;\vdash \Delta}$$

This case distinction rule may not always be a good choice for achieving most readable specifications, though. When we end up in subsumption cases or identical consequence cases, cut should perhaps not have been applied. But still its application does not disturb local equivalence.

"+" cut-derived

$$\frac{\Gamma \;\vdash \phi, \Delta \quad \Gamma \;\vdash \neg\phi, \Delta}{\Gamma \;\vdash \Delta}$$

"+" unwind loop once

$$\frac{\Gamma \;\vdash \langle \mathtt{if(b)}\{\mathtt{a; while(b)a}\} \rangle \phi, \Delta}{\Gamma \;\vdash \langle \mathtt{while(b)a} \rangle \phi, \Delta}$$

"−" loop induction with arbitrary invariant I

$$\frac{\Gamma \;\vdash I, \Delta \quad I, b \;\vdash [\mathtt{a}] I \quad I, \neg b \;\vdash \phi}{\Gamma \;\vdash [\mathtt{while(b)a}] \phi, \Delta}$$

"−" assignment rule

$$\frac{\Gamma[\mathtt{x} \mapsto y], \mathtt{x} \doteq t[\mathtt{x} \mapsto y] \;\vdash \phi, \Delta[\mathtt{x} \mapsto y]}{\Gamma \;\vdash \langle \mathtt{x = t} \rangle \phi, \Delta}$$

where y is a new variable. This rule is only globally equivalent, not locally equivalent.

“+” assignment rule with substitutions[7]

$$\frac{\Gamma \ \vdash \phi[\mathtt{x} \mapsto t], \Delta}{\Gamma \ \vdash \langle \mathtt{x} = \mathtt{t} \rangle \phi, \Delta}$$

“+” Further examples of equivalence transformations are the usual axioms and inference rules of propositional logic, equations, or reordering .

□

Remark 5.4.6 (Proofs of local equivalence results) *1. The assignment rule with new variable y is only globally equivalent, not locally equivalent.*

$$\frac{\Gamma[\mathtt{x} \mapsto y], \mathtt{x} \doteq t[\mathtt{x} \mapsto y] \ \vdash \phi, \Delta[\mathtt{x} \mapsto y]}{\Gamma \ \vdash \langle \mathtt{x} = \mathtt{t} \rangle \phi, \Delta}$$

2. The assignment rule with substitutions is locally equivalent.

$$\frac{\Gamma \ \vdash \phi[\mathtt{x} \mapsto t], \Delta}{\Gamma \ \vdash \langle \mathtt{x} = \mathtt{t} \rangle \phi, \Delta}$$

Proof:

1. First we prove that local consequence does not hold, and thus local equivalence holds less than ever.

   “→” $\Gamma := \top, \Delta := \bot, t := \mathtt{x} - 1, \phi := \mathtt{x} \doteq 0$ then in the state $s := [\mathtt{x} \mapsto 1]$ we have that

   $$s \vDash \top \wedge \langle \mathtt{x} = \mathtt{x} - \mathtt{1} \rangle \mathtt{x} \doteq 0$$

   but

   $$s \nvDash \forall y \, (\top \wedge \mathtt{x} \doteq y - 1 \supset \mathtt{x} \doteq 0)$$

   because

   $$s[y \mapsto 2] \vDash \top \wedge \mathtt{x} \doteq y - 1$$
   $$s[y \mapsto 2] \nvDash \mathtt{x} \doteq 0$$

   Then we prove that global equivalence holds nevertheless.

[7] In fact, this rule is not correct if ϕ still contains further modalities. If restricting the applicability to innermost modalities does not suffice, then we need a concept like KeY updates which are “substitutions that stick to modal operators”. Even though those nested modality cases are thus more complex, the principles are still the same.

"$\rightarrow$" We have to show that

$$s \models \forall y\,(\Gamma[\mathtt{x} \mapsto y] \wedge \mathtt{x} \doteq t[\mathtt{x} \mapsto y] \supset \phi)$$

Assuming that $s \models \Gamma[\mathtt{x} \mapsto y] \wedge \mathtt{x} \doteq t[\mathtt{x} \mapsto y]$ we have $s[\mathtt{x} \mapsto \mathrm{val}(s, y)] \models \Gamma$ by Res. A.1.1. By our global premise we have

$$s[\mathtt{x} \mapsto \mathrm{val}(s, y)] \models \Gamma \supset \langle \mathtt{x} = \mathtt{t}\rangle\phi$$

$\Rightarrow\ s[\mathtt{x} \mapsto \mathrm{val}(s, y)] \models \langle \mathtt{x} = \mathtt{t}\rangle\phi\ \Rightarrow$

$$\underbrace{s[\mathtt{x} \mapsto \mathrm{val}(s, y)][\mathtt{x} \mapsto s[\mathtt{x} \mapsto \mathrm{val}(s, y)](t)]}_{s[\mathtt{x}\mapsto \underbrace{s[\mathtt{x} \mapsto \mathrm{val}(s,y)](t)}_{\underbrace{\mathrm{val}(s, t[\mathtt{x} \mapsto y])}_{\mathrm{val}(s,\mathtt{x})}}]} \models \phi$$

thus $s \models \phi$.

"$\leftarrow$" Assuming that $s \models \Gamma$ we have to show that $s \models \langle \mathtt{x} = \mathtt{t}\rangle\phi$, i.e. that $s[\mathtt{x} \mapsto \mathrm{val}(s, t)] \models \phi$. According to our global premise we have

$$s[\mathtt{x} \mapsto \mathrm{val}(s, t)] \models \forall y\,(\Gamma[\mathtt{x} \mapsto y] \wedge \mathtt{x} \doteq t[\mathtt{x} \mapsto y] \supset \phi)$$

For $s' := s[\mathtt{x} \mapsto \mathrm{val}(s, t)][y \mapsto \mathrm{val}(s, \mathtt{x})]$ we have $s' \models \mathtt{x} \doteq t[\mathtt{x} \mapsto y]$ by construction. $s' \models \Gamma[\mathtt{x} \mapsto y]$ because of $s \models \Gamma$ and $\mathrm{val}(s', y) = \mathrm{val}(s, x)$ whilst $\mathtt{x} \notin FV(\Gamma[\mathtt{x} \mapsto y])$. Then by the premise $s' \models \phi \overset{y \notin FV(\phi)}{\Rightarrow} s[\mathtt{x} \mapsto \mathrm{val}(s, t)] \models \phi$

2. We prove that $\langle \mathtt{x} = \mathtt{t}\rangle\phi \equiv \phi[\mathtt{x} \mapsto t]$.

$$\begin{array}{rrcl} & s & \models & \langle \mathtt{x} = \mathtt{t}\rangle\phi \\ \Longleftrightarrow & s[\mathtt{x} \mapsto \mathrm{val}(s, t)] & \models & \phi \\ \underset{\text{Res. A.1.1}}{\Longleftrightarrow} & s & \models & \phi[\mathtt{x} \mapsto t] \end{array}$$

Thereby note that assignments are deterministic: there is a unique s' with $s\rho(\mathtt{x} = t)s'$, namely $s' := s[x \mapsto \mathrm{val}(s, t)]$. Especially, $\langle \mathtt{x} = \mathtt{t}\rangle\phi \equiv [\mathtt{x} = \mathtt{t}]\phi$.

∎

5.5 Quintessence

In this section we reconcile the individual steps, see what we have achieved altogether so far, and express what the overall statement of our approach is. With the detailed knowledge collected in the previous sections, we will conceive how the individual pieces of the puzzle finally coalesce to a sensible whole.

We started by proving a specification computation proof obligation from condition 5.2 that has an arbitrary state change accumulator $\mathcal{C}$. Meanwhile we have been performing only equivalent transformations during the proof, and then have extracted the specification $\psi_\mathcal{C}$ as a conjunct of the open goals. Thereby we have proven the following central higher-order statement.

Lemma 5.5.1 *Let $\psi_\mathcal{C}$ be the conjunct of the (open) goals of a proof attempt of condition 5.2 that only involves equivalent transformations. Then*

$$\text{for each formula } \mathcal{C} \quad \text{for each interpretation } I \quad \text{for each state } s$$
$$I, s \quad \vDash \quad \forall \mathbf{x}^{\mathrm{pre}} \big(\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset (\langle\alpha\rangle\mathcal{C} \leftrightarrow \psi_\mathcal{C})\big) \tag{5.4}$$

Proof: Since we only applied equivalent transformations, premise and conclusion of every rule application are locally equivalent. By iteration, the initial proof obligation and $\psi_\mathcal{C}$ are locally equivalent as well. Therefore (when leaving quantifiers implicit for simplicity)

$$\begin{aligned} & \vDash \quad (\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \langle\alpha\rangle\mathcal{C}) \leftrightarrow \psi_\mathcal{C} \\ \Rightarrow \quad & \vDash \quad \mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \big((\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset \langle\alpha\rangle\mathcal{C}) \leftrightarrow \psi_\mathcal{C}\big) \\ \Longleftrightarrow \quad & \vDash \quad \mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \supset (\langle\alpha\rangle\mathcal{C} \leftrightarrow \psi_\mathcal{C}) \end{aligned}$$

■

This result represents the essential statement distilled from what is true about the process. Also it is quite intuitive that it really provides what we need, because condition 5.4 expresses what is equivalent to a program run, or, more precisely, equivalent to holding after a program run.

What still has to be achieved in the following is the particular connection with strongest specifications, and readable presentation alternatives. For this several possibilities exist. First, a natural option is to take $\psi_\mathcal{C}$ directly as an equivalent formulation of the effect of a program run is a natural option, though perhaps not the most simple representation for proving statements about. Second, translating the formulas back to corresponding JAVA statements[8] presents an equivalent program, which achieves a program normal form. Third, to keep the dynamic logic formula structure of $\psi_\mathcal{C}$ while

[8] For example translating conditional implications to ⌜**if** (b) s **else** t⌝, and embedding the remaining modalities directly into the source code.

imposing some naming standards like the use of x^{pre} and x^{post} instead of $\mathtt{x}$, thereby rigidifying the specification. And fourth, to further reduce all remaining single-step modalities to equations (we will investigate this approach in Sect. 6.1).

For user output we prefer the first option, while for our further proofs, the third option comes in handier. The second possibility with program normalisation could be worth a closer look in case of minimising formal methods on the surface of the user interaction, or perhaps for optimising compilers. The fourth option often is an extension of the third and may sometimes lead to more convincing specifications, that can directly be adapted to an OCL model.

In order to see what Res. 5.5.1 has to do with computing specifications and why calling the conjunct $\psi_{\mathcal{C}}$ of open goals a specification really is justified, we consult Res. 3.2.1. Furthermore the extracted specification $\psi_{\mathcal{C}}$ will usually contain some reference to the initial specification accumulator $\mathcal{C}$ from the postcondition. If we now specialise $\mathcal{C}$ to a proposition that makes $\psi_{\mathcal{C}}$ true, the latter will be a specification[9]. If we specialise $\mathcal{C}$ "sufficiently strong" we will have "the" strongest specification of the program. We will make this line of thought more precise in the following central result of our work. The proof is simple because it combines many of the above results.

Proposition 5.5.2 *With* $\mathcal{C} := \mathbf{x} \doteq \mathbf{x}^{\text{post}}$*, (a variant of)* $\psi_{\mathbf{x}\doteq\mathbf{x}^{\text{post}}}$ *is rigid for* α *and equivalent to "the" maximal specification* ψ_α*.*

Proof: According to Res. 5.5.1, $\psi_{\mathcal{C}}$ satisfies the condition 3.1 of Res. 3.2.1. Thus if we could prove that $\psi_{\mathcal{C}}$ is rigid for α, Res. 3.2.5 would imply that $\psi_{\mathcal{C}} \equiv \psi_\alpha$ is "the" maximal specification. Unfortunately, $\psi_{\mathcal{C}}$ is not rigid in general such that the premises of Res. 3.2.5 are not fulfilled. Therefore, our strategy for proving this conjecture has to be refined a bit. We will still start from the formula $\psi_{\mathcal{C}}$, which satisfies condition 3.1, and successively perform small transformations which sustain condition 3.1 while getting closer to rigidity.

So it remains to construct a variant $\psi'_{\mathcal{C}}$ of $\psi_{\mathcal{C}}$ that is rigid for α and still satisfies condition 3.1.

By Res. A.1.3, prepending $\langle \mathtt{x} = x^{\text{pre}} \rangle$ to $\psi_{\mathcal{C}}$ does not disturb condition 3.1, since $\psi_{\mathcal{C}}$ occurs under the predication of $\mathtt{x} \doteq x^{\text{pre}}$.[10] Then replacing $\mathbf{x}$ by $\mathbf{x}'$ on all modal quantification levels[11] does not disturb condition 3.1 according

[9]because it originates from equivalent transformations only.

[10]Here an admissible intermediate step is to replace $\mathtt{x}$ with x^{pre} on modal quantification level of depth 0 because $\mathtt{x}$ and x^{pre} share the same value. Even though this additional step might be preferable from a systematic point of view, it is not necessary and will be omitted here for simplicity.

[11]The replacements also have to take place within the programs of modalities. Further

to Res. 3.2.2. Call $\psi'_{\mathcal{C}}$ the resulting variant of $\psi_{\mathcal{C}}$. The last renaming step rigidifies $\psi'_{\mathcal{C}}$ for α because of Res. A.4.2 and $MV(\alpha) \cap FV(\psi'_{\mathcal{C}}) = \emptyset$ by construction. ■

Example 5.5.1 Let us examine a small example for illustrating the overall specification construction process. Consider again, the program α from our motivating example 5.1.1.

```
if (x > 0) {
    x = x + 1;
}
x = x + 2;
```

When leaving out the initial assignment condition $\mathtt{x} \doteq x^{\mathrm{pre}}$ for readability, the proof attempt for the program α above can continue from condition 5.2 up to the following situation.

$$\dfrac{\dfrac{\dfrac{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 3\rangle \mathcal{C}}{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 1 + 2\rangle \mathcal{C}}}{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 1\rangle\langle \mathtt{x} = \mathtt{x} + 2\rangle \mathcal{C}} \qquad \dfrac{\mathtt{x} \le 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 2\rangle \mathcal{C}}{\neg \mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 2\rangle \mathcal{C}}}{\vdash \langle \alpha \rangle \mathcal{C}}$$

Since no more (helpful) inference rules are applicable, we replace $\mathcal{C}$ by $x^{\mathrm{post}} \doteq \mathtt{x}$. This is possible, because the remaining elementary diamonds (or updates) promise that only x changes, so that $MV(\alpha) = \{\mathtt{x}\}$. Then we extract

$$\begin{aligned} \psi_{\mathcal{C}} \quad := \quad & (\mathtt{x} > 0 \supset \langle \mathtt{x} = \mathtt{x} + 3\rangle \mathcal{C}) \\ & \wedge(\mathtt{x} \le 0 \supset \langle \mathtt{x} = \mathtt{x} + 2\rangle \mathcal{C}) \end{aligned}$$

Which formally reveals the proper specification according to the proof of Res. 5.5.2 after some renaming transformations.

$$\begin{aligned} \psi'_{x^{\mathrm{post}} \doteq \mathtt{x}'} \quad := \quad & (x^{\mathrm{pre}} > 0 \supset \langle \mathtt{x}' = x^{\mathrm{pre}} + 3\rangle x^{\mathrm{post}} \doteq \mathtt{x}') \\ & \wedge(x^{\mathrm{pre}} \le 0 \supset \langle \mathtt{x}' = x^{\mathrm{pre}} + 2\rangle x^{\mathrm{post}} \doteq \mathtt{x}') \end{aligned}$$

For avoiding the new variables $\mathtt{x}'$, this can be simplified again to the following specification formula, as expected.

$$\begin{aligned} \psi_{x^{\mathrm{post}} \doteq \mathtt{x}} \quad := \quad & (x^{\mathrm{pre}} > 0 \supset x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 3) \\ & \wedge(x^{\mathrm{pre}} \le 0 \supset x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 2) \end{aligned}$$

note that this renaming construction is a consequent enhancement of the ideas behind the proof of Res. 3.2.2. Here the same effect of replacing remaining modalities $\langle \mathtt{a} \rangle$ with $\langle \mathtt{a}' \rangle$ etc. will take place.

From this example, we can conclude that even though the variable renaming has been a technical trick in the proof, it is not necessary for an implementation. The translation from $\psi_{\mathcal{C}}$ to $\psi_{x^{\text{post}} \doteq \mathbf{x}}$ can also be achieved without intermediate variable renaming. $\square$

Remark 5.5.3 *The modifies list $MV(\alpha) = \{x_1, \ldots, x_n\}$ for α that is required for the construction of $\mathbf{x} \doteq \mathbf{x}^{\text{post}}$ can be computed in retrospect by accumulating left-hand sides of variable assignments occurring in $\psi_{\mathcal{C}}$.*

Experience shows that the computed specification attains more readability if we achieve previous state reference through other built-in mechanisms that are equivalent to the $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ solution, which is so close to the theory. The reason for this preference is that users usually will not profit much from specifications bloated by additional conditions like "if $\mathtt{x}$ and x^{pre} initially have the same value then ...". It is simpler for a human to remember the relationship between $\mathtt{x}$ and x^{pre} once and for all, in favour of a more concise notation that highlights the essential information without concealing them within a pile of technical details. There are at least two ways to achieve this focus. First, to track the course of the $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ formulas through the proof tree and to omit them from the final depiction whenever appropriate. And second, to use a built-in mechanism equivalent to mentioning $\mathbf{x} \doteq \mathbf{x}^{\text{pre}}$ that does not have the disadvantages of an illegible result. Possible solutions include, for example, to use a built-in dedicated @*pre*-operator, or to start from an initial program variable assignment of rigid constant symbols like after $\langle \mathbf{x} = \mathbf{x}_0 \rangle$.

So far we have only said *that* equivalent formulations of the strongest program specification can be computed, not how simple they are, nor which precise logical form they take. What specifications typically reduce to is a set of conditional single-step transitions, perhaps still embedded inside a loop. Of course, this depends on the particular underlying calculus and is not an inherent property of the computing specifications approach.

5.6 Examples

First we start with an example that already displays some typical behaviour and change accumulation, but that still is manageable.

Example 5.6.1 In the following abbreviated JAVA program α, $\ulcorner 2 \mid \mathtt{x} \urcorner$ is short for the condition that 2 divides $\mathtt{x}$, which could have been written as the longer term $\ulcorner$x % 2 == 0$\urcorner$. But that would have blown up the following formulas rendering them unnecessarily illegible.

```
if (2 | x) {
    x = x + 2;
} else {
    x = x + 3;
}
x = x + 1;
```

Leaving out the initial assignment condition $\mathtt{x} \doteq x^{\text{pre}}$ for readability, the proof attempt for the program α above can continue from condition 5.2 up to the following situation.

$$\dfrac{\dfrac{\dfrac{2 \mid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 3 \rangle \mathcal{C}}{2 \mid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 2 + 1 \rangle \mathcal{C}}}{2 \mid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 2 \rangle \langle \mathtt{x} = \mathtt{x} + 1 \rangle \mathcal{C}} \quad \dfrac{\dfrac{2 \nmid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 4 \rangle \mathcal{C}}{2 \nmid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 3 + 1 \rangle \mathcal{C}}}{2 \nmid \mathtt{x} \ \vdash \langle \mathtt{x} = \mathtt{x} + 3 \rangle \langle \mathtt{x} = \mathtt{x} + 1 \rangle \mathcal{C}}}{\vdash \langle \alpha \rangle \mathcal{C}}$$

Since no more (helpful) inference rules are applicable, we replace $\mathcal{C}$ by $x^{\text{post}} \doteq \mathtt{x}$ here. This is possible because the remaining elementary diamonds (or updates) promise that only x changes, so that $MV(\alpha) = \{\mathtt{x}\}$. Then we extract

$$\begin{aligned} \psi_{\mathcal{C}} \quad := \quad & (2 \mid \mathtt{x} \supset \langle \mathtt{x} = \mathtt{x} + 3 \rangle \mathcal{C}) \\ & \wedge (2 \nmid \mathtt{x} \supset \langle \mathtt{x} = \mathtt{x} + 4 \rangle \mathcal{C}) \end{aligned}$$

which formally reveals the proper specification after some renaming transformations.

$$\begin{aligned} \psi'_{x^{\text{post}} \doteq \mathtt{x}'} \quad := \quad & (2 \mid x^{\text{pre}} \supset \langle \mathtt{x}' = x^{\text{pre}} + 3 \rangle x^{\text{post}} \doteq \mathtt{x}') \\ & \wedge (2 \nmid x^{\text{pre}} \supset \langle \mathtt{x}' = x^{\text{pre}} + 4 \rangle x^{\text{post}} \doteq \mathtt{x}') \end{aligned}$$

For avoiding the new variables x', this can be simplified again to the following formula, as expected.

$$\begin{aligned} \psi_{x^{\text{post}} \doteq \mathtt{x}} \quad := \quad & (2 \mid x^{\text{pre}} \supset x^{\text{post}} \doteq x^{\text{pre}} + 3) \\ & \wedge (2 \nmid x^{\text{pre}} \supset x^{\text{post}} \doteq x^{\text{pre}} + 4) \end{aligned}$$

$\square$

Now we present an example where things get more complicated due to the presence of proper irreducible while loops.

Example 5.6.2 Examine again the list traversion example 4.1.1, inherently retaining loop-like constructions in the specification. Call the program of example 4.1.1 α. Again we will leave out the initial assignment condition $\mathbf{x} \doteq \mathbf{x}^{\mathrm{pre}} \;:=\; \mathtt{x} \doteq x^{\mathrm{pre}} \;:=\; \forall o\, \mathtt{x}(o) \doteq x^{\mathrm{pre}}(o)$, which corresponds to the modifies list $MV(\alpha) = \{\mathtt{x}\}$, for readability. The proof for the program α reasonably only continues from condition 5.2 to the following situation.

$$\frac{\vdash \langle \mathtt{n} = \mathtt{root}\rangle\langle \mathtt{while(n! = empty)\{...\}}\rangle\mathcal{C}}{\vdash \langle\alpha\rangle\mathcal{C}}$$

Then almost the only[12] applicable rule unfolds the loop once. Suppressing the simple assignment $\langle \mathtt{n} = \mathtt{root}\rangle$ this would lead to the following.

$$\frac{\vdash \langle \mathtt{if(n! = empty)\{n.x = n.x + 1; n = n.next; while(n! = empty)\{...\}\}}\rangle\mathcal{C}}{\vdash \langle \mathtt{while(n! = empty)\{...\}}\rangle\mathcal{C}}$$

From this situation the proof could in principle continue with assimilating the branch and elementary assignments into ordinary sequent formulas expressing the same circumstances, for example to the following two open goals.

$$\begin{array}{l} \mathtt{n} \neq \mathtt{empty} \;\vdash \langle \mathtt{n.x = n.x + 1; n = n.next}\rangle\langle \mathtt{while(n! = empty)\{...\}}\rangle\mathcal{C} \\ \mathtt{n} \doteq \mathtt{empty} \;\vdash \mathcal{C} \end{array}$$

But even after encoding the outer diamond's elementary assignments with means of classical formulas, the initial problem of the while loop remains. So instead of unfolding the loop to infinite unreadability the proof should stop leaving the following extracted formula, and corresponding rigid specification after some renaming acrobatics.

$$\begin{array}{rcl} \psi_{\mathcal{C}} & := & \langle \mathtt{n} = \mathtt{root}\rangle\langle \mathtt{while(n! = empty)\{...\}}\rangle\mathcal{C} \\ \psi_{x^{\mathrm{post}} \doteq \mathtt{x}'} & := & \langle \mathtt{n} = \mathtt{root}\rangle \\ & & \langle \mathtt{while(n! = empty)\{n.x' = n.x' + 1; n = n.next\}}\rangle x^{\mathrm{post}} \doteq \mathtt{x}' \end{array}$$

Because of the missing invariant knowledge and no further transformation with the loop body, this only is a remote approximation of the far more readable (also strongest) specification ψ of example 4.1.3. Also, intuitively, that particular ψ offers more insight into what really happens than all variants of $\psi_{\mathcal{C}}$ accomplish, even though they are strongest specifications altogether. This is because of its processing of the loop body and a separate concept for expressing the application of this transformation to all affected list elements. □

[12]No loop induction is possible without knowing a loop invariant. Even though some trivial rules could be applicable, they do not really help.

5.7 Specification Construction Calculus

Specification construction works by starting from the proof obligation condition 5.2, continues by performing some inferences, and concludes by extracting the specification from the proof attempt. Obvious questions are: How many inferences? And which calculus at all?

Which particular kind of calculus to apply is not a decisive question for the feasibility of the specification construction process, as long as the inference rules are equivalent transformations. Still the quality of the specification depends on the nature of the rules. For example, a sequent calculus that is able to close goals is superior to one without closing axioms, since it can omit information that reduces to evident truths from the specification. The application of inference rules establishes a normal form of the program, or its specification, respectively. However, what properties of the calculus guarantee which particular normal form of the specification, is probably a very difficult question to prove formally. Nevertheless, there are some desirable qualitative properties, which can be made plausible informally.

Whether the calculus underlying proof system is a sequent calculus is insignificant. Still we prefer to use sequent calculus notation. That the inference rules should analyse and decompose modalities, is a natural demand, and essential to the fundamental ideas of specification construction with theorem proving. Lengthy programs within the modalities should be consequently reduced in size, with their effect simulated in logical language. Otherwise, the specification construction process would never accomplish a sufficient logical analysis of the computer program. This is, for example, the initial problem with loops, which complicate the further program decomposition. Since branching conditions have a natural counterpart in logic, what remains for the calculus to achieve is an adequate processing of assignments.

Less immediate properties of a specification construction calculus are concerned with simplification. Closing tautological cases is helpful for omitting logically unnecessary information from a specification. Communicating and combining information about the program from different parts of a formula in the proof attempt is vital for removing subcases of the program's effect that refer to incompatible branches of the program, which no program run can ever follow, anyway. This communication is usually ensured by inference rules that decompose complex formulas into simpler subcases, thereby combining distributed information on one branch or subgoal, systematically.

Example 5.7.1 Consider the following JAVA program α. It has branching conditions of mutual effect, which exclude some paths.

```
if (0 < x) {
```

```
        x += 1;              // short form of x = x + 1;
    }
    if (x > 0) {
        x = 3;
    }
```

We abbreviate the second ⌜**if** (x > 0)...⌝ statement with α_2. When leaving out the initial assignment condition $\mathtt{x} \doteq x^{\mathrm{pre}}$ for readability, and when abbreviating some rule applications, the specification construction proof for the program α reaches the following situation.

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{\dfrac{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = 3\rangle \mathcal{C}}{\mathtt{x} > 0, \mathtt{x} > -1 \ \vdash \langle \mathtt{x} = 3\rangle \mathcal{C}}}{0 < \mathtt{x}, \mathtt{x}+1 > 0 \ \vdash \langle \mathtt{x} = 3\rangle \mathcal{C}}}{0 < \mathtt{x}, \langle \mathtt{x}+ = 1\rangle \mathtt{x} > 0 \ \vdash \langle \mathtt{x}+ = 1\rangle\langle \mathtt{x} = 3\rangle \mathcal{C}}
\quad
\dfrac{\dfrac{\dfrac{\dfrac{*}{\bot \ \vdash \ldots}}{\mathtt{x} > 0, \mathtt{x} \leq -1 \ \vdash \ldots}}{0 < \mathtt{x}, \mathtt{x}+1 \leq 0 \ \vdash \ldots}}{0 < \mathtt{x}, \langle \mathtt{x}+ = 1\rangle \mathtt{x} \leq 0 \ \vdash \ldots}
}{0 < \mathtt{x} \ \vdash \langle \mathtt{x}+ = 1\rangle\langle \alpha_2\rangle \mathcal{C}}
\quad
\dfrac{\dfrac{0 \geq \mathtt{x} \ \vdash \mathcal{C}}{\vdots}}{0 \geq \mathtt{x} \ \vdash \langle \alpha_2\rangle \mathcal{C}}
}{\vdash \langle \alpha\rangle \mathcal{C}}
$$

Since no more inference rules are applicable, we replace $\mathcal{C}$ by $x^{\mathrm{post}} \doteq \mathtt{x}$ here. This is possible, because the remaining elementary diamonds promise that only x changes, so that $MV(\alpha) = \{\mathtt{x}\}$. Then we extract

$$
\begin{aligned}
\psi_{\mathcal{C}} \quad &:= \quad (\mathtt{x} > 0 \supset \langle \mathtt{x} = 3\rangle \mathcal{C}) \\
&\qquad \wedge(\mathtt{x} \leq 0 \supset \mathcal{C}) \\
&\hat{=} \quad \mathtt{x} > 0 \supset \langle \mathtt{x} = 3\rangle \mathcal{C}
\end{aligned}
$$

□

Example 5.7.2 The same principle of example 5.7.1 extends to the following program.

```
    if (x < 0) {
        x = 0;
    } else {
        x = x + 1;
    }
    if (0 > x) {
        x = 17;
    } else {
        x = x + 2;
    }
```

Here the specification construction proof is similar, but has more consisable branches. One that can be closed by arithmetic using the tautology $x + 1 <$

$0 \supset x < 0$, and one branch that contains subsumption cases $a+1 < 0 \vee a < 0$. It leads to the following specification.

$$\begin{array}{lll} \psi_{\mathcal{C}} & := & (\mathtt{x} < 0 \supset \langle \mathtt{x} = 2 \rangle \mathcal{C}) \\ & & \wedge (\mathtt{x} \geq 0 \supset \langle \mathtt{x} = \mathtt{x} + 3 \rangle \mathcal{C}) \end{array}$$

□

Further simplification results from inference rules that reconcile information to a briefer representation. On the contrary, inference rules that unnecessarily duplicate information in one formula, are undesirable. So an inference rule that replaces a subformula A by $A \vee A$ would not be a good idea, for example, because its application bloats the resulting specification without any benefit.

Example 5.7.3 Specification construction very often profits from rules for algebraic simplification. Consider the following program α.

```
for (int i = 0; i < 4; i++) {
    s = s + n*i;
}
```

Let W denote the following code snippet, which we will need in the proof.

```
while (i < 4) {
    s = s + n*i;
    i++;
}
```

Then α is equivalent to

```
{
    int i = 0;
    W
}
```

When computing the specification of α, an abbreviated proof attempt – with many closing branches left out – could look like

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\vdash \langle \mathtt{s = s + n * 6}\rangle\langle \mathtt{i = 5}\rangle\mathcal{C}
}{
\vdash \langle \mathtt{s = s + n * 3}\rangle\langle \mathtt{s = s + n * 3; i = 5}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{s = s + n * 3}\rangle\langle \mathtt{i = 4}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{s = s + n + n * 2}\rangle\langle \mathtt{i = 4}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{s = s + n}\rangle\langle \mathtt{i = 2}\rangle\langle \mathtt{s = s + n * i; i + +}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{s = s + n}\rangle\langle \mathtt{i = 2}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{i = 1}\rangle\langle \mathtt{s = s + n * 1; i + +}\rangle\langle W\rangle\mathcal{C}
}
}{
\vdash \langle \mathtt{i = 1}\rangle\langle W\rangle\mathcal{C}
}
}{
\top \ \vdash \langle \mathtt{i = 0}\rangle\langle \mathtt{i = 0 + 1}\rangle\langle W\rangle\mathcal{C}
}
}{
0 < 4 \ \vdash \langle \mathtt{i = 0}\rangle\langle \mathtt{s = s + n * 0; i = i + 1}\rangle\langle W\rangle\mathcal{C}
}
}{
\langle \mathtt{i = 0}\rangle\mathtt{i} < 4 \ \vdash \langle \mathtt{i = 0}\rangle\langle \mathtt{s = s + n * i; i + +}\rangle\langle W\rangle\mathcal{C}
}
\qquad
\dfrac{
\dfrac{
\dfrac{*}{\bot \ \vdash \ldots}
}{
0 \geq 4 \ \vdash \ldots
}
}{
\langle \mathtt{i = 0}\rangle\mathtt{i} \geq 4 \ \vdash \ldots
}
}{
\dfrac{
\vdash \langle \mathtt{i = 0}\rangle\langle W\rangle\mathcal{C}
}{
\vdash \langle \alpha\rangle\mathcal{C}
}
}
$$

The readability of the specification obtained profits decisively from the following simplification inference rule that we have applied during the proof attempt.

$$\frac{(a + b) * x}{a * x + b * x}$$

The specification extracted from the above proof attempt is as follows.

$$\psi_{\mathcal{C}} \quad := \quad s^{\text{post}} \doteq s^{\text{pre}} + 6 * n \wedge i^{\text{post}} \doteq 5$$

Here the information about `i` (or i^{post}, respectively) can also be removed since `i` is a variable local to α only, and thus of no effect that is visible outside. □

Chapter 6

Extensions

In this chapter, we describe extensions of the specification construction approach. Based on theorem proving technology, we continue the specification construction process in order to reach more intuitive equations for describing the effect of a program. Further, we consider an extension of specification construction to partial specification completion, where the task is to automatically add formulas to a partial method specification until the strongest specification is attained.

6.1 Change Equations

We will now describe a way for creating a more intuitive though, perhaps, less precise presentation of the computed specification. Instead of specifying a program by single-step substitutions or updates this section deals with a somewhat more anticipated approach of change equations. Change equations are ordinary equations describing the transitional effect in logical terms and are thus closer to classical first-order logic. So to say, instead of contracting to conditional (single-step) modalities of depth 1, contraction continues further to modalities of depth 0 (no modalities). Since without modalities there is no built-in separation of prestate and poststate, the prestate and poststate values explicitly have to be distinguished syntactically. Therefor, we intend x^{post} to denote the poststate value of `x`.

In order to produce change equations, we add two inference rules to the usual proof system, which perform the transformation from depth 1 modalities to depth 0. One for accumulating state change information as (change) equations, and one for getting rid of the state change accumulation symbol once it occurs on the top-level of a sequent.

$$\langle \mathtt{x} = \mathtt{t} \rangle \mathcal{C} \quad \mapsto \quad \mathcal{C} \wedge x^{\mathrm{post}} \doteq t \tag{6.1}$$

$$\vdash \mathcal{C} \quad \mapsto \quad \vdash \top \tag{6.2}$$

Example 6.1.1 Let us continue the specification construction process of example 5.5.1 by extending the proof attempt as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{x} > 0 \ \vdash x^{\mathrm{post}} \doteq \mathtt{x} + 3}{\mathtt{x} > 0 \ \vdash \mathcal{C} \wedge x^{\mathrm{post}} \doteq \mathtt{x} + 3}}{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 3 \rangle \mathcal{C}}}{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 1 + 2 \rangle \mathcal{C}}}{\mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 1 \rangle \langle \mathtt{x} = \mathtt{x} + 2 \rangle \mathcal{C}} \qquad \dfrac{\dfrac{\dfrac{\mathtt{x} \leq 0 \ \vdash x^{\mathrm{post}} \doteq \mathtt{x} + 2}{\mathtt{x} \leq 0 \ \vdash \mathcal{C} \wedge x^{\mathrm{post}} \doteq \mathtt{x} + 2}}{\mathtt{x} \leq 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 2 \rangle \mathcal{C}}}{\neg \mathtt{x} > 0 \ \vdash \langle \mathtt{x} = \mathtt{x} + 2 \rangle \mathcal{C}}}{\vdash \langle \alpha \rangle \mathcal{C}}$$

From this, we directly extract the specification in change equation form.

$$\begin{aligned} \psi'_{x^{\mathrm{post}} \doteq \mathtt{x}} \quad := \quad & (\mathtt{x} > 0 \supset x^{\mathrm{post}} \doteq \mathtt{x} + 3) \\ & \wedge (\mathtt{x} \leq 0 \supset x^{\mathrm{post}} \doteq \mathtt{x} + 2) \end{aligned}$$

respectively

$$\begin{aligned} \psi_{x^{\mathrm{post}} \doteq \mathtt{x}} \quad := \quad & (x^{\mathrm{pre}} > 0 \supset x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 3) \\ & \wedge (x^{\mathrm{pre}} \leq 0 \supset x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 2) \end{aligned}$$

□

Note that the above simple rules 6.1 and 6.2 rely on flat modalities. This does impose restrictions on generality, though, since the transformation in Chapt. 5 ensures a contraction of modalities. If, however, we would prefer to apply those rules before this has been established, we would have to employ multiple variables $\mathbf{x}', \mathbf{x}'', \mathbf{x}''', \ldots$ for denoting the intermediate values of $\mathbf{x}$. But as soon as we rename the last variable to x^{post}, the same results still hold.

6.2 Specification Completion

Now we consider the slightly more general setting of a partially specified program and the task to complete the specification to the strongest specification given the specified precondition and invariants. The problem of specification completion reduces to ordinary specification construction with a slightly modified initial proof obligation.

Let ϕ_P be the pre-specified (partial) precondition, and ϕ_I the (class) invariant. Instead of condition 2.1 a specification ψ now has to fulfil

$$\models \quad \forall \mathbf{x}^{\text{pre}} \; \left(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \wedge \phi_P \wedge \phi_I \supset [\alpha]\psi\right)$$

Similarly, instead of condition 5.2, we start the sequent calculus construction from the following proof obligation and continue in the usual fashion.

$$\models \quad \forall \mathbf{x}^{\text{pre}} \; \left(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \wedge \phi_P \wedge \phi_I \;\supset\; \langle\alpha\rangle\mathcal{C}\right)$$

Like in condition 5.3 we could specialise $\mathcal{C}$ to $\mathtt{x_1} \doteq {x_1}^{\text{post}} \wedge \ldots \wedge \mathtt{x_n} \doteq {x_n}^{\text{post}}$. Then we can obtain the specification $\psi_\mathcal{C}$ from the proof attempt in the usual manner. In order to check whether α preserves the class invariant, a proof of $\psi_\mathcal{C} \models \phi_I$ is sufficient. A minor inconvenience is that the common invariant ϕ_I will then be repeated in each computed method specification. But when we think of changing the class invariant after adopting the computed specification into the formal model, we could otherwise end up with the wrong method specification if the invariant has been weakened. Also if we think of non-conform subclassing (thus no subtyping) with invariant weakening, then an automatically specified method of the super class should certainly retain the (stronger) class invariant of the super class. Otherwise the scope of the specification validity could be confused with the subclass, and thus invalidated.

User-specified postconditions currently will not be incorporated into the specification construction process. Rather they could be tested for being a consequence of the computed specification. The reasons for this include the following. If we would extend a pre-specified postcondition ψ_P to the strongest specification, a natural wish could be that the computed specification should neither repeat nor subsume the pre-specified part, but only cover additional information. However, the condition that the constructed specification $\psi_\mathcal{C}$ should not subsume ψ, could be fulfilled very easily by coming up with the specification $\neg\psi_P \supset \psi_\alpha$. But this is, of course, not what has been intended, originally. Therefore the wish that $\psi_\mathcal{C}$ shell not repeat the pre-specified postcondition seems ill-defined.

Instead, when we allow the computed specification to repeat the part that ψ_P already knew, the specification construction process consequently does not have to respect nor know ψ_P at all. Then we are back to the original specification construction and the sole use of ψ_P would be a check for entailment by ψ_α.

6.3 Weak Specifications

It may be tempting to suspect that applying inference rules that are only correct for an ordinary proof calculus but are no equivalent transformations, is no harm and would still result in a weaker specification. Unfortunately, this is not the case. From Res. 3.1.6 in conjunction with Res. 3.2.1 it is easy to see that inference rules that are no equivalent transformations but only their premise is sufficient for their consequence will prohibit coming up with a proper specification. So when we want to reduce our demand of aspiring to find the *strongest* specification we have to adopt "necessary" rules that are not "sufficient", but not the other way around. The sole inference rule that really posed a problem when demanding local equivalence has been the loop induction rule. Finding any substitute for it is easy as the proof of Res. 3.2.1 shows. We just need a formula of dynamic logic (including programs) that expresses the iterative effect of a while loop. So even leaving out all rules for while loops will do, since according to Res. 5.5.2 the result is the strongest specification, nevertheless. But reaching a readable and simple formula for a loop still is an open research problem and may require weakening specifications along these lines.

Example 6.3.1 In order to demonstrate the effect that the application of those rules, which are no equivalence transformations, have on the specification, let us continue example 5.5.1. When we would continue the proof attempt with weakening we could reach the following situation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \langle \mathtt{x = x + 3} \rangle \mathcal{C}}{\mathtt{x > 0} \ \vdash \langle \mathtt{x = x + 3} \rangle \mathcal{C}}}{\mathtt{x > 0} \ \vdash \langle \mathtt{x = x + 1 + 2} \rangle \mathcal{C}}}{\mathtt{x > 0} \ \vdash \langle \mathtt{x = x + 1} \rangle \langle \mathtt{x = x + 2} \rangle \mathcal{C}} \qquad \dfrac{\mathtt{x \leq 0} \ \vdash \langle \mathtt{x = x + 2} \rangle \mathcal{C}}{\neg \mathtt{x > 0} \ \vdash \langle \mathtt{x = x + 2} \rangle \mathcal{C}}}{\vdash \langle \alpha \rangle \mathcal{C}}$$

From this proof attempt, which involves non-equivalence transformations, we extract

$$\begin{array}{rcl} \psi_{\mathcal{C}} & := & (\langle \mathtt{x = x + 3} \rangle \mathcal{C}) \\ & & \wedge(\mathtt{x \leq 0} \supset \langle \mathtt{x = x + 2} \rangle \mathcal{C}) \end{array}$$

And would receive the following formula, which fails to qualify as a proper specification of α completely, since it implies $3 \doteq 2$ under the satisfiable predication that $x^{\mathrm{pre}} \leq 0$.

$$\begin{array}{rcl} \psi_{x^{\mathrm{post}} \doteq \mathtt{x}} & := & (x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 3) \\ & & \wedge(x^{\mathrm{pre}} \leq 0 \supset x^{\mathrm{post}} \doteq x^{\mathrm{pre}} + 2) \end{array}$$

Thus weakening is no admissible rule during specification construction, and our notion of equivalence transformations correctly classifies the weakening rule as non-applicable. $\square$

The loop induction rule bears obstacles for specification construction of a completely different quality. Wrong choices of the loop induction invariant I give rise to counter-factuals and contra-intuitive propositions within the specification. For example, if the loop initialisation condition $\Gamma \vdash I, \Delta$ fails to close (or even reduces to $\perp$), it will appear within the extracted specification, much to the confusion[1] of the user. Variations of inductionless induction [Comon, 2001] and rippling [Bundy et al., 1993, Bundy and Lombart, 1995] techniques could provide a suitable solution for loop specification.

Skolemisation is a less troublesome case. Application of the non-equivalence transformation Skolemisation during specification construction leaves “undeclared” constant Skolemisation symbols. They can be thought of as implicitly existentially quantified by the user. The only remaining problem is that the precise conditions that the Skolemisation symbols had to satisfy is lost without Hilbert ϵ-rules.

[1] $\Gamma \vdash I, \Delta$ most probably will not have an apparent link to its meaning for the program's effect.

Chapter 7

Implementation

In this chapter, we briefly describe the implementation of the specification construction process, its user interface and current limitations.

7.1 User interaction

The practical sequent calculus approach of Chapt. 5 has been implemented as an additional module for the KeY System. Automatic specification construction support shows up as an additional entry to the context menu of the CASE tool modelling environment. An automatic specification construction process is initiated from the case tool by activation of that menu action on the desired JAVA method implementation. In the appearing interactive prover window, a click to the "Extract Specification" entry of the tools menu starts the automatic prover and finally extracts and displays the resulting specification.

7.2 Details

Behind the curtain, the automatic prover relies on the application of heuristics – assuming non-equivalent transformation rules to be deactivated, for example δ-rules and weakening[1]. Having reached a "good" final proof situation, perhaps by running out of applicable inference rules from the heuristics, the specification extracts from the proof situation as a conjunction of the open goals[2]. Somewhat involved from a purely technical point of view, is

[1]Loop induction rules currently require user interaction, such that heuristics will not apply them unintentionally.

[2]As a peculiarity to KeY , constraints, as resulting from γ-rules, have to be lowered then, i.e. translated back to ordinary dynamic logic without constraints. This can be

the assembly of the initial specification construction proof obligation. First, there are several settings that control the precise treatment of prestate remembrance and poststate accumulation. And second, the isolation of the dynamic logic representation of the program diamond from an UML and OCL based modelling environment is somewhat involved. But apart from these technical details the implementation directly continues along the lines of the process described in Chapt. 5.

7.3 Limitations

The current state of the implementation still has some simple technical limitations, though none poses an inherent difficulty. As far as adjusting the heuristic settings is concerned, our implementation does not yet automatically remove δ-rules temporarily from the set of applicable heuristics. Also a reasonable setting for the maximum depth of heuristics applied before coming up with a specification, needs user interaction. In conjunction with the missing heuristic adjustment, this may sound improper but actually is of great advantage to the user for controlling the precise form that automatically constructed specifications will reduce to. About the most severe theoretical limitation is the coarse treatment of KeY constraints (as resulting from γ-rules) in our implementation. But just like δ-rules, they almost never seem to occur in practical examples of specification construction without lemma introduction. The most severe practical limitation, however, is simplification. Currently, loads of state change information about internal variables sneak their way into the final specification displayed. Due to a missing program variable scope concept in KeY , removing this variable noise from the output bears unnecessarily complicated technical implementation details, but is simple in principle, and probably a worthwhile future extension of the implementation.

achieved by replacing them with their "declarative" meaning, a disjunction of conjunctions of equalities.

Chapter 8

Summary

In the previous sections we have seen that the problem of automatically specifying a program given its implementation can be solved in full generality. When embedding program specifications into dynamic logic, a natural construction guarantees to reach "the" strongest specification of any program. So even though strongest specifications *per se* are undecidable, their construction is effective. This means that automatically checking arbitrary formulas for being a strongest specification of a program is impossible on Turing machines, but, nevertheless, producing a single formula that is a strongest specification can in fact be done. However, even though this constructed specification shares all computationally relevant conceptual properties with the original program, deciding on the basis of the specification which program properties really hold, is impossible on a Turing machine.[1] Even worse, the constructed strongest specification does not necessarily reach a higher degree of readability than the original program source. Achieving legibility really is the true challenge. Anyhow, a major advantage of this approach is that it is no ad-hoc method, but embedded into a well-understood logic setting. Additionally, the corresponding algorithm is close to the theory and experiences a natural embedding into logic.

A practical approach for constructing specifications automatically results from using the automatic theorem proving capabilities for the dynamic logic into which program specifications have been embedded. Therewith the advantages of the well-understood precise semantics and mechanisms of logic extend to the specification construction process. Also the specific form of the constructed formulas can be adjusted simply by changing the concise local representation of the rewrite process, namely inference rules, instead of having to alter a complex construction algorithm. Of course, this explicit

[1] Of course, Turing machines cannot decide those properties when given the program source code, either.

formulation with rewrite rules does not completely alleviate the problem of investigating the resulting normal forms. So the problem of readability has a well-established setting, but still is highly non-trivial (undecidable). Especially in imperative languages, by far the hardest readability problem happens to impose the (unbounded) while loop, or equivalent means of unlimited repetition. Further readability concerns involve reconciling branching cases, especially subsumption cases.

Roughly speaking, what our practical sequent calculus approach can currently specify fully automatically are at least all those programs with *a priori* constant loops. It most certainly is not a coincidence that this is essentially as far as a single step[2] of abstract state machines [Gurevich, 2000, Börger and Stärk, 2003] can go. Incomplete, still, is the concept of iteration or repetition in our practical approach. Our theoretical investigations though have shown that there is no principle obstacle with iteration. On the other hand, how to achieve and express specifications of unbounded iteration also in a readable manner is still an open research problem. Following the course drafted in the proofs of our theoretical results, perhaps, would not lead to the goal of simplicity and readability, despite the guarantee of yielding the strongest specification. Even though simulating the theoretical proof result would be simple to achieve in practice: just remove all while rules.

Of course, the central conceptual difficulty common to all specification construction approaches also applies to our approach. After extracting the specification from an implementation, all bugs of the implementation will persist in the specification. Therefore, the main advantage of formal specifications, namely improving program correctness by detecting discrepancies between specification and implementation, is impossible for computed specifications. Still, automated specification construction helps to improve program stability. In the face of an object-oriented program consisting of a mixture of specified and unspecified classes, specification construction allows an integration into one formal model. And proofs about that formal model amount to relative correctness statements about the user-specified classes. Even though there cannot be a proper notion of correctness for the unspecified classes, the specified classes can be verified to work as expected, given the actual effect of the unspecified classes – independent of whether the unspecified classes have the effect originally intended by the designer, or not. Furthermore, hu-

[2]A single step of an abstract state machine (ASM) consists of nested conditions and parallel updates, i.e. elementary modifications to the interpretation of a non-rigid function symbol at one position. The normal form attained by performing specification construction on the basis of the current KeY calculus from [Ahrendt et al., 2004] consists of conditional elementary modifications as well. The refined simultaneous updates of KeY even work in parallel, reaching an apparent similarity with abstract state machines.

mans can check far simpler whether a program has the intended effect on the basis of a (computed) specification, when the particular details of an implementation are ignored. For example, computed specifications can disregard some performance optimisations, tricky special case checks or unnecessarily complex implementations. Thus, under pragmatic aspects, automatic specification construction can be a valuable tool for the construction of program systems.

What is worth noting is that all our specification computation does *not* solve the frame problem completely, even though it may appear to at first sight. Of course, only due to the additional knowledge of dealing with strongest specifications, we can attempt to conclude that some state, which has not been mentioned within the specification, should be free of change.[3] Thus, with the resulting precise description of what changes and how, it is tempting to assume that every other thing remains unchanged. This circumstance might be mistaken for a solution of the frame problem. But in the presence of references, our statements about the code are subject to the aliasing problem. So even though we are sure about every object that does refer to or contain a value affected by the modifies list, it is difficult (undecidable) to isolate all other objects that are free of any indirect influence. So in the end we find ourselves in the surprising situation of having a precise logical specification of what any program does without being able to infer all program properties from it. On the other side, this situation is not worse than with the initial program. A closer look even reveals that since we have an effective procedure for constructing the strongest specification of every program containing all computationally relevant information, but all nontrivial properties of programs are undecidable, then we will still not be able to decide them with an intermediate automatic construction of specifications. In short we conclude that the intrinsic problems do not lie in constructing specifications automatically from the implementation, but in proving theorems about the specification. And so our approach is in a way "relatively complete". Still, from a pragmatic point of view, proving properties of the specification usually is far simpler than proving complex statements just by looking at the source code. Specifications, as constructed from the implementation algorithmically, at least form an appropriate intermediate formal representation for proving statements about the program.

[3]For weak specifications, this will never be true because they could simply have forgotten to mention some particular effect, but the strongest specification would not be allowed to ignore any.

Appendix A

Properties

In this chapter, we state and prove properties of the basic concepts of substitutions, rigidity, and specifications, which we need during our investigations.

A.1 Substitution

Lemma A.1.1 (Lemma of Substitution) *If $\sigma(z)$ is rigid for every modal operator passed by during $\sigma := [z \mapsto \sigma(z)]$* [1] *then for each interpretation I for each state s*

$$\mathrm{val}_I(s, \sigma(\phi)) = \mathrm{val}_{I[z \mapsto \mathrm{val}_I(s,\sigma(z))]}(s, \phi)$$

Proof: Note that we implicitly presume that σ is admissible for ϕ, i.e. it does not introduce new illegal bindings inside the scope of a quantifier. This can always be achieved by α-conversion. The proof follows an induction over the formula structure of ϕ. We take $\{\star, \exists, \langle\rangle\}$ as a logical basis where $\star$ denotes an arbitrary binary logical connective, for example $\supset$.

IH As induction hypothesis we use that the conjecture is true for each formula ψ of a simpler structure than ϕ (so it is true for any state and interpretation). In the following always let I be an arbitrary interpretation, s an arbitrary state, and call $\tilde{I} := I[z \mapsto \mathrm{val}_I(s, \sigma(z))]$.

(I) If ϕ is an atomic formula then $\mathrm{val}_I(s, \sigma(\phi)) = \mathrm{val}_{\tilde{I}}(s, \phi)$ by the substitution lemma of classical logic.

[1] This is short saying for: If $\sigma(z)$ is rigid for every modal operator passed by during the application of the substitution σ to ϕ, i.e. whenever replacing an occurrence of z by $\sigma(z)$, the latter is rigid for any modal operators within whose scope z appeared in.

(II) If $z \notin FV(\phi)$ then because of $\sigma(\phi) = \phi$ and the fact that the variable assignment for z does not have any effect on the evaluation of ϕ, the conjecture is obvious.

(III) If $\phi = \psi \star \chi$ for a logical connective $\star$ then $\mathrm{val}_I(s, \sigma(\phi)) = \mathrm{val}(\star)\big(\mathrm{val}_I(s, \sigma(\psi)), \mathrm{val}_I(s, \sigma(\chi))\big) \overset{IH}{=} \mathrm{val}(\star)\big(\mathrm{val}_{\tilde{I}}(s, \psi), \mathrm{val}_{\tilde{I}}(s, \chi)\big) = \mathrm{val}_{\tilde{I}}(s, \phi)$.

(IV) If $\phi = \exists x\, \psi$ then because of II we only have to prove the case of $z \in FV(\phi)$. Then $z \neq x$ as well as $x \notin \sigma(z)$ or else there would have been a collision. Thus $\mathrm{val}_I(s, \sigma(\phi)) = \mathit{true} \overset{z \neq x}{\iff}$ there is d $\mathit{true} = \mathrm{val}_{I[x \mapsto d]}(s, \sigma(\psi)) \overset{IH}{=} \mathrm{val}_{I[x \mapsto d][z \mapsto \mathrm{val}_{I[x \mapsto d]}(s, \sigma(z))]}(s, \psi) \overset{z \neq x}{=} \mathrm{val}_{I[z \mapsto \mathrm{val}_{I[x \mapsto d]}(s, \sigma(z))][x \mapsto d]}(s, \psi) \overset{x \notin \sigma(z)}{=} \mathrm{val}_{I[z \mapsto \mathrm{val}_I(s, \sigma(z))][x \mapsto d]}(s, \psi) = \mathrm{val}_{\tilde{I}[x \mapsto d]}(s, \psi) \iff \mathrm{val}_{\tilde{I}}(s, \phi) = \mathit{true}$.

(V) If $\phi = [\alpha]\psi$ then there are two possibilities.

(a) $z \in FV(\psi) \Rightarrow \sigma(z)$ is rigid for α by premise $\Rightarrow \mathrm{val}_I(s, \sigma(\phi)) = \mathit{true} \iff$ for each $s\rho(\alpha)s'$ $\mathit{true} = \mathrm{val}_I(s', \sigma(\psi)) \overset{IH}{=} \mathrm{val}_{I[z \mapsto \mathrm{val}_I(s', \sigma(z))]}(s', \psi) \overset{\text{rigid}}{=} \mathrm{val}_{\tilde{I}}(s', \psi) \iff \mathrm{val}_{\tilde{I}}(s, \phi) = \mathit{true}$.

(b) $z \notin FV(\psi) \Rightarrow$ according to II applied to ψ, the conjecture is immediate.

■

Note that there is just one simple induction left to generalise this lemma to arbitrary substitutions of a greater set of support than just $\{z\}$. But because of its awkward precise formulation we prefer to leave it at this intuitive level.

Example A.1.1 $[i \mapsto x]$ is rigid for modal operators passed by in $i \doteq 1$, and in $i \doteq 1 \wedge [\mathtt{x} = \mathtt{x} + \mathtt{1}]x \doteq 2$, but not for $i \doteq 1 \wedge [\mathtt{x} = \mathtt{x} + \mathtt{1}]x \doteq i$ or $i \doteq 1 \wedge [\mathtt{x} = \mathtt{x} + \mathtt{1}]a \doteq i$. □

Remark A.1.2 *If $z \notin t$ and t is rigid for every modal operator passed by during the substitution*[2] *then* $\phi(t) \equiv \forall z\, (z \doteq t \supset \phi(z)) \equiv \exists z\, (z \doteq t \wedge \phi(z))$ [3].

Proof: $\mathrm{val}_I(s, \exists z\, (z \doteq t \wedge \phi(z))) = \mathrm{val}_I(s, \forall z\, (z \doteq t \supset \phi(z))) = \mathrm{val}_{I[z \mapsto \mathrm{val}_I(s,t)]}(s, \phi(z)) \overset{\text{Res. A.1.1}}{=} \mathrm{val}_I(s, \phi(t))$. ■

[2] which is implicit in the application $\phi(t)$, i.e. $[\lambda_1 \mapsto t]$

[3] $\equiv \big(\lambda z.\phi(z)\big)(t)$ [Fitting and Mendelsohn, 1998]

Remark A.1.3 *Let t be any term, then* $\models \langle \mathtt{x} = \mathtt{t}\rangle\phi \iff \models \mathtt{x} \doteq t \supset \phi$

Proof:

"$\Rightarrow$" Let s be any state with $s \models \mathtt{x} \doteq t \Rightarrow s\rho(\mathtt{x} = t)s$ where $s \models \phi$ by premise.

"$\Leftarrow$" Let s be any state, and let $s\rho(\mathtt{x} = t)\underbrace{s[\mathtt{x} \mapsto \mathrm{val}_I(s,t)]}_{s'} \Rightarrow s' \models \mathtt{x} \doteq t \overset{\text{premise}}{\Rightarrow} s' \models \phi$.

■

A.2 Rigidity

Remark A.2.1 *A formula ϕ that is rigid for α satisfies for $\star \in \{\vee, \supset\}$ (but not for $\star \in \{\wedge, \leftrightarrow, \subset\}$)*

$$\phi \star [\alpha]\psi \quad\equiv\quad [\alpha](\phi \star \psi)$$

Proof: by definition of rigid and the semantics of modal operators. More precisely

$\star = \vee$ "$\rightarrow$" $\models \phi \vee [\alpha]\psi \overset{\text{Res. A.2.2}}{\Rightarrow} \models [\alpha]\phi \vee [\alpha]\psi \Rightarrow \models [\alpha](\phi \vee \psi)$.

"$\leftarrow$" $\models [\alpha](\phi \vee \psi)$. Let s be any state. Assuming that $s \models \neg\phi$ we have to show that $s \models [\alpha]\psi$. Because of rigidity it is for each $s\rho(\alpha)t$ $t \models \neg\phi \Rightarrow t \models \psi \Rightarrow s \models [\alpha]\psi$.

$\star =\supset$ $\phi \supset [\alpha]\psi \equiv \neg\phi \vee [\alpha]\psi \overset{\star=\vee}{\equiv} [\alpha](\neg\phi \vee \psi) \equiv [\alpha](\phi \supset \psi)$.

$\star =\leftrightarrow$ A counter-example is a world with $s \models \neg\phi$, $s\rho(\alpha)t$, $s\rho(\alpha)t'$, $t \models \neg\phi \wedge \psi$, $t' \models \neg\phi \wedge \neg\psi$. Thus $s \models \neg[\alpha]\psi \wedge \neg\phi \wedge ([\alpha]\psi \leftrightarrow \phi)$. But $s \not\models [\alpha](\psi \supset \phi)$ since $s\rho(\alpha)t$, $t \models \psi \wedge \neg\phi$.

$\star = \wedge$ One direction is simple: $\phi \wedge [\alpha]\psi \overset{\text{Res. A.2.2}}{\Rightarrow} [\alpha]\phi \wedge [\alpha]\psi \equiv [\alpha](\phi \wedge \psi)$. But the other direction can be refused by this counter example of a nonterminating program. Let $s \models \neg\phi$ and assume that there is no state t with $s\rho(\alpha)t$ then $s \models [\alpha](\phi \wedge \psi) \wedge [\alpha]\psi$ but $s \not\models \phi$.[4]

[4]Be aware that $s \models \neg\phi$ further implies $s \models [\alpha]\neg\phi$ for rigid α. Still this does not contradict the asserted $s \models [\alpha]\phi$, because dynamic logic generally does not satisfy the axioms of system D of modal logic.

$\blacksquare$

Remark A.2.2 *If ϕ is rigid for α then*

$$\begin{aligned} \phi &\models [\alpha]\phi \\ [\alpha]\phi, \langle\alpha\rangle\top &\models \phi \\ \langle\alpha\rangle\phi &\equiv \phi \wedge \langle\alpha\rangle\top \\ &\equiv [\alpha]\phi \wedge \langle\alpha\rangle\top \end{aligned}$$

Proof:

1. $\phi \models [\alpha]\phi$ by definition and the semantics of modal operators.

2. $[\alpha]\phi, \langle\alpha\rangle\top \models \phi$: Assuming the premises hold in some state s we know that for each $s\rho(\alpha)t$ $t \models \phi$. So let t be some state such that $s\rho(\alpha)t$, which we know must exist only by $\langle\alpha\rangle\top$. Then because ϕ is rigid for α it is $\mathrm{val}_I(s,\phi) = \mathrm{val}_I(t,\phi) = \mathit{true}$.

3. $[\alpha]\phi \wedge \langle\alpha\rangle\top \equiv \phi \wedge \langle\alpha\rangle\top$ is a consequence of the above two cases.

4. $\langle\alpha\rangle\phi \equiv \phi \wedge \langle\alpha\rangle\top$

   "$\rightarrow$" Let s be any state in which $\langle\alpha\rangle\phi$ is true, then there is $s\rho(\alpha)t$ $t \models \phi \overset{\text{rigid}}{\Rightarrow} \mathrm{val}_I(s,\phi) = \mathrm{val}_I(t,\phi) = \mathit{true}$, $t \models \top$.

   "$\leftarrow$" Let s be any state in which the premises are true. Then $s \models \phi$, there is $s\rho(\alpha)t$ $t \models \top \overset{\text{rigid}}{\Rightarrow} \mathrm{val}_I(s,\phi) = \mathrm{val}_I(t,\phi) = \mathit{true}$, $s \models \langle\alpha\rangle\phi$.

$\blacksquare$

A.3 Elementary Properties

Remark A.3.1 $i \notin FV(A) \Rightarrow \forall i\,(A \leftrightarrow B) \equiv A \leftrightarrow \forall i\,B$

Proof:

1. $I \models A \Rightarrow$ for each d $I[i \mapsto d] \models A$. Thus since $I[i \mapsto d] \models A \leftrightarrow B \Rightarrow I[i \mapsto d] \models B \overset{d\,\text{arbitrary}}{\Rightarrow} I \models \forall i\,B \Rightarrow I \models A \leftrightarrow \forall i\,B$

2. $I \not\models A \Rightarrow$ for each d $I[i \mapsto d] \not\models A$. Thus since $I[i \mapsto d] \models A \leftrightarrow B \Rightarrow I[i \mapsto d] \not\models B \overset{d \text{ arbitrary}}{\Rightarrow} I \models \forall i \, \neg B \overset{\text{presupposition of existence}}{\Rightarrow} I \models \neg \forall i \, B \Rightarrow I \models A \leftrightarrow \forall i \, B$

■

A.4 Specific Variations

Let us collect simple properties and variations of our formalisation which will prove convenient to have, in some situations.

Remark A.4.1 *Equivalent formulations of program specifications, which all assume* $\mathbf{x}^{\text{pre}}$ *to be a rigid constant or (rigid) variable include*

$$
\begin{array}{lll}
 & & (\textit{condition 2.1}) \\
\Longleftrightarrow & \models & \mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha]\psi \\
\Longleftrightarrow & \models & \forall \mathbf{x}^{\text{pre}} \left(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha]\psi\right) \\
\Longleftrightarrow & \models & Cl_{const(rigid)\mathbf{x}^{\text{pre}}} \left(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \supset [\alpha]\psi\right) \\
\Longleftrightarrow & \models & \exists \mathbf{x}^{\text{pre}} \left(\mathbf{x} \doteq \mathbf{x}^{\text{pre}} \land [\alpha]\psi\right)
\end{array}
$$

Proof: by Res. A.1.2 ■

What we want the modifies list $MV(\alpha)$ to achieve is precisely that we know in advance that every formula not referring to any symbol of $MV(\alpha)$ is rigid with respect to α. More formally, we could take the assertion of Res. A.4.2 as a defining condition for modifies lists. Only then can the search for more manageable criteria start.

Remark A.4.2 *Let* ϕ *be a formula with* $FV(\phi) \cap MV(\alpha) = \emptyset$*, then* ϕ *is rigid for* α*.*

Proof: ■

Bibliography

[Ahrendt et al., 2004] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P. H. (2004). The KeY Tool. *Software and System Modeling.* To appear.

[Ammarguellat, 1992] Ammarguellat, Z. (1992). A control-flow normalization algorithm and its complexity. *Software Engineering*, 18(3):237–251.

[Beckert and Schmitt, 2003] Beckert, B. and Schmitt, P. H. (2003). Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM)*. Brisbane.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag.

[Bundy and Lombart, 1995] Bundy, A. and Lombart, V. (1995). Relational rippling: A general approach. In *IJCAI*, pages 175–181.

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253.

[Comon, 2001] Comon, H. (2001). *Inductionless Induction*, volume 1 of *Handbook of Automated Reasoning*, pages 913–962. Elsevier and MIT Press.

[Detlefs et al., 1998] Detlefs, D. L., Rustan, K., Leino, M., Nelson, G., and Saxe, J. B. (1998). Extended static checking. Research Report 159, Compaq Systems Reseach Center.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

[Fitting and Mendelsohn, 1998] Fitting, M. and Mendelsohn, R. L. (1998). *First-Order Modal Logic*. Kluwer Academic Publishers, 1st edition.

[Flanagan and Leino, 2001] Flanagan, C. and Leino, K. R. M. (2001). Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021:500 et seqq.

[Flanagan, 2002] Flanagan, C. amd Qadeer, S. (2002). Predicate abstraction for software verification. In *29th POPL*. ACM.

[Gabbay and Ohlbach, 1992] Gabbay, D. M. and Ohlbach, H. J. (1992). Quantifier elimination in second–order predicate logic. In Nebel, B., Rich, C., and Swartout, W., editors, *Principles of Knowledge Representation and Reasoning (KR92)*, pages 425–435. Morgan Kaufmann.

[Gannod et al., 1998] Gannod, G. C., Chen, Y., and Cheng, B. H. C. (1998). An automated approach for supporting software reuse via reverse engineering. In *Automated Software Engineering*, page 94 et seqq.

[Gannod and Cheng, 1997] Gannod, G. C. and Cheng, B. H. C. (1997). A formal automated approach for reverse engineering programs with pointers. In *Automated Software Engineering*, pages 219–226.

[Gannod and Cheng, 1995] Gannod, G. C. and Cheng, B. H. C. (July 1995). Strongest postcondition semantics as the formal basis for reverse engineering. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering,* (Toronto, Ontario; July 14-16, 1995), pages 188–197. IEEE Computer Society Press.

[Gosling et al., 1996] Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (1996). *The Java Language Specification*. The Java Series. Addison-Wesley, Massachusetts.

[Gurevich, 2000] Gurevich, Y. (2000). Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111.

[Harel, 1984] Harel, D. (1984). *Dynamic Logic*, volume II of *Handbook of Philosophical Logic*, chapter 10, pages 497–604. Reidel, Dordrecht, 1 edition.

[Harel et al., 2000] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic logic*. MIT Press.

[Harel et al., 2001] Harel, D., Kozen, D., and Tiuryn, J. (2001). *Dynamic Logic*, volume 4 of *Handbook of Philosophical Logic*, chapter 3. Kluwer Academic Publishers, 2 edition.

[J2ME, 2003] J2ME (2003). Java 2 Platform, Micro Edition. http://java.sun.com/j2me/.

[JavaCard, 1999] JavaCard (1999). Java Card technology. http://java.sun.com/products.javacard/.

[Rumbaugh et al., 1998] Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual.* Addison-Wesley.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language User Guide.* Addison-Wesley.

[Schlager, 2000] Schlager, S. (2000). Erweiterung der Dynamischen Logik um temporallogische Operatoren. Master's thesis, Universität Karlsruhe, Fakultät für Informatik. In German. Available at: `http://i12www.ira.uka.de/~schlager/publications/Studienarbeit.ps.gz`.

[Schmitt, 2000] Schmitt, P. H. (2000). Formale Systeme. Vorlesungsskriptum Fakultät für Informatik , Universität Karlsruhe.

[TogetherSoft, 2003] TogetherSoft (2003). TogetherSoft WWW homepage. http://www.togethersoft.com/.

[van Benthem, 2001] van Benthem, J. (2001). *Correspondence Theory*, volume 3 of *Handbook of Philosophical Logic*, chapter 4. Kluwer Academic Publishers, 2 edition.

Acknowledgements

At this place I would like to thank all those who kindly supported me during the work on this thesis. First of all, I am very grateful to Bernhard Beckert, whose advice in the area of logic was crucial for the development of this approach, and who always knew how to come to the essential part. Bernhard Beckert also contributed to a great extent in the creation of the initial concepts. I would also like to express my highest gratitude to Professor Peter Schmitt for providing the opportunity to investigate such an exciting and theoretically inspired topic, and for comments on the work. Many thanks deserve Andreas Roth and Richard Bubel for their continuous help on the implementation, and their extensive knowledge of the KeY System internals. Also I would like to thank Philipp Rümmer for worthwhile discussions about how things really are and for indispensable support with complex TeX macros. And last but not least, I want to say thanks to my wife for her patience, and to my parents for their support.