Test Generation and Animation Based on Object-Oriented Specifications

Thèse

présentée et soutenue publiquement le 9 décembre 2011 par

Matthias P. Krieger

pour l'obtention du Doctorat de l'université Paris-Sud

Jury

Bernhard Rumpe, RWTH Aachen, *Rapporteur*

Catherine Dubois, Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIIE), *Rapporteur*

Burkhart Wolff, Université Paris-Sud, *Directeur de Thèse*

Christine Paulin-Mohring, Université Paris-Sud, *Examinatrice*

Bruno Marre, Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA), *Examinateur*

Laboratoire de Recherche en Informatique
Université Paris-Sud 11

Abstract

The goal of this thesis is the development of support for test generation and animation based on object-oriented specifications. We aim particularly to take advantage of state-of-the-art satisfiability solving techniques by using an appropriate representation of object-oriented data. While automated test generation seeks a large set of data to execute an implementation on, animation performs computations that comply with a specification based on user-provided input data. Animation is a valuable technique for validating specifications.

As a foundation of this work, we present clarifications and a partial formalization of the Object Constraint Language (OCL) as well as some extensions in order to allow for test generation and animation based on OCL specifications.

For test generation, we have implemented several enhancements to HOL-TestGen, a tool built on top of the Isabelle theorem proving system that generates tests from specifications in Higher-Order Logic (HOL). We show how SMT solvers can be used to solve various types of constraints in HOL and present a modular approach to case splitting for deriving test cases. The latter facilitates the introduction of splitting rules that are tailored to object-oriented specifications.

For animation, we implemented the tool OCLexec for animating OCL specifications. OCLexec generates from operation contracts corresponding Java implementations that call an SMT-based constraint solver at runtime.

Keywords: Test generation, Animation, Model execution, UML, OCL, SAT solvers, Isabelle/HOL

Acknowledgements

I would like to thank many people who provided valuable support during the years in which I was working on this thesis.

My thesis advisor Burkhart Wolff took plenty of time to give me advice, guided me through difficult times and allowed me some freedom for pursuing my own research interests. Alexander Knapp helped me get started with my work on animation while I was in Munich and made several valuable suggestions. I would also like to thank my further co-author Achim D. Brucker for the good collaboration.

I am grateful to Bernhard Rumpe and Catherine Dubois for the hard task of reading my thesis and providing a review.

During my time at University Paris-Sud, I was welcomed by the "ForTesSE" group, which was a very pleasant environment to work in. I am very happy about the acquaintances I made there, on a professional as well as a personal level. Makarius Wenzel, in particular, always was ready to take a few minutes, or a few hours, to introduce me to the secrets of the Isabelle system. Frédéric Voisin helped a lot by proof-reading the French part of the thesis.

Last but not least, I would like to thank my wife Florence and my parents for their constant support.

Abstract (in French)

L'objectif de cette thèse est l'assistance à la génération de tests et à l'animation de spécifications orientées objet. Nous cherchons en particulier à profiter de l'état de l'art des techniques de résolution de satisfaisabilité en utilisant une représentation appropriée des données orientées objet. Alors que la génération automatique de cas de tests recherche un large ensemble de valeurs à fournir en entrée d'une application, l'animation de spécifications effectue les calculs qui sont conformes à une spécification à partir de valeurs fournies par l'utilisateur. L'animation est une technique importante pour la validation des spécifications.

Comme fondement de ce travail, nous présentons des clarifications et une formalisation partielle du langage de spécification OCL (Object Constraint Language) ainsi que quelques extensions, afin de permettre la génération de tests et l'animation à partir de spécifications OCL.

Pour la génération de tests, nous avons implémenté plusieurs améliorations à HOL-TestGen, outil basé sur le démonstrateur de théorème Isabelle, qui engendre des tests à partir de spécifications en Logique d'Ordre Supérieure (Higher-Order Logic ou HOL). Nous montrons comment des solveurs SMT peuvent être utilisés pour résoudre différents types de contraintes en HOL et nous présentons une approche modulaire de raisonnement par cas pour dériver des cas de tests. Cette dernière approche facilite l'introduction de règles de decomposition par cas qui sont adaptées aux spécifications orientées objet.

Pour l'animation de spécifications, nous avons développé OCLexec, outil d'animation de spécifications en OCL. A partir de contrats de fonctions OCLexec produit les implémentations Java correspondantes qui appellent un solveur de contraintes SMT lors de leur exécution.

Mots clés : Génération de tests, Animation, Execution de modèle, UML, OCL, Solveurs SAT, Isabelle/HOL

Contents

Chapter 1

Introduction

Testing is a primary means for achieving that computer systems function correctly. Due to the large number of test cases necessary to achieve satisfactory test coverage, it is desirable to automate testing activities as far as possible. The use of *formal specifications*, that specify the required behavior of the system in a machine-readable formal language, facilitates the automation of testing. Formal specifications can help testing frameworks determine whether output returned by the system under test meets the requirements. Moreover, test cases can often be generated automatically from formal specifications. It is argued that formal specifications in themselves improve the quality of software by describing precisely and unambiguously the required behavior of the system, and thus preventing misunderstandings between users and developers. However, the use of formal specifications is still often not considered worthwhile.

Another technique that can help improve the quality of software is *animation* [60]. Animation is the execution of computations that comply with a formal specification based on user-provided input data. If support for animation is available, users can validate the specification by animating it on sample sets of input data (*scenarios*). This is already possible before implementation of the system. For incomplete, faulty or inadequate specifications, animation will typically lead to strange and alarming results. This allows the discovery of errors and misunderstandings before they have lead to a buggy implementation. Animation is a longstanding research problem (see [107] for an overview).

In contrast to automatically generated test cases, animation based on user-supplied input avoids scenarios that are too artificial. Animation can help users gain confidence in the specification by allowing the execution of scenarios that are common for the application domain. Animation is particularly powerful if it is accomplished by generating a prototype implementation. The generated code can be linked with components of the system that are already finished. This allows the system to be tested as a whole, although some of its parts are not yet available. It may even be possible to entirely omit the manual implementation of certain operations if they can be animated efficiently enough.

This thesis proposes novel techniques for the automatic generation of tests from formal specifications and for animation. Due to the widespread use of object-oriented design techniques, we decide to focus on object-oriented specifications. We choose the Object Constraint Language (OCL [125]) as a representative object-oriented specification language. OCL is an established specification language that is closely integrated with the Unified Modeling Language (UML), the de-facto standard for object-oriented modeling.

Both test generation and animation depend fundamentally on constraint solving techniques. In recent years, satisfiability (SAT) solving has emerged as an astonishingly efficient constraint solving approach. SAT solving has already been applied successfully to some testing scenarios, e.g., white box testing with Pex [142]. However, it is still nontrivial to implement tool support based on SAT solvers. We show how SAT solvers, in particular SMT (satisfiability modulo theories) solvers, can be used effectively to solve the constraints we encounter during test generation and animation.

For test generation, we have implemented several enhancements to HOL-TestGen [34], a tool built on top of the Isabelle theorem proving system [120] that generates tests from specifications in Higher-Order Logic (HOL). We show how SMT solvers can be used to solve various types of constraints in HOL and present a modular approach to case splitting for deriving test cases. The latter facilitates the introduction of splitting rules that are tailored to object-oriented specifications.

For animation, we implemented the tool OCLexec for animating specifications in OCL. OCLexec generates from operation contracts corresponding Java implementations that call an SMT-based constraint solver at runtime.

1.1 Contributions

The precise contributions of this thesis are the following.

1.1.1 Fundamental Techniques

These are contributions concerning fundamental techniques used for testing and animation.

Modular Test Theorem Derivation in HOL-TestGen Test generation typically involves some sort of case splitting based on the test specification in order to identify desired test cases. A very common example of such a transformation step is the generation of a disjunctive normal form (DNF) from the specification. We observed that the set of appropriate case splits may depend on various factors such as the application domain or the datatypes used in the specification. In order to take this into account, we present a new modular design of HOL-TestGen's test theorem derivation procedure that can be parametrized with different specification transformations. This allows the application of new case splitting rules in HOL-TestGen and results in a flexible test generation procedure that can easily be adapted to different application domains or variations in the specification language.

Solving Constraints in HOL with SMT Solvers from Isabelle In order to solve the constraints arising in the test data generation phase of HOL-TestGen, we use an interface between Isabelle and an SMT solver. We show how counterexamples returned from the solver can be converted to variable instantiations that are usable within Isabelle. The instantiations are checked in a safe LCF-style manner, i.e., they are ,,reconstructed“ in Isabelle. Recursive functions are omnipresent and indispensable in HOL. In order to solve constraints with recursive predicates, we apply an approach that is based on under-approximating the set of solutions of the constraint. Finally, we give an example of an application of techniques from interactive theorem proving to ,,interactive constraint solving" for test generation. We give experimental results for the efficiency of our constraint solving technique based on case studies with HOL-TestGen.

Solving Arithmetic Constraints with Bounded Quantifiers Bounded quantifiers over integers, which can be regarded as a special case of a recursive definition, are powerful enough to express nearly any kind of constraint. We will see later that it is straightforward to map most OCL constraints to a formula representation with bounded quantifiers. Eager SMT solving is a method for deciding the satisfiability of a constraint by translating it to a Boolean SAT problem. The resulting SAT problem is solved using an off-the-shelf SAT solver. We show how existing approaches to eager SMT solving can be adapted to handle bounded quantifiers. In particular, we pay attention to optimizing the time consumed by translating to a SAT problem. The satisfiability of integer constraints is merely semi-decidable when bounded quantifiers and function symbols are allowed. Hence, every decision procedure for such a language must revert to some kind of potentially non-terminating enumeration. We define a form of enumeration that, based on a prior analysis of the formula, tends to find solutions quickly. Finally, we show how these techniques can be extended to allow for optimization according to an objective function.

1.1.2 Applications

We now turn to the application to concrete test and animation scenarios.

A Semantics for Undefined Values in OCL Both model-based testing and animation are based on a formal specification of the system. Test generation as well as animation necessarily depend on a precise semantics of the specification language. We build on previous work on defining a formal semantics for OCL, in particular the approach based on embedding OCL into HOL [33]. We extend previous work by clarifying the semantics of undefined values [27]. The semantics of undefined values in OCL is particularly intricate since OCL features two distinct undefined values with different semantics.

Extensions to OCL We observe that extensions to OCL are necessary in order to allow for satisfactory animation of OCL specifications. These extensions include a simplified definition of frame conditions by *invariability clauses* and objective functions as a part of operation contracts. We achieve tool interoperability by specifying invariability clauses and objective functions in a UML profile. We present several application examples of objective functions in order to substantiate that objective functions are a useful general-purpose specification instrument [101].

Animation of OCL Operation Contracts We present an efficient as well as fully automatic approach to the animation of OCL operation contracts [102]. It is implemented in the tool OCLexec that generates from OCL operation contracts corresponding Java implementations which call a constraint solver at runtime. We show how OCL constraints can be mapped to semantically equivalent arithmetic formulas with bounded quantifiers. For animation, the operation contract is translated to an arithmetic formula with bounded quantifiers, which can then be solved with an adapted eager SMT approach. Case studies demonstrate that our approach can handle problem instances of considerable size.

Generating Tests from Object-Oriented Specifications As an application of our modular design of HOL-TestGen's test case generation phase, we present a setup tailored to testing based on object-oriented specifications [26]. Taking into account that object identity is one of the most important aspects of object-oriented modeling, we define rewrite rules that distinguish test cases based on object identity. By also considering the multiplicities of object attributes, we are able to systematically enumerate all object graphs that may represent test cases. Moreover, we show how recursive operation specifications can be handled with similar techniques.

Chapter 2

Preliminaries

2.1 SAT and SMT Solving

Satisfiability (SAT) solvers have had an enormous impact on the application areas in which they are used. In this section we review the techniques used by state of the art satisfiability solvers. We first outline the basic DPLL algorithm for deciding Boolean satisfiability which is still at the core of most solving procedures. Then we sketch how this foundation can be extended to deal with richer logics for deciding Satisfiability Modulo Theories (SMT).

2.1.1 The DPLL Algorithm

A Boolean formula is *satisfiable* if there exists an assignment of the Boolean values true and false to the variables occurring in the formula for which the formula evaluates to true. Algorithms for decidability often suppose that the formula is in *conjunctive normal form* (CNF), i.e., a conjunction of disjunctive clauses. Clauses consist of a set of *literals*, i.e., variables or negated variables.

The satisfiability of a general Boolean formula can be reduced easily and polynomially to the satisfiability of a CNF by introducing new variables. Such a reduction also applies more generally to Boolean *circuits*. A Boolean circuit consists of a collection of *gates*, with every gate computing a Boolean function of its inputs. A circuit in which the output of every gate is an input to at most one gate is a formula. Using an encoding due to Tseitin [146], a Boolean circuit can be converted to a CNF with equivalent satisfiability and of size that is linear in the size of the circuit.

The DPLL (Davis-Putnam-Logemann-Loveland) algorithm [57] for deciding satisfiability of a CNF is still the basis of most SAT solving procedures. Figure 2.1 shows the essence of the DPLL algorithm. A key element is a simplification of the CNF through the detection of clauses with only one literal (unit clauses). A unit clause forces the literal l it consists of to assume the value *true* in any satisfying assignment, so all other clauses containing l can be deleted from the

```
procedure DPLL($\varphi$: CNF):
  $\varphi'$ := unit_propagate($\varphi$)
  if $\varphi'$ contains the empty clause then
    return unsatisfiable
  else if $\varphi'$ only contains unit clauses then
    return satisfiable
  else
    $l$ := choose_literal($\varphi'$)
    if DPLL($\varphi' \wedge l$) = satisfiable
        or DPLL($\varphi' \wedge \neg l$) = satisfiable
    then
      return satisfiable
    else
      return unsatisfiable
    end if
  end if
end procedure
```

Figure 2.1: Basic DPLL algorithm

CNF, and all occurrences of $\neg l$ in the CNF can be deleted as well. The resulting simplified CNF is satisfiable iff the original CNF was satisfiable. The deletion of a literal may reduce other clauses to unit clauses, which in turn leads to a further simplification, and so on. This process is called *unit propagation*. If an empty clause is detected at any time during unit propagation, we can conclude that the CNF is unsatisfiable and stop propagating. If all clauses are reduced to unit clauses, we have found a satisfying variable assignment. Otherwise, unit propagation stops with a CNF in which all clauses have at least two literals.

The DPLL algorithm proceeds by recursively setting a Boolean variable to true or false and then performing unit propagation. If unsatisfiability is detected, it is known that the current partial assignment cannot be completed to a satisfying assignment. Then the algorithm *backtracks* up the search tree and continues with a different partial assignment. If no satisfying assignment is found during the search and there are no possible further assignments to try, it can be concluded that the CNF is unsatisfiable and the algorithm terminates.

Implementations of the DPLL algorithm have attained impressive efficiency by thorough optimization of several aspects such as the following:

- When backtracking, it can usually be deduced that assignments to a limited set of variables are responsible for reaching an unsatisfiable partial assignment. Appropriate clauses can be added to the CNF that allow this reason for unsatisfiability to be avoided by unit propagation. This kind of *learning* can be accomplished in various ways [155].

- The choice of the literal selected for assignment in the recursive step is essential for efficiency. Several heuristics have been proposed [139].

- Datastructures for representing the CNF must be designed carefully, consider e.g., the two literal watching scheme [114] for easily detecting clauses relevant for unit propagation.

2.1.2 Eager SMT Solving

Most naturally arising satisfiability problems are not concerned exclusively with Boolean values. Rather, real applications usually also involve other entities like numbers, strings, lists etc. Thus, we are interested in solving the problem of satisfiability modulo the theories of these additional datatypes. An approach to deal with such data is to encode it into Boolean variables, as is done for example by the UCLID tool [104]. By expressing operations on the new types by Boolean circuits, as would be done for computing the operations in hardware, the satisfiability problem can be reduced to a Boolean one. This technique is called eager SMT solving since the satisfiability problem is translated at once to a Boolean representation rather than waiting “lazily” and performing theory-specific reasoning during the Boolean search. Eager SMT solving allows the efficiency of highly optimized Boolean SAT solvers to be exploited for solving problems involving richer logics.

Eager SMT can be used for solving constraints over bitvectors. This technique is called “bit-blasting” since, depending on the maximum bitwidth, a large number of Boolean variables may be necessary to encode a bitvector value or an arithmetic operation. Bit-blasting is a widely used approach for analyzing systems that employ finite-precision bit-vector arithmetic; see [39] for a recent overview. Programming languages like Java specify precise bounds for integer types, so bit-blasting is in principle suitable for solving problems expressed in the context of such languages. Moreover, tailored techniques have been developed for bounding the bitwidth required for solving linear constraints over integers [135]. However, in general such bounds cannot be derived for nonlinear constraints.

When performing eager SMT solving, the time consumed by the SAT solver for solving the resulting Boolean satisfiability problem grows quickly with the size of the input formula and the bitwidth used for encoding integer values. Therefore *abstraction* techniques are of interest that create smaller Boolean satisfiability problems which only partially express the original satisfiability problem. Such abstractions can be under-approximations exhibiting a subset of the solutions of the exact satisfiability problem, or over-approximations whose solutions are a superset of the actual solutions. Hence, if a SAT solver finds a solution to an under-approximation, this solution corresponds to a solution to the original satisfiability problem. If a SAT solver determines that an over-approximation is unsatisfiable, it can be concluded that the original satisfiability problem is unsatisfiable as well. If the approximations defined are insufficient for determining satisfiability of the original satisfiability problem, they can often be *refined* to yield larger Boolean satisfiability problems that express the exact satisfiability problem more accurately.

Although a satisfiable over-approximation yields no direct conclusion about the satisfiability of the original satisfiability problem, the counterexample returned by the SAT solver can guide the further refinement of the abstractions by, e.g., indicating value ranges that are likely to include solutions to the exact problem. Similarly, the subset of the clauses used by the SAT solver for establishing unsatisfiability of an under-approximation, the "unsatisfiable core", can be used for constructing tailored over-approximations.

A class of abstractions proposed for accelerating SMT solving is based on restricting the bitwidth used for encoding integers [39, 36]. Another kind of useful over-approximation can be computed by ignoring certain array axioms [69, 37]. For example, if $a[x]$ and $a[y]$ occur in the input formula, the over-approximation can be constructed by replacing the array accesses by distinct variables. This is an approximation since a counterexample in which $a[x] \neq a[y]$ but $x = y$ is spurious.

2.1.3 Lazy SMT Solving

Lazy SMT solving has been proposed as a technique for allowing theory-specific reasoning to be carried out by dedicated *theory solvers* rather than encoding the theories into Boolean logic (see e.g. [132] for a survey). This approach allows to avoid the blow-up of the Boolean representation inherent to eager SMT solving and makes it possible to handle theories which are difficult to encode like real linear arithmetic.

Lazy SMT solving proceeds by creating a Boolean abstraction of the satisfiability problem. Atomic theory-specific subformulas are represented by Boolean variables in the abstraction. Boolean search according to the DPLL procedure is used to find a conjunction of literals (a *monome*) that implies the abstraction. If such a monome is found, the theory solver is invoked in order to decide whether the monome is satisfiable modulo the theory. If this is not the case, the theory solver returns a clause of literals that expresses the cause of this unsatisfiability. The Boolean search is then resumed with the newly learned clause.

Example Consider the formula

$$x > 2 \wedge (x < 1 \vee x > 3)$$

in the theory of linear arithmetic. Boolean abstraction yields the CNF

$$a \wedge (b \vee c)\,.$$

A first Boolean search could choose the monome $a \wedge b$ which corresponds to $x > 2 \wedge x < 1$. The theory solver for linear arithmetic responds that this monome is unsatisfiable. Then the Boolean search is resumed with the learnt clause $(\neg a \vee \neg b)$. Unit propagation yields the monome $a \wedge c$ which corresponds to $x > 2 \wedge x > 3$. The theory solver constructs the model $x = 4$.

Usually it is desired to decide satisfiability modulo a combination of different theories, each with a different theory solver. Typical theories that are combined are the theory of arrays, the theory of uninterpreted functions, the theories of linear arithmetic as well as theories of bitvectors and of recursive datatypes.

A well-known scheme for combining theories that has been adopted by many solver implementations is due to Nelson and Oppen [116]. In the sequel, we sketch how theories can be combined according to this scheme. We restrict ourselves without loss of generality to the combination of two theories $\mathcal{T}_1$ and $\mathcal{T}_2$. First, the original formula φ is transformed to an equisatisfiable formula $\varphi_1 \wedge \varphi_2$ such that φ_i only contains function symbols of the theory $\mathcal{T}_i$. This can be achieved by substituting a sufficient set of subexpressions by fresh variables and conjoining corresponding equalities with the input formula.

We call a formula ar an *arrangement* of a set X of variables if there is an equivalence relation R on X such that

$$ar = \bigwedge_{(x,y) \in R} x = y \quad \wedge \bigwedge_{(x,y) \notin R} x \neq y.$$

Under sufficient conditions on $\mathcal{T}_1$ and $\mathcal{T}_2$ it can be shown that $\varphi_1 \wedge \varphi_2$ is satisfiable iff there is an arrangement ar of the common variables of φ_1 and φ_2 such that $ar \wedge \varphi_1$ and $ar \wedge \varphi_2$ are satisfiable [143]. Thus, the satisfiability of $\varphi_1 \wedge \varphi_2$ can be determined by checking the satisfiability of $ar \wedge \varphi_1$ and $ar \wedge \varphi_2$ for all possible arrangements. This can be done within a DPLL framework by conjoining $x = y \vee x \neq y$ with $\varphi_1 \wedge \varphi_2$ for all pairs x, y of common variables [22]. Then the satisfiability of a monome encountered during DPLL search can be determined by calling the theory solvers on the respective subsets of the monome's literals.

Example Consider the formula

$$f(x) < f(y) \wedge (x \geq y \vee x < y - 1)$$

in the combined theory of uninterpreted functions and linear arithmetic. We introduce the fresh variables t_1 and t_2 in order to partition the formula according to theories which yields

$$\begin{gathered} t_1 < t_2 \wedge (x \geq y \vee x < y - 1) \\ \wedge\, t_1 = f(x) \wedge t_2 = f(y)\,. \end{gathered}$$

Adding additional equality distinctions for representing arrangements results in

$$\begin{aligned}
&t_1 < t_2 \wedge (x \geq y \vee x < y - 1)\\
&\wedge t_1 = f(x) \wedge t_2 = f(y)\\
&\wedge (x = y \vee x \neq y) \wedge (x = t_1 \vee x \neq t_1) \wedge (x = t_2 \vee x \neq t_2)\\
&\wedge (y = t_1 \vee y \neq t_1) \wedge (y = t_2 \vee y \neq t_2) \wedge (t_1 = t_2 \vee t_1 \neq t_2)\,.
\end{aligned}$$

During the search the constraint $t_1 < t_2$ causes the theory solver for linear arithmetic to object whenever the variables t_1 and t_2 are assigned to the same equivalence class by the arrangement. Suppose the monome

$$\begin{aligned}
&t_1 < t_2 \wedge x \geq y\\
&\wedge t_1 = f(x) \wedge t_2 = f(y)\\
&\wedge x = y \wedge x \neq t_1 \wedge x \neq t_2\\
&\wedge y \neq t_1 \wedge y \neq t_2 \wedge t_1 \neq t_2
\end{aligned}$$

is selected during the search. The theory solver for linear arithmetic accepts this monome, but the theory solver for uninterpreted functions finds a contradiction in $t_1 = f(x) \wedge t_2 = f(y) \wedge x = y \wedge t_1 \neq t_2$.

2.1.4 E-Matching

Formulas arising naturally in applications often include quantifiers. However, for many theories of interest only the quantifier-free fragment is decidable.[1] A common approach of lazy SMT solvers for an approximate treatment of quantified formulas is to instantiate quantifiers with heuristically selected ground terms that are encountered during the search. It can be assumed without loss of generality that the only quantifiers in the formula are positively occurring universal quantifiers, since other quantifiers can be moved or skolemized. For such a formula it can be possible to derive unsatisfiability by heuristic quantifier instantiation if the right ground terms are selected for instantiation and an unsatisfiable ground formula is constructed. Otherwise, however, we cannot make any conclusion about the satisfiability of the formula.

Heuristic quantifier instantiation in lazy SMT solvers, which was pioneered by the Simplify theorem prover [59], typically proceeds as follows. We call a ground term *active* if it occurs in a literal that is currently asserted in the search process. Note that monomes for which a theory solver is called only consist of active terms. Every quantified subformula is associated with a set of formulas with free variables which is called a *trigger*[2]. A quantifier is instantiated with all active terms encountered during the search that match its trigger. Triggers can be generated automatically according to heuristics or can be user-specified.

[1] See [70] for some theory fragments that are decidable despite the presence of quantifiers.

[2] A trigger is also called pattern in some publications and in the SMT-LIB standard.

Solvers maintain so-called *E-graphs* which represent the equalities that are implied by the asserted literals. An E-graph describes an equivalence relation on the active terms. Two ground terms are equivalent if the current assertions imply that they are equal. Some theory reasoning, e.g., in the theory of uninterpreted functions, can be carried out directly on the E-graph by propagating knowledge about equalities and inequalities. Trigger matching is typically not performed syntactically, but modulo the equivalence relation expressed by the current E-graph, which is called *E-matching*. This enlarges the set of matching ground terms and makes quantifier instantiation more powerful.

Example Consider the formula

$$f(0) < f(f(0)) \wedge \forall x. f(f(x)) = f(x) .$$

Suppose that the trigger $\{f(f(x))\}$ is used for the quantifier, and that only the literal $f(0) < f(f(0))$ has been asserted so far in the search process. The ground term $f(f(0))$ matches this trigger, so the instantiation $f(f(0)) = f(0)$ is obtained, and unsatisfiability is derived by linear arithmetic. Consider the equivalent, but syntactically distinct formula

$$y = f(0) \wedge y < f(y) \wedge \forall x. f(f(x)) = f(x) .$$

Now none of the occurring ground terms match the trigger $\{f(f(x))\}$ syntactically. After $y = f(0)$ has been added to the E-graph, however, the term y matches the trigger modulo the equalities in the E-graph, so unsatisfiability can be derived as before.

There are basically two variants of trigger-driven quantifier instantiation: eager and lazy instantiation. When performing eager instantiation, quantifiers are instantiated as early as possible, i.e., as soon as literals are asserted that contain new matching terms. Such a strategy restricts the search space and favors constraint propagation. However, eager instantiation can also be counterproductive if there are many matching terms. In particular, since an instantiation may itself contain ground terms that match the trigger used, infinite *matching loops* can occur.

Figure 2.2 shows a simplified lazy SMT algorithm with eager trigger-based quantifier instantiation. Some of the notation is inspired by [65]. The algorithm is based on the DPLL procedure shown in Figure 2.1. Now we allow the CNF argument to also include quantified variables. When called on a satisfiable CNF, the procedure returns `null`. When called on an unsatisfiable CNF, the procedure returns a set of clauses that, when added to the CNF, enables unit propagation to prevent the configuration of unit clauses in the CNF of reoccurring. This is the DPLL learning mechanism mentioned in Section 2.1.1 that is essential for efficiency, but omitted in Figure 2.1 for simplicity. Learning is particularly important for lazy SMT solving.

```
procedure SMT($\varphi$: CNF):
  $\varphi'$ := unit_propagate($\varphi$)
  if $\varphi'$ contains the empty clause then
    return reason for unsatisfiability
  else if $\varphi'$ only contains unit clauses then
    return theory_solve($\varphi'$)
  else
    $l$ := choose_literal($\varphi'$)
    $\varphi''$ := $\varphi' \wedge \mathrm{match}(\varphi', l)$
    $L$ := SMT($\varphi'' \wedge l$)
    if $L$ = null then
      return null
    else if unit_propagate($\varphi'' \wedge L$) contains the empty clause then
      return $L \wedge \mathrm{match}(\varphi', l)$
    else
      /* $\neg l$ follows from $\varphi'' \wedge L$ by unit propagation */
      $R$ := SMT($\varphi'' \wedge L \wedge \mathrm{match}(\varphi'', L)$)
      if $R$ = null then
        return null
      else
        return $L \wedge R \wedge \mathrm{match}(\varphi', l) \wedge \mathrm{match}(\varphi'', L)$
    end if
  end if
end procedure
```

Figure 2.2: Simplified lazy SMT algorithm with quantifier instantiation

The subroutine `theory_solve` determines the satisfiability of a monome modulo the theory used. Just as the main procedure, it can return `null` or a learned clause. The function match returns the set of clauses to be added to the CNF for quantifier instantiation. The set match (φ, S) are the new clauses that result from adding the set of clauses S to the CNF φ, performing unit propagation and matching ground terms in unit clauses to triggers modulo the E-graph for the unit clauses, and instantiating the corresponding quantifiers.

In its simplest form, the subroutine `unit_propagate` can ignore the theory semantics of the literals and work as in the plain Boolean DPLL procedure. It is desirable for efficiency, however, that `unit_propagate` also performs some theory-specific reasoning. A very basic form of such reasoning is to check the satisfiability of the occurring unit clauses modulo the theory used. This is the same kind of analysis that the subroutine `theory_solve` performs on a set of unit clauses that implies the entire formula φ. This technique called *early pruning* (see e.g. [38]) prevents further expensive Boolean reasoning to be carried out on a configuration of unit clauses that can be shown to be unsatisfiable modulo the theory used. A more sophisticated form of reasoning that can be integrated in `unit_propagate` is *theory propagation* (see e.g. [118]). Theory propagation can deduce literals from sets of unit clauses by theory-specific reasoning, analogous to unit propagation in the Boolean case.

Trigger-driven quantifier instantiation gives a kind of execution model to the language processed by SMT solvers. Thus, in this sense the language of SMT solvers adheres to the slogan *Algorithm = Logic + Control* (see e.g., [97]) of logic programming and somewhat resembles a logic programming language.

2.2 Isabelle/HOL

Isabelle [120] is a well-known interactive theorem proving system. Isabelle follows the "LCF approach", i.e., basic inferences are performed by a small inner kernel. More complex inferences are composed into elementary ones that can be carried out by the kernel. Isabelle's kernel is considered trusted due it to its simplicity and its rigorous validation. Hence, there is very high confidence in theorems proven within Isabelle.

Isabelle is implemented in the functional programming language ML. The term language used by Isabelle is a typed λ-calculus. Theorems proved by the kernel are terms of propositional type that are represented as ML values of type `Thm`. These theorem objects are the basic building blocks of proofs. Theorems can be manipulated and composed in order to create new theorems. A high-level instrument for obtaining new theorems are *tactics*. Tactics are ML functions that map theorems to sequences of theorems. Many Isabelle tactics require an additional argument of type `Proof.context` that provides additional bookkeeping information. The result of a tactic is a sequence of all theorems that can result from the operation represented by the tactic. Examples of commonly used tactics are tactics for resolution, instantiation and simplification. Tactics can be composed using *tacticals*. Tacticals are functions that create new tactics from existing ones. They can be regarded as control structures that guide the way to a proof. Commonly used tacticals are, for example, `THEN` for sequential application of tactics, `COND` for conditional application, and `REPEAT` for iterative application. Tacticals can be used to build complex proving procedures from the basic inference steps provided by Isabelle. An example of such an elaborate high-level tactic implemented in Isabelle is Cooper's algorithm for deciding Presburger arithmetic [45].

Most often proofs in Isabelle are performed in a backward manner. For proving a proposition C, the *goal* $C \Longrightarrow C$ is created. While the conclusion of the implication is left unchanged, the premises are manipulated by tactics, and the original premise C is possibly decomposed into several *subgoals* so that an implication of the form

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C \tag{2.1}$$

results. After all proof obligations have been discharged, the desired theorem C is obtained.

Isabelle is a generic theorem prover, meaning that it can be instantiated with different logics. The most important logic used with Isabelle is Higher-Order Logic (HOL), a variant of Church's type theory [48]. HOL is powerful enough to express nearly all common mathematical theorems. It offers function and predicate symbols, quantifiers over arbitrary sets, the usual logical connectives, and lambda abstractions. Basically HOL can be used like other variants of higher-order predicate logic. HOL has been extended with facilities that ease the development of formal theories. In particular, there is support for conveniently defining recursive datatypes and recursive functions. As a result, HOL can also be informally described as a "functional programming language with quantifiers". New definitions of datatypes and functions are usually added to a theory by *conservative definitions* that ensure that no inconsistencies are introduced. Many generally useful theories have been integrated into HOL, such as lists, sets, natural and rational numbers etc. These theories can be used as universal libraries for developing new theories or specifications.

Interfaces to Isabelle have been created for several third-party theorem proving tools like SAT and SMT solvers [128, 152, 17]. A fundamental problem is that such tools often do not justify the theorems they generate. In this case such theorems can be consumed by Isabelle using the *oracle* mechanism that accepts external theorems but marks them as potentially untrusted, as well as theorems derived from these external theorems. Third-party tools that do justify their theorems usually provide proofs that are very different from Isabelle proofs. In this case it can however be feasible to transform the external proof into a genuine Isabelle proof by mapping the provided proof steps to appropriate Isabelle tactics. This task is referred to as *proof reconstruction* since the external proof is "reconstructed" by the Isabelle kernel. If proof reconstruction is possible, the oracle mechanism is not needed, and the generated theorem should be considered as trustworthy as any other theorem derived within Isabelle. Thus, proof reconstruction can allow to combine the efficiency of highly optimized specialized theorem provers, like SAT solvers, with the safety of Isabelle's LCF architecture. Moreover, a failed attempt to reconstruct a proof can be an indication of an unsoundness flaw in a theorem proving tool.

Isabelle comes with a wide range of features that make the development of large theories and applications practical. The Isabelle system provides a structured proof language [154], an Emacs-based editing environment that displays remaining proof obligations [7], counterexample generators for detecting non-theorems [12, 16] and code generators that can generate implementations of functions defined in Isabelle [11]. As as result, Isabelle could be successfully applied in large projects, such as formalizations of the prime number theorem [8], of the relative consistency of the axiom of choice [127] and of parts of Hales' proof of the Kepler conjecture [119]. Another area of successful applications of Isabelle has been the verification of hardware and system software [4, 80].

2.3 Specification Based Testing with HOL-TestGen

In this section we review the most important notions of testing and introduce the HOL-TestGen tool for automated test generation.

Testing is the process of exercising a system on a selected set of inputs and thereby observing whether the system behaves as required. This general definition of testing applies to the testing of software as well as hardware, or combined hardware/software systems. Testing is an essential part of the software development process. Software development is highly error-prone, so it would be illusionary to expect even a simple program to serve a useful purpose without being tested. For the special case of software systems employed in areas like transportation and medicine in which failures may potentially be life-threatening, it is to be expected that an immense testing effort is required to achieve a satisfactory level of safety and security, and that testing is the most expensive part of the development process.

Since testing only takes into account the behavior of a system on a finite set of inputs, testing alone cannot ensure unconditionally that the system works correctly under all circumstances. The only exception to this rule are hardware components with a memory of only a couple of bits for which exhaustive testing is feasible. A potential alternative to testing are *formal verification* techniques that can guarantee by theorem proving that program code satisfies certain formalized requirements. Although formal verification is attractive because it delivers unconditional assurances of correctness, it also has the following drawbacks when used alone without supplementary testing:

- Formal verification can only guarantee compliance with requirements that can be expressed in the formalisms supported by the respective verification technique. Formal specifications often do not capture important non-functional requirements. E.g., even if formal verification proves that a system always gives a correct response, the system may nevertheless exhibit unacceptable runtime for some inputs.

- Formal verification techniques are based on abstractions of the system that idealize components like the operating system, CPU, compiler etc. Testing may reveal errors in this periphery of the system.

- Due to their complexity, formal verification techniques are tailored to certain implementation languages. However, large systems are often implemented in multiple languages and may depend on third-party binaries, which limits their suitability for formal verification.

- Since by Rice's theorem it is undecidable for any property whether a function computed by a program in a Turing-complete language satisfies it, formal verification is necessarily not fully automatic or limited to a restricted subset of programs. As a result, formal verification requires immense effort and cannot be applied in practice to large systems as a whole.

Due to these limitations of formal verification, testing remains an indispensable element of the software development process.

2.3.1 Test Coverage

Although testing cannot ensure unconditionally that a system works correctly, there are approaches to selecting a set of test cases that exercises a sufficient portion of the system, and thus achieves satisfactory *test coverage*. Different *coverage criteria* have been defined to characterize suitable test suites. The most modest requirement is that every piece of code should be executed at least once by the test suite (*statement coverage*), since an error in a certain piece of code cannot possibly be detected by a test suite if this code is never executed. It is reasonable to demand the slightly stronger criterion of *decision coverage* that requires the test suite to make the condition of every decision in the system evaluate to true as well as false at least once. Thus, for code enclosed in an if-statement, decision coverage requires a test case that executes this code as well as a test case that avoids its execution. Note that statement coverage may not require a test case that avoids the execution.

Of course decision coverage is likely to miss some errors, since an error may only be revealed for specific input values, for example an error caused by a division by zero. *Boundary value analysis* is a technique that seeks values for testing that are likely to reveal certain errors. *Condition coverage* is a coverage criterion that targets errors in Boolean logic by requiring every condition in a Boolean expression to be evaluated to true as well as false at least once. Some coverage criteria, although justified in principle, may be impossible to achieve in practice. *Path coverage*, for example, requires every path through different branches of the program to be executed at least once, unless the path is infeasible, i.e., is not executed for any input. This is a sensible coverage criterion, since some errors can only be revealed by executing certain branches in combination. However, for all but the most primitive programs, path coverage is only a theoretical concept because the number of paths (feasible and infeasible) grows exponentially with the number of decisions encountered during execution, and typically is infinite for programs with loops.

Coverage criteria can be applied to the system under test itself as well as its specification. A specification defines the requirements demanded from the system under test. In general, different implementations of the system with different behavior may conform to the same specification. The precise definition of conformance with a specification depends on the type of specification. An example of a kind of conformance which has attracted attention for testing purposes is input-output conformance, which was introduced by Tretmans [145] for specifications of automata with input and output. In this thesis, we will use HOL and UML/OCL as specification formalisms. It is important to consider coverage criteria related to the specification in addition to criteria concerning the implementation in order to cover features that are present in the specification, but have erroneously been omitted from the implementation. Testing approaches that are only based on the specification are referred to as *black-box testing* since they do not take into account the internals of the system under test. Analogously, approaches that are based on the implementation are characterized as *white-box testing* techniques.

Although the coverage criteria described above may appeal to intuition, none of them can guarantee the detection of all errors in the system under test. Since intuition can be misleading, it is desirable to be able to make formal statements about the quality of certain test suites. An important step in this direction has been the definition of *test hypotheses* [13]. Test hypotheses are assumptions under which it is possible to infer that a system works correctly from the fact that a system passes all tests in a certain test suite. Thus, test hypotheses are a logical description of the gap between the set of correct systems and the set of systems that pass all tests. Test hypotheses are a formal concept for characterizing the quality of test suites. A test suite that requires test hypotheses that are "strong" in some sense for inferring correctness of the system is to be considered inferior to a test suite which allows "weaker" test hypotheses. In other words, test hypotheses are a formal measure of test coverage.

To introduce test hypotheses more formally, let the function $p : I \to O$ model the behavior of an implementation that returns output values in O on input values in I. Let the relation $S \subseteq I \times O$ represent a specification. We say that a program under test p conforms to the specification S if $\forall x \in I.\ (x, p(x)) \in S$. This is a quite general definition of conformance and is adequate e.g., for specifications based on pre- and postconditions. If the program p is specified to satisfy the postcondition post $\subseteq I \times O$ whenever the precondition pre $\subseteq I$ is satisfied, we can define S to be

$$\{(x, y) \in I \times O \mid x \in \text{pre} \Longrightarrow (x, y) \in \text{post}\}.$$

A test is simply a certain input from I for which the output of the program p is checked for conformance with the specification S. Let $T \subseteq I$ denote a set of tests constituting a test suite. The program p passes all tests in the test suite if $\forall x \in T. (x, p(x)) \in S$. The predicate H_T is a sufficient test hypothesis for the test suite T if

$$H_T \wedge \forall x \in T.\ (x, p(x)) \in S \Longrightarrow \forall x \in I.\ (x, p(x)) \in S.$$

The so-called *uniformity* and *regularity* hypotheses are two specific types of test hypothesis that are commonly combined to form a hypothesis H_T that is sufficient for a certain test suite T [13]. The uniformity hypothesis is of the form

$$\forall x_0 \in X.\ ((x_0, p(x_0)) \in S \Longrightarrow \forall x \in X.\ (x, p(x)) \in S), \tag{2.2}$$

or equivalently

$$(\exists x_0 \in X.\ (x_0, p(x_0)) \in S) \Longrightarrow (\forall x \in X.\ (x, p(x)) \in S). \tag{2.3}$$

This hypothesis expresses the assumption that the set X of possible tests is "uniform" in the sense that if any test in X detects an error in the program p, then so will all tests. Thus, under this assumption, it is sufficient to test the program on just one arbitrarily chosen input x_0 from X. The set X that is regarded as uniform typically consists of closely related inputs, such as a set of inputs that lead to the same decision outcomes during execution.

Regularity hypotheses are applicable when a function *size* on the set X of tests under consideration is available that expresses in some sense the structural complexity of its argument. For example, for integers *size* could be defined as absolute value and for lists it could be the length of the list. A regularity hypothesis is of the form

$$(\forall x \in X.\ \mathit{size}(x) \leq k \Longrightarrow (x, p(x)) \in S) \Longrightarrow (\forall x \in X.\ (x, p(x)) \in S). \tag{2.4}$$

Thus, this hypothesis expresses the assumption that the set X of possible tests is "regular" in the sense that if any test in X detects an error in the program p, then so will a test whose *size* is at most k. This corresponds to the intuition that a bug should be able to be exhibited with a small "minimal" example that only includes those features that are relevant for the code containing the bug. In other words, under the assumption that the regularity hypothesis (2.4) is true, it is sufficient to test the program only on the inputs whose *size* is at most k. If this set is still infinite, it is necessary to assume further hypotheses in order to obtain a finite set of tests in the end. The regularity hypothesis can be regarded as a formal variant of the *small scope hypothesis* [87] that some authors refer to.

2.3.2 Automating Testing

Testing is regarded as a tedious activity by many developers. Even for testing small pieces of code, it is often necessary to design many similar test cases in order to sufficiently exercise the code. Determining the correctness of the results returned on these test cases by the system under test also belongs to the task of testing. Moreover, substantial amounts of testing often need to be repeated after changes to the system, even if the modifications are minor.

For these reasons, it is desirable to automate the various steps of testing as far as possible. Frameworks are in widespread use that allow for executing a whole test suite with a single command, e.g. JUnit [91]. Such tools can also check the correctness of the test results as long as this is feasible, e.g., when it is sufficient to compare the results to previously returned ones. A very challenging area of test automation is the problem of generating test cases themselves. The difficulty of generating test cases is due to the fact that it can be very hard to determine inputs that satisfy certain conditions in a specification or exercise certain parts of code during execution of an implementation, or only determine if such inputs exist. The standard approach to generating a test case that tests a certain part of code is to define a formula called path condition that expresses the fact that execution reaches the code when the system is run on a variable input. Solving the path condition for the input then yields a suitable input that can be used for a test case, unless the code in question is dead code and no such input exists. However, for typical systems under test such path conditions become much too large and too complex to be in the reach of the currently available constraint solving techniques.

The automated generation of test cases is a longstanding research problem. Several test case generation tools have been implemented that generate tests from suitable implementations, such as implementations that are coded in Java [5, 21, 18] or a functional programming language [49, 95, 47], or that run on the .NET platform [142]. Another class of tools generate test cases from various kinds of specifications [89, 88, 82, 149]. Generating tests from specifications has the advantage that specifications can be more succinct than implementations that may need to read their input in a complex format or distinguish additional cases for optimization purposes. As a result, the arising path conditions are often simpler and can be solved more easily. Also, the analysis of a system implemented in several different languages is difficult, while it may be possible to provide its specification in a single, less complex language. Another advantage of specifications is that they define the allowed behavior of the system under test. Thus, they also facilitate automated checking of the test results. Altogether, the use of formal specifications has many advantages for testing.

2.3.3 The HOL-TestGen Tool

In the remainder of this chapter, we introduce the HOL-TestGen tool that generates tests from specifications in HOL. This presentation refers to the original version of HOL-TestGen described for example in [34].

For using HOL-TestGen, a property to test is specified by a HOL formula that contains a free variable representing the program under test. This specification expresses that the formula should hold, i.e., be a theorem, when the free variable representing the program is substituted by a function modeling the program under test. HOL specifications can make use of any types, predicates and functions that have been defined properly in HOL. Such HOL theories that describe the background of a test specification can be referred to as *test theories*.

A specification consisting of a precondition pre and a postcondition post could be mapped to the HOL formula

$$\mathrm{pre}\,(x) \Longrightarrow \mathrm{post}\,(x, p\,(x))\,,$$

in which the variable p represents the program under test. The variable x represents the input to the program under test, and is regarded as implicitly universally quantified. Another type of HOL specification that has been used are specifications for testing programs on inputs sequences [31]. These specifications have been designed by modeling input sequences as lists and defining the semantics of executing the program on such a sequence in a dedicated HOL theory.

HOL-TestGen is built on top of the Isabelle theorem prover. Its main component is the Isabelle tactic `gen_test_cases` that transforms a test specification into a *test theorem*. A test theorem is a snapshot of an Isabelle proof state (2.1). The original test specification remains as the conclusion of the test theorem. The premises of the test theorem, i.e., the subgoals of the proof state, are designated as either test cases or test hypotheses. Thus, a test theorem has the form

$$[TC_1, \ldots, TC_n, H_1, \ldots, H_m] \implies TS,$$

where the premises $TC_1, \ldots, TC_n$ are the test cases, the premises $H_1, \ldots, H_m$ are the test hypotheses and the conclusion TS is the test specification. The test cases $TC_1, \ldots, TC_n$ in a test theorem returned by `gen_test_cases` still contain variables. They can be turned into executable tests by instantiating their variables adequately. A test theorem expresses that a program under test conforms to the test specification if it passes all tests and the test hypotheses $H_1, \ldots, H_m$ are true.

When invoking the tactic `gen_test_cases` on a test specification TS, the proof state $TS \Longrightarrow TS$ is created. This is the standard way in Isabelle of creating a proof state from a conjecture. Then the proof state is transformed by applying Boolean equalities to create a conjunctive normal form (CNF). Note that a proof state (2.1) can be written equivalently as

$$\neg A_1 \wedge \ldots \wedge \neg A_n \Longrightarrow C. \tag{2.5}$$

Thus, the subgoals already form a conjunction. A CNF is obtained by transforming the subgoals to disjunctions by applying De Morgan's laws. The Isabelle provides standard tactics for this task. The transformation to a DNF or CNF is a standard approach to generating test cases from specifications [61].

The next step taken by `gen_test_cases` for deriving the test theorem is *case splitting* on free variables whose type is a recursive datatype. The case splitting is based on the *size* function provided by Isabelle/HOL for all recursive datatypes. This function counts the number of constructor applications that are necessary to obtain a value. For every possible *size* of the variable's value from zero to a user-defined constant k, a separate case represented by its own subgoal is introduced. The remaining case of a *size* that is larger than k is taken into account by introducing a subgoal containing a regularity hypothesis (2.4). For example, in the case of a list variable x and $k = 2$, this case split can be expressed by the derivation rule

$$\frac{\begin{array}{c}[x = [\,]]\\ \vdots\\ P(x)\end{array} \qquad \forall a. \ \begin{array}{c}[x = [a]]\\ \vdots\\ P(x)\end{array} \qquad \forall a, b. \ \begin{array}{c}[x = [a, b]]\\ \vdots\\ P(x)\end{array} \qquad \mathit{size}\,(x) > 2 \implies P(x)}{P(x)}. \tag{2.6}$$

Here new variables a, b are introduced for holding the elements of lists considered for the cases $k = 1$ and $k = 2$. These new variables will be processed in the further course of the `gen_test_cases` procedure, depending on their type. The proposition $\mathit{size}\,(x) > 2 \implies P(x)$ represents the regularity hypothesis.

The steps of CNF generation and case splitting are repeated for a customizable number of times. Simplification stages can be inserted if appropriate. After this iteration, any remaining free variables are dealt with by adding subgoals containing uniformity hypotheses (2.3). These uniformity hypotheses allow free variables, that are implicitly universally quantified, to be eliminated by introducing existentially quantified variables. Furthermore, an existentially quantified variable in a subgoal is logically equivalent to an Isabelle metavariable that can be instantiated. These metavariables remain in the test theorem and are instantiated later for generating executable tests.

As an example, we present a test theorem generated by HOL-TestGen for testing a procedure that sorts lists of integers. First, we need to define what "sorted" means in order to specify the desired behavior of the procedure. We achieve this by means of a straightforward recursive definition in HOL:

```
primrec is_sorted:: "int list ⇒ bool"
  where "is_sorted  []       = True" |
        "is_sorted  (x#xs)   = (case xs of
                                  []    ⇒ True
                                | y#ys ⇒ x ≤ y ∧ is_sorted xs)"
```

This recursive definition can be regarded as a very small test theory that forms the background of our specification. We are now ready to specify the property that is to be tested. We intend to simply test that the list returned is sorted:

```
test_spec "is_sorted(PUT (l::(int list)))"
```

Here *PUT* stands for the implementation under test, and *l* represents an arbitrary input list. In order to keep the example simple, we omitted stating that the list returned by the implementation under test must be a permutation of the input list.

After defining the test specification, we can now apply the `gen_test_cases` tactic:

```
  apply (gen_test_cases "PUT")
```

The test theorem returned by `gen_test_cases` consists of the following subgoals:

```
 1. is_sorted (PUT [])
 2. THYP (is_sorted (PUT []) ⟶ is_sorted (PUT []))
 3. is_sorted (PUT [??X8X2])
 4. THYP
      ((∃a. is_sorted (PUT [a])) ⟶
     (∀a. is_sorted (PUT [a])))
 5. is_sorted (PUT [??X5X2, ??X6X3])
 6. THYP
     ((∃a aa. is_sorted (PUT [a, aa])) ⟶
     (∀a aa. is_sorted (PUT [a, aa])))
 7. is_sorted (PUT [??X1X2, ??X2X3, ??X3X4])
 8. THYP
     ((∃a aa ab. is_sorted (PUT [a, aa, ab])) ⟶
      (∀a aa ab. is_sorted (PUT [a, aa, ab])))
 9. THYP (3 < length l ⟶ is_sorted (PUT l))
```

Subgoals marked by *THYP* constitute test hypotheses. The last subgoal is a regularity hypothesis for lists and expresses that the result for an input list larger than 3 is considered to be sorted. The other test hypotheses are uniformity hypotheses. The subgoals that are not marked with *THYP* correspond to test cases. The variables remaining in the test theorem can be instantiated by arbitrary integers (since this test theorem does not prescribe any constraints on the variables' values) in order to obtain executable tests.

The test theorem is stored when `gen_test_cases` terminates. It can later be retrieved and inspected by the user like any other theorem derived in Isabelle. The command `gen_test_data` provided by HOL-TestGen applies constraint solving techniques to instantiate the variables that still are present in the test theorem's subgoals for generating executable tests. The tests obtained by `gen_test_data` are stored in a dedicated data structure. HOL-TestGen also provides a command `gen_test_script` that packages the test cases in a ML program. This ML code can execute the tests on any implementation that can be called from ML and analyze the results. For deciding the correctness of the test results, HOL-TestGen makes use of Isabelle's support for generating code from HOL [11].

A notable application of HOL-TestGen has been the test of security policies in various domains [31, 25, 23, 24]. To facilitate this task, a tailored simplification method for security policies has been designed in Isabelle [23].

Other tools similar to HOL-TestGen are LOFT [110] and FocalTest [42]. LOFT is a deduction system for test generation that makes test hypotheses explicit. FocalTest generates tests from specifications that are defined within a proof environment.

2.4 The Object Constraint Language (OCL)

The Unified Modeling Language (UML) is a graphical modeling language that is regarded as the de-facto standard for object-oriented modeling. OCL [125] is a textual language complementing UML that can be used for specification tasks that are difficult or impossible to accomplish with UML diagrams alone. OCL has a variety of applications at different modeling levels. In this thesis we use OCL for querying and constraining UML system states.

The expressions of OCL constitute the core of the language. The evaluation of OCL expressions is free of side effects. OCL provides convenient means for navigating across associations and retrieving objects. For an OCL expression `e`, the expression `e.a` denotes the value of the attribute `a` for the object `e`. In some contexts it is also meaningful to refer to the value `e.a`**`@pre`** in the pre-state of an operation, i.e., the value before its potential modification by the operation call.

In addition to user-defined classes and the primitive types `Boolean`, `Integer`, `Real` and `String`, OCL offers the collection types `Set`, `Bag` (i.e., multiset), `Sequence` and `OrderedSet`. Collections appear usually by evaluating attributes with a corresponding multiplicity, but OCL also features dedicated collection constructors. The OCL standard library provides numerous collection operations. These operations are accessed by means of the `->` operator, e.g., `e->size()` denotes the size of the collection-valued expression `e`. Special operations called *iterators* allow variables to refer to collection elements. A type of such iterators are comprehensions, which are named `collect` in OCL. The mathematical expression $\{x + 1 \mid x \in A\}$ would be written as `A->collect(x | x + 1)` in OCL. Variables in iterators can also be left implicit, e.g., `e->collect(a)` denotes a collection of all values the attribute `a` can assume for any object in the collection `e`. The precise type of the resulting collection depends on the type of the source `e`.

Another important class of iterators allow quantification over collections. Due to this focus on collection manipulation, OCL somewhat resembles SQL.

Pre- and postconditions are OCL expressions of type `Boolean` that specify operations. A precondition has to be fulfilled when the operation specified by it is called and a postcondition has to hold when the operation returns. An OCL operation contract consists of an arbitrary number of pre- and postconditions. Another specification instrument are class invariants that have to always hold for all objects of a certain class, independent from any operation calls.

The OCL standard [125] explicitly allows pre- and postconditions to refer to functions that are specified by OCL operation contracts. In particular, a contract can refer to the operation that it specifies, which corresponds to a recursive definition. However, this kind of recursive definition in OCL differs substantially from recursive definitions in functional programming languages, since OCL operation contracts, also recursive ones, are satisfied by any implementation (function) that conforms to the pre- and postconditions. Thus, in general, an OCL operation contract can be satisfied by several different implementations. Therefore recursive OCL operation contracts bear more similarity to algebraic specifications. The OCL standard forbids “infinite” recursion without giving a definition of this notion.

Chapter 3

Modular Test Theorem Derivation in HOL-TestGen

The original procedure of HOL-TestGen for deriving a test theorem, which was described in Section 2.3.3, discriminates between variables whose type is a recursive datatype and variables that have a different type. For recursive datatypes, separate test cases for different *size*s of the variable's value are introduced in combination with a regularity hypothesis, whereas for other types a single case is produced by means of a uniformity hypothesis. It is evident that testing is likely to benefit from adding custom treatment for further types in addition to recursive datatypes. For example, for bitvector types boundary values like 0, 1, -1 as well as the maximal and minimal values of the types are particularly interesting for testing, and could be considered separately in different test cases. This would constitute a kind of case splitting that is tailored to the respective bitvector types. Set and collection types are further interesting targets. For these types, exhaustion lemmas like (2.6) cannot be derived the same way as for recursive datatypes. It is however conceivable to introduce an analogous kind of case splitting for such types that is based on using the collection's size as *size* function in the regularity hypothesis.

In this chapter, we present an improved test derivation procedure for HOL-TestGen with a modular design. The modularity allows tailored derivation rules for arbitrary types to be plugged into a generic kernel. This facilitates the customization of HOL-TestGen to new application domains and avoids a complicated implementation that is cluttered with special cases for handling various types. Furthermore, we highlight the affinity of theorem proving and testing by presenting tactical code for test derivation. As additional contribution of this chapter, we discuss the relationship between case splitting and test hypotheses and give some advice for producing effective test suites.

```
procedure gen_test_cases(test_spec, rules, breadth):
  proof_state := test_spec
  transformations := $\emptyset$
  for i := 1 to breadth do
    for rule in rules do
      transformations := transformations $\cup$ rule(proof_state)
    end for
    sort(transformations)
    for transformation in transformations do
      proof_state := transformation(proof_state)
    end for
  end for
  insert_uniformity(proof_state)
  return proof_state
end procedure
```

Figure 3.1: High-level description of test derivation procedure

3.1 Overview of Test Theorem Derivation

We represent test derivation rules as functions from proof states to sets of transformations on proof states. Every such transformation is associated with a priority. To derive the test theorem, we apply the provided test derivation rules to the test specification, and then perform the transformations returned by these rules in order of their priority. We repeat this process for a number of times that is configurable by the user. At the end of the procedure, the uniformity hypothesis (2.3) is inserted for any remaining variables. Figure 3.1 shows this overall procedure as pseudo-code. The user-defined parameter `breadth` determines the number of times the rules are applied.

3.2 An Interface for Test Derivation Rules

The original procedure of HOL-TestGen for deriving a test theorem discriminates variables based on their type. For variables whose type is recursive datatype, a particular kind of case splitting is performed. Another element of the original procedure is the generation of a CNF by applying De Morgan's laws. This can be regarded as a kind of case splitting which is not based on variables, but rather on certain Boolean operations. Thus, a generic interface that is suitable for these forms of splitting cannot be solely based on types. Rather, a more universal notion of test derivation rule is necessary. Therefore we define a test derivation rule to be a general mapping from proof states represented as `Thms` in ML to lists of tactics:

```
  type test_derivation_rule
         = Proof.context -> int -> thm -> (int * tactic) list
```

The integer argument taken by a test derivation rule denotes a subgoal index that the rule should operate on. This is a common indexing convention used in Isabelle code for tactics that manipulate an individual subgoal of a proof state. Similarly, we intend test derivation rules to only consider the subgoal that is denoted by the integer argument. The third argument represents the proof state passed to the rule. This argument can be understood as counterpart to the Thm taken by tactics. In contrast to tactics, which must return a sequence of theorems, a test derivation rule is declared to return a list of pairs of integers and tactics. The tactics returned are operations that implement the rule if they are applied to the proof state. These tactics can constitute case splits, simplifications, or any other operations that serve the purpose of deriving a test theorem. Thus, our notion of test derivation is very general. The integers associated with the tactics returned denote priorities. The higher the priority, the sooner the tactic should be applied to proof state. These priorities have an important function. For example, it can make a difference whether a simplification is carried out before or after a case split.

Note that the tactics returned by a test derivation rule do not take a subgoal index as argument. Rather, they operate on an entire proof state. This is not a restriction, since a subgoal-specific tactic can be wrapped with the ALLGOALS tactical provided by the Isabelle library in order to obtain a tactic that operates on all subgoals of a proof state. The reason for this design decision is that tactics returned by a test derivation rule may be applied in an unforeseeable order to the proof state, depending on the priorities they are associated with. Tactics that are applied before some particular test derivation rule, such as those used for CNF generation, may split a subgoal into new ones. Thus, subgoal indices determined in advance become obsolete.

Many test derivation rules do not use all features of this interface. For example, we define a basic simplification rule that returns the same simplification tactic independently from the proof state as follows:

```
fun minimize_rule ctxt no state =
   [(10, distinct_subgoals_tac
         THEN (ALLGOALS(asm_full_simp_tac (simpset_of ctxt)))]
```

This rule returns a singleton list containing a tactic that first carries out distinct_subgoals_tac that merges identical subgoals and then applies the simplification tactic asm_full_simp_tac to all subgoals. The returned tactic is associated with the priority “10”. Since the tactic returned does not depend on the proof state, the argument state is ignored.

3.3 Tactical Test Theorem Derivation

We now describe the generic test generation procedure that can apply provided test derivation rules in order to yield a test theorem. We define some custom tacticals that are useful for this purpose. One such tactical is

```
ALLCASES: (int -> tactic) -> tactic,
```

as we decide to name it. This tactical applies the tactic passed to it as argument to all subgoals that have not been marked particularly by HOL-TestGen. These subgoals represent classes of test cases, and, in particular, have not been marked as test hypotheses. The other custom tactical that we will use is

```
SORT: Proof.context -> test_derivation_rule list
                                              -> int -> tactic
```

This tactical returns a tactic that applies the provided test derivation rules to the proof state, and then applies the tactics returned by the test derivation rules in order of the priority they have been associated with by the rules.

The tactical `SORT` can be implemented easily. First we apply the passed test derivation rules `rules` to the subgoal `n` of the proof state `thm` and collect the resulting pairs of tactics and priorities:

```
val matches = maps (fn rule => rule ctxt n thm) rules
```

The combinator `maps` provided by the Isabelle standard library applies a list-valued function to a list and merges the results to a single list. As next step, we obtain a sorted list of the tactics to apply by sorting the list `matches`:

```
val sorted = map (fn (i,tac) => tac)
   (sort (fn (x,y) =>
      rev_order (int_ord (fst x, fst y))) matches)
```

Here we use the `sort` function provided by the Isabelle standard library, parametrized with a suitable order on the list elements. We obtain this order by reversing the standard integer order on the priorities, so we achieve a sorted result with tactics of higher priority at the beginning of the list. After sorting, we trim off the priorities in order to construct a list consisting only of the tactics. Then we use the `EVERY` tactical for obtaining a composition of these tactics as `EVERY sorted`. The `EVERY` tactical provided by Isabelle applies the tactics in the list passed to it in the order they appear in the list. Finally, we enclose this sequential composition in the `SELECT_GOAL` tactical that is also provided by the Isabelle standard library. This tactical allows to apply a tactic that operates on an entire proof state to a single subgoal. Using `SELECT_GOAL` ensures that `SORT` only affects the subgoal that has been passed to it as argument. Other subgoals are not altered, at most their number changes if the modified subgoal is split into several ones. Altogether, the tactic returned by SORT amounts to

```
SELECT_GOAL (EVERY sorted) n
```

We are now ready to give a definition of the gen_test_cases tactic:

```
fun gen_test_case_tac ctxt rules breadth =
   ALLCASES((asm_full_simp_tac (simpset_of ctxt))
   THEN REPEAT_DETERM_N breadth (ALLCASES (SORT ctxt rules)))
   THEN ALLCASES(uniformityI_tac ctxt))
```

Thus, the gen_test_cases tactic is a sequential composition of three tactics. The first tactic performs a general-purpose simplification of the test specification. Then, using the SORT tactical, the provided test derivation rules are applied for a number of times that is specified by the breadth parameter. The value of this parameter is user-defined. For achieving this repetition, we use the tactical REPEAT_DETERM_N provided by the Isabelle standard library. The final step of gen_test_cases is to insert the uniformity hypothesis (2.3). This is performed by the auxiliary tactic uniformityI_tac.

We do not give the implementation of the tactic uniformityI_tac here because it is quite technical, but rather give a high-level description. The tactic uniformityI_tac assumes that the proof state is in conjunctive normal form. Thus, every subgoal is a disjunction, or equivalently, a series of implications $A_1 \Longrightarrow \ldots \Longrightarrow A_n$. The tactic uniformityI_tac permutes this series, if necessary, by applying the identity $A \Longrightarrow B \equiv \neg B \Longrightarrow \neg A$, such that all propositions referencing the program under test are moved to the right end of the series. At this point the subgoal is of the form

$$\mathrm{pre}\,(x_1, \ldots, x_m) \Longrightarrow \mathrm{post}\,(x_1, \ldots, x_m, PUT\,(x))\,, \tag{3.1}$$

where PUT is the program under test, $x_1, \ldots, x_m$ are the variables occurring in the subgoal, and $\mathrm{pre}\,(x_1, \ldots, x_m)$ does not contain any references to PUT. We then insert the subgoal

$$\begin{aligned}(\exists x_1, \ldots, x_m.\ &\mathrm{pre}\,(x_1, \ldots, x_m) \wedge \mathrm{post}\,(x_1, \ldots, x_m, PUT\,(x)))\\ &\Longrightarrow (\forall x_1, \ldots, x_m.\ \mathrm{pre}\,(x_1, \ldots, x_m) \Longrightarrow \mathrm{post}\,(x_1, \ldots, x_m, PUT\,(x)))\,,\end{aligned}$$

which represents a uniformity hypothesis. Adding this hypothesis allows us to replace the subgoal (3.1) by the two subgoals

$$\mathrm{pre}\,(x_1, \ldots, x_m) \tag{3.2}$$

and

$$\mathrm{post}\,(x_1, \ldots, x_m, PUT\,(x))\,, \tag{3.3}$$

in which the variables $x_1, \ldots, x_m$ occur as Isabelle metavariables, i.e., are implicitly universally quantified and can be instantiated. The subgoal (3.2) will be treated as constraint on the variables $x_1, \ldots, x_m$. The other resulting subgoal (3.3) represents a test case. Once the variables $x_1, \ldots, x_m$ are instantiated, the subgoal (3.3) is executable and evaluates to true if the program under test passes the test. The subgoal (3.2) should be eliminated by instantiating the variables $x_1, \ldots, x_m$ such that it evaluates to true. Such an instantiation operates simultaneously on both subgoals (3.2) and (3.3). It can be performed by the constraint solving tactics presented in Chapter 4.

In order to give an example for test derivation, we revisit the sorting specification presented in Section 2.3.3. Now, we additionally specify that the list returned by the implementation under test must be a permutation of the input list. We achieve this by specifying that the output of the implementation under test is equal to the output produced by an insertion sort. We define an insertion sort function in HOL by first recursively defining insertion into a sorted list:

```
fun   ins :: ”int ⇒ int list ⇒ int list”
where ”ins x [] = [x]”
   —”ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys)))”
```

Using this function, we can now recursively define insertion sort as follows:

```
fun   sort:: ”int list ⇒ int list”
where ”sort [] = [] ”
   — ”sort (x#xs) = ins x (sort xs)”
```

With the insertion sort function at our disposal, we can specify the behavior of the implementation under test simply by $PUT\,(l) = \mathrm{sort}\,(l)$. As in the first part of this example, PUT represents the implementation under test and l the input list to be sorted. Given this specification, the tactic `gen_test_cases` first applies a test derivation rule that carries out case splitting for variables of a recursive datatype. This rule applies the derivation rule (2.6) for the variable l, which transforms the specification into the following list of subgoals:

```
1. PUT [] = List_test.sort []
2. ⋀a. PUT [a] = List_test.sort [a]
3. ⋀a aa. PUT [a, aa] = List_test.sort [a, aa]
4. ⋀a aa ab list.
      length (a # aa # ab # list) ≤ 2 ⟹
      PUT (a # aa # ab # list)
      = List_test.sort (a # aa # ab # list)
5. ⋀a aa ab list.
      THYP (2 < length l ⟶ PUT l = List_test.sort l)
```

These subgoals represent, from top to bottom, the cases of an empty input list, a list of length one, a list of length two, a list of length larger than two and a corresponding regularity hypothesis. Simplifying this proof state inserts the definition of the `sort` function where this is possible. The fourth subgoal is eliminated by this simplification:

```
1. PUT [] = []
2. ⋀a. PUT [a] = [a]
3. ⋀a aa.
      (a < aa ⟶ PUT [a, aa] = [a, aa]) ∧
      (¬ a < aa ⟶ PUT [a, aa] = [aa, a])
4. THYP (2 < length l ⟶ PUT l = List_test.sort l)
```

Then `gen_test_cases` performs a transformation to conjunctive normal form, which yields the following proof state:

```
1. PUT [] = []
2. ⋀a. PUT [a] = [a]
3. ⋀a aa. a < aa ⟹ PUT [a, aa] = [a, aa]
4. ⋀a aa. ¬ a < aa ⟹ PUT [a, aa] = [aa, a]
5. THYP (2 < length l ⟶ PUT l = List_test.sort l)
```

In this proof state, every subgoal, except for the one representing the regularity hypothesis, constitutes a clause of the CNF. The clauses correspond to every possible permutation of a list with up to two elements. By using different parameters for case splitting, we could make this proof state enumerate the permutations of arbitrarily long lists. The final step of `gen_test_cases` is the application of uniformity hypotheses, which results in the following test theorem:

```
1. PUT [] = []
2. THYP (PUT [] = [] ⟶ PUT [] = [])
3. PUT [??X7X2] = [??X7X2]
4. THYP ((∃a. PUT [a] = [a]) ⟶ (∀a. PUT [a] = [a]))
5. PO (??X4X2 < ??X5X3)
6. PUT [??X4X2, ??X5X3] = [??X4X2, ??X5X3]
7. THYP
    ((∃a aa. a < aa ∧ PUT [a, aa] = [a, aa]) ⟶
     (∀a aa. a < aa ⟶ PUT [a, aa] = [a, aa]))
8. PO (¬ ??X1X2 < ??X2X3)
9. PUT [??X1X2, ??X2X3] = [??X2X3, ??X1X2]
10. THYP
     ((∃a aa. ¬ a < aa ∧ PUT [a, aa] = [aa, a]) ⟶
      (∀a aa. ¬ a < aa ⟶ PUT [a, aa] = [aa, a]))
11. THYP (2 < length l ⟶ PUT l = List_test.sort l)
```

This step generates a test case for every clause of the CNF. Moreover, constraints are added for test cases with nontrivial constraints. In this example, these are the test cases that consider lists of length two. The subgoals in the test theorem that represent constraints are marked with "PO" ("proof obligation"). These constraints can be tackled with the constraint solving tactics presented in Chapter 4.

3.4 How Useful is the Regularity Hypothesis?

Regularity hypotheses (2.4) are applicable when a function *size* on the set X of tests under consideration is available that expresses in some sense the structural complexity of its argument. A regularity hypothesis can be described as the assumption that the set X of possible tests is "regular" in the sense that if any test in X detects an error in the program p, then so will a test whose *size* is at most k. Thus, under the assumption that the regularity hypothesis is true, it is sufficient to test the program only on those inputs whose *size* is at most k. At first glance, a regularity hypothesis may appear to be a weaker hypothesis than a corresponding uniformity hypothesis that only leads to the generation of a single test case, because the regularity hypothesis requires tests that cover all inputs whose *size* is at most k. This, however, is not true in general, since the set X may not contain any tests at all whose *size* is at most k. In this case, the regularity hypothesis does not require any tests, while using a corresponding uniformity hypothesis can never eliminate the need for tests.

Thus, a regularity hypothesis is not a priori preferable to a corresponding uniformity hypothesis. Considering the interaction with case splitting for test generation gives another argument against using regularity hypotheses. To see this, note that assuming a regularity hypothesis (2.4) still requires the generation of test cases for checking

$$\forall x \in X.\ \mathit{size}\,(x) \leq k \Longrightarrow (x, p\,(x)) \in S. \tag{3.4}$$

If (3.4) holds, i.e., the implementation passes these test cases, and any hypotheses assumed for establishing (3.4) are true, then (2.4) is equivalent to

$$\forall x \in X.\ \mathit{size}\,(x) > k \Longrightarrow (x, p\,(x)) \in S. \tag{3.5}$$

Note that this is the interesting case, since the hypotheses do not matter when the implementation does not pass the tests.

Now, consider instead a case split of the test specification into

$$\forall x \in X.\ \mathit{size}\,(x) \leq k \Longrightarrow (x, p\,(x)) \in S \tag{3.6}$$

and

$$\forall x \in X.\ \mathit{size}\,(x) > k \Longrightarrow (x, p\,(x)) \in S. \tag{3.7}$$

The case (3.6) is identical to the formula (3.4), and the other case (3.7) is identical to the modified regularity hypothesis (3.5). When generating a test suite, the case

(3.6) can be handled in the same way as the identical case (3.4) when introducing the regularity hypothesis. Let us turn to the other case (3.7). By introducing the uniformity hypothesis

$$\begin{aligned}(\exists x_0 \in X.\ \mathit{size}\,(x_0) > k \wedge (x_0, p\,(x_0)) \in S)\\ \Longrightarrow (\forall x \in X.\ \mathit{size}\,(x) > k \Longrightarrow (x, p\,(x)) \in S)\,,\end{aligned} \tag{3.8}$$

we can reduce case (3.7) to a single test case $(x_0, p\,(x_0)) \in S$ for an x_0 that satisfies $\mathit{size}\,(x_0) > k$. Thus, we avoided assuming the regularity hypothesis (3.5) by assuming the uniformity hypothesis (3.8) and adding one additional test case. Note that the regularity hypothesis (3.5) implies (3.8), while the converse is not true. Hence, by performing the case split and avoiding the regularity hypothesis, we were able to weaken the hypotheses that we must assume.

So should we simply forget about the regularity hypothesis? A nice characteristic of the regularity hypothesis is that it allows us to avoid constraints of the form $\mathit{size}\,(x_0) > k$. When deducing test cases using regularity hypotheses and exhaustion lemmas like (2.6), we instead face constraints like $\mathit{size}\,(x_0) = k$. Such more restrictive constraints can allow useful simplifications. E.g., in the case of a list x_0 and $k = 2$, we can write $x_0 = [a, b]$, and $\mathit{sort}\,(x_0) = y$ can be rewritten by the Isabelle simplifier to $(a < b \Longrightarrow [a, b] = y) \wedge (\neg a < b \Longrightarrow [b, a] = y)$. In contrast, if the constraint were $\mathit{size}\,(x_0) > 2$ instead of $\mathit{size}\,(x_0) = 2$, we would have no means of simplifying the term $\mathit{sort}\,(x_0)$, and would have to deal with the presence of recursive functions during the constraint solving phase. However, the techniques for tackling recursive constraints presented in Section 4.4 allow us to cope with such a situation. Note that it is possible to substitute any disliked test case by a suitable test hypothesis if desired. Thus, this recourse should be used with care. In all, we conclude that skepticism towards the regularity hypothesis is warranted in order to achieve test suites that are as effective as possible.

Chapter 4

Solving Constraints in Isabelle

In this chapter we first discuss the intricacy of solving constraints in a theorem proving environment. Then we introduce different types of methods that we use for solving constraints in Isabelle. These include methods based on the random generation of candidate solutions, as well as methods based on SMT solving. We show how recursive constraints can be solved with SMT solvers, and describe an interactive approach to make constraints amenable to SMT solving. Finally, we present experimental results for these constraint solving techniques and review related work.

4.1 Constraint Solving versus Theorem Proving

Isabelle is a *theorem proving* tool. For the purposes of this thesis, testing and animation, we are faced however with the task of *constraint solving*, which at first sight appears to be considerably different from theorem proving. To prove a theorem typically means to show that some assertions hold for all possible values that one or more variables can assume. Moreover, we usually know which theorem we want to prove before we prove it. We may record the proof found for later reference, but often, e.g, when performing system verification, we are only interested in the certain knowledge that a theorem holds, independently of the type of proof carried out.

Generally, to solve a constraint means to determine a set of values $X_1, \ldots, X_n$ such that $P\,(X_1, \ldots, X_n)$ holds for some predicate P. If we have found a concrete solution $X_1 = x_1, \ldots, X_n = x_n$, we have actually proved the theorem $P\,(x_1, \ldots, x_n)$. In contrast to the typical theorem proving scenario, we did not know which theorem we would prove before we found the solution, i.e., proved the theorem. Another possible point of view is that we proved the more general theorem $\exists X_1, \ldots, X_n.\ P\,(X_1, \ldots, X_n)$ by solving the constraint. Such a theorem stating that some solution exists, however, is in itself often not of much use. For applications like testing and animation, the proof of this theorem that conveys which values solve the constraint are of crucial importance.

Thus, the question arises whether Isabelle as a theorem proving environment can meet the requirements that are necessary for dealing with constraint solving tasks. Our approach exploits the support of Isabelle for *metavariables*. Metavariables are variables in a theorem (which is represented internally in Isabelle as an ML value of type `Thm`) that can be instantiated if desired, resulting in a new theorem. We represent a constraint by a subgoal $P\,(X_1,\ldots,X_n)$ with metavariables $X_1,\ldots,X_n$. The metavariables are the unknowns that need to be determined such that the constraint is satisfied. Instantiating the metavariables with values that satisfy the constraint allows the subgoal to be simplified to *True* and thus to be eliminated. This corresponds to the usual way of proving theorems in Isabelle by successively eliminating subgoals. Since further occurrences of the metavariables outside of the eliminated subgoal are instantiated at the same time, the solution to the constraint is not necessarily lost when the subgoal is eliminated. The simplest way to obtain such a subgoal $P\,(X_1,\ldots,X_n)$ is to create the trivial theorem $P\,(X_1,\ldots,X_n) \Longrightarrow P\,(X_1,\ldots,X_n)$. Eliminating the subgoal would then yield the theorem $P\,(x_1,\ldots,x_n)$ in which all metavariables have been replaced by concrete values $x_1,\ldots,x_n$. This theorem expresses that the constraint has been solved by $x_1,\ldots,x_n$. For a suitable predicate P this instantiation can be recovered from the term structure of $P\,(x_1,\ldots,x_n)$.

We illustrate this approach with a toy example. Our constraint just asks for natural numbers *x* and *y* such that *x* is smaller than *y*.

```
schematic_lemma  "(?x::nat) < (?y::nat)"
```

In this statement of the lemma, the variables *x* and *y* are metavariables. We obtain a single subgoal representing the constraint:

```
  1. ?x < ?y
```

We guess the solution *x = 3*, *y = 7* and represent this instantiation as a list on the ML level as required by Isabelle's low level instantiation tactic that we will use.

```
ML_prf {*
val insts = [(("x",0),"(3::nat)"), (("y",0),"(7::nat)")]
*}

apply (tactic "RuleInsts.instantiate_tac @{context} insts")
```

Performing the instantiation yields the expected subgoal:

```
  1. 3 < 7
```

```
apply (simp)
done
```

Simplification eliminates the subgoal and results in the desired theorem expressing a solution to the constraint: *3 < 7*. The instantiation used can be recovered by matching this theorem to the constraint *x < y*.

4.2 Random Constraint Solving Techniques

Random constraint solving proceeds by first generating a candidate solution randomly and then determining whether it is a valid solution to the constraint. The random generation is repeated until a valid solution is found or a time limit is reached. This approach, which is also referred to as the "generate-and-test" technique, is a brute-force approach since it does not employ advanced reasoning like constraint propagation and learning to find a solution quickly. Also, the unsatisfiability of a constraint cannot be shown with this technique. An advantage of the random constraint solving approach is its simplicity – it is applicable whenever means for the random generation of candidate solutions and the detection of valid solutions is available. For many kinds of data these building blocks have already been made available for other applications, while a tailored constraint solving algorithm would require substantial additional effort.

When used for the generation of test data, the random approach has further advantages. Sometimes random test data is preferred since random data often appears to be complex and therefore challenging to an implementation under test. In contrast, "intelligent" constraint solving techniques are likely to yield solutions that exhibit unnatural structure, such as all-zero solutions or other border cases. Moreover, for some applications it is believed that the actual input data is itself random in some sense. Finally, by constructing a sufficiently large test suite from test data generated uniformly at random, it can be argued that an appropriate test coverage has been achieved, provided that the real inputs to the system under test exhibit the same distribution. A problem with randomly generated test data is that some parts of the system under test are likely to never be exercised by random inputs. However, such parts may be critical for common uses of the system. When testing a compiler on random data, for example, it is to be expected that a syntax error is detected for the overwhelming majority of inputs. On the other hand, it is very unlikely that valid source code of a program with input and output is generated.

The random approach to test data generation has been investigated extensively for testing functional programs [49, 95, 47]. Inspired by the *QuickCheck* tool [49] for randomly testing Haskell programs, Berghofer and Nipkow [12] implemented Isabelle's `quickcheck` command for detecting non-theorems. Adapting the approach of its predecessor for Haskell, the `quickcheck` command generates random values for free values in a conjecture and then tests the validity of the conjecture for the random assignment. In order to perform repeated validity tests efficiently, `quickcheck` uses Isabelle's code generator to obtain a validity test as compiled code. Thus, `quickcheck` can benefit from any available compiler optimizations. The drawback is that only "executable" conjectures amenable to code generation can be analyzed this way.

Brucker and Wolff have employed a different form of random constraint solving for test data generation in the HOL-TestGen tool [28]. In their approach, candidate solutions are generated randomly as does the `quickcheck` command, but instead of generated compiled code, the Isabelle simplifier is used for testing the validity of the solutions. While this approach is potentially less efficient when checking a large number of candidate solutions, it can benefit from more powerful reasoning capabilities provided by the Isabelle simplifier. When testing for valid solutions during random constraint solving, the Isabelle simplifier is used as a general-purpose reasoner that is more powerful than a plain evaluation engine. For example, the Isabelle simplifier can eliminate quantifiers in some situations, such as in the formula $\forall x \in \mathbb{Z}.\ x = 1 \Longrightarrow P\,(x)$, which it simplifies to $P\,(1)$. Note that quantified formulas like this one are likely to make an SMT solver fail since SMT solvers often do not apply many heuristics for dealing with quantifiers. Thus, this variant of random constraint solving can be beneficial even if more sophisticated constraint solving techniques such as SMT solving are available.

We have packaged both the quickcheck-based random solving technique as well as the alternative technique using the Isabelle simplifier in a tactic that allow a constraint to be solved by a single call to the tactic. An invocation of the quickcheck-based tactic `quickcheck_tac` from Isabelle's structured proof language Isar, using a maximum of 50 solving attempts, looks like this:

```
schematic_lemma  "(?x::nat) < (?y::nat)"
by (tactic "quickcheck_tac @{context} 50 1")
```

For applying the other tactic `random_solve_tac` that is based on the Isabelle simplifier, it is enough to change the name of the tactic:

```
schematic_lemma  "(?x::nat) < (?y::nat)"
by (tactic "random_solve_tac @{context} 50 1")
```

These tactics yield theorems that express a solution to the constraint like the theorem `0 < 2`.

It is interesting to note that the tactic `random_solve_tac` can itself be defined by tactical code. As basic building block, we assume the availability of a tactic `random_inst_tac: Proof.context -> int -> tactic` that instantiates all metavariables in the indicated subgoal of the theorem passed with suitably generated random constant terms. We assume that this tactic returns a sequence consisting of a single theorem. Thus, in order to generate several random instantiations, we have to call `random_inst_tac` repeatedly.

We introduce the tactic

```
solve_by_simp_tac: Proof.context -> int -> tactic
```

as another auxiliary tactic. This tactic returns a theorem if the subgoal passed as the second argument can be eliminated by the simplifier. This tactic can be defined as follows:

```
fun solve_by_simp_tac ctxt
  = SOLVED' (full_simp_tac (simpset_of ctxt))
```

Here `SOLVED': (int -> tactic) -> int -> tactic` is a tactical from Isabelle's library that only returns theorems for which the corresponding subgoal has been eliminated. The tactic `full_simp_tac` invokes the Isabelle simplifier. With the help of `solve_by_simp_tac`, we can define the tactic `single_tac: Proof.context -> int -> tactic` that makes an attempt to solve the constraint using a single random instantiation:

```
val single_tac ctxt =
   (random_inst_tac ctxt)
       THEN' (solve_by_simp_tac ctxt)
```

This tactic simply generates a random instantiation with `single_rand_inst_tac` and then checks for a valid solution with `solve_by_simp_tac`. The desired tactic

```
random_solve_tac: Proof.context -> int -> int -> tactic
```

can then be defined as a repeated application of `single_tac`:

```
fun random_solve_tac ctxt iterations n thm
   = (FIRST (replicate iterations (single_tac ctxt n))) thm
```

The function `replicate` belongs to the standard library and constructs a list containing `single_tac` `iterations` number of times. The tactical `FIRST` is also from the library and selects the first invocation of `single_tac` from the list that succeeds, i.e., solves the constraint. If no call to `single_tac` succeeds, `FIRST` returns an empty theorem sequence indicating failure.

4.3 An SMT Interface Exploiting Counterexamples

We now present an approach to constraint solving in Isabelle that is based on exploiting counterexamples returned by SMT solvers. We build on the interface that has been implemented between Isabelle and the SMT solver Z3 [17]. This interface provides the `smt` tactic to Isabelle for using Z3 to prove theorems. Z3 [58] is a state-of-the art SMT solver that follows the lazy SMT paradigm and includes theory solvers for real and integer linear arithmetic, bitvectors, recursive datatypes, function symbols and arrays. Z3 supports quantifier reasoning based on user-defined triggers and E-matching.

The `smt` tactic encodes a conjecture in HOL into a constraint that can be processed by the solver and whose unsatisfiability implies the truth of the conjecture. In the case that Z3 determines that the constraint is unsatisfiable, Z3 also returns a description of a proof for its unsatisfiability. The `smt` tactic then reconstructs this proof in order to obtain a proof of the conjecture within Isabelle.

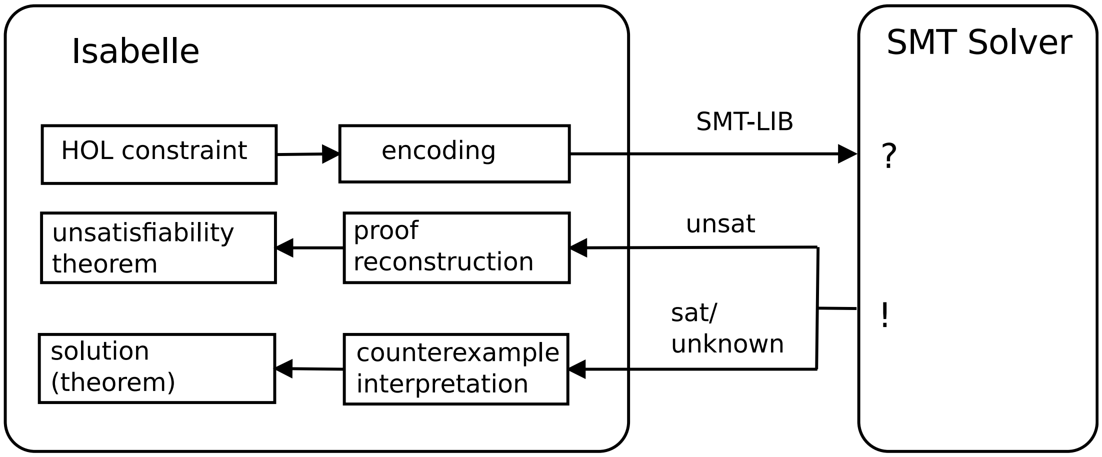


Figure 4.1: Architecture of the extended interface between Isabelle and the Z3 solver

The tactic also supports a mode of operation in which theorems are created with Isabelle's oracle mechanism in order to allow for faster execution.

SMT solvers usually can provide *counterexamples* when they fail to show that a constraint is unsatisfiable. A counterexample is an assignment of concrete functions to the uninterpreted functions occurring in the constraint which satisfies the constraint. The `smt` tactic presents such counterexamples to the user as an aid to detecting non-theorems and debugging proofs. Thus, this feature of the tactic is similar to counterexample generators like Quickcheck [12] and Nitpick [16]. We will use counterexamples provided by SMT solvers also for proving theorems in order to derive further benefit from them. The resulting extended architecture of the interface between Isabelle and Z3 is depicted in Figure 4.1.

Our approach for exploiting counterexamples works as follows. In order to prove and eliminate a subgoal A_k of a proof state

$$A_1 \Longrightarrow \ldots \Longrightarrow A_k \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C, \tag{4.1}$$

we pass the negated subgoal $A_k \Longrightarrow False$ to the existing `smt` tactic of Isabelle in the hope of obtaining a counterexample. Any such counterexample must assign values to the metavariables of A_k that make A_k evaluate to $True$. From the counterexample we extract an instantiation of these metavariables with HOL terms and instantiate the proof state (4.1) correspondingly. Finally we apply Isabelle's simplifying tactic to the subgoal and hope that this results in its elimination. In practice this may not be the case because the correctness of the instantiation may rely on parts of the background theory that were not passed to the SMT solver.

Note that this technique to exploit counterexamples is based on the instantiation of metavariables, so it fits nicely with our Isabelle-compatible constraint solving approach described above. Moreover, the derivation of a proof from a counterexample in Isabelle constitutes a trusted way of checking the correctness of a counterexample, analogous to proof reconstruction techniques implemented in Isabelle [128, 152, 17]. Although it is arguably easier to check a counterexample than a proof since counterexample checking essentially amounts to term evaluation and does not require, e.g., resolution steps, using the established Isabelle kernel for this purpose can provide a particular level of confidence in the correctness of the counterexample. We have packaged our counterexample-based proof technique in a tactic called `smt_solve_tac`. This tactic can be invoked by the user in the same way as the other constraint solving tactics `quickcheck_tac` and `random_solve_tac` introduced above.

4.3.1 Interpreting Counterexamples

The extraction of an instantiation from a counterexample still presents some difficulty because counterexamples often contain references to fresh values of types that the solver regards as uninterpreted. The counterexamples are based on the assumption that these values are distinct. It is problematical to map these values in the counterexample to HOL terms suitable for the instantiation of metavariables. Moreover, the constraint passed to the SMT solver typically contains references to functions that the solver regards as uninterpreted, but that have a concrete semantic interpretation in HOL. The semantic interpretation of these functions further constrains the HOL terms that can denote values occurring in a counterexample.

Recall our example $x < y$ from above. The SMT interface of Isabelle 2011 generates the following textual description in so-called SMT-LIB format for processing by Z3.

```
(benchmark Isabelle
:status unknown
:logic AUFLIA
:extrasorts ( S1 S2)
:extrafuns (
  (f1 S1)
  (f2 S1)
  (f3 S2 Int)
  (f4 S2)
  (f5 S2)
  (f6 Int S2)
)
:assumption (not (= f1 f2))
:assumption (< (f3 f4) (f3 f5))
:assumption (not false)
:assumption (forall (?v0 S2) (= (f6 (f3 ?v0)) ?v0))
:assumption (forall (?v0 Int) (implies (<= 0 ?v0) (= (f3 (f6 ?v0)) ?v0)))
:assumption (forall (?v0 Int) (implies (< ?v0 0) (= (f3 (f6 ?v0)) 0)))
:formula true)
```

Since natural numbers are not a builtin theory of Z3, the additional sort `S2` is declared for denoting natural numbers. The function `f3` represents the HOL function *int* that embeds the natural numbers in the integers. The function `f6` represents its inverse *nat*. These are examples of functions that have a concrete semantic interpretation in HOL, but are declared as uninterpreted in the language processed by the SMT solver. The function `f4` represents the value of the variable *x* and `f5` the value of the variable *y*. The remaining functions and sorts in the constraint have been introduced for internal purposes of the SMT interface.

Given this input, Z3 produces the following textual description of the counterexample it found.

```
f1 -> val!0
f2 -> val!1
f4 -> val!2
f5 -> val!3
f3 ->
  val!2 -> 0
  val!3 -> 1
  else -> 1

f6 ->
  0 -> val!2
  1 -> val!3
  else -> val!3

unknown
```

Here, the verdict "unknown" at the end indicates that the solver was not able to establish the correctness of the counterexample due to the presence of quantifiers. In the counterexample, symbols starting with `val!` represent fresh values that are identified by the number at the end of the symbol. The counterexample above assigns to the variables *x* and *y* represented by `f4` and `f5` the values `val!2` and `val!3`. These values are intended to represent distinct natural numbers. We cannot create HOL terms for instantiating *x* and *y* by choosing an arbitrary pair of distinct natural numbers because `val!2` and `val!3` are also referenced by other parts of the counterexample. In particular, the counterexample states that the function `f6` representing the HOL function *nat* maps 0 to `val!2` and 1 to `val!3`. Note that we can associate concrete HOL terms with `f6(0)` and `f6(1)`, namely *nat 0* and *nat 1*. This observation allows us to derive that `val!2` and `val!3` can be denoted by *nat 0* and *nat 1*, respectively.

We generalize this approach for extracting instantiations from counterexamples as follows. We represent every fresh value in the counterexample by a free variable in HOL. This allows us to express the counterexample as a list of HOL equations. The left-hand sides of the equations represent function applications in the counterexample, which may correspond to arbitrary terms in HOL, including metavariables. The right hand sides represent the values assigned to the function applications and may correspond to numeric constants or free variables representing fresh values in HOL.

We now manipulate the list of equations by substitutions in order to obtain assignments to metavariables without free variables. For every equation of the form $e = a$, where a is a free variable and e is an expression without free variables, we delete the equation and substitute all occurrences of a in the remaining equations, on the left hand as well as the right hand sides, by e. This eliminates all references to the fresh value represented by a from the equations. We perform these kinds of substitutions as long as possible. This results in a set of equations with the property that for every equation $e = a$ for a free variable a, the expression e also contains free variables. There is no hope to eliminate these remaining equations, since definitions of any free variables in e will depend on further free variables, and so on. Thus, we have done the best possible to obtain instantiations of metavariables by suitable HOL terms.

Note that it was essential for our substitution approach that we limited ourselves to substitutions that do not contain other free variables, since there may be several equations defining the same free variable. However, not all definitions may be fruitful in the end, e.g., some definitions could refer to free variables that are not defined anywhere. By restricting ourselves to substitutions that do not contain other free variables, we ensure that only definitions are applied that are guaranteed to be useful.

The counterexample presented above is represented as follows by HOL equations.

```
x = a
y = b
int a = 0
int b = 1
nat 0 = a
nat 1 = b
```

Here we have discarded equations involving functions symbols used for internal purposes of the SMT interface. When applying our substitution method, we observe that `nat 0 = a` and `nat 1 = b` are definitions of `a` and `b` that do not contain other free variables. Performing the corresponding substitutions and deleting these definitions yields the following set of equations.

```
x = nat 0
y = nat 1
int (nat 0) = 0
int (nat 1) = 1
```

Our transformation of the counterexample is now finished, since there are no more free variables left that represent fresh values. As desired, we have obtained the instantiations `x = nat 0` and `y = nat 1` of the metavariables `x` and `y`.

4.4 Solving Recursive Constraints

Recursive functions are indispensable in Isabelle/HOL. Comprehensive support for recursive functions makes HOL resemble a functional programming language. Thus, when generating tests from HOL specifications, we need to be prepared to solve constraints involving recursive functions and predicates. In this section we first give an introduction to recursive definitions in HOL. Then we discuss the difficulties that arise when dealing with recursive constraints and present an approach for solving a large class of recursive constraints with SMT solvers.

4.4.1 Recursion in HOL

A recursive function is defined in HOL by stating several equations with the function on the left-hand side. A classical example of a function that can be defined recursively is the factorial function, which can be defined as follows in HOL. The factorial function is first declared as a function that maps natural numbers to natural numbers and then defined uniquely by a set of equations.

```
fun fac::"nat ⇒ nat"

where

"fac (Suc x) = (x+1) * (fac x)" |
"fac 0 = 1"
```

Here *Suc* denotes the successor function of the natural numbers. The package for recursive function definitions included in Isabelle/HOL turns such a recursive definition into a conservative definition of the function, i.e., a definition that does not risk to introduce inconsistencies.

There may be more than two equations in a recursive definition, as for example in the following definition taken from [120].

```
fun sep::"'a ⇒ 'a list ⇒ 'a list"

where

"sep a [] = []" |
"sep a [x] = [x]" |
"sep a (x # xs) = x # a # sep a (xs)"
```

This function *sep* inserts the element *a* between the elements in a list. In this definition of *sep*, the *patterns* on the left-hand sides of the second and third equations overlap since they both match lists of length one. When interpreting a recursive function definition, Isabelle/HOL matches the patterns on the left-hand sides from top to bottom. Thus, for determining the value of the *sep* function, lists of length one are preserved and the last equation only applies to lists of length at least two. Internally Isabelle/HOL creates equations from such a definition whose patterns do not overlap. See [100] for a description of the internals of Isabelle/HOL's recursive function package.

Another feature of recursive definitions in HOL is support for explicit conditions that guide the application of equations in addition to pattern matching. For simplicity we will not separately consider this advanced feature in this section.

In recursive definitions, the function being defined may not occur in the patterns on the left-hand sides. Otherwise, the set of patterns that may appear on the left-hand sides is basically not restricted. However, it is essential that the HOL package for recursive definitions can establish termination of the definition. Termination means that for all possible function arguments, the equations in the recursive definitions need only be applied a finite number of times until a base case is reached in which the function being defined does not occur on the right-hand side.

A common tool for proving the termination of recursive definitions are *measure functions*. Consider a recursive definition of a function $f : A \to B$. The existence of a suitable measure function $\mu : A \to \mathbb{N}$ guarantees that the recursive definition terminates. A measure function μ is suitable if $\mu(l) > \mu(r)$ for every pair l,r of function arguments that can appear simultaneously on the left and right-hand side of an equation in the recursive definition. Thus, the value of the measure function for the function arguments reduces by at least 1 every time we apply an equation in the recursive definition from left to right. Since the value of the measure function cannot become negative, we are certain to reach a base case after a finite number of steps. The HOL package for recursive function definitions attempts to prove termination automatically by performing a systematic search for a suitable measure function. The starting point are "size" functions for recursive datatypes. Basically these functions count the number of datatype constructor applications that are necessary to obtain the respective value. Thus, the size of a list is defined to be the list's length, and the size of a natural number is the number itself. These size functions are usually good candidates for measure functions because most recursive function definitions peel off datatype constructors and use recursive calls on the arguments to the constructor. HOL's datatype package searches for a suitable composition of size functions for the arguments of a recursive function in order to find a measure function that works for the entire recursive definition.

4.4.2 Tackling Recursive Constraints

SMT solvers cannot deal with recursive functions out-of-the-box. The reason is that a recursive definition is a universally quantified formula, and the kind of trigger-driven quantifier instantiation used by SMT solvers is incomplete. For most decidable theories supported by SMT solvers, there is no hope for a complete decision procedure for recursive constraints over these theories. For example, a solution to an instance of the Post correspondence problem can be expressed using the append function on lists. The append function can easily be defined recursively using only language from the theory of recursive datatypes which some SMT solvers support as a subtheory. Thus, the satisfiability of recursive constraints over lists can be shown undecidable by a reduction to post's correspondence problem. A similar argument applies to recursive constraints over integers.

We can however define an instrumentation of recursive definitions that brings us closer to our goal of finding solutions to recursive constraints. In the sequel we define a *recursive definition* to be a set of formulas of the form

$$\begin{aligned}
\forall u. \quad & f_1\left(p_{1,1}\left(u\right)\right) &&= r_{1,1}\left(u\right) \\
& && \vdots \\
\forall u. \quad & f_1\left(p_{1,n_1}\left(u\right)\right) &&= r_{1,n_1}\left(u\right) \\
& && \vdots \\
\forall u. \quad & f_m\left(p_{m,1}\left(u\right)\right) &&= r_{m,1}\left(u\right) \\
& && \vdots \\
\forall u. \quad & f_m\left(p_{m,n_m}\left(u\right)\right) &&= r_{m,n_m}\left(u\right)
\end{aligned} \tag{4.2}$$

that uniquely define the referenced functions f_i, i.e., have a unique model, and whose left-hand sides are complete, i.e., the formula

$$\forall x. \quad \left(\left(\exists u.\ x = p_{i,1}\left(u\right)\right) \vee \ldots \vee \left(\exists u.\ x = p_{i,n_i}\left(u\right)\right)\right) \tag{4.3}$$

is a tautology for all $i = 1, \ldots, m$. The terms $p_{1,1}, \ldots, p_{m,n_m}$ represent patterns, and $r_{1,1}, \ldots, r_{m,n_m}$ are the right-hand sides of the definition. The condition (4.3) expresses that every argument of a recursive function matches one of the function's patterns on the left-hand side. This class of recursive definitions corresponds to the representation of recursive definitions that Isabelle/HOL uses internally, except that we also allow overlapping patterns. Allowing recursive definitions to refer to several functions allows mutual recursion as well as the definition of functions that are based on another recursive function. Since the quantified variable u and the patterns $p_{i,j}$ in the equations (4.2) may range over Cartesian products, it is possible to restrict ourselves to using only one quantified variable in recursive definitions without loss of generality.

Proposition 4.1. *Let a set of formulas Γ consist of a set of terminating recursive definitions and a set of quantifier-free formulas Φ. Let Γ' be the set of formulas obtained from Γ by adding the completeness assertion (4.3) for every recursively defined function f_i instrumented with a trigger as follows:*

$$\forall x \quad \left\{f_i\left(x\right)\right\}.\ \left(\left(\exists u.\ x = p_{i,1}\left(u\right)\right) \vee \ldots \vee \left(\exists u.\ x = p_{i,n_i}\left(u\right)\right)\right) \tag{4.4}$$

Moreover, we instrument every recursive definition in Γ' as follows:

$$\begin{aligned}
\forall u \quad & \{f_1\left(p_{1,1}\left(u\right)\right)\}. \quad f_1\left(p_{1,1}\left(u\right)\right) && = r_{1,1}\left(u\right) \\
& && \vdots \\
\forall u \quad & \{f_1\left(p_{1,n_1}\left(u\right)\right)\}. \quad f_1\left(p_{1,n_1}\left(u\right)\right) && = r_{1,n_1}\left(u\right) \\
& && \vdots \\
\forall u \quad & \{f_m\left(p_{m,1}\left(u\right)\right)\}. \quad f_m\left(p_{m,1}\left(u\right)\right) && = r_{m,1}\left(u\right) \\
& && \vdots \\
\forall u \quad & \{f_m\left(p_{m,n_m}\left(u\right)\right)\}. \quad f_m\left(p_{m,n_m}\left(u\right)\right) && = r_{m,n_m}\left(u\right)
\end{aligned} \tag{4.5}$$

Then the conjunction of the formulas in Γ' is logically equivalent to the conjunction of the formulas in Γ, and if an SMT solver employing the algorithm of Figure 2.2 terminates when called on the conjunction of formulas in Γ', then the result returned by the solver corresponds to the satisfiability of the conjunction of the formulas in Γ.

We cannot expect the algorithm of Figure 2.2 to terminate because of potential matching loops. But the proposition above tells us that if the algorithm does terminate, then in this special case the result returned by the algorithm is guaranteed to be correct, although this is not the case in general because trigger-based quantifier instantiation is incomplete.

Proof of Proposition 4.1. The conjunction of the formulas in Γ' is logically equivalent to the conjunction of the formulas in Γ because the formulas (4.3) are tautologies. If the algorithm of Figure 2.2 returns a non-null result to indicate that the problem is unsatisfiable, then the problem is indeed unsatisfiable.

It remains to show that the satisfiability problem is satisfiable if the algorithm of Figure 2.2 returns null. In this case a potential model M has been constructed that satisfies the quantifier-free formulas Φ as well as the instantiations of the recursive definitions and completeness assertions in Γ' that have been triggered. By construction the recursive definitions are satisfied by the functions they uniquely define. Let $R\left(f_i\right)$ denote the function the recursive definition specifies for the function symbol f_i. If we can show that for every function symbol f_i representing a recursive function, its interpretation $M\left(f_i\right)$ by the model M can be extended to the function $R\left(f_i\right)$, we have shown that an extended model M' based on M exists that satisfies Γ' and therefore Γ.

It is enough to show that for every assignment

$$M\left(f_i\right)\left(a\right) = b \tag{4.6}$$

prescribed by M, we have $M\left(f_i\right)\left(a\right) = R\left(f_i\right)\left(a\right)$. Suppose that for some a, we have $M\left(f_i\right)\left(a\right) \neq R\left(f_i\right)\left(a\right)$. Since M prescribes the assignment (4.6), there must be some ground term $f_i\,(t)$ such that $M\,(t) = a$. Otherwise, this assignment would be irrelevant. The ground term $f_i\,(t)$ matches the trigger attached to the completeness assertion (4.4) for f_i. Thus, this completeness assertion was instantiated with t. It follows that M satisfies $t = p_{i,j}\,(u)$ for some j and u, and this equality was included in the E-graph. As a result, the ground term $f_i\,(t)$ matches the trigger of the equation

$$\forall u \quad \{f_i\,(p_{i,j}\,(u))\}\,. \quad f_i\,(p_{i,j}\,(u)) = r_{i,j}\,(u) \tag{4.7}$$

of the recursive definition of f_i modulo E. Thus, this equation is instantiated with u. Since the pattern $p_{i,j}$ does not contain the function symbol f_i[1], we have $R\,(p_{i,j}\,(u)) = M\,(p_{i,j}\,(u)) = M\,(t) = a$. Hence, because both M and R satisfy (4.7), it follows that $M\,(f_i)\,(a) = M\,(r_{i,j}\,(u))$ and $R\,(f_i)\,(a) = R\,(r_{i,j}\,(u))$. Together with $M\,(f_i)\,(a) \neq R\,(f_i)\,(a)$, we obtain $M\,(r_{i,j}\,(u)) \neq R\,(r_{i,j}\,(u))$. Thus, the right-hand side $r_{i,j}\,(u)$ must contain an occurrence of f_i[1], leading to a new inequality $M\,(f_i)\,(a') \neq R\,(f_i)\,(a')$. However, since the recursive definition of f_i terminates, we reach a base case after a finite number of steps. This base case gives an inequality $M\,(r_{i,k}\,(u')) \neq R\,(r_{i,k}\,(u'))$ for a right-hand side $r_{i,k}$ that does not contain f_i – contradiction.

□

Note that the addition of the completeness assertions (4.3) to the constraint was essential in this proof, although as tautologies they have no effect on the satisfiability of the constraint. Moreover, we exploited the capability of SMT solvers to match triggers modulo the set E of asserted equalities. Plain syntactic matching would not have been sufficient.

4.4.3 Enforcing Termination by Under-Approximation

Proposition 4.1 brings us closer to solving recursive constraints. However, this proposition only applies to cases in which the solver terminates. Passing a constraint that is instrumented according to Proposition 4.1 to an SMT solver employing Figure 2.2 may lead to a nonterminating call to the solver.

The reason for this potential nontermination are matching loops that occur naturally in the constraints (4.5). The function symbols f_i may occur on the right-hand sides in (4.5), and these occurrences can match the triggers in (4.5). Even if these occurrences do not match the triggers in (4.5) syntactically, it is essential for recursion that they can match the triggers modulo equalities from (4.4).

This potential nontermination cannot be avoided without sacrifices because the satisfiability of recursive constraints over theories of interest is undecidable.

[1] If it contains an occurrence of another function symbol f_k, $k < i$, that the recursive definition of f_i depends on and $M\,(f_k)$ cannot be extended to $R\,(f_k)$, we can start over with a new inequality $M\,(f_k)\,(a_k) \neq R\,(f_k)\,(a_k)$, and so on.

Our approach will be based on under-approximating the constraint in order to eliminate non-terminating matching loops. We will restrict solutions of the constraint to ones in which the measures of the arguments to recursive functions do not exceed a certain bound. This will allow us to derive that the measures of any instantiated arguments to recursive functions are bounded as well. As a result, the number of quantifier instantiations can be bounded accordingly.

A tool we will use to achieve a terminating instrumentation are “transformed” definitions of recursive functions. For a function f_i defined recursively as in (4.2), we define the *transformed* function f'_i by

$$\begin{aligned} \forall u, k.\quad f'_i\left(p_{i,1}\left(u\right), k+1\right) &= r'_{i,1}\left(u, k\right) \\ &\vdots \\ \forall u, k.\quad f'_i\left(p_{i,n_i}\left(u\right), k+1\right) &= r'_{i,n_i}\left(u, k\right) \end{aligned} \tag{4.8}$$

where the transformed right-hand sides $r'_{i,j}$ are simply the original right-hand sides with occurrences of any function f from the same recursive definition substituted by f' with the additional argument k. Furthermore, we define

$$\forall u.\quad f'_i\left(p_{i,j}\left(u\right), 0\right) = r_{i,j}\left(u\right) \tag{4.9}$$

if $r_{i,j}$ is a base case, i.e., does not contain any occurrences of f_i, and

$$\forall u.\quad f'_i\left(p_{i,j}\left(u\right), 0\right) = d \tag{4.10}$$

otherwise, for an arbitrary constant d. It is easy to see that μ'_i defined by $\mu'_i\left(x, k\right) = k$ is a measure function for a transformed function f'_i. Thus, the number of steps required to evaluate f'_i can be bounded by the new second argument k. Moreover, if μ_i is a measure function for the original function f_i, then we have $f'_i\left(x, k\right) = f_i\left(x\right)$ if $k \geq \mu_i\left(x\right)$. Here we assume that μ_i also counts the steps used for evaluating any functions f that are defined in the same recursive definition as f_i and that f_i depends on. In other words, regarding the suitability of a measure function, we may treat f_i and the functions f_i depends on as mutually recursive, even if f_i is not actually mutually recursive. As a result, the measure functions used by Isabelle for proving termination may not always be adequate for our purposes.

In order to guarantee termination, we will need to adapt the triggers in Proposition 4.1. Otherwise, we cannot be sure that only intended quantifier instantiations occur, since matching is performed modulo the set of equalities in asserted ground formulas, and the user-provided constraints may contain arbitrary equalities. For example, the term $f'\left(x, k\right)$ can match the trigger $\{f'\left(x, y+1\right)\}$ modulo the user-specified equality $k = y+1$. In order to be able to trace all instantiations of recursive definitions to specific occurrences of recursive functions, we introduce an auxiliary function symbol Pred_i for every recursive function f_i and use this function symbol in triggers.

These considerations lead to the following proposition.

Proposition 4.2. *Let a set of formulas Γ consist of a set of recursive definitions and a set of quantifier-free formulas Φ. Let Γ' be the set of formulas obtained from Γ by transforming the recursive definitions according to (4.8), (4.9) and (4.10), and instrumenting the transformed definitions with triggers as follows:*

$$\forall u, v \quad \{f_i' (p_{i,j} (u), \mathrm{Pred}_i (v) + 1)\}. \qquad f_i' (p_{i,j} (u), \mathrm{Pred}_i (v) + 1) = r_{i,j}' (u, \mathrm{Pred}_i (v)) \tag{4.11}$$

For the definitions (4.9) and (4.10) that specify the value of a transformed function for a second argument of zero, we can avoid using Pred_i *in the trigger.*

We also add to Γ' the completeness assertions (4.4) instrumented as in Proposition 4.1 and the following analogous assertions for every auxiliary function symbol Pred_i*:*

$$\forall k \quad \{f_i' (x, k)\}. \quad k = 0 \lor k > 0 \land k = \mathrm{Pred}_i (k) + 1 \tag{4.12}$$

Moreover, for every occurrence of a recursive function $f_i (t)$ in Φ, let Γ' contain the equation

$$f_i (t) = f_i' (t, l), \tag{4.13}$$

where l is some arbitrary fixed constant.

Let Ψ be a set of quantifier-free formulas that contains for every argument t used in an application of a recursive function f_i in Φ a constraint $\mu_i (t) \leq l$, where μ_i is a measure function for f_i.

Then, if the conjunction of the formulas in $\Gamma \cup \Psi$ is satisfiable, then the conjunction of the formulas in $\Gamma' \cup \Psi$ is satisfiable. Furthermore, an SMT solver employing the algorithm of Figure 2.2 terminates when called on the conjunction of formulas in $\Gamma' \cup \Psi$, and the result returned by the solver corresponds to the satisfiability of the conjunction of the formulas in $\Gamma \cup \Psi$.

This proposition improves over Proposition 4.1 by also guaranteeing termination. The price we have to pay is the under-approximation introduced by the restrictions $\mu_i (t) \leq l$.

Proof of Proposition 4.2. The transformed recursive definitions in Γ' and equations (4.12) involving Pred_i are satisfiable by construction. Also, the equations $f_i (t) = f_i' (t, l)$ in Γ' are implied by the constraints $\mu_i (t) \leq l$ in Ψ. Hence, if the conjunction of the formulas in $\Gamma \cup \Psi$ is satisfiable, then so is the conjunction of the formulas in $\Gamma' \cup \Psi$. Thus, if the algorithm of Figure 2.2 returns a non-null result to indicate that $\Gamma' \cup \Psi$ is unsatisfiable, then $\Gamma \cup \Psi$ is also unsatisfiable. If null is returned, then it can be shown in the same way as in the proof of Proposition 4.1 that $\Gamma \cup \Psi$ is satisfiable.

It remains to show that the algorithm of Figure 2.2 terminates when called on $\Gamma' \cup \Psi$. According to the triggers applied, the transformed recursive definitions (4.11) can only be instantiated by the terms $\mathrm{Pred}_i (k)$ in the equations (4.12), which can themselves only be instantiated by arguments to transformed recursive functions. Other than in the equations (4.13), applications of transformed recursive functions can only occur in the right-hand sides of the transformed recursive definitions (4.11). As a result, we can trace every application of a transformed recursive function in an instantiated ground term to the *evaluation* of an application of a transformed recursive function in the equations (4.13). For every such application $f'_i(x, k)$ in an instantiated ground term, the second argument to the function corresponds to the distance of this application to the original application in (4.13). If this argument m reaches zero, the condition $k > 0$ in (4.12) causes the arithmetic component of the SMT solver to prevent the equation $\mathrm{Pred}_i(m) + 1 = m$ from being asserted. Since the triggers of the recursive definitions require an active occurrence of Pred_i for instantiation, the instantiation chain of recursive definitions is interrupted. Thus, termination is ensured. □

Note that we exploited quite specific features of the SMT solver in this proof. In particular, we required that triggers are not matched against

$$k > 0 \wedge \mathrm{Pred}_i(k) + 1 = k$$

if the assertion of these literals, in combination with the other already asserted literals, leads to a contradiction modulo the theory of linear arithmetic (or difference logic). This requirement is met by a solver that employs a suitable form of theory propagation in the `unit_propagate` step in Figure 2.2. A less elaborate solver could meet the requirement by performing a form of early pruning within the `choose_literal` step to prevent a literal to be selected that leads to a contradiction.

Example: Exponential Diophantine Equations With this support for recursive functions, it is possible to handle constraints with nonlinear arithmetic by defining nonlinear functions recursively. We start with the multiplication function, for which we define the following transformed variant on $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$:

$$\begin{aligned} \mathrm{mult}(x+1, y, k+1) &= y + \mathrm{mult}(x, y, k) \\ \mathrm{mult}(0, y, k) &= 0 \\ \mathrm{mult}(x+1, y, 0) &= 0 \end{aligned} \tag{4.14}$$

Here the last equation defines an arbitrary function value as default for the case that the number of evaluation steps specified by the third argument is not sufficient. Based on this function for multiplication, we can define a function for exponentiation in the same manner:

$$\begin{aligned} \mathrm{pow}(x, y+1, k+1) &= \mathrm{mult}(x, \mathrm{pow}(x, y, k), k) \\ \mathrm{pow}(x, 0, k) &= 1 \\ \mathrm{pow}(x, y+1, 0) &= 1 \end{aligned} \tag{4.15}$$

It is easy to see that using these definitions, the required recursive depth to evaluate x^y is $x + y$. Thus, we have $\mathrm{pow}\,(x, y, l) = x^y$ if $l \geq x + y$. Hence, we can find all solutions (x, y) between 0 and 3 to the equation $x^y = 8$ by choosing $l = 3 + 3 = 6$ and passing (4.14), (4.15) and corresponding versions of (4.12) together with appropriate triggers and the following system of constraints to an SMT solver:

$$\begin{array}{rcl} \mathrm{pow}\,(x, y, 6) & = & 8 \\ x & \leq & 3 \\ y & \leq & 3 \end{array} \tag{4.16}$$

The SMT solver Z3 yields as desired the solution $(x, y) = (2, 3)$.

4.5 Towards Interactive Constraint Solving

Isabelle is a theorem prover that favors interactive use. Users can freely define intermediate lemmas, examine proof states and decide which tactics to apply. These forms of interaction allow theorems to be proved that are out of reach of non-interactive theorem proving procedures. In principle, interaction enables the proof of any theorem that is provable if a sufficiently rich set of elementary inference steps is made available to the user. However, in order for a proof to actually be carried out, it is necessary that the user is ingenious enough to find a successful approach to prove the theorem. A nice feature of the type of interaction provided by Isabelle is that a user can choose and apply appropriate automated theorem proving procedures that are known to work well for a particular class of conjectures.

In contrast, typical constraint solving procedures are fully automatic and do not provide for user interaction after computation has started. Users may be able to influence constraint solving by altering the representation of the constraints before starting the solver. E.g., a user may define custom triggers that are later taken into account for quantifier instantiation during SMT solving. Also, a constraint solving algorithm may be customizable through parameters and options. Finally, a constraint solver may allow users to supply solving subroutines that are used for a specific type of constraints. Such subroutines could e.g. be solvers for a custom theory for an SMT solver featuring theory combination.

Using Isabelle for constraint solving problems raises the question whether Isabelle's support for interaction can aid constraint solving in a similar way as it aids theorem proving. Brucker and Wolff [29] have demonstrated that interaction in Isabelle can contribute to constraint solving when applying their random constraint solving technique. They used a custom rewrite rule for simplifying constraints that define test cases for red-black trees. The correctness of the rewrite rule was proved interactively. The tailored simplifications were shown to improve the performance of random constraint solving remarkably. However, we have no indication that this kind of approach can provide similar advantages when applying more sophisticated constraint solving techniques, such as those used by SMT solvers.

In this thesis we present a different way in which constraint solving can benefit from interaction in Isabelle. Above we have described how SMT solvers can be used within Isabelle for solving quantifier-free constraints, possibly involving recursive functions. However, occurrences of quantifiers in constraints, other than those in recursive function definitions, remain problematic for SMT solvers, since the kind of trigger-driven quantifier instantiation used by SMT solvers is incomplete. We now present a case in which interaction in Isabelle can help alleviate this shortcoming of SMT solvers. We use the same red-black tree example that Brucker and Wolff [29] used for making a case for interactive constraint solving. The most natural specification of red-black tree operations involves quantifiers that would be problematic for SMT solvers without special precautions. We show how these quantifiers can be eliminated by means of interactive proofs. The test cases generated from the modified specification can be handled by the SMT solving techniques we presented above.

In the sequel, we present the original specification of red-black trees and then show how the occurring problematic quantifiers can be eliminated by means of interactive proofs.

4.5.1 Case Study: Red-Black Trees

A red-black tree [10] is a binary tree that satisfies additional balancing invariants to ensure fast lookups. Each node is associated with a color (i.e., red or black) to allow for balancing. Thus, we introduce a corresponding color datatype in HOL.

```
datatype  color = R | B
```

We define red-black trees as a polymorphic datatype to allow for storage of different types of values. However, for simplicity we will restrict ourselves in the sequel to trees that store integers.

```
datatype 'a tree = E | T color "'a tree" "'a" "'a tree"
```

The type constructor `E` represents an empty tree, non-empty trees can be built with the type constructor `T`.

In order to describe the datastructure of red-black trees, we still need to define the balancing invariants. We start with the so-called weak red invariant that prevents a red node from having a red child:

```
fun    redinv        :: "int tree ⇒ bool"
where   redinv_1:    "redinv E = True"
      | redinv_2:    "redinv (T B a y b) = (redinv a ∧ redinv b)"
      | redinv_3:    "redinv (T R (T R a x b) y c) = False"
      | redinv_4:    "redinv (T R a x (T R b y c)) = False"
      | redinv_5:   "redinv (T R a x b) = (redinv a ∧ redinv b)"
```

We continue with the strong red invariant. This invariant requires every red node to have a black parent. This is implied by the weak red invariant if in addition the root is black. Hence, the strong red invariant can be defined as follows:

```
fun   strong_redinv :: "int tree ⇒ bool"
where
    Rinv_1:  "strong_redinv E = True"
|   Rinv_2:  "strong_redinv (T R a y b) = False"
|   Rinv_3:  "strong_redinv (T B a y b) = (redinv a ∧ redinv b)"
```

In order to define the black invariant, we introduce the auxiliary function max_B_height on trees whose value is equal to the greatest number of black nodes on any path from the root to a leaf:

```
fun  max_B_height :: "int tree ⇒ nat"
where
  maxB_height_1:  "max_B_height E = 0"
| maxB_height_3:  "max_B_height (T B a y b)
                   = Suc(max (max_B_height a)
                             (max_B_height b))"
|  maxB_height_2:  "max_B_height (T R a y b)
                   = (max (max_B_height a) (max_B_height b))"
```

Now we can express the black invariant that requires all paths from the root to any leaf to exhibit the same number of black nodes:

```
fun blackinv      :: "int tree ⇒ bool"
where
     blackinv_1:  "blackinv E = True"
|    blackinv_2:  "blackinv (T color a y b)
                  = ((blackinv a) ∧ (blackinv b)
                     ∧ ((max_B_height a) = (max_B_height b)))"
```

The following recursive predicate that determines whether a certain integer belongs to a red-black tree can be used in test specifications:

```
fun isin          :: "int ⇒ int tree ⇒ bool"
where
     isin_empty :  "isin x E = False"
 |   isin_branch : "isin x (T c a y b)
                     = ((x = y) | (isin x a) | (isin x b))"
```

Of course, we also need to ensure that the integers stored in red-black trees are ordered:

```
fun isord         :: "int tree ⇒ bool"
where
     isord_empty  : "isord E = True"
```

```
 |  isord_branch : "isord (T c a y b) =
                    (isord a ∧ isord b
                     ∧ (∀ x. isin x a ⟶ (x < y))
                     ∧ (∀ x. isin x b ⟶ (x > y)))"
```

4.5.2 Modifying the Specification Interactively

The problem in the specification of red-black trees above are the quantifiers in the definition of *isord* that are difficult for SMT solvers to deal with. We will use the following definition in order to eliminate these quantifiers.

```
fun tree_all_comp
  :: "int tree ⇒ (int ⇒ int ⇒ bool) ⇒ int ⇒ bool"
where
    tree_all_comp_empty :  "tree_all_comp E c y = True"
 |  tree_all_comp_branch :
  "tree_all_comp (T rb a x b) c y
  = ((c x y) ∧ (tree_all_comp a c y) ∧ (tree_all_comp b c y))"
```

This function *tree_all_comp* compares its third argument to all nodes in the tree passed as its first argument, thereby using the comparison function passed as its second argument. By choosing an appropriate recursive definition, we are able to express this function without quantifiers like in the definition of *isord* above.

The following lemma states how the quantifier in the definition of *isord* can be expressed using the function *tree_all_comp*.

```
lemma all_eq: "(tree_all_comp a c y)
                 = (∀ x. isin x a ⟶ c x y)"
```

All proof obligations resulting from this conjecture can be discharged by the following sequence of tactics that was found interactively. The proof is essentially based on induction according to the recursive definition of *tree_all_comp*.

```
apply (auto)
apply (erule rev_mp)+
apply (induct_tac a)
apply (auto)
apply (erule rev_mp)+
apply (induct_tac a)
apply (auto)
done
```

In order to avoid lambda-expressions in the test specification that may be problematic for SMT solvers, we define the following two specific tree comparison functions.

```
fun tree_all_less :: "int tree ⇒ int ⇒ bool"
```

```
where
    tree_all_less_d:
  "tree_all_less a y = (tree_all_comp a (λu v .u < v)) y"


fun tree_all_greater :: "int tree ⇒ int ⇒ bool"
where
    tree_all_greater_d:
  "tree_all_greater a y = (tree_all_comp a (λu v. u > v)) y"
```

Using these new functions, we are able to provide a new definition of `isord` that does not rely on an unbounded quantifier.

```
fun isord'         :: "int tree ⇒ bool"
where
    isord_empty'  : "isord' E = True"
 |  isord_branch' : "isord' (T c a y b) =
                    (isord' a ∧ isord' b
                     ∧ (tree_all_less a y)
                     ∧ (tree_all_greater b y))"
```

Using our lemma above, we can prove interactively that the new predicate `isord'` is equivalent to its predecessor `isord`. Again, the proof is essentially based on induction.

```
lemma isord_subst: "isord t = isord' t"
apply (induct_tac t)
apply (auto)
apply (simp_all add: all_eq)
done
```

By inserting this lemma `isord ?t = isord' ?t` in the Isabelle simplifier, we can eliminate any occurrence of `isord` by the equivalent predicate `isord'` that can be handled more easily by SMT solvers.

4.6 Experimental Results

We evaluated the constraint solving techniques discussed in this chapter on the following test generation problems:

- The classical "triangle" example of testing a program that classifies a triangle as equilateral, isosceles or scalene [115].
- The sorting specification presented in Section 3.3. However, we chose different parameters in order to obtain a larger test suite. The resulting test suite consists of 34 test cases that correspond to all possible permutations of lists with lengths up to 4.

Test suite	# tests	Random solver		Quickcheck		SMT	
		# solved	Runtime	# solved	Runtime	# solved	Runtime
Triangle	18	18	0.13	18	4.4	18	1.1
Listsort	34	34	1.1	34	14	34	1.1
RB trees (original)	30	0	–	0	–	0	–
RB trees (modified)	30	14	23	11	20	30	418
AVL trees	14	14	3.2	14	6.5	14	2.5
Firewall	238	8	0.054	8	3.4	238	8.5

Table 4.1: Test data generation with different constraint solving techniques: experimental results

- A test specification expressing that the deletion of an element from a red-black tree preserves the black invariant:

$$\mathit{isord}\ t \wedge \mathit{isin}\ y\ t \wedge \mathit{strong\_redinv}\ t \wedge \mathit{blackinv}\ t \\ \Longrightarrow \mathit{blackinv}\,(\mathit{delete}\,(y, t))$$

  See Section 4.5.1 for definitions of the predicates used in this specification. We removed the recursive definition of *isord* from the Isabelle simplifier in order to obtain test cases that are independent from this definition. As a result, applications of *isord* remained in the constraints associated with the test cases. We carried out our evaluation on these original constraints as well as modified constraints in which *isord* was substituted by its equivalent variant isord' introduced in Section 4.5.2.

- A similar specification about AVL trees stating that a tree remains balanced after insertion. However, here we did not specify that the trees are ordered, which yields simpler constraints. Nevertheless, the resulting constraints still contain applications of recursive functions.

- A test specification taken from a large case study for HOL-TestGen that is concerned with the test of firewalls [23]. The resulting constraints are rich in arithmetic inequalities, e.g.,

$$\neg x \leq 4096 \wedge 1 < y \wedge y < 256 \wedge 1001 < z \wedge z < 1256 \wedge x \leq 6544.$$

For solving the constraints generated by HOL-TestGen for these test specifications, we applied the random solving technique of Brucker and Wolff [28], a technique based on Isabelle's `quickcheck` command [12] and the SMT approach described in Section 4.3 with instrumentation for recursive functions following the presentation in Section 4.4.

The results of the evaluation are shown in Table 4.1. The table gives for every test suite the number of tests in the test suite and the number of tests that each constraint solving technique could generate as well as the time in seconds consumed by constraint solving. The times given only refer to those constraints that were actually solved by the respective techniques. The number of random solving attempts was limited to 1000 per constraint. The `quickcheck` procedure was configured for a maximum of 10000 attempts per constraint, except for the red-black tree example, for which we reduced this parameter to 1000 in order to be able to carry out the measurements within reasonable time despite the complexity of the constraints. The timeout for SMT solving was set to 300 seconds. The measurements were carried out on a machine with 2 GB RAM and a dual-core 2.4 GHz P8600 mobile CPU. Parallel processing in Isabelle [111] was turned off.

We conclude that, at least for the kinds of test generation problems we dealt with in this evaluation, SMT solving is the method of choice. In particular, this is the only technique that succeeded in generating all tests of the red-black tree and firewall test suites. However, for the triangle test suite the random solving technique consumed less time, which may be an indication that random solving can solve some easy constraints faster than the SMT-based technique. Moreover, random solving using the Isabelle simplifier for detecting solutions compared favorably to `quickcheck` in this evaluation. Reasons for this could be the distributions used for the random generation of candidate solutions, or that `quickcheck` suffers from an overhead due to code generation.

4.7 Related Work

The random approach to test data generation for functional programs was first introduced by the Quickcheck tool [49] and has been applied since in several variations [95, 47, 42]. Because HOL resembles a functional programming language, the application of random solving in HOL-TestGen can be regarded as a continuation of this work.

The instrumentation of formulas with suitable triggers for SMT solving has already been investigated before [106, 113]. This prior work has been concerned with defining triggers that are sufficient for finding proofs in certain situations, notably in the domain of program verification. In contrast, the goal of our work was to define triggers that not only ensure the correctness of an “unsat” response of the solver, but also the correctness of a counterexample in case of a “sat” response.

Another tool that heavily relies on solving recursive constraints with a SMT solver is the SQL “query explorer” Qex [151] that generates test cases for databases. Database tables are represented as recursive datastructures, and SQL constructs are mapped to recursive functions that operate on tables. A difference to our approach is that the bounds on the size of the tables in the constraint encoding are not expressed numerically, but by constraints that are specific to the datastructure. Such custom constraints may become very large for certain datastructures.

Carlier et al. [43] have presented a different approach to solving constraints that involve functions defined via pattern matching. They integrated dedicated constraint combinators for pattern matching into a framework for constraint logic programming. With this approach, they also were able to observe massive speedups for test data generation when compared to random solving methods.

Nitpick [16] is a counterexample generator integrated in Isabelle that solves HOL constraints using an eager SMT approach, i.e., HOL formulas are encoded in several steps into Boolean constraints that are passed to a propositional SAT solver. The purpose of Nitpick is to detect false conjectures during interactive proofs, it does not establish new Isabelle theorems. Nitpick ensures termination in presence of recursive functions by under-approximating types, e.g., the set of natural numbers could be approximated by a finite range [15]. In contrast, our approximation approach targets the arguments of recursive functions individually. The two kinds of approximation are, however, very similar. Following the eager SMT methodology, Nitpick performs the instantiation of recursive definitions before calling the SAT solver. Our trigger-based approach to the instantiation of recursive definitions can potentially benefit from optimizations in the solver. In particular, instantiations are performed "on demand" only when a trigger matches.

Chapter 5

Arithmetic Formulas with Bounded Quantifiers

In this chapter we introduce arithmetic formulas with bounded quantifiers as intermediate representation for constraints. As in compiler construction, intermediate representations play a crucial role in animation tools. Intermediate languages must be general enough to be able to express all constructs supported by the tool and at the same time allow for efficient processing in the back-end. In the context of animating OCL specifications, arithmetic formulas with bounded quantifiers turn out to possess these desirable properties.

OCL constraints have a complex structure. They deal with a rich set of types including classes and several kinds of collections. Analysis is simplified substantially if constraints can first be translated to a simpler intermediate representation.

In the sequel, we define the syntax and semantics of arithmetic formulas with bounded quantifiers and discuss its expressiveness. We then present an efficient eager SMT approach for solving constraints represented in this language. Since it is essential for animation that objective functions can be taken into account, we also show how an objective function can be optimized when solving. We will describe later how most of the OCL language can be mapped conveniently to arithmetic formulas with bounded quantifiers.

5.1 Syntax and Semantics

We define the syntax of our intermediate language, that we refer to as the language of arithmetic formulas with bounded quantifiers, as follows:

1. Function symbols represent uninterpreted functions mapping $\mathbb{Z}^k$ to $\mathbb{Z}$.
2. Additional function symbols represent any desired unary and binary arithmetic operations such as addition, subtraction, multiplication and integer

division. We will assume in the sequel that at least addition is included.

3. Constants from $\mathbb{Z}$ are terms.

4. Variables are terms and assume values in $\mathbb{Z}$.

5. A function symbol applied to terms is a term.

6. Binary predicates $=$, $<$, $\leq$, $>$ and $\geq$ applied to terms are formulas.

7. Formulas can be connected using the usual boolean operations.

8. For a formula p and terms t_1 and t_2, $\text{if } \ p \ \text{ then } \ t_1 \ \text{ else } \ t_2$ is a term.

9. If p is a formula and t_1, t_2 are terms, then $\forall t_1 \leq x \leq t_2 \, . \, p$ is a formula. Here t_1 is the lower bound and t_2 is the upper bound of the quantifier. Similarly, $\exists t_1 \leq x \leq t_2 \, . \, p$ is a formula.

The semantics of this language is the ordinary semantics associated with the combined theory of arithmetic and uninterpreted functions. Thus, the language of arithmetic formulas with bounded quantifiers can be understood as a syntactically defined subset of this combined theory.

5.2 Expressiveness and Decidability

The language of arithmetic formulas with bounded quantifiers has some useful characteristics. As is usual terminology, we call a set of function symbols and variables a *signature*. We refer to an assignment of specific functions to function symbols and variables in a signature as *model*. A formula is *satisfiable* if there is a model for which the formula evaluates to true. The following proposition states that arithmetic formulas with bounded quantifiers are very expressive.

Proposition 5.1. *Every recursively enumerable predicate $P\left(x_1,\ldots,x_k\right) \subseteq \mathbb{N}^k$ has an arithmetic formula $\varphi\left(x_1,\ldots,x_k\right)$ such that for all $x_1,\ldots,x_k \in \mathbb{N}$, $\varphi\left(x_1,\ldots,x_k\right)$ is satisfiable iff $P\left(x_1,\ldots,x_k\right)$ is true.*

Proof. The proposition follows from Matiyasevič's theorem [56] stating that every recursively enumerable predicate $P\left(x_1,\ldots,x_k\right) \subseteq \mathbb{N}^k$ can be expressed in the form $\exists y_1,\ldots,y_l \in \mathbb{N}. \ p\left(x_1,\ldots,x_k,y_1,\ldots,y_l\right) = 0$, where $p\left(x_1,\ldots,x_k,y_1,\ldots,y_l\right)$ is a polynomial with integer coefficients. We can take $\varphi := p \wedge y_1 \geq 0 \wedge \ldots y_l \geq 0$, and leave $y_1,\ldots,y_l$ as free variables.

Note that it is not essential that we allow multiplication as elementary operation in arithmetic formulas, since we can replace a multiplication $s \cdot t$ by

$$\text{if } \ t \geq 0 \ \text{ then } \ f\left(|t|\right) \ \text{ else } \ -f\left(|t|\right)$$

for a fresh function symbol f if we add the constraints $f(0) = 0$ and

$$\forall 1 \leq x \leq |t| \,.\, f(x) = f(x-1) + s.$$

The absolute value $|\cdot|$ can be expressed by

$$|t| = \text{if } \ t > 0 \ \text{ then } \ t \ \text{ else } \ -t.$$

□

It follows immediately from Proposition 5.1 that the satisfiability of arithmetic formulas with bounded quantifiers is undecidable since there are recursively enumerable sets that are not recursive.

Proposition 5.2. *Given a representation of a model M for a signature Σ that allows every function value $M(f)(a)$ of an interpretation $M(f)$ of a function symbol f in Σ by M to be computed effectively, the value of every arithmetic formula with function symbols from Σ can be computed effectively.*

Proof. It is enough to note that the formula can be evaluated bottom-up in the obvious way. □

Proposition 5.3. *The set of satisfiable arithmetic formulas with bounded quantifiers is recursively enumerable.*

Proof. We call a model M for a signature Σ *bounded* if there is a number $n \in \mathbb{N}$ such that for every function f in Σ, $|M(f)(a)| \leq n$ for all $a \in \mathbb{Z}^k$, $M(f)(a) = 0$ if the absolute value of a component of $a \in \mathbb{Z}^k$ is larger than n, and $|x| \leq n$ for every variable x in Σ. It is easy to see that the set of bounded models can be enumerated. Hence, in order to infer that the set of satisfiable arithmetic formulas with bounded quantifiers is recursively enumerable, it is enough to show that every satisfiable arithmetic formula with bounded quantifiers has a bounded model. To see this, observe that since an arithmetic formula with bounded quantifiers can be evaluated for any model in a finite number of steps, the largest absolute value of any integer used during such an evaluation is bounded. □

Thus, although the satisfiability of arithmetic formula with bounded quantifiers is undecidable, the set of satisfiable formulas is at least recursively enumerable, or *semi-decidable*. In the next section, we present an approach based on SMT solving for performing this enumeration in an efficient manner.

5.3 Solving Using Eager SMT

The satisfiability of arithmetic formulas with bounded quantifiers can be analyzed using the techniques based on lazy SMT that we gave in Section 4.4 for constraints with recursive functions. To see this, note that the value of a quantified formula $\forall t_1 \leq x \leq t_2 \,.\, p\,(x)$ equals $q\,(\max\,(t_2 - t_1 + 1, 0))$, where the predicate q is defined recursively by

$$\begin{array}{lcl} q\,(0) & = & \text{true} \\ q\,(n+1) & = & q\,(n) \wedge p\,(n + t_1)\,. \end{array}$$

Thus, bounded quantifiers in an arithmetic formula can be expressed with recursive definitions. However, in this section we present a different technique based on eager SMT for solving arithmetic formulas with bounded quantifiers. Using a technique based on eager SMT has the advantage that a wide variety of propositional SAT solvers are available, while many of today's top lazy SMT solvers are closed-source. On the other hand, an advantage of lazy SMT solvers is that built-in solvers for linear arithmetic can handle real numbers with arbitrary precision. We do not claim here that eager or lazy SMT is more efficient in general, but observe that in either case efficiency depends on a carefully tuned implementation of the solver.

We proceed as follows to find a model of an arithmetic formula with bounded quantifiers using eager SMT. First, the formula is simplified in order to remove redundant subexpressions. Second, we construct a Boolean circuit that computes the validity of the formula. In order to construct the circuit, we may need to bound certain values. Third, the Boolean circuit is converted to conjunctive normal form (CNF). Fourth, the resulting CNF is solved by an off-the-shelf SAT solver. A solution to the Boolean satisfiability problem yields a model for the original arithmetic formula. If no solution to the Boolean problem exists and we had to bound any values in order to construct the circuit, we repeat the analysis with less restrictive bounds. The result is a semi-decision procedure that always finds a model if one exists. However, termination is not guaranteed. This basic procedure is depicted in Figure 5.1.

In the sequel we describe this procedure in more detail.

5.3.1 Encoding as a Boolean Circuit

Our encoding of arithmetic formulas as Boolean circuits does not differ significantly from the encodings employed by other SAT-based analysis tools [50, 144, 69, 35]. However, we describe our encoding here for completeness.

Encoding of Function Symbols Recall that function symbols represent uninterpreted functions mapping $\mathbb{Z}^n$ to $\mathbb{Z}$. We encode function symbols as vectors of Boolean variables. For every function value these vectors contain as subvector a bit-vector that is long enough to represent all values in the range of the function (we discuss the problem of fixing this range below in Section 5.3.2). Through an analysis of the formula we determine the set of possible arguments the function may be evaluated for during an evaluation of the formula. The length of the

```
procedure solve($\varphi$):
  bounds := initial_bounds($\varphi$)
  forever do
    CNF := generate_CNF($\varphi$, bounds)
    (solved, solution) := call_SAT_solver(CNF)
    if solved then
      return solution
    end if

    increase_bounds(bounds)
  end forever
end procedure
```

Figure 5.1: Basic procedure for finding a model of an arithmetic formula with bounded quantifiers

vector for a function symbol is the product of the number of possible arguments and the number of bits necessary for representing a function value.

Encoding of Terms We encode integer terms as vectors of Boolean circuits which represent the bits of the integer value. Arithmetic operations like addition and multiplication are dealt with by constructing a Boolean circuit for the operation, as would be done for computing the operation in hardware. We do not allow arithmetic overflow, e.g., we encode the sum of two 32-bit integers as a vector of 33 bits, so all values that can result from the addition of two 32-bit integers can be represented. We translate function application to a multiplexer circuit that selects the bit-vector which corresponds to the value of the function argument. If a term contains free variables, we perform the encoding for every possible variable assignment. This results in a map that assigns a vector of Boolean circuits to every variable assignment.

Encoding of Formulas Boolean operations in arithmetic formulas can be mapped directly to corresponding gates in the generated Boolean circuit. For quantifiers, we encode the body of the quantified formula together with a guard checking the quantifier bounds for all possible assignments to the quantified variable. The resulting Boolean circuits are fed into the respective gate ($\wedge$ or $\vee$).[1]

5.3.2 Choosing Suitable Ranges for Function Symbols

Recall that in the Boolean encoding of arithmetic formulas outlined above, a subexpression with free variables is encoded separately for all values the variables

[1] Of course this is not necessary for quantifiers that can be eliminated by skolemization.

can assume during an evaluation of the formula. It is clearly not feasible to always perform the encoding for all values in the largest possible quantifier range, e.g., all 32-bit integers.

Existing analysis tools for UML/OCL operation contracts like UML2Alloy [6] and UMLtoCSP [40] depend on bounds provided by the user for restricting quantifier ranges. The results of the analysis only make a statement about states that comply with the provided bounds. However, for the purpose of animation it is highly desirable to use a form of analysis that is complete in the sense that valid animation results are obtained if they exist. We aim to relieve the user from the burden of providing adequate bounds. In particular, the necessity of specifying bounds is a considerable obstacle to the integration of animated operations with other code, since it requires a modification of the operation interface.

Trigger-based quantifier instantiation used by solvers following the lazy SMT approach can alone not overcome this difficulty. In general, that quantifier instantiation cannot derive unsatisfiability does not imply that a correct model of the formula can be obtained.

We propose an iterative approach that is based on restricting the ranges of certain function symbols occurring in the arithmetic formula. We restrict the ranges of those function symbols that occur in the quantifier bounds that are made explicit by the syntax of our language. Through interval arithmetic, we can then derive a restricted range for each lower and upper quantifier bound. Thus, we can obtain a sufficient translation of a quantified formula by instantiating the quantified variable only for the restricted set of values that can be between the quantifier bounds. If the function symbol ranges are chosen to be small enough, this set of values the quantified variable can assume is manageable. If no model is found for the first choice of restricted function symbol ranges, a more expensive attempt with larger ranges is made, and so on. Restricting the range of a function symbol results in an under-approximation of the original satisfiability problem, i.e., certain models are excluded, whereas every solution to the under-approximation is a valid model for the formula. Note that simply restricting the quantifier ranges considered during the translation while leaving function symbol ranges unchanged does not necessarily yield an under-approximation, and thus may give rise to solutions that are not valid models of the formula.

This technique of restricting ranges of function symbols is analogous to our under-approximation technique described in Section 4.4 for solving constraints with recursive functions.

We restrict the ranges of function symbols occurring in function arguments as we do for function symbols occurring in quantifier bounds. As a result, we can derive through interval arithmetic sufficiently bounded ranges for all terms that are function arguments. This allows us to encode function symbols as vectors of Boolean variables that are of manageable size.

Since in an eager SMT approach a fixed number of bits has to be allocated for every integer value in order to obtain a Boolean encoding of the formula, the range of every integer value has to be bounded. In many applications, it is possible to restrict integer values that do not occur in quantifier bounds or function arguments to ranges which are certainly sufficient. Consider as an example an integer value modeling a Java field of type `int`. Its values may be restricted to 32-bit numbers, which is sufficient to represent all values of the Java `int` type.

As a result of this approach to bounding function values, we search for models using different bounds for different integer values. In the context of animating OCL specifications, these integer values can represent values of integer attributes, numbers of instances of a class or collection sizes. The bounds for these values are chosen depending on the contexts in which the values are used in the constraints. Consider for example the constraint

$$\begin{aligned} & s\,(0) = 0 \\ & \wedge \forall 1 \leq x \leq n.\ s\,(x) = s\,(x-1) + c\,(x-1)\,. \end{aligned}$$

Here s and c are unary function symbols, and n is a constant (nullary function). This kind of constraint could be used to express summation over collection elements represented as $c\,(0)\,, c\,(1)\,, \ldots, c\,(n-1)$. The sum equals $s\,(n)$. For analyzing the satisfiability of such a constraint, we would introduce restrictive bounds for the constant n, since n occurs in a quantifier bound. These bounds would, if necessary, be increased in future iterations in order to find a model. On the other hand, since the functions s and c occur neither in quantifier bounds nor function arguments, we could consider, e.g., all possible 32-bit values for values of these functions already in the first call to the SAT solver. Thus, even models with very large values for these functions are not necessarily problematic for our approach.

5.3.3 Efficient Translation of Formulas to Boolean Circuits

In our approach, the actual constraint solving is performed by the SAT solver that receives the CNF. The preceding computation that generates the Boolean circuit from the arithmetic formula and converts the circuit to a CNF is deterministic and has a complexity that is polynomial in the size of the circuit. These facts suggest that the SAT solving is the bottleneck regarding runtime, whereas the preprocessing steps are uncritical for performance. Nevertheless, in our experience the cost of generating the input to the SAT solver is for many simulation problems far more expensive than the execution of the SAT solver itself.[2] We observed that many SAT instances arising during animation are solved in a fraction of a second. The main factor that determines the size of the Boolean circuit and the CNF, and thus the preprocessing time, are the quantifiers that are present

[2] See [41, 144] for measurements showing that preprocessing consumed more time than SAT solving.

in the input formula and the ranges for which they are instantiated. Nested quantifiers are particularly expensive.

In order to reduce the time used for preprocessing, we have implemented an improved algorithm for translating quantified formulas to Boolean circuits. Figure 5.2(a) shows a typical approach to perform an encoding like the one described in Section 5.3.1. In this pseudo-code, `BINARY_OP` stands for any kind of binary operation such as addition. The assignment `env` to the free variables of the formula is a parameter to the translation. This assignment can then be passed on to recursive calls of the procedure for translating subexpressions. The resulting subexpression translations can be used for obtaining a translation of the entire formula, e.g., by feeding them into a circuit constructed by the function `make_BINARY_OP` that computes the operation `BINARY_OP`. For translating a quantified formula, a loop iterates over the values for which the quantifier is instantiated. For every value, the body of the quantified formula is translated with the quantified variable set to this value.[3] The translations of the body are then aggregated according to the type of the quantifier. The remaining connectives of our language such as if-then-else can be handled analogously. This approach to formula translation is straightforward to implement. It is also suggested by some semantics definitions that define the semantics of quantifiers by constructs that resemble loops. This is also the case for the OCL standard [125].

However, it turns out the straightforward approach depicted in Figure 5.2(a) is not optimal concerning efficiency. Note that it causes a separate translation of every subexpression in the scope of a quantifier for every value the quantified variable can assume—even for subexpressions in which the quantified variable does not occur. Consider for example the constraint

$$\forall 1 \leq x \leq 10.\ \forall 1 \leq y \leq 10.\ f\left(g\left(x\right)\right) \neq h\left(k\left(y\right)\right).$$

The subexpression $f\left(g\left(x\right)\right)$ would be translated $10 \cdot 10$ times, although the variable x can only assume 10 different values. The subexpression $h\left(k\left(y\right)\right)$ would also be translated $10 \cdot 10$ times, although the variable y can only assume 10 different values as well. This clearly is a waste of resources. It would be much more efficient to perform translations of subexpressions depending on their free variables.

This observation leads to the algorithm sketched in Figure 5.2(b). We call this approach bottom-up in contrast to the top-down method in Figure 5.2(a). The bottom-up algorithm first translates subexpressions for all possible assignments to their free variables. The resulting translations are stored in a map data structure that supports lookups based on a variable assignment. When translating an application of an arithmetic operation, the translations of the arguments are retrieved for every assignment to the free variables of the entire formula. When performing these map lookups, we discard any values for variables that

[3] Here we assume that this body already contains the guard checking the quantifier bounds.

```
procedure translate(expr, env):
  case expr of BINARY_OP(in1, in2):
    return make_BINARY_OP(translate(in1, env), translate(in2, env))

  case expr of FORALL(x, body):
     inputs := ∅
     for i in [lowerBound(x)..upperBound(x)] do
       inputs := inputs ∪ translate(body, env[x:=i])
     end for
     return make_AND(inputs)

  case expr of ...
end procedure
```

(a) Top-down

```
procedure translate(expr)
  case expr of BINARY_OP(in1, in2):
    translations1 := translate(in1)
    translations2 := translate(in2)
    for env in Assignments(FreeVars(expr)) do
      translations[env]
         := make_BINARY_OP(translations1[env], translations2[env])
    end for
    return translations

  case expr of FORALL(x, body):
    body_translations := translate(body)
    for env in Assignments(FreeVars(expr)) do
      inputs := ∅
      for i in [lowerBound(x)..upperBound(x)] do
        inputs := inputs ∪ body_translations[env[x:=i]]
      end for
      translations[env] := make_AND(inputs)
    end for

  case expr of ...
end procedure
```

(b) Bottom-up

Figure 5.2: Approaches to Generating Circuits from Formulas with Quantifiers (pseudo-code)

are not free variables of the respective argument. The retrieved subexpression translations are then used for computing corresponding translations of the entire formula. For translating a quantified formula, the translations of the body are aggregated according to the type of the quantifier.

The bottom-up approach has the advantage that the number of times a subexpression is translated only depends on the values that its free variables can assume. The translation procedure visits every subexpression only once. All translations of a subexpression are performed in an efficient loop structure. This promotes optimizations like the elimination of loop invariant computations and caching of memory.

In the implementation of our approach, we use an adapted version of the kodkod solver for constructing the circuit as a *Compact Boolean Circuit* [144], a compressed representation of a Boolean circuit. Kodkod includes effective algorithms for constructing and compressing Compact Boolean Circuits. We do not make use of higher-level features of kodkod such as symmetry breaking or encoding of relations.

5.3.4 Extension to Support Objective Functions

For many applications, it is essential that constraints can not only be solved, but also that solutions can be determined that are optimal according to provided objective functions, or at least close to optimal. We now show how our approach to finding models for arithmetic formulas can be extended to support objective functions. As objective functions we allow arbitrary arithmetic terms in our language.

The fundamental problem encountered when considering objective functions is that it is generally undecidable whether a better solution exists that improves the value of the objective function. Such a superior solution may only be constructible using much larger bounds. Therefore we encourage the user to supply a time limit for solving with an objective function in order to ensure termination. In order to promote early termination, we also consider over-approximations of the satisfiability problem based on the same bounds used for under-approximation as described above. If we determine that an over-approximation of the problem of finding an improved solution is unsatisfiable, we know that such a solution does not exist for any bounds and can stop the search. However, solving over-approximations is always incomplete, so unfortunately user-provided time limits are in general unavoidable.

We construct over-approximations by adding fresh Boolean variables to formulas resulting from the translation of quantifiers. These new variables represent the unknown value of the quantified formula in case the quantifier bounds exceed the bounds used for the translation. For a universal quantifier $\forall t_1 \leq x \leq t_2 \, . \, p\,(x)$

we construct the Boolean formula

$$\begin{array}{llllll} & (t_1 \leq \mathrm{lb}\,(t_1) & \wedge & \mathrm{lb}\,(t_1) & \leq t_2 \implies & p\,(\mathrm{lb}\,(t_1))) \\ \wedge & (t_1 \leq \mathrm{lb}\,(t_1) + 1 & \wedge & \mathrm{lb}\,(t_1) + 1 & \leq t_2 \implies & p\,(\mathrm{lb}\,(t_1) + 1)) \\ \ldots & & & & & \\ \wedge & (t_1 \leq \mathrm{ub}\,(t_2) & \wedge & \mathrm{ub}\,(t_2) & \leq t_2 \implies & p\,(\mathrm{ub}\,(t_2))) \\ \wedge & (t_1 < \mathrm{lb}\,(t_1) & \vee & \mathrm{ub}\,(t_2) & > t_2 \implies & a) \end{array} \tag{5.1}$$

where $\mathrm{lb}\,(t)$ and $\mathrm{ub}\,(t)$ are a lower and upper bound, respectively, of the values the term t can assume when the function values that t depends on are within the bounds used for the translation. The Boolean variable a is a fresh variable that represents the unknown value of the quantified formula in case the lower bound t_1 of the quantifier is smaller than $\mathrm{lb}\,(t_1)$ or the upper bound t_2 is larger than $\mathrm{ub}\,(t_2)$. The new variable a does not have any effect if $t_1 \geq \mathrm{lb}\,(t_1)$ and $t_2 \leq \mathrm{ub}\,(t_2)$ or the quantified formula evaluates to false due to the values of $p\,(\mathrm{lb}\,(t_1))\,, \ldots, p\,(\mathrm{ub}\,(t_2))$.

Example Consider a quantified formula

$$\varphi = \forall\, 0 \leq x \leq f + g \,.\, h\,(x) = 1. \tag{5.2}$$

Here f and g are constants (nullary functions) and h is a unary function. If we use bounds that restrict f and g to assume values in $[-2, 2]$, we obtain through interval arithmetic an upper bound of 4 for $f + g$. We apply the scheme (5.1) with $t_1 = 0$ and $t_2 = f + g$. Since the lower bound t_1 is constant, comparisons with it can be eliminated, and we obtain the following unfolding of φ:

$$\begin{array}{lll} & (0 \leq f + g & \implies h\,(0) = 1) \\ \wedge & (1 \leq f + g & \implies h\,(1) = 1) \\ \wedge & (2 \leq f + g & \implies h\,(2) = 1) \\ \wedge & (3 \leq f + g & \implies h\,(3) = 1) \\ \wedge & (4 \leq f + g & \implies h\,(4) = 1) \\ \wedge & (4 > f + g & \implies a). \end{array} \tag{5.3}$$

The overall procedure for animating with an objective function is shown in Figure 5.3. We can test the existence of a solution meeting a certain bound on the value of the objective function by calling the SAT solver with an appropriate set of Boolean literals as *assumptions*. When called with assumptions, a SAT solver restricts the search to models in which the assumptions are true. Clauses learned by the solver remain and can be used when solving later under different assumptions. Both MiniSat [63] and SAT4J [14] are SAT solvers that provide an interface for solving under assumptions.

For every choice of bounds we compute a solution by binary search that is optimal among all solutions within these bounds. We assume that the objective function assumes values within a certain interval [f_min, f_max] that we determine e.g. by restricting the values of all function symbols to 32-bit numbers. This has the additional benefit of preventing infinite looping in case there is no optimal solution.

We maintain a lower bound on the value of the objective function, which is to be minimized, with the hope of observing at some point that we have obtained a solution that is optimal unconditionally. After having searched for a solution that is as good as possible for the chosen bounds, we again perform a binary search for determining the best lower bound. For determining the lower bound we can reuse the same CNF since we can also control over-approximation over the assumption interface. We terminate after a solution has been found and the timeout is reached or a solution has been found whose value of the objective function matches the lower bound.

5.4 Related Work

The benefit of using intermediate languages for implementing animation tools has been recognized, and led to the intermediate languages μZ [77] for animating Z and CLPS-B [20] for B. These intermediate languages do not directly address the problem of quantifier bounding. On the other hand, bounded quantifiers have received widespread attention in computational complexity theory [98, 54].

So-called model finders like Paradox [50] and Nitpick [16] that search for models of quantified formulas with a SAT solver proceed by generating SAT problems that do not cover all potential models. If no model is found, they can generate a larger SAT problem to cover more models, and so on. This constitutes an iterative approach that is similar to our approach to finding models for arithmetic formulas with bounded quantifiers. Nitpick deals with problematic quantifiers by computing an undefined value when the value of the quantified formula is unknown. Our over-approximation approach can detect slightly more unsatisfiable constraints. The technique of under- and over-approximation that we use corresponds to abstraction-refinement techniques in verification [52]. Such techniques can be more powerful if the under-approximation is computed according to the result of the over-approximation and vice versa. An application of this idea to bounded quantifiers may be able to improve our approach.

Our bottom-up approach to translating quantifiers described in Section 5.3.3 is similar to a technique used by a past version of the Alloy tool [138] which augments the basic top-down approach with a cache of the generated Boolean subcircuits in order to prevent unnecessary quantifier instantiations. This technique can potentially save more quantifier instantiations than ours since it also takes identities obtained by constant folding into account. In contrast, we avoid

```
procedure solve($\varphi$, $f$, max_time):
  initialize_timer_for_timeout
  bounds := initial_bounds($\varphi$)
  found_solution := false
  best_value := f_max + 1
  lower_bound := f_min
  do
    CNF := generate_CNF($\varphi$, bounds)
    local_min := lower_bound
    do
      mid := local_min + (best_value − local_min) div 2
      (solved, solution)
        := call_SAT_solver(CNF, $f \leq$ mid, under−approximate)
      if solved then
        found_solution := true
        best_solution := solution
        best_value := mid
      else
        local_min := mid + 1
      end if
    while best_value > local_min

    local_max := best_value
    while lower_bound < local_max
      mid := lower_bound + (local_max − lower_bound) div 2
      (solved, solution)
        := call_SAT_solver(CNF, $f \leq$ mid, over−approximate)
      if not solved then
        lower_bound := mid + 1
      else
        local_max := mid
      end if
    end while

    increase_bounds(bounds)
  while not found_solution or timeout(max_time)
          or lower_bound = best_value

  return best_solution
end procedure
```

Figure 5.3: Procedure for solving with an objective function f

the overhead of cache misses by performing all translations of a subexpression at once. This also allows us to process subexpressions in a predictable order, which facilitates optimizations. Our observation that a bottom-up translation can be more efficient than a straightforward top-down approach has an analogue in the area of XPath query evaluation [75].

Another type of solver applicable to satisfiability problems with objective functions are MAX-SAT solvers, e.g., one of the solvers in the SAT4J [14] package. Since they are tailored to optimization, such solvers are likely to be more efficient than the technique of optimizing via assumptions that we use. Sugar [141] is a SAT-based constraint solver that also performs optimization over an assumption interface to a SAT solver. The Z3 [58] SMT solver supports a notion of so-called *contexts* that provide similar functionality as the assumption interface of MiniSat [63] and SAT4J [14].

Chapter 6

Extending OCL Operation Contracts with Objective Functions

OCL has been wisely conceived with executability in mind. The language omits constructs like unbounded quantifiers ranging over all integers that make expression evaluation intractable or entirely impossible. Recursion used for defining operations in postconditions is restricted by the OCL standard [125] to be finite, so uncomputable operations are avoided. The collection constructors and operations are designed in a way that prevents an uncontrolled explosion of collection size. As a result, evaluators for OCL that check the conformance of an implementation to its OCL specification at runtime could be implemented without resorting to sophisticated reasoning techniques (e.g., [83]). This is in contrast to other specification languages like Z [84] that offer more powerful constructs. For such languages constraint evaluation can be highly nontrivial, not to mention more difficult kinds of specification analysis like animation or test generation.

Naturally, the choice to restrict the expressive power of OCL comes at a price: some specification tasks may be impossible to accomplish or require considerably higher effort. Working around limitations of the language may also lead to specifications that obscure the problem that was to be specified originally. This may be one of the reasons why the use of OCL operation contracts appears to still not have gained widespread acceptance, although OCL has established itself as a language for model well-formedness rules and is also widely employed for queries in model-transformation and action languages.

A shortcoming of OCL in this respect that we identified is the difficulty to express optimization tasks. Such problems ask for operation results for which an objective function assumes an optimal value. Optimization problems arise naturally in application domains like operations research and constitute some of the most elementary algorithmic problems. A basic example is the problem of finding a shortest path in a graph. Figure 6.1 lists fundamental optimization algorithms covered in a classic introductory algorithms textbook [133]. Consequently, objective functions are widely used in mathematical modeling languages like AMPL [66] or GAMS [1]. Sugar [141] is an example of a constraint programming language that features objective functions.

We propose to facilitate the specification of optimization tasks in OCL by adding objective functions to operation contracts. Thus, rather than enriching the expression language of OCL, we introduce an additional constituent of operation contracts. Objective functions in operation contracts would provide immediate support in OCL for specifying optimization problems. Moreover, objective functions can also be used as a convenient means to specify that an operation should return a solution to a certain constraint if a solution exists: make the objective function evaluate whether the constraint holds and return the optimal value only if this is the case. Altogether, we think that this extension would make OCL operation contracts more attractive by facilitating the specification of many operations.

Objective functions strictly increase the expressiveness of OCL operation contracts. With the presence of an objective function, it is no longer decidable whether a set of returned operation results conforms to an operation contract. We propose to achieve tool interoperability by specifying objective functions in a UML profile. Existing tools can simply ignore the additional information represented by objective functions. Thus, existing applications of OCL are not compromised by the introduction of objective functions.

In the remainder of this chapter, we precisely define the syntax and semantics of this extension to OCL. We also discuss applications of objective functions to different specification tasks by means of an example specification.

- Closest pair among a set of points
- Minimum spanning tree of a graph
- Shortest path in a graph
- Maximum network flow
- Maximum matching of a graph
- Regression: Least Squares
- Knapsack problem
- Linear programming

Figure 6.1: Optimization algorithms covered in a classical algorithms textbook [133]

6.1 Syntax

Together with class invariants and different kinds of value definitions, operation contracts are a major ingredient of OCL specifications. Recall that an OCL operation contract consists of pre- and postconditions which are Boolean expressions. We propose to allow objective functions as a further element of operation contracts. Obviously, the type of an objective function must support comparison, so we restrict objective functions to expressions of type `Integer` or `Real`.[1] This corresponds to the restrictions imposed on body expressions of the `sortedBy` iterator. Furthermore, every operation contract may include at most one objective function.

Another useful extension proposed to OCL operation contracts are invariability clauses [96] that specify which parts of the system state an operation may modify. We will also consider this extension since invariability clauses are essential for animation support that we will discuss later.

In all, an OCL operation contract for an operation *op* with the arguments $x_1, \ldots, x_n$ with these extensions has the form:

$$\begin{aligned}
&\texttt{context}\ C\ ::\ \mathit{op}(x_1, \ldots, x_n) : T\\
&\quad\texttt{pre} : \phi(\mathit{self}, x_1, \ldots, x_n)\\
&\quad\texttt{pre} : \ldots\\
&\quad\texttt{post} : \psi(\mathit{self}, x_1, \ldots, x_n, \mathit{result})\\
&\quad\texttt{post} : \ldots\\
&\quad\texttt{minimize} : \theta(\mathit{self}, x_1, \ldots, x_n, \mathit{result})\\
&\quad\texttt{modifies only} : t_1(\mathit{self}, x_1, \ldots, x_n) :: a_1, \ldots, t_m(\mathit{self}, x_1, \ldots, x_n) :: a_m
\end{aligned} \tag{6.1}$$

Here, the OCL expressions ϕ and ψ are of type Boolean and θ is an OCL expression of type `Integer` or `Real`. The OCL expressions $t_1, \ldots, t_m$ denote sets of objects in the pre-state. We require that **`@pre`** does not occur in ϕ or $t_1, \ldots, t_m$. The operation contract (6.1) requires the operation to minimize the function θ. Thus, we extend the concrete syntax of operation contracts with the keyword **`minimize`**. Of course, a corresponding **`maximize`** keyword can be introduced as syntactic sugar as well. In short, the **`modifies only`** clause in (6.1) specifies that the operation may only change the attribute a_i for the objects in t_i. Attributes not mentioned in the **`modifies only`** clause may not be changed for any object. A richer syntax for **`modifies only`** clauses is presented in [96].

OCL operation contracts are often defined in UML models by storing OCL expressions as specifications of `Constraint` model elements. `Operation` elements can then reference such constraints through associations provided for by the UML metamodel. In order to avoid metamodel incompatibilities and to ensure the interoperability with OCL tools that do not use our operation contract extensions, we define the new operation contract elements in a UML profile. Figure 6.2 shows a UML profile for extending operation contracts with objective functions and invariability clauses. The objective function is stored as a string

[1] A further syntax extension could allow user-defined comparison functions.

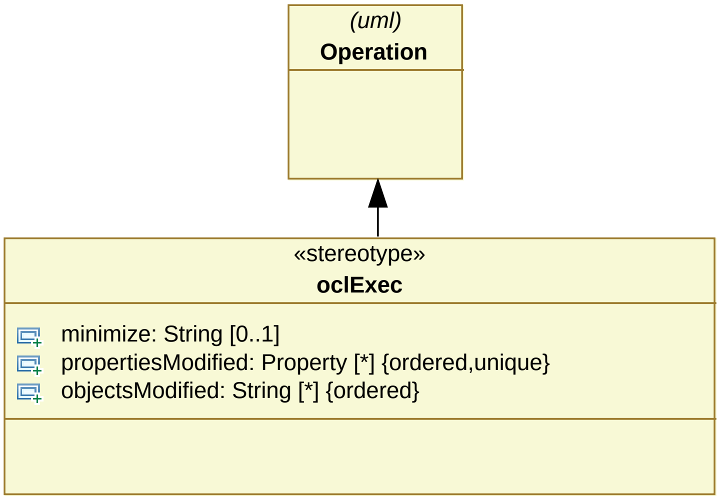

(uml)
Operation
«stereotype»
oclExec
minimize: String [0..1]
propertiesModified: Property [*] {ordered,unique}
objectsModified: String [*] {ordered}

Figure 6.2: A UML profile for operation contract extensions

and can be parsed when needed. The **`modifies only`** clause is defined by listing the expressions $t_1, \ldots, t_m$ and the attributes $a_1, \ldots, a_m$ in separate attributes of the stereotype.

It may be more systematic to rather model objective functions and **`modifies only`** clauses as separate stereotypes of the `Constraint` metaclass. However, the profile in Figure 6.2 is simple and can be applied easily in UML editors. A more comprehensive approach to extending OCL that is based on modularization instead of UML profiles has been presented in [3].

6.2 Semantics

When discussing the semantics of operation contracts we assume without loss of generality that there is exactly one precondition ϕ and one postcondition ψ. Following Annex A of the OCL standard [125], we define the *semantics* of an operation contract to be a relation R between pre-*environments* r_{pre} and post-environments r_{post}. An environment includes the system state and values of parameters to the operation. R is defined by

$$R = \{(r_{\text{pre}}, r_{\text{post}}) \mid \phi(r_{\text{pre}}) \wedge \psi\,(r_{\text{pre}}, r_{\text{post}})\}\,. \tag{6.2}$$

Thus, a transition from a pre- to a post-environment is permitted by the semantics if both the pre- and the postcondition are satisfied. The behavior of an operation implementation can be described by a function f that maps pre-environments to post-environments. The operation implementation *satisfies* the contract if and only if the graph of f is contained in R ($\text{graph}(f) \subseteq R$).

We show how this original definition of R can be modified in order to take an objective function θ into account. Let the contract require that the operation minimizes θ. This is expressed by the following definition of the modified semantics R':

$$\begin{aligned} R' = \{(r_{\text{pre}}, r_{\text{post}}) \ \mid \phi\left(r_{\text{pre}}\right) \wedge \psi\left(r_{\text{pre}}, r_{\text{post}}\right) \\ \wedge \quad \forall\, r'_{\text{post}}.\ \psi\left(r_{\text{pre}}, r'_{\text{post}}\right) \implies \theta\left(r_{\text{pre}}, r_{\text{post}}\right) \leq \theta\left(r_{\text{pre}}, r'_{\text{post}}\right)\}\,. \end{aligned} \tag{6.3}$$

Thus, a pair of a pre- and post-environment $(r_{\text{pre}}, r_{\text{post}})$ can only belong to R' if there is no other post-environment r'_{post} satisfying the postcondition with a smaller objective value. Hence, the objective function constrains the set of permitted transitions and may forbid transitions that are allowed by the original semantics R. Note that there only is a difference to the original semantics if the contract is underspecified, i.e., there is a valid pre-environment for which there is more than one post-environment satisfying the postcondition. Otherwise, $r'_{\text{post}} = r_{\text{post}}$ whenever $\psi\left(r_{\text{pre}}, r'_{\text{post}}\right)$, and the additional condition in (6.3) could never be violated. Thus, the objective function selects preferred post-states in case several are permitted by the postconditions.

Also note that the addition of the objective function increased the expressiveness of OCL operation contracts since the new semantics R' cannot in general be obtained by simply adding a postcondition $\theta\left(r_{\text{pre}}, r_{\text{post}}\right) \leq c\left(r_{\text{pre}}\right)$ for some OCL expression c. This only is an alternative if there exists such an expression c that computes the optimal value of the objective function from the pre-environment. However, this cannot always be the case, since the values of OCL expressions are effectively computable, but the existence of a post-environment r'_{post} violating (6.3) is in general undecidable. Even in cases in which it is possible to designate such an expression c, it is likely that this expression is much more complex than the addition of an objective function to the operation contract.

Finally, note that objective functions in operation contracts differ considerably from the `min` and `max` operations on collections that are provided by the OCL standard library. While these operations select the minimum and maximum from a finite collection, objective functions operate on the set of all possible post-states, which tends to be much larger than an OCL collection or can even be infinite.

If a subclass redefines the operation, the semantics of the redefined operation must conform to the Liskov substitution principle. Specifically, the set of possible post-states r_{post} that can result from calling the redefined operation in a certain pre-state r_{pre} satisfying the precondition ϕ must be a subset of the set of post-states reachable from this pre-state r_{pre} by calling the operation in the superclass. The objective function can be redefined in the subclass as long as this requirement is met.

The **`modifies only`** clause further constrains the semantics of the operation contract. However, unlike for objective functions, it is also possible to express this restriction through postconditions. Such a transformation is described in [96].

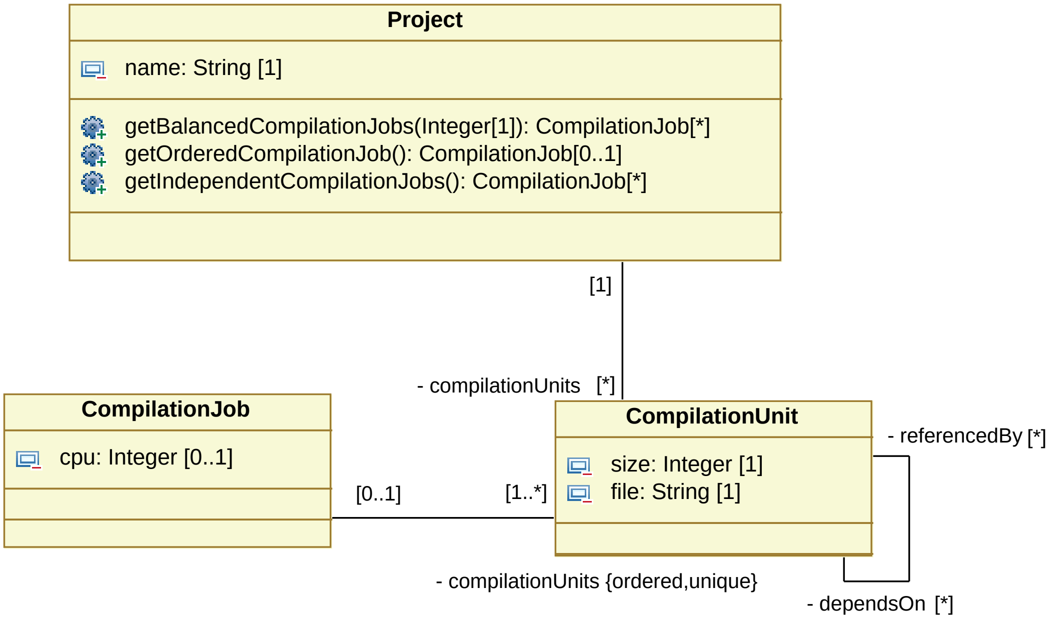

Project
name: String [1]
getBalancedCompilationJobs(Integer[1]): CompilationJob[*]
getOrderedCompilationJob(): CompilationJob[0..1]
getIndependentCompilationJobs(): CompilationJob[*]
[1]
- compilationUnits [*]
CompilationJob
cpu: Integer [0..1]
CompilationUnit
size: Integer [1]
file: String [1]
[0..1]
[1..*]
- referencedBy [*]
- compilationUnits {ordered,unique}
- dependsOn [*]

Figure 6.3: Excerpt from a possible UML model of a build tool

6.3 Applications

In this section we demonstrate the usefulness of objective functions by presenting several application examples. As running example we use the simplified model of a build tool shown in Figure 6.3. A `Project` comprises several `CompilationUnits`. In general, the task of the build tool is to arrange all `CompilationUnits` into adequate `CompilationJobs`. A `CompilationJob` designates an ordered sequence of `CompilationUnits` that are to be processed in order to complete the job. The assignment of `CompilationUnits` to `CompilationJobs` may be driven by various considerations. In particular, there can be dependencies between `CompilationUnits`, which is modeled by an association. Moreover, `CompilationUnits` can have different `sizes`.

6.3.1 Ordinary Optimization Problems

As first example, we consider a plain optimization task that is representative for many similar ones with an operations research background. The goal is to achieve efficient parallel processing of `CompilationJobs` by assigning `CompilationUnits` to `CompilationJobs` such that the longest execution time of any `CompilationJob` is minimized and the entire process is completed as early as possible. This is accomplished by the operation `getBalancedCompilationJobs` specified in Figure 6.4, which returns a set of `CompilationJobs` that arrange the

```
context Project::getBalancedCompilationJobs(n: Integer):
   Set(CompilationJob)

post allUnitsReturned:
   result->collect(compilationUnits)->asSet() = compilationUnits@pre

post jobLimitMet: result->size() <= n

minimize: let
             jobSizes: Bag(Integer)
                = result->collectNested(job |
                   job->compilationUnits
                      ->collectNested(size@pre)->sum())
          in
            jobSizes->max()

modifies only: compilationUnits::compilationJob
```

Figure 6.4: Operation contract for evenly distributing compilation units among processes

`CompilationUnits` of the `Project` such that total execution time is minimized. The parameter `n` indicates the maximal number of `CompilationJobs` created and would usually correspond to the number of CPUs available. The postcondition `allUnitsReturned` specifies that the set of `compilationUnits` included in the returned `CompilationJobs` are exactly the `CompilationJobs` of the `Project` for which the operation is called. The postcondition `jobLimitMet` limits the number of returned `CompilationJobs` to the argument passed to the operation. In this operation contract, most of the behavior is specified in the objective function. In the objective function, we first compute the total sizes of all returned `CompilationJobs` by applying `collectNested` and `sum`. We use `collectNested` here instead of `collect` in order to make explicit for readability that no flattening is performed. Then the value of the objective function is defined to be the maximum of all total job sizes, which is an estimate of the total execution time. Finally, the **`modifies only`** clause specifies that the attribute `compilationJob` may only be changed for the `compilationUnits` belonging to the `Project` for which the operation is called and that the values of all other attributes may not be affected by the operation. This invariability clause makes some previous **`@pre`** decorations superfluous, but we decided to keep these in order to already make explicit in the postconditions that these are values from the pre-state.

This problem of optimizing parallel execution is NP-hard since the subset sum problem can be reduced to it. Thus, even a relatively simple implementation is likely to be substantially more complex than the operation contract.

```
context Project::getOrderedCompilationJobs(): CompilationJob

post allUnitsReturned: (not result.oclIsUndefined())
                       implies
                       result.compilationUnits->asSet()
                          = compilationUnits@pre

post resultSorted:
   (not result.oclIsUndefined())
   implies
   let
      units: OrderedSet(CompilationUnit) = result.compilationUnits
   in
      Sequence{2..units->size()}
         ->forAll(i | Sequence{1..i-1}
           ->forAll(j | units->at(j).dependsOn@pre
                             ->excludes(units->at(i))))

minimize: if result.oclIsUndefined() then 1 else 0 endif

modifies only: compilationUnits::compilationJob
```

Figure 6.5: Operation contract for ordering compilation units by dependencies

6.3.2 Problems that do not always have a Solution

The task of the next operation `getOrderedCompilationJobs` that we specify is to find a compilation order that respects the dependencies of the `CompilationUnits`. In other words, we desire a topologically sorted sequence of the compilation units. The contract is shown in Figure 6.5. An interesting aspect of this operation is that there is no valid compilation order if the dependency graph has a cycle. In this case the operation should return `null`. The postcondition `allUnitsReturned` specifies that if the result is not `null`, i.e., a solution exists, then the set of `CompilationJobs` returned are exactly the `CompilationJobs` of the `Project`, as in the previous contract. The postcondition `resultSorted` states that, if the result is not `null`, no `CompilationUnit` depends on another unit occurring later in the returned sequence.

This operation does not solve an optimization problem at first sight. The objective function of the contract expresses that the operation should return a non-null result whenever possible, i.e., when it is not excluded by the postconditions. The presence of the objective function is essential for the contract to be complete, since otherwise an implementation that always returns `null` even if a valid compilation order exists would satisfy the contract. Note that it is not sufficient to simply add the additional postcondition **`post: not`** `result.oclIsUndefined()`, since this would make the postconditions unsatisfiable in case there is no valid compilation order, and the contract would not be implementable. An admissible

```
context Project::getIndependentCompilationJobs():
   Set(CompilationJob)

post allUnitsReturned:
   result->collect(compilationUnits)->asSet() = compilationUnits@pre

post jobsIndependent:
   result->forAll(compilationUnits->forAll(
      compilationUnits->includesAll(dependsOn@pre)))

minimize: -result->size() -- this maximizes the size of the result

modifies only: compilationUnits::compilationJob
```

Figure 6.6: Operation contract for grouping compilation units into independent jobs

alternative would be to add a precondition expressing that the postconditions are satisfiable, i.e., that the dependency graph is acyclic. This choice would be possible for this operation because the satisfiability of the postconditions is decidable in this case. It is clear, however, that such a precondition testing whether the dependency graph has a cycle would be much more lengthy then the objective function in Figure 6.5. Thus, the possibility of adding an objective function to the contract helped considerably to specify the operation in a concise and comprehensible manner. This technique of preferring non-null results by means of an objective function is applicable in general to problems that do not always have a solution.

6.3.3 Other Disguised Optimization Problems

Next we consider another task that is usually not regarded as an optimization problem. We are seeking a set of `CompilationJobs` that can be processed independently, e.g., for allowing parallel execution as discussed above for the operation `getBalancedCompilationJobs`. But this time we do not arrange the `CompilationUnits` based on their size but according to their dependencies. We require that there is no dependency between any two `CompilationUnits` belonging to distinct `CompilationJobs`. In order to be as flexible as possible for execution, we desire to have as many independent `CompilationJobs` as possible. This amounts to finding the connected components of the dependency graph. The operation `getIndependentCompilationJobs`, whose contract is shown in Figure 6.6, is specified to return a set of `CompilationUnits` that meets these requirements. As for the previous operations, the first postcondition expresses that the `CompilationUnits` included in the returned `CompilationJobs` are exactly the `CompilationJobs` of the `Project`. The next postcondition `jobsIndependent` states that every `CompilationJob` returned includes all `CompilationUnits` that depend on any unit in the job. This rules out any dependencies between any two `CompilationUnits` belonging to distinct `CompilationJobs`.

In order to ensure that the returned `CompilationJobs` correspond to the connected components of the dependency graph, we still need to specify that there actually is a dependency between every pair of `CompilationUnits` that belong to the same `CompilationJob`. Otherwise, an implementation may always return just a single `CompilationJob` that includes all `CompilationUnits`. However, we find this requirement difficult to express using postconditions, since two `CompilationUnits` may depend on each other via several other intermediate `CompilationUnits`. This is where the objective function comes in handy. The objective function defined in Figure 6.6 requires the operation to return a maximal number of `CompilationJobs`. This implies that every job corresponds to exactly one connected component of the dependency graph. Thus, adding an objective function to the contract helped again to keep the contract simple.

Chapter 7

Undefined Values in OCL

From its beginnings, OCL has been equipped with the notion of an undefined value (called `OclUndefined` in previous versions of the standard, e. g., [123]; in recent versions of the OCL standard, e. g., [125], this constant is called **`invalid`**) to deal with exceptions occurring during expression evaluation. A classical example of such an exception is a division by zero. In OCL such an erroneous division is specified to yield an undefined value. Other reasons for exceptions include attempts to retrieve elements from empty collections, illegal type conversions and attribute calls on objects that do not exist. Most operations in OCL are defined to be *strict*, i. e., they evaluate to **`invalid`** if they are called with an undefined argument. This ensures that errors are propagated during expression evaluation so they are visible and can be handled later on. Naturally, OCL collections are not allowed to have undefined elements, since errors are more easily signaled by marking the entire collection value as undefined.

During the development of OCL, it became clear that convenient object-oriented navigation required a second exception element in addition to **`invalid`**. This second exception element, called **`null`**, represents the *absence of value* rather than indicating evaluation errors. The need to express the absence of value arises naturally when dealing with object attributes with a multiplicity of zero or one. These attributes, that occur frequently in models, are not required to yield a value when evaluated. Representing this absence of value with the original undefined value **`invalid`** would be inconvenient and counter-intuitive. To prevent a propagation of undefined values, it would be necessary to handle all cases of value absence immediately. In particular, it would not be possible to pass potentially null values to strict operations. Since nearly all operations of OCL are strict, even the most basic operations such as equality testing would not be realizable without checking for an absence of value. These difficulties can be avoided by introducing the **`null`** element as a valid operation argument and collection element.

Recent versions of OCL 2.0 standard [124] (and later versions of OCL, in particular OCL 2.2 [125]) introduce **`null`** as a second exception element representing the absence of a value. Unfortunately, this extension has been done in an ad hoc manner, which results in several inconsistencies and ambiguities. For example, the standard does not exactly define when `null` operation arguments are treated as exceptions, i. e., lead to `invalid` operation results. It is not clear when object attributes can evaluate to `null` values and how this depends on the multiplicity of the attribute. There is also no indication in the standard whether objects that do not exist ("dangling references") are treated the same way as `null` or not. Unsurprisingly, an evaluation [73] of OCL tools identified the handling of undefined values as a major weakness of most tools.

In this chapter we propose a formal semantics that overcomes these problems in the current version of the OCL standard. We build on the "HOL-OCL" approach [33] of defining OCL semantics by means of an embedding of OCL into HOL. We first provide a summary of the essentials of the HOL-OCL semantics as it could be found in textbooks. Nevertheless, our semantics is a strong formal semantics largely following [123, Annex A]. Then we present as an increment our proposal for OCL 2.2 [125], focusing on the key issue of null-elements and null-types. In particular we analyze the consequences for an omnipresent feature of UML, namely multiplicities, and its pragmatics. Finally, we discuss some problematic aspects of the OCL standard that concern undefined values.

7.1 An Overview over OCL Semantics

In this section, we will briefly introduce to OCL semantics from the HOL-OCL perspective. The main differences between the OCL 2.0 formal semantics description [123, Annex A] and HOL-OCL are

1. that the latter is a machine-checked, "strong" formal semantics which is itself based

2. on a typed meta-language (i. e., HOL) instead of an untyped one (i. e., naïve set theory), and

3. various technical simplifications: instead of three different semantic interpretation functions $I(x)$, $I[\![e]\!]\tau$, and $I_{\mathrm{ATT}}[\![e]\!]\tau$, we use only one.

The first difference enables us to give a semantic consistency guarantee: Since all definitions of our formal semantics are conservative extensions, the consistency of HOL-OCL is reduced to the consistency of HOL, i. e., a widely accepted small system of seven axioms. The second difference dramatically reduces the number of rules necessary for formal reasoning.

For modeling undefined values in OCL, we will make use of the type constructor $\alpha_\bot := \bot \mid \lfloor\_\rfloor : \alpha$ provided by the HOL library that assigns to each type α a type $\alpha_\bot$ *disjointly extended* by the exceptional element $\bot$. The function $\ulcorner\_\urcorner : \alpha_\bot \Rightarrow \alpha$ is the inverse of $\lfloor\_\rfloor$ (unspecified for $\bot$). Partial functions $\alpha \rightharpoonup \beta$ are defined as functions $\alpha \Rightarrow \beta_\perp$ supporting the usual concepts of domain (dom _) and range (ran _).

7.1.1 Valid Transitions and Evaluations

We recall that OCL expressions (which we will describe below in more detail) form a typed assertion language whose syntactic elements are composed of

1. operators on built-in data structures such as Boolean or collection types like `Set`,
2. operators of the user-defined data-model such as attribute accessors, type-casts and tests, and
3. calls to user-defined, potentially recursive, side-effect-free methods.

The topmost goal of the formal semantics for OCL expressions is to define the notion of a *valid transition* over system states; even concepts like *object invariants* can be derived from this notion. Let σ be a pre-state and σ' a post-state and let and ϕ a Boolean OCL expression, then we write

$$(\sigma, \sigma') \vDash \phi$$

for "the transition from σ to σ' is valid in ϕ." A formula ϕ is valid if and only if its evaluation in the context (σ, σ') yields true. As all types in HOL-OCL are extended by the special element $\perp$ denoting undefinedness, we define formally:

$$(\sigma, \sigma') \vDash \phi \equiv \left(I[\![\phi]\!](\sigma, \sigma') = \llcorner \mathrm{true} \lrcorner \right) .$$

Since all operators of the assertion language depend on the context (σ, σ') and result in values that can be $\perp$, all expressions can be viewed as *evaluations* from (σ, σ') to a type $\alpha_\perp$. All types of expressions have a form captured by the following type abbreviation:

$$V(\alpha) := \sigma \times \sigma \Rightarrow \alpha_\perp ,$$

where σ stands for the type of system states and _ := _ denotes type abbreviation. In the following, we will use the abbreviation $\tau = (\sigma, \sigma')$ whenever there is no need to refer to the pre-state and post-state of a state transition explicitly.

The OCL semantics [123, Annex A] uses different interpretation functions for invariants and pre-conditions; instead, we achieve their semantic effect by a syntactic transformation $_{\mathrm{pre}}$ which replaces all accessor functions _. a by their counterparts _. a `@pre`. For example, $(\mathit{self}.\,\mathrm{a} > 5)_{\mathrm{pre}}$ is just $(\mathit{self}.\,\mathrm{a}\,$`@pre`$\, > 5)$. The operation _`.allInstances()` is also substituted by its `@pre` counterpart.

Thus, we can re-formulate the semantics of the two OCL top-level constructs, invariant specification and method specification, as follows:

$$\begin{aligned} &I[\![\texttt{context}\ c : C\ \texttt{inv}\ n : \phi(c)]\!]\tau \equiv \\ &\quad \tau \vDash (C\ \texttt{.allInstances()-->forall(}x\text{—}\phi(x)\texttt{))} \wedge \\ &\quad \tau \vDash (C\ \texttt{.allInstances()-->forall(}x\text{—}\phi(x)\texttt{))}_{\text{pre}}\,. \end{aligned} \tag{7.1}$$

The standard forbids expressions containing _ `@pre` in invariants or preconditions syntactically; thus, mixed forms can not arise. The semantics for a specification of a *op* with the arguments $a_1, \ldots, a_n$ reads as follows:

$$\begin{aligned} &I[\![\texttt{context}\ C\ :: op(a_1, \ldots, a_n) : T \\ &\qquad \texttt{pre}\ \phi(\mathit{self}, a_1, \ldots, a_n) \\ &\qquad \texttt{post}\ \psi(\mathit{self}, a_1, \ldots, a_n, \mathit{result})]\!]\tau \equiv \\ &\quad \forall s, x_1, \ldots, x_n. \\ &\qquad \Delta(s, x_1, \ldots, x_n) \wedge \tau \vDash \phi(s, x_1, \ldots, x_n)_{\text{pre}} \\ &\qquad\quad \rightarrow \tau \vDash \psi(s, x_1, \ldots, x_n, s.op(x_1, \ldots, x_n)) \\ &\quad\ \wedge \neg\Delta(s, x_1, \ldots, x_n) \\ &\qquad\quad \rightarrow \tau \vDash s.op(x_1, \ldots, x_n) \triangleq \texttt{invalid} \end{aligned} \tag{7.2}$$

where $\Delta(s, x_1, \ldots, x_n)$ is an abbreviation for

$$\tau \vDash \texttt{not}\ s\ \texttt{.oclIsInvalid()} \wedge \ldots \wedge \tau \vDash \texttt{not}\ x_n\ \texttt{.oclIsInvalid()}$$

and where the symbol $\_ \triangleq \_$ stands for "strong equality" which will be formally introduced in Section 7.1.3 together with the constant `invalid` and its test for it.

This definition captures two cases: if the arguments of an operation are defined, the result of a method call must satisfy the specification; otherwise the operation will be strict and return invalid. Note that we *specify* the interpretation function for operation calls occurring in OCL expressions this way: $I[\![s.op(a_1, \ldots, a_n)]\!]\tau$ must be chosen such that it satisfies the constraints of the postconditions.

Summing up, by these definitions an OCL specification, i. e., a sequence of invariant declarations and operation contracts, can be transformed into a set of (logically conjoined) statements which is called the *context* Γ_τ. The *theory* of an OCL specification is the set of all valid transitions $\tau \vDash \phi$ that can be derived from Γ_τ. For the logical connectives of OCL , a conventional Gentzen-style calculus for pairs of the form $\Gamma_\tau \vdash \phi$ can be developed that allows for inferring valid transitions from Γ_τ by deduction (cf. [33, 30])). Due to the inclusion of arithmetic, any calculus for OCL is necessarily incomplete. Note further that it is straight-forward to extend our notion of context to multi-transition contexts such as:

$$\Gamma \equiv \{(\sigma, \sigma') \vDash \phi, (\sigma', \sigma'') \vDash \psi\}\,.$$

7.1.2 Strict Operations

Following common terminology, an operation that returns $\bot$ if one of its arguments is $\bot$ is called *strict*. The majority (including all user-defined operations, e. g., defined in the underlying class model) of operations is strict. For example, the Boolean negation is formally presented as:

$$I[\![\texttt{not}\ X]\!]\tau \equiv \begin{cases} \llcorner \neg \ulcorner I[\![X]\!]\tau \urcorner \lrcorner & \text{if } I[\![X]\!]\tau {\neq} \bot, \\ \bot & \text{otherwise,} \end{cases}$$

where $\tau = (\sigma, \sigma')$ and $I[\![\_]\!]$ is a notation marking the OCL constructs to be defined. This notation is motivated by our goal to achieve the maximal textual similarity to the textbook-style semantics in the OCL standard [123].

The binary case of the integer addition is analogous:

$$I[\![X + Y]\!]\,\tau \equiv \begin{cases} \llcorner \ulcorner X' \urcorner + \ulcorner Y' \urcorner \lrcorner & \text{if } X' {\neq} \bot \text{ and } Y' {\neq} \bot, \\ \bot & \text{otherwise,} \end{cases}$$

where X' is an abbreviation for $I[\![X]\!]\tau$ and Y' for $I[\![Y]\!]\tau$. The operator $\_ {+} \_$ on the right refers to the integer HOL operation with type $[\text{int}, \text{int}] \Rightarrow \text{int}$. The type of the corresponding strict OCL operator $\_ {+} \_$ is $[V(\text{int}), V(\text{int})] \Rightarrow V(\text{int})$. From the bulk of definitions of this kind, a large number of theorems (derived rules) such as associativity, commutativity, etc. were formally proven in HOL-OCL (omitted here).

A variation of this definition scheme is used for the operators on collection types such as OCL sets or sequences:

$$I[\![X\texttt{-->union}(Y)]\!]\tau \equiv \begin{cases} S \llcorner \ulcorner X' \urcorner {\cup} \ulcorner Y' \urcorner \lrcorner & \text{if } X' {\neq} \bot \text{ and } Y' {\neq} \bot, \\ \bot & \text{otherwise,} \end{cases}$$

where $X' \equiv I[\![X]\!]\,\tau$ and $Y' \equiv I[\![Y]\!]\,\tau$. Here, S ("smash") is a function that maps a lifted set $\llcorner X \lrcorner$ to $\bot$ if and only if $\bot \in X$ and is the identity otherwise. Smashedness of collection types is the natural extension of the strictness principle for data structures.

Intuitively, the type expression $V(\tau)$ is a representation of the type that corresponds to the OCL type τ. We introduce the following type abbreviations:

$$\texttt{Boolean} := V(\text{bool})\,, \qquad \texttt{Set}(\alpha) := V(\alpha\,\text{set})\,,$$
$$\texttt{Integer} := V(\text{int})\,, \text{ and} \qquad \texttt{Sequence}(\alpha) := V(\alpha\,\text{list})\,.$$

The mapping of an expression `E` of OCL type `T` to an expression E of type T in our meta-language HOL is injective and preserves well-typedness.

7.1.3 Boolean Operators

There is a small number of explicitly stated exceptions from the general rule that OCL operators are strict: the strong equality, the undefinedness operator and the logical connectives. As a prerequisite, we define the logical constants for truth, absurdity and undefinedness as follows:

$$\begin{aligned} I[\![\texttt{true}]\!]\tau &\equiv \llcorner\text{true}\lrcorner\,, \\ I[\![\texttt{false}]\!]\tau &\equiv \llcorner\text{false}\lrcorner\,, \text{ and} \\ I[\![\texttt{invalid}]\!]\tau &\equiv \bot\,. \end{aligned}$$

OCL has a *strict equality*, written $\_ = \_$, which we denote $\_ \doteq \_$ throughout this paper. On the primitive types, it is defined similarly to the integer addition; the case for objects is discussed later. For logical purposes, we introduce also a *strong equality* $\_ \triangleq \_$ which is defined as follows:

$$I[\![X \triangleq Y]\!]\,\tau \equiv (I[\![X]\!]\,\tau = I[\![Y]\!]\,\tau)\,,$$

where the $\_ = \_$ operator on the right denotes the logical equality of HOL. The undefinedness test is defined by X `.oclIsInvalid()` $\equiv (X \triangleq \texttt{invalid})$. Strong equality can be explained as a syntactic abbreviation via strict equality and undefinedness:

$$\begin{aligned} X \triangleq Y \equiv & \\ & (X \doteq Y \;\texttt{or}\; (X \doteq Y)\,\texttt{.oclIsInvalid()})\;\texttt{and} \\ & \qquad (X\,\texttt{.oclIsInvalid()} \doteq Y\,\texttt{.oclIsInvalid()}) \end{aligned} \tag{7.3}$$

The major purpose of strong equality is a concise formulation of the Leibniz rule (equals may be substituted in any context) adapted to OCL :

$$\frac{\tau \vDash e \triangleq e' \in \Gamma \qquad \Gamma \vdash \tau \vDash \phi(e)}{\Gamma \vdash \tau \vDash \phi(e'),}$$

This is the foundation of term-rewriting in OCL. The equivalence symbol in Equation 7.2 is also syntactically equivalent to strong equality (on Boolean).

The strong equality can be used to state reduction rules like: $\tau \vDash (\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$. The OCL standard requires a Strong Kleene Logic. In particular:

$$I[\![X \;\texttt{and}\; Y]\!]\tau \equiv \begin{cases} \llcorner \ulcorner X' \urcorner \wedge \ulcorner Y' \urcorner \lrcorner & \text{if } X' \neq \bot \text{ and } Y' \neq \bot, \\ \llcorner\text{false}\lrcorner & \text{if } X' \text{ or } Y' \text{ are } \llcorner\text{false}\lrcorner, \\ \bot & \text{otherwise,} \end{cases}$$

where $X' \equiv I[\![X]\!]\tau$ and $Y' \equiv I[\![Y]\!]\tau$. Thus, the Boolean operation $\_\;\texttt{and}\;\_$ has type $[V(\text{bool}), V(\text{bool})] \Rightarrow V(\text{bool})$.

The other Boolean connectives are just shortcuts:

$$\begin{aligned} X \;\texttt{or}\;\; Y &\equiv \texttt{not}\,(\texttt{not}\; X \;\texttt{and}\,\texttt{not}\; Y), \text{and}\\ X \;\texttt{implies}\; Y &\equiv \texttt{not}\; X \;\texttt{or}\;\; Y\,. \end{aligned}$$

The logical quantifiers are viewed as special operations on the collection types $\texttt{Set}(\alpha)$ or $\texttt{Sequence}(\alpha)$. Their definition in the OCL standard is very operational and restricted to the finite case; instead, we define the universal quantification as generalization of the conjunction:

$$I[\![X\texttt{-->forall}(x \mid P(x))]\!]\,\tau \equiv$$
$$\begin{cases} \bot & \text{if } X' = \bot,\\ \llcorner\forall a \in \ulcorner X'\urcorner.\ \ulcorner p(a)\urcorner\lrcorner & \text{if } \forall a \in \ulcorner X'\urcorner.\ p(a){\neq}\bot,\\ \llcorner\text{false}\lrcorner & \text{if } \exists a \in \ulcorner X'\urcorner.\ p(a) = \llcorner\text{false}\lrcorner,\\ \bot & \text{otherwise,} \end{cases}$$

where $X' \equiv I[\![X]\!]\tau$ and $p(a) \equiv I[\![P(\lambda\,\tau.\ a)]\!]\tau$. As usual, the existential quantification is introduced as abbreviation:

$$X\texttt{-->exists}(x \mid P(x))$$
$$\equiv \texttt{not}\;\; X\texttt{-->forall}(x \mid \texttt{not}\;\; P(x))\,.$$

7.1.4 Object-oriented Data Structures

Above we described various built-in operations on datatypes and the logic. Now we turn to several families of operations that the user implicitly defines when stating a class model as logical context of a specification. This is the part of the language where object-oriented features such as type casts, accessor functions, and tests for dynamic types come into play. Syntactically, a class model provides a collection of classes C, an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes. Semantically, a class model means a collection of accessor functions (denoted $\_.\mathrm{a} :: A \to B$ and $\_.\,\mathrm{a}\,\texttt{@pre} :: A \to B$ for $\mathrm{a} \in A$ and $A, B \in C$), type casts that can change the static type of an object of a class (denoted $\_\texttt{.oclAsType}(C)$ of type $A \to C$) and dynamic type tests (denoted $\_\texttt{.oclIsTypeOf}(C)$). A precise formalization of the syntactic side of a class system can be found in [32].

Class Models: A Simplified Semantics

We now clarify the notions of *object identifiers*, *object representations*, *class types* and *state*.

First, object identifiers are captured by just an abstract type oid comprising countably many elements and a special element **`nullid`**.

Second, object representations model "a piece of typed memory," i. e., a kind of record comprising some administrative information and the information for all the attributes of an object; here, the basic types `Boolean`$_\tau$, `Integer`$_\tau$, etc. as well as collections over them are stored directly in the object representations, class types and collections over them are represented by oid's (respectively lifted collections over them; such collections may be $\bot$).

Third, the class type C will be the type of such an object representation. It is a Cartesian product:

$$C := (\mathrm{oid} \times \mathrm{C}_\mathrm{t} \times A_1 \times \cdots \times A_k)$$

where a unique tag-type C_t (ensuring type-safety) is created for each class type, and where the types $A_1, \ldots, A_k$ are the attribute types (including all inherited attributes) with class types substituted by the type oid. The function OidOf projects the first component, the oid, out of an object representation.

Fourth, for a class model M with the classes $C_1, \ldots, C_n$, we define states as partial functions from oids to object representations satisfying a *state invariant* inv_σ:

$$\sigma := \{f :: \mathrm{oid} \rightharpoonup (C_1 + \ldots + C_n) \mid \mathrm{inv}_\sigma(f)\}$$

where $\mathrm{inv}_\sigma(f)$ states two conditions:

1. there is no object representation for `nullid`:

   $$\texttt{nullid} \notin (\mathrm{dom}\, f)\,.$$

2. there is a "one-to-one" correspondence between object representations and object identifiers (oid):

   $$\forall \mathit{oid} \in \mathrm{dom}\, f.\ \mathit{oid} = \mathrm{OidOf}\ \ulcorner f(\mathit{oid})\urcorner\,.$$

The latter condition is also mentioned in [123, Annex A] and goes back to Mark Richters [130].

7.1.5 The Accessors

On states built over object universes we can now define accessors, casts, and type tests of an object model. We consider the case of an attribute a of a class C which has the simple class type D (not a basic type, not a collection):

$$I[\![\mathit{self}\,.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } o = \bot \lor \mathit{id} \notin \mathrm{dom}\ \sigma' \\ \mathrm{get}_\mathrm{D}\ u & \text{if } \sigma'(\mathrm{get}_\mathrm{C}\ulcorner\sigma'\ \mathit{id}\urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise;} \end{cases}$$

and for accessing the attribute in the previous state:

$$I[\![\mathit{self}\,.\,\mathrm{a@pre}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } o = \bot \vee \mathit{id} \notin \mathrm{dom}\ \sigma \\ \mathrm{get}_{\mathrm{D}}\ u & \text{if } \sigma(\mathrm{get}_{\mathrm{C}} \ulcorner \sigma\ \mathit{id} \urcorner .\, \mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise,} \end{cases}$$

where $o = I[\![\mathit{self}]\!](\sigma,\sigma')$ and $\mathit{id} = \mathrm{OidOf} \ulcorner o \urcorner$. Here, $\mathrm{get}_{\mathrm{D}}$ is the projection function from the object universe to $D_\bot$, and $x.\,\mathrm{a}$ is the projection of the attribute from the class type (the Cartesian product). In the case of a class type, we have to evaluate the expression *self*, get an object representation (or `invalid` if the evaluation is not possible), project the attribute, de-reference it in the pre or post state, respectively, and project the class object from the object universe ($\mathrm{get}_{\mathrm{D}}$ may yield $\bot$ if the element in the universe does not correspond to a D object representation). In the case of a basic type attribute, the de-referentiation step is left out.

In our model accessors always yield (type-safe) object representations; not oids. This has the consequence that a reference that is *not* in dom σ, i. e., that is a "dangling reference," immediately results in `invalid` (this is a subtle difference to [123, Annex A] where undefinedness is detected one de-referentiation step later). The strict equality $\_ \doteq \_$ must be defined via OidOf when applied to objects. It satisfies $(\texttt{invalid} \doteq X) \triangleq \texttt{invalid}$.

The definitions of casts and type tests are straightforward and can be found in [32], together with other details of the construction above and its automation in HOL-OCL. Strict equality of objects that are not undefined amounts to a comparison of the respective oids of their object representations.

7.2 A Formal Semantics for OCL 2.2

In this section, we describe our proposal for a formal semantics of OCL 2.2 [125] as an increment to the OCL 2.0 semantics (currently underlying HOL-OCL and essentially formalizing [123, Annex A]). In later versions of the standard [125], the semantics annex is only slightly updated and does not reflect all changes made to the mandatory parts of the standard.

7.2.1 Revised Operations on Basic Types

In UML, and since [124] also in OCL, all basic types comprise the `null`-element, modeling a possible absence of value. Seen from a functional language perspective, this corresponds to the view that each basic value is a type like `int option` as in SML [126]. Technically, this results in lifting any basic type twice:

$$\texttt{Integer} := V(\mathrm{int}_\bot)\,, \text{ etc.}$$

and basic operations have to take the null elements into account. The distinguishable **invalid** and **null** elements are defined as follows:

$$I[\![\texttt{invalid}]\!]\tau \equiv \bot \quad \text{and} \quad I[\![\texttt{null}]\!]\tau \equiv \llcorner\bot\lrcorner\,.$$

As example for elementary constants, we present:

$$I[\![\texttt{true}]\!]\tau \equiv \llcorner\llcorner\text{true}\lrcorner\lrcorner \quad \text{and} \quad I[\![\texttt{false}]\!]\tau \equiv \llcorner\llcorner\text{false}\lrcorner\lrcorner\,.$$

Consistent with the OCL 2.2 standard [125], we give an interpretation such that $\texttt{null}+3 = \texttt{invalid}$, and due to commutativity, we postulate $3+\texttt{null} = \texttt{invalid}$, too. The necessary modification of the semantic interpretation reads as follows:

$$I[\![X+Y]\!]\,\tau \equiv \begin{cases} \llcorner\llcorner \ulcorner\ulcorner X' \urcorner\urcorner + \ulcorner\ulcorner Y' \urcorner\urcorner \lrcorner\lrcorner & \text{if } X', Y' \notin \{\bot, \llcorner\bot\lrcorner\} \\ \bot & \text{otherwise}\,, \end{cases}$$

where $X' \equiv I[\![X]\!]\,\tau$ and $Y' \equiv I[\![Y]\!]\,\tau$. The resulting principle here is that operations on the primitive types Boolean, Integer, Real, and String treat **null** as **invalid** (except $\_ \triangleq \_$, $\_ \doteq \_$, $\_\,\texttt{.oclIsInvalid()}$, $\_\,\texttt{.oclIsUndefined()}$, and type-tests).

This principle is motivated by our intuition that invalid represents known errors, while null-arguments of operations for Boolean, Integer, Real, and String are *optional* data. Thus, we must also modify the logical operators such that

$$\texttt{null and false} \triangleq \texttt{false}$$

and, in an analogous case,

$$\texttt{null and true} \triangleq \texttt{invalid}$$

holds. As another consequence of this principle, we define

$$X\,\texttt{.oclIsUndefined()} \equiv (X \triangleq \texttt{invalid}) \quad \texttt{or} \quad (X \triangleq \texttt{null})\,.$$

Now, the question arises how the test for **invalid** has to be redefined in an OCL 2.2 setting. We see two options:

1. Since the standard currently requires:

$$\texttt{null}\,\texttt{.oclIsInvalid()} \triangleq \texttt{invalid}\,,$$

   an awkward definition would be:

$$\begin{aligned} X\,\texttt{.oclIsInvalid()} \equiv\ & \texttt{if}\, X \triangleq \texttt{null} \\ & \texttt{then}\, \texttt{invalid} \\ & \texttt{else}\ X \triangleq \texttt{invalid} \\ & \texttt{endif} \end{aligned}$$

2. Easier to handle from a deductional point of view would be:

$$X\ \texttt{.oclIsInvalid()} \equiv X \triangleq \texttt{invalid},$$

which also maintains Equation 7.3.

7.2.2 Null in Class Types

It is a viable option to rule out inherent undefinedness in object graphs *as such*. The source of such undefinedness are oids which do not occur in the state, i. e., which represent "dangling references." Ruling out potentially undefined object accessors would correspond to a world where constructors always set attributes to `null` or an object that is not undefined and to a programming language without explicit deletion (as, for instance, in Spec# [9]). Semantically, this can be modeled by strengthening the state invariant inv_σ by adding clauses that state that in each object representation all oids are either `nullid` or element of the domain of the state. We deliberately decided against this option for the following reasons:

1. *methodologically* we do not like to constrain the semantics of OCL without clear reason; in particular, "dangling references" exist in C and C++ programs and it might be necessary to write contracts for them, and

2. *semantically*, the condition "no dangling references" can only be formulated with the complete knowledge of all classes and their layout in form of object representations. This restricts the OCL semantics to a closed world model.[1]

We can model `null`-elements as object representations with `nullid` as their oid:

Definition 7.1 (Representation of `null`-Elements)**.** Let C_i be a class type with the attributes $A_1, \ldots, A_n$. Then we define its null object representation by:

$$\mathrm{nullrep}_{Ci} := \llcorner(\texttt{nullid}, \mathrm{arb}_\mathrm{t}, a_1, \ldots, a_n)\lrcorner$$

where the a_i are $\perp$ for primitive types and collection types, and `nullid` for simple class types. The term arb_t is an arbitrary underspecified constant of the tag-type.

[1] In our presentation, the definition of `state` in Section 7.1 assumes a closed world. This limitation can be easily overcome by leaving "polymorphic holes" in our object representation universe, i. e., by extending the type sum in the state definition to $C_1 + \cdots + C_n + \alpha$. The details of the management of universe extensions are involved, but implemented in HOL-OCL (see [32] for details). However, these constructions exclude knowing the set of "reachable oids" in advance.

7.2.3 Revised Accessors

Having introduced `null`-elements, the modification of the accessor functions is now straight-forward:

$$I[\![\mathit{obj}\,.\,\mathrm{a}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \begin{array}{l}\text{if } A = \bot,\\ \text{or } \mathit{id} \notin \operatorname{dom} \sigma'\end{array} \\ \mathrm{nullrep}_D & \text{if } \mathrm{get}_{\mathrm{C}} \ulcorner \sigma'(\mathit{id}) \urcorner.\,\mathrm{a}^{(0)} = \mathtt{nullid} \\ \mathrm{get}_{\mathrm{D}}\, u & \text{if } \sigma'(\mathrm{get}_{\mathrm{C}} \ulcorner \sigma'(\mathit{id}) \urcorner.\,\mathrm{a}^{(0)}) = \llcorner u \lrcorner, \\ \bot & \text{otherwise,} \end{cases}$$

where $A \equiv I[\![\mathit{obj}]\!](\sigma,\sigma')$ and $\mathit{id} \equiv \mathrm{OidOf}\ulcorner A \urcorner$.

The definitions for type cast and dynamic type test, which are not explicitly shown in this thesis (see [32] for details) can be generalized accordingly. In the sequel, we will discuss the resulting properties of these modified accessors.

Accessors, type casts and type tests are strict:

$$\mathtt{invalid}.\,\mathrm{a} \triangleq \mathtt{invalid}$$
$$\mathtt{invalid}.\mathtt{oclAsType}(C) \triangleq \mathtt{invalid}$$
$$\mathtt{invalid}.\mathtt{oclIsTypeOf}(C) \triangleq \mathtt{invalid}$$

Furthermore, the following rule schemes express that the dynamic type remains unchanged while casting:

$$\mathit{obj}.\mathtt{oclAsType}(B).\mathtt{oclIsTypeOf}(A) = \mathit{obj}.\mathtt{oclIsTypeOf}(A)$$

Moreover, we can "re-cast" an object safely, i. e., up and down casts are idempotent. Casting an object deeper in the subclass hierarchy than its dynamic type results in invalid. Furthermore, casting is transitive:

$$\frac{\tau \vDash \mathit{obj}.\mathtt{oclIsTypeOf}(B)}{\tau \vDash \big((\mathit{obj}.\mathtt{oclAsType}(A)).\mathtt{oclAsType}(B)\big) \triangleq \mathit{obj}}$$

$$\frac{\tau \vDash \mathit{obj}.\mathtt{oclIsTypeOf}(A)}{\tau \vDash \mathit{obj}.\mathtt{oclAsType}(B) \triangleq \mathtt{invalid}}$$

$$\frac{\tau \vDash \mathit{obj}.\mathtt{oclIsTypeOf}(C)}{\tau \vDash \begin{array}{l}(\mathit{obj}.\mathtt{oclAsType}(B)).\mathtt{oclAsType}(A)\\ \triangleq \mathit{obj}.\mathtt{oclAsType}(A)\end{array}}$$

where we assume classes A, B, C with $C < B < A$.

7.2.4 Null and Collection Types

According to the OCL 2.2 standard, both `null` and `invalid` also belong to the collection types. Moreover, `null` is admissible as collection element. In contrast, `invalid` is not permitted as collection element, since exceptions are more easily signaled by marking the entire collection value as undefined. Hence, we obtain, e. g., the OCL `Set` type with integer elements as:

$$\texttt{Set(Integer)} := V(\mathrm{int}_\bot \ \mathrm{set})$$

The element type of the set is only lifted once, since `invalid` collection elements are not permitted.

This raises the question how collection operations should behave when called with undefined values as arguments. Since collection operations cannot insert `invalid` into a collection, it appears reasonable that they return `invalid` when called with `invalid` arguments in order to propagate the exception. On the other hand, when called with `null` arguments, collection operations should not necessarily evaluate to `invalid`, since otherwise `null` could not be inserted into collections, and their presence would be cumbersome to detect. Hence, collection operations should be strict with respect to `invalid`, but not respect to `null`. Consider as example the `->includes` operation for testing membership in a collection. Following our previous reasoning concerning strictness, we obtain the following definition of this operation:

$$\begin{aligned} &I[\![a\texttt{-->includes}(X)]\!]\,\tau \\ &\qquad\qquad \equiv \begin{cases} \llcorner\llcorner \ulcorner X' \urcorner \in \ulcorner\ulcorner A \urcorner\urcorner \lrcorner\lrcorner & \text{if } X' \neq \bot \text{ and } A \notin \{\bot, \llcorner\bot\lrcorner\} \\ \bot & \text{otherwise,} \end{cases} \end{aligned}$$

where $A \equiv I[\![a]\!]\,\tau$ and $X' \equiv I[\![X]\!]\,\tau$. We apply these strictness conventions only to arguments of collection operations that are of the element type of the collection. Thus, collection operations should be strict for `null` arguments that are collections (e. g., the argument of `->union`) or other types not related to the collection (e. g., the index argument of the sequence access function `_->at(_)`).

7.3 Attribute Values

The evaluation of an attribute for an object can yield a value or a collection of values. The type of the evaluation result depends on the multiplicity specified for the attribute. A multiplicity defines a lower bound as well as a possibly infinite upper bound on the cardinality of the attribute's values.

7.3.1 Single-Valued Attributes

If the upper bound specified by the attribute's multiplicity is one, then an evaluation of the attribute yields a single value. If the lower bound specified by the multiplicity is zero, the evaluation is not required to yield a non-null value. In this case an evaluation of the attribute can return `null` to indicate an absence of value.

To facilitate accessing attributes with multiplicity `0..1`, the OCL standard states that single values can be used as sets by calling collection operations on them. However, the implicit conversion of a value to a `Set` is not defined by the standard. We argue that the resulting set cannot be constructed the same way as when evaluating a `Set` literal. Otherwise, `null` would be mapped to the singleton set containing `null`, but the standard demands that the resulting set is empty in this case. The conversion should instead be defined for all non-collection types T as follows:

```
context T::asSet():T
  post: if self ≐ null then result ≐ Set{}
        else result ≐ Set{self} endif
```

7.3.2 Collection-Valued Attributes

If the upper bound specified by the attribute's multiplicity is larger than one, then an evaluation of the attribute yields a collection of values. This raises the question whether `null` can belong to this collection. The OCL standard states that `null` can be owned by collections. However, if an attribute can evaluate to a collection containing `null`, it is not clear how multiplicity constraints should be interpreted for this attribute. The question arises whether the `null` element should be counted or not when determining the cardinality of the collection. Recall that `null` denotes the absence of value in the case of a cardinality upper bound of one, so we would assume that `null` is not counted. On the other hand, the operation `size` defined for collections in OCL does count `null`.

We propose to resolve this dilemma by regarding multiplicities as optional. This point of view complies with the UML standard, that does not require lower and upper bounds to be defined for multiplicities.[2] In case a multiplicity is specified for an attribute, i. e., a lower and an upper bound are provided, we require any collection the attribute evaluates to to not contain `null`. This allows for a straightforward interpretation of the multiplicity constraint. If bounds are not provided for an attribute, we consider the attribute values to not be restricted in any way. Because in particular the cardinality of the attribute's values is not bounded, the result of an evaluation of the attribute is of collection type. As

[2] We are however aware that a well-formedness rule of the UML standard does define a default bound of one in case a lower or upper bound is not specified.

the range of values that the attribute can assume is not restricted, the attribute can evaluate to a collection containing `null`. The attribute can also evaluate to `invalid`. Allowing multiplicities to be optional in this way gives the modeler the freedom to define attributes that can assume the full ranges of values provided by their types. However, we do not permit the omission of multiplicities for association ends, since the values of association ends are not only restricted by multiplicities, but also by other constraints enforcing the semantics of associations. Hence, the values of association ends cannot be completely unrestricted.

7.3.3 The Precise Meaning of Multiplicity Constraints

We are now ready to define the meaning of multiplicity constraints by giving equivalent invariants written in OCL. Let `a` be an attribute of a class `C` with a multiplicity specifying a lower bound m and an upper bound n. Then we can define the multiplicity constraint on the values of attribute `a` to be equivalent to the following invariants written in OCL :

```
context C inv lowerBound: a->size() >= m
          inv upperBound: a->size() <= n
          inv notNull: not a->includes(null)
```

If the upper bound n is infinite, the second invariant is omitted. For the definition of these invariants we are making use of the conversion of single values to sets described in Section 7.3.1. If $n \leq 1$, the attribute `a` evaluates to a single value, which is then converted to a `Set` on which the `size` operation is called.

If a value of the attribute `a` includes a reference to a non-existent object, the attribute call evaluates to `invalid`. As a result, the entire expressions evaluate to `invalid`, and the invariants are not satisfied. Thus, references to non-existent objects are ruled out by these invariants. We believe that this result is appropriate, since we argue that the presence of such references in a system state is usually not intended and likely to be the result of an error. If the modeler wishes to allow references to non-existent objects, she can make use of the possibility described above to omit the multiplicity.

7.3.4 Semantics of Operation Contracts

With the addition of `null` values, we need to adapt the semantic interpretation of operation contracts defined in (7.2). As we already mentioned in Section 7.1.1, an operation call should evaluate to an undefined value (i. e., `invalid`) if any of the arguments are undefined in order to ensure the propagation of undefined values. As `invalid` arguments are always undefined, and `invalid` is also returned if the self-argument is `null`, we modify the Δ in (7.2) as follows:

$$\Delta' \equiv \tau \models \texttt{not}\ s\,\texttt{.oclIsUndefined()}$$

$$\begin{aligned}&\land\ \tau \models \texttt{not}\ x_1\texttt{.oclIsInvalid()}\\ &\land\ \cdots \land\ \tau\,\texttt{not}\ \models x_n\texttt{.oclIsInvalid()}\ .\end{aligned}$$

This interpretation of operation contracts allows **null** as argument (except *self*) of an operation, i. e., the parameters $x_1, \ldots, x_n$ are, in the underlying class model, specified with a multiplicity `0..1`.

For the case that some parameter x_i $(1 \leq i \leq n)$ of an operation with the postcondition $\phi(x_1, \ldots, x_n)$ has multiplicity `1..1`, we replace ϕ with the postcondition

$$\begin{aligned}&x_i = \texttt{null}\ \texttt{and}\ \mathit{result}\texttt{.oclIsInvalid()}\\ &\qquad\qquad\qquad\qquad \texttt{or}\ \ \texttt{not}\ x_i = \texttt{null}\ \texttt{and}\ \phi(x_1, \ldots, x_n)\end{aligned}$$

that explicitly captures the case where x_i is **null**. Such a representation of diagrammatic UML features as textual OCL constraints is also used for other features of UML, e. g., the multiplicity of attributes and association ends.

7.4 Compliance with the OCL Standard

While the latest version of the OCL standard [125] improved a lot with respect to the handling of `null` values (e. g., compared to [124]), many details are still underspecified. In the following, we will discuss the relation of our proposal to the OCL standard [125]. In particular, we focus on decisions that are either only implicitly mentioned in the OCL standard or in which different parts of the standard contradict each other.

The OCL standard does not clearly define the semantics of null values for basic datatypes (e. g., Boolean, Integer). While we strongly suggest (see Section 7.2.1) to treat `null` and `invalid` equivalently for basic datatypes (e. g., `5 + null` $\triangleq$ `invalid` holds), this is not required by the standard. For example, a semantics where `5 + null` $\triangleq$ `5` holds is not explicitly excluded by the standard. As the OCL standard still requires that `null + 5` $\triangleq$ `invalid` holds (recall that `null + 5` is a shorthand for `null.+(5)`) this would result in an addition that is not commutative.

In Section 7.2.3 we argued that casts of **invalid** should result in **invalid**. While languages such as JML [105] define a similar behavior (i. e., casts of undefined values raise a runtime exception), the OCL standard is inconsistent: On the one hand, it states "*If the actual type of self at evaluation time does not conform to t, then the oclAsType operation evaluates to null.*" [125, p. 141] and on the other hand, it states "*An object can only be re-typed to a type to which it conforms. If the actual type of the object, at evaluation time, is not a subtype of the type to which it is re-typed, then the result of oclAsType is invalid.*" [125, p. 13]. In this case, we clearly suggest to follow the semantics of languages like JML or Java and change the requirement on page 141 of the standard to "…evaluates to *invalid*."

Due to the pragmatics described in Section 7.3.1, the collection types are potential candidates for extending the semantics of null-values, i. e., a **null**-collection could be semantically equivalent to the corresponding empty collection. As the description of the operation `->isEmpty()` [125, p. 148] states, "*`null->isEmpty()` returns 'true' in virtue of the implicit casting from null to `Bag`{}*," the OCL standard seems, at the first sight, to intend such a semantical equivalence. Since such an "implicit casting to `Bag`" is only defined for non-collection types, this example can only describe cases in which the operation is called on non-collection values. Consequently, our semantics handles operation calls on **null** values of collection-types similar to any other operation call on **null** (i. e., resulting in `invalid`). In Section 7.3.1 we presented a well-defined implicit conversion `asSet()` which facilitates the access to non-collection type attributes and association ends with multiplicity `0..1`. The flattening of collections is another class of operations for which we propose to interpret the collection-type `null` value as empty collection such that `Set{Set{1}, null}->flatten()}` $\triangleq$ `Set{1}}` holds.

Already the OCL Manifesto [53] (and earlier versions dating back to at least 1998) recognized the need for both a strong and a strict equality, however, the OCL standard only defines the strict equality. As the strong equality is mainly important for formal reasoning over OCL specifications, we see no strong need to include a strong equality in the OCL standard. Thus, we propose to define the standard equality to be strict with respect to `invalid` and non-strict with respect to `null`. Similar to other object-oriented specification languages (e. g., JML) we propose to add the equality to the list of exceptions to the rule that calling an operation on **null** results in `invalid`. Thus, in our semantics $(\texttt{null} \doteq \texttt{null}) \triangleq \texttt{true}$ holds. This, for example, also ensures the equivalence

```
A->includes(x) = A->exists(a | x = a)
```

for all values of `x` that are not undefined (including the case `x = null`). With respect to the handling of `null` and `invalid` the intention of the recent standard is unclear to us. In more detail, the standard [125, p. 141] seems for the types `OclVoid` and `OclInvalid` to override the equality of the type `OclAny`. Still, as these overridden operations are defined within the list of operations of the type `OclAny`, the intended semantics remains unclear.

7.5 Related Work

Although there have been several approaches to defining a formal semantics of UML and OCL, e.g., [81, 64, 109, 136, 78], to the best of our knowledge none of them addresses the interplay between `invalid` and `null`. There are other object-oriented specification languages that support null elements, namely JML [105] or Spec# [9]. Notably, both languages limit null elements to class types and provide a type system supporting non-null types. In the case of JML, the non-null types are chosen as the default types [46]. Supporting non-null types simplifies the analysis of specifications drastically, as many cases that result in undefined values (e. g., de-referencing a null) are already ruled out by the type system.

Recently three alternative semantics for handling undefinedness have been proposed for languages processed by finite-domain constraint solvers [67], including the three-valued Kleene semantics adopted by the Boolean operations of OCL. However, this proposal only considers a single undefined value. Alloy [85] is a language that is often compared to OCL. Since objects in Alloy can only occur as members of a set or a relation, an undefined object value is modeled by an empty set. This very natural representation of the absence of value is also employed in OCL by the conversions of values which can potentially be `null` to sets. Undefined Boolean values do not occur in Alloy.

Chapter 8

Animation of OCL Operation Contracts

In this chapter we present an approach to animating OCL operation contracts. By translating OCL constraints to arithmetic formulas with bounded quantifiers and solving these using our techniques presented in Chapter 5, we can perform animation efficiently without relying on additional guidance from the user. We implemented our approach in the tool OCLexec that generates from OCL operation contracts corresponding Java implementations which call a constraint solver at runtime. The generated code can serve as a prototype.

In the sequel, we first demonstrate the benefits of animation by means of a case study. Then we present our animation technique in detail. We describe a preliminary analysis of operation contracts that narrows down the set of classes for which new instances may need to be created and the set of constraints that need to be considered for animation. Moreover, we show how OCL expressions can be mapped to arithmetic formulas with bounded quantifiers. Finally, we give experimental results for our animation tool OCLexec.

8.1 A Case Study

In this section we present an example of a specification that could benefit from animation.

8.1.1 The Task

Figure 8.1 shows an excerpt from a possible UML model of a company. Employees are temporarily assigned to customers to carry out the customers' orders. Customers specify the skills that they would like the employee to have for handling their order (association end `requestedSkills`). Also, employees may give a list of customers that they prefer to work for (attribute `preferredCustomers`).

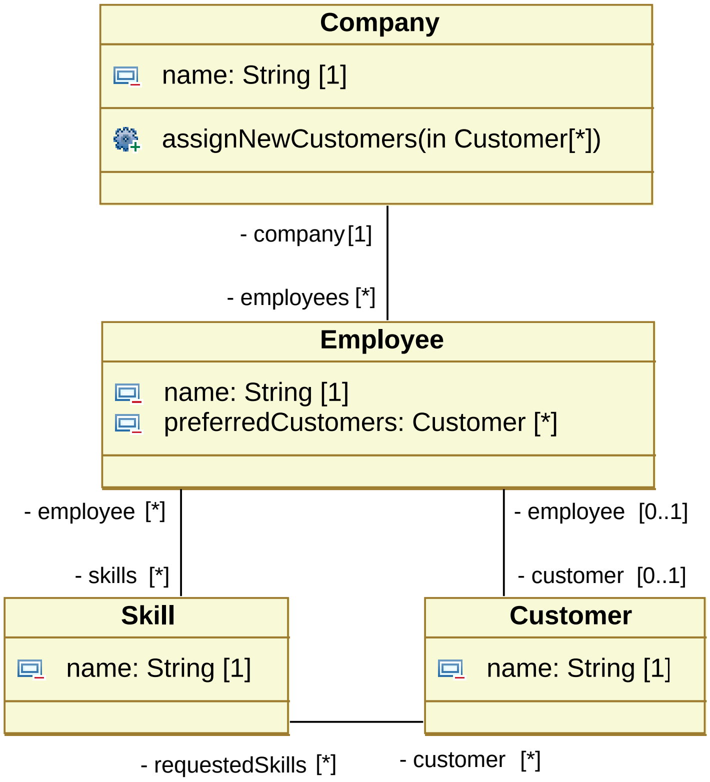


Figure 8.1: Excerpt from a possible UML model of a company

A task of the system which we are specifying is to perform an adequate assignment of employees to customers. What is sought is an assignment that respects the preferences of both the customers and the employees. This kind of assignment problem can be regarded as an instance of the prominent stable marriage problem [68]. The term *stable marriage* is inspired by the idea of matching men to women in a consistent manner. It is well-known that if the numbers of men and women are equal, it is always possible to find a *stable* assignment, i.e., an assignment in which no man and woman leave their assigned partners in order to form a new couple because they both prefer their new partner to the one that was assigned to them.

8.1.2 Anatomy of the Operation Contract

Since the operation contract in Figure 8.2 is nontrivial, we explain why it expresses the requirements. The precondition of the operation contract states that there are at least as many available employees, i.e., employees that are currently not assigned to a customer, as customers that are supposed to be matched. This condition is obviously necessary for the existence of any assignment of available

```
context Company::assignNewCustomers(newCustomers: Set(Customer)):

pre enoughEmployees: employees->select(customer.oclIsUndefined())->size()
                        >= newCustomers->size()

post allCustomersAssigned:
   employees@pre->select(customer@pre.oclIsUndefined()
                            and not customer.oclIsUndefined())
                ->collect(customer)->asSet() = newCustomers

post assignmentStable:
   employees@pre->select(customer@pre.oclIsUndefined())
      ->forAll(e | newCustomers->forAll(c |
         let
            matchedSkills : Set(Skill)
               = c.requestedSkills@pre->intersection(c.employee.skills@pre),

            potentialSkills : Set(Skill)
               = c.requestedSkills@pre->intersection(e.skills@pre)
         in
            (potentialSkills->includesAll(matchedSkills)
                implies potentialSkills = matchedSkills)
         or
            (e.preferredCustomers@pre->includes(c)
                implies e.preferredCustomers@pre->includes(e.customer))))

modifies only: employees->select(customer.oclIsUndefined())::customer,
               newCustomers::employee
```

Figure 8.2: Operation contract for assigning new customers to available employees

employees to all new customers. As mentioned above, this condition is also sufficient for the existence of a stable assignment. In the precondition, we use the built-in operation `oclIsUndefined` for testing whether the value of the `customer` attribute of an `Employee` object is `null` or a reference to a `Customer` object. Using this test, we can form the set of available employees and apply the built-in operation `size` to it.

The first postcondition of the operation contract asserts that after completion of the operation all customers have in fact been assigned to available employees. In this postcondition, first the set of employees that were available in the pre-state but are no longer available in the post-state is defined. Then we use the `collect` comprehension of OCL to obtain the collection of customers that are assigned to this set of employees. This collection is a bag, since OCL semantics is based on the general case that several employees may be assigned to the same customer, although this is excluded by the multiplicities in the class diagram. We use the built-in operation `asSet` to convert the bag to a set, so it can be compared to the set of new customers.

The second postcondition asserts that the assignment performed by the operation is stable. We quantify over all pairs `e`, `c` of available employees and new customers and consider the employee assigned to the customer (`c.employee`) as well as the customer assigned to the available employee (`e.customer`). This postcondition rules out that the pair `e-c` is a better match than both `c-c.employee` and `e-e.customer`. It does so by stating that the skills potentially provided by employee `e` to customer `c` are not a proper superset of the skills provided by `c.employee` to customer `c`, or that employee `e` also prefers `e.customer` if employee `e` lists customer `c` as preferred.

To complete the operation contract, we still need to specify which attribute values may be changed by the operation. We do this by adding a modifies only clause which states that the operation may only modify the attribute `customer` for the available employees and the attribute `employee` for the new customers. All other attribute values must be left unchanged by the operation. The operation contract in Figure 8.2 does not contain an objective function, but our animation technique would also be able to take an objective function into account. Modifies only clauses and objective functions have not yet been incorporated into the OCL standard. See Section 6.1 for details about these extensions to OCL.

We have now obtained an operation contract that precisely reflects the requirements. Note that the contract is underspecified, i.e., it does not prescribe a unique result, but allows the operation to perform any stable assignment. Moreover, the contract does not indicate how such an assignment can be found.

8.1.3 Animating the Operation Contract

The tool OCLexec we implemented our approach in generates Java method bodies. It inserts code that enforces the postconditions of the operation and all class invariants. OCLexec serializes an intermediate representation of the operation contract to a file that the generated method body can access as a resource. This intermediate representation is based on the language of arithmetic formulas with bounded quantifiers introduced in Chapter 5. The method body only reads the serialized file and calls a library routine responsible for animating the operation. The pre-state considered for animation is simply the pre-state of the method call. In principle, the results returned by this generated method body cannot be distinguished from results returned from a manually implemented method body that conforms to the operation contract. Note that inserting code in method bodies should not interfere with other code that may have been generated for the model. Thus, the developer can use her favorite tool for the overall code generation and then use our tool only for selected method bodies.

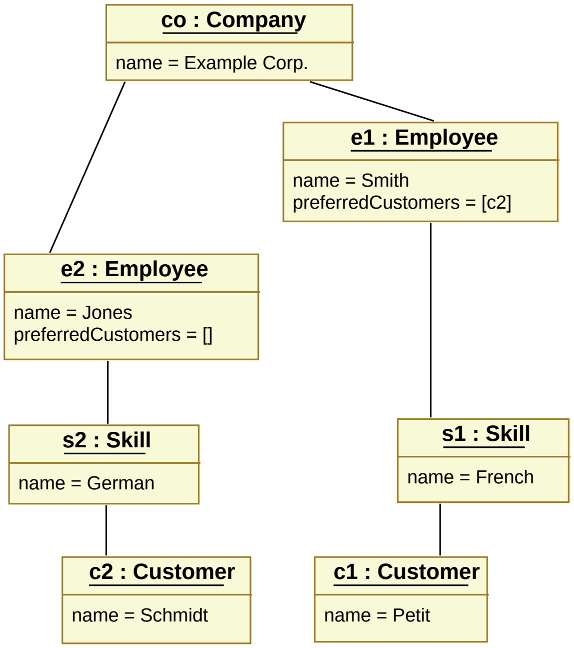
Figure 8.3(a) depicts a very simple system state in which the operation `assignNewCustomers` can be called. The company employs two staff members whose names are Smith and Jones. There are two skills: French and German language skills. Smith speaks French while Jones speaks German. There are two customers, called Petit and Schmidt, who ask for French and German language skills, respectively, from the employee that is assigned to them. Moreover, employee Smith prefers to work for customer Schmidt. Figure 8.3(b) shows a possible outcome of calling the generated method for the two customers in this system state. Employee Smith is assigned to customer Schmidt and employee Jones is assigned to customer Petit. Unfortunately, neither customers' request for language skills is met. However, the assignment is stable, since employee Smith is now assigned to his preferred customer Schmidt and therefore not interested in changing the assignment.

Depending on the needs of the company, this result of the operation call may not be sufficient. It may well be that the customers' demands for skills are deemed more important than the preferences of the employees. If this is the case, animation would have revealed an important flaw of the specification. Note that this kind of unforeseen behavior cannot be discovered if the constraints are only tested on system states that the specifier has designed to be correct or incorrect.

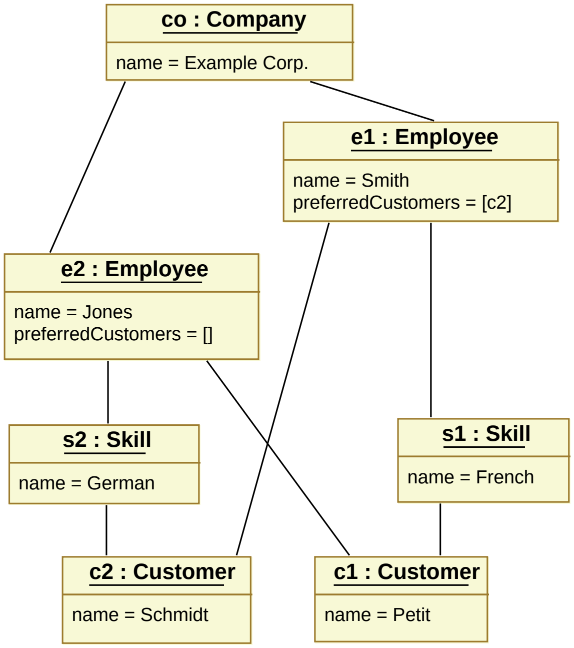
If the operation is not performance-critical and sufficiently efficient code can be generated for it, animation may allow to skip or postpone its implementation. Such an opportunity saves implementation effort and helps avoid coding errors. Moreover, a larger part of the development can be carried out on a higher and platform independent level of abstraction. In this sense, animation of operation contracts can be regarded as a contribution to Model-Driven Development.

8.2 Execution of Animation

We implemented an animation technique that is based on a translation of the operation contract to an arithmetic formula with bounded quantifiers as described



(a) State before animation



(b) State after animation

Figure 8.3: Effect of animating a call to the operation `assignNewCustomers` on a system state

in Chapter 5. First, we narrow down the set of classes for which new instances may need to be created and the set of constraints that need to be considered for animation. Using this information, we can the express the set of animation results that are permitted by the operation contract by an arithmetic formula with bounded quantifiers. A concrete animation result can then be obtained by solving this formula using the techniques described in Section 5.3.

8.2.1 Preliminary Analysis: Reasoning about New Class Instances

As a first step of animation, we determine for which classes new instances may need to be generated. It is beneficial to restrict this set of classes as much as possible. Knowing that no instances need to be created for a certain class allows to reduce the search space that has to be explored. Such an observation also concerns the handling of class invariants. Class invariants have to always hold for all objects of a certain class. The following OCL invariant definition stating that the `name` attribute of an object of class `Skill` may never be empty could be added to the specification presented in Section 8.1:

```
context Skill inv: name <> ""
```

If no instances are created for a class that an invariant belongs to and the invariant references no attributes that can be modified by the operation, then we can conclude that the invariant can never be violated in the post-state if it was satisfied in the pre-state. Hence, such invariants do not need to be considered when searching for animation results, which simplifies the computation.

The creation of new instances can be restricted by the operation contract. OCL provides the `oclIsNew` test for querying whether an object has been created by the operation call. By using this language feature, postconditions can express that certain objects must have already existed before the operation call. However, it is usually not possible to observe that this test has been applied to a set of object references that is sufficient to completely rule out the creation of new instances for a certain class. In this case we must take into account that it may be necessary to create new instances of the class to satisfy the operation contract. Our goal is to compute an approximation of the set of classes requiring new instances that is as good as possible.

For defining a suitable approximation, we observe that, in order to be relevant for animation results, a reference to a new object needs to be (i) the value of an output parameter, (ii) a value of a modified attribute of an object existing before the operation call, or (iii) a value of an attribute of a freshly created object.

We can approximate the set of classes for which new instances need to be created by analyzing for which types of objects these cases can occur. In order to limit the impact of case (iii), we demand that attributes of a freshly created object are assigned `null` when this is possible without compromising satisfiability of the operation contract. This is the case when the multiplicity lower bound of the attribute is zero and the attribute is not referenced by any constraint considered during animation.

Let T denote the set of classes for which new instances may be created and C be the set of constraints considered for animation. The observations above yield that the following conditions on T and C are sufficient for ensuring correct animation.

1. Every postcondition belongs to C.
2. If a modifies only clause lists an attribute a, then every invariant referencing a belongs to C.
3. Every invariant of a class in T belongs to C.
4. If a class t is the type of an out-parameter of the operation or a subtype of the parameter's type, then $t \in T$. This corresponds to case (i) above.
5. If a modifies only clause lists an attribute a, then every class t which is the type of a or a subtype of the type of a belongs to T. This corresponds to case (ii) above.
6. For every attribute a of a class in T, if a is referenced by a constraint in C in the post-state or a does not have a multiplicity lower bound of zero, then every class t which is the type of a or a subtype of the type of a belongs to T. This condition corresponds to case (iii) above.

The smallest sets T and C that satisfy these conditions can easily be found by a closure computation. We initialize T and C to empty sets and augment the sets according to the conditions until a fixed point is reached.

When performing preliminary constraint analysis on the operation contract in Figure 8.2, the set C of constraints to consider consists of the two postconditions, since the specification does not define any invariants. The set T is set to $\{\texttt{Employee}, \texttt{Customer}\}$ according to Condition 5. The class `Company` is added to T by Condition 6 because the association end `company` of class `Employee` does not have a multiplicity lower bound of zero, and the procedure terminates with

$$T = \{\texttt{Employee}, \texttt{Customer}, \texttt{Company}\} .$$

Thus, no new instances of class `Skill` need to be created in the post-state. Hence, even if the example invariant given at the beginning of this section is included in the specification, it would not need to be considered during animation. However, if the invariant was defined for the class `Company` instead of `Skill`, it would be added to the set C by Condition 3.

8.2.2 Translating OCL Expressions to Arithmetic Formulas with Bounded Quantifiers

We describe how OCL expressions can be translated to nested tuples of the arithmetic formulas with bounded quantifiers defined in Section 5.1. For animating an operation, we translate the postconditions of the operation and all relevant invariants[1] to arithmetic constraints. The conjunction of the resulting formulas expresses the condition that must be satisfied when the operation returns.

Representing OCL expressions of type `Integer` by arithmetic terms is straightforward; we cope with undefinedness in OCL separately (see below). Similarly, OCL expressions of type Boolean can be compiled directly to formulas.

Expressions whose type is a class are mapped to a pair (t_1, t_2) of arithmetic terms, where t_1 describes the dynamic type of the object which is the expression value. For this purpose, we assign an integer to every non-abstract class in the model. Note that due to subtyping t_1 can become quite complex. The second term t_2 gives an identifier of the object.

An expression of a collection type is mapped to the comprehension

$$\langle t_1 \leq x \leq t_2 \,|\, p(x) \bullet E(x) \rangle \,, \tag{8.1}$$

where x is a variable, t_1 and t_2 are terms, $p(x)$ is a formula and $E(x)$ is itself an encoding of an OCL expression whose type is the element type of the collection. The terms t_1 and t_2 are the lower and upper bound of the variable x. As indicated by the notation, x may occur in $p(x)$ and $E(x)$. For every integer i between t_1 and t_2, the value described by $E(i)$ belongs to the collection iff $p(i)$ is true. Here we denote by $p(i)$ and $E(i)$ the formula $p(x)$ and the encoding $E(x)$, respectively, with i substituted for x.

Collection operations are translated by manipulating the tuple that represents the collection expression. For example, a `select` operation on a collection represented by a comprehension of the form (8.1) is translated by conjoining the body of the `select` construct with the predicate $p(x)$. A `collect` operation can be translated by substituting $E(x)$ by the body of the `collect` construct.

For expressing undefined values, we add to every translation of an OCL expression e two formulas $invalid_e$ and $null_e$ such that e evaluates to `invalid` iff $invalid_e$ is true, and to `null` iff $null_e \wedge \neg invalid_e$ is true.

There are a limited number of OCL language features like recursive operations and real numbers that we do not support due to the effort required to encode them using this kind of framework.

We show how the translation is derived for the first postcondition `allCustomersAssigned` of the operation contract in Figure 8.2. We adhere to the OCL convention that the variable holding the object the operation is called on is named `self`. We assign the integer 0 to the class `Company` and introduce the

[1] These are the constraints in the set C defined in Section 8.2.1.

function symbol $f_{\texttt{self}}$ in order to translate the self variable of type Company to the term pair $(0, f_{\texttt{self}})$. Note that self is the implicit source of the attribute call employees**@pre** in this postcondition. We assign the integer 1 to the class Employee and introduce the 0-1-valued function symbol $f_{\texttt{employees}}$ with arity four for representing a characteristic function that indicates whether an Employee object is associated with a Company object in the pre-state. The function symbol $f_{\texttt{Employee}}$ represents the number of Employee instances in the pre-state.[2] Thus, the expression employees**@pre** of type Set can be translated to the comprehension

$$\big\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, f_{\texttt{employees}}\,(0, f_{\texttt{self}}, 1, x) = 1 \bullet (1, x) \big\rangle\,.$$

We use the 0-1-valued function symbols $f_{\mathit{null}(\texttt{customer})}$ and $f'_{\mathit{null}(\texttt{customer})}$ for indicating whether the values of the attribute customer are null in the pre- and post-state, respectively. Thus, the translation of the expression

```
employees@pre->select(customer@pre.oclIsUndefined()
                               and not customer.oclIsUndefined())
```

becomes

$$\big\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, p(x) \bullet (1, x) \big\rangle$$

with

$$\begin{aligned} p(x) := {} & f_{\texttt{employees}}\,(0, f_{\texttt{self}}, 1, x) = 1 \\ & \qquad\qquad \wedge f_{\mathit{null}(\texttt{customer})}\,(1, x) = 1 \wedge \neg f'_{\mathit{null}(\texttt{customer})}\,(1, x) = 1. \end{aligned}$$

We assign the integer 2 to the class Customer and introduce the function symbol $f'_{\texttt{customer}}$ with arity two for representing a function which maps Employee instances to identifiers of the associated Customer objects in the post-state. As a result, applying the collect construct with body expression customer to this source yields the translation

$$\big\langle 0 \leq x \leq f_{\texttt{Employee}} - 1 \,|\, p(x) \bullet (2, f'_{\texttt{customer}}\,(1, x)) \big\rangle$$

with the same formula $p(x)$ as in the previous comprehension.

Using the function symbol $f_{\texttt{newCustomers}}$ as characteristic function for the set-valued parameter newCustomers and applying the same translation scheme as above for this expression gives us the comprehension

$$\langle 0 \leq x \leq f_{\texttt{Customer}} - 1 \,|\, f_{\texttt{newCustomers}}\,(2, x) = 1 \bullet (2, x) \rangle$$

[2] Since the class Employee belongs to the set T of classes for which new instances may need to be created for animation, a separate function symbol would be necessary for representing the number of instances of this class in the post-state.

Based on these translations, we obtain the following formula that expresses that all elements of `newCustomers` belong to the collection on the left-hand side of the equality:

$$\begin{aligned} &\forall\, 0 \le x \le f_{\texttt{Customer}} - 1\,. \\ &\quad f_{\texttt{newCustomers}}(2, x) = 1 \\ \Longrightarrow\ &\exists\, 0 \le y \le f_{\texttt{Employee}} - 1\,.\ p(y) \wedge x = f'_{\texttt{customer}}(1, y)\,. \end{aligned}$$

In order to translate the entire equality, we only need to conjoin an analogous formula for the containment in the other direction.

8.3 Experimental Results

We evaluated the efficiency of OCLexec by animating the operation contract of Figure 8.2 in system states of increasing size. Specifically, these are states with a number n of `Employee` and `Customer` objects, respectively, and $\lfloor \log_2 n \rfloor$ `Skill` objects. Employees and customers are associated independently to every skill with probability 1/2. This achieves that every subset of skills is quite likely to be associated with at least one employee or customer, which favors conflicts between the different actors. Similarly, an employee lists every customer as preferred with probability 1/2.

Customers	Employees	Skills	Runtime
5	5	2	0.4 sec
10	10	3	0.5 sec
20	20	4	0.8 sec
30	30	4	2.2 sec
40	40	5	23 sec
50	50	5	78 sec

Table 8.1: Runtime required for animating the operation contract of Figure 8.2

The results of the evaluation are shown in Table 8.1. The measurements were performed on a machine with 2 GB RAM and a dual-core 2.4 GHz P8600 mobile CPU. The SAT solver used was MiniSat [63], a well-known state-of-the-art SAT solver.

We note that for states with up to 20 employees and customers, animation is efficient enough for certain applications like prototyping. States of this size already include many interesting application scenarios. However, for larger states the time consumed increases quickly, and animation becomes infeasible. These runtimes may seem disappointing, considering that a polynomial-time algorithm exists for the stable marriage problem. Note that the number of generated Boolean constraints grows faster than linearly in the number of employees and customers, due to e.g., the nested quantifiers in the specification. Also recall

Operation	$n = 3$	$n = 5$	$n = 7$	$n = 10$	$n = 15$	$n = 20$
`getBalancedCompilationJobs(2)`	3.0	3.0	50	>1000	>1000	>1000
`getOrderedCompilationJob()`	2.5	12	37	>1000	>1000	>1000
`getIndependentCompilationJobs()`	3.0	9	10	140	>1000	>1000
`getCompilationOrder()` (input cyclic)	1.0	1.0	1.0	1.5	1.5	>1000
`getCompilationOrder()` (input acyclic)	1.0	1.0	1.0	1.5	1.5	>1000

Table 8.2: Runtime (in seconds) required for animating the specification in Section 6.3

```
context Project::getCompilationOrder(): OrderedSet(CompilationUnit)

post allUnitsReturned: (not result->isEmpty())
                       implies
                       result->asSet() = compilationUnits@pre

post resultSorted:
   (not result->isEmpty())
   implies
   Sequence{2..result->size()}
      ->forAll(i | Sequence{1..i-1}
         ->forAll(j | result->at(j).dependsOn@pre
                         ->excludes(result->at(i))))

minimize: if result->isEmpty() then 1 else 0 endif

modifies only: nothing
```

Figure 8.4: “Tuned” variant of the operation contract in Figure 6.5

that we are processing a high-level specification in a relatively general-purpose language.

We also evaluated OCLexec on the specification presented in Section 6.3 with random inputs. When generating an input with n compilation units, we added every possible dependency between compilation units with probability $0.1/n$, so it was sufficiently unlikely that a cyclic dependency graph was generated. File sizes were sampled uniformly between 0 and 10000. Unfortunately, our over-approximation scheme was not able to identify optimal solutions for any of the operations, so user-defined timeouts were necessary for animating these operations. Table 8.2 shows the total execution times for animating the operations obtained by adding the best attainable value of the objective function as an additional postcondition.

In order to evaluate our approach to over-approximation, we modified the contract of the operation `getOrderedCompilationJob` as shown in Figure 8.4.

The modified contract returns a sequence of compilation units directly rather than returning an object holding this sequence. Cyclic dependency graphs are indicated by returning an empty sequence. Over-approximation succeeds for this modified contract, which we animated with dependencies added with probability $10/n$, so the dependency graph very likely had a cycle. Table 8.1 shows the time required to determine that 1 is the optimal value of the objective function and the operation can return an empty result. The table also shows the time consumed by animating the modified contract when processing the acyclic graphs used for animating the other contracts, which are essentially identical to the runtimes for animating the cyclic dependency graphs with the same number of vertices. We note that animation of the modified contract in Figure 8.4 is more efficient than the original contract in Figure 6.5. The reason may be that, in contrast to the original contract, the modified contract does not specify the creation of new objects.

8.4 Related Work

There has been constant interest in animation as a research problem. Work on animation has focused particularly on the specification languages Z [62, 147, 77] and B [20, 134, 107]. Pioneering work on animating OCL can be found in [122, 76].

Animators for operation contracts have also been implemented, for example, for the specification language JML [19, 99, 44]. These animators for JML do not use SAT solving like OCLexec does, but rely on other constraint solving techniques. They are automatic in the sense that they do not explicitly require the user to provide additional information such as bounds. However, they cannot handle certain constraints. Specifically, the JML-TT animator [19] lacks support for quantifiers, which severely restricts the class of specifications it can process. The jmle animator [99, 44] works by generating a prototype implementation in Java, as does OCLexec, and throws an exception at runtime for certain constraints it cannot handle. In contrast, OCLexec identifies unsupported constraints at compile-time, and an operation implementation generated by OCLexec always terminates successfully if valid operation results exist. A recent animator that uses a SAT solver is Squander [112], which animates operation contracts in a tailored specification language.

The power available through SAT solvers as constraint solving engines has long been recognized. Alloy [86], NP-SPEC [41], answer set programming systems (e.g., smodels [117]) and SAT-based CSP solvers (see e.g., [140]) are tools that process constraint languages using SAT solvers or similar search techniques. These languages avoid constructs that are difficult to encode in Boolean constraints, like multisets and nested collections which are available in OCL, and usually require narrow bounds on integer values that allow for explicit enumeration of the considered integers. As a result, these languages offer an attractive trade-off between expressiveness and efficiency. This comes at a price: since these languages are not tightly integrated into a large-scale language like UML or Java, developers need to write tedious glue code to interface with the constraint solver. We view OCLexec as complementary to these tools; after animating an operation with OCLexec, a developer could seek a more efficient execution of the operation using such a SAT-based tool. The Alloy annotation language [94] could be regarded as an approach to close the gap between Alloy as a constraint language and Java as a large-scale language.

UML2Alloy [6], UMLtoCSP [40] and USE [71, 103] are tools for analyzing OCL specifications. They aim to verify or validate certain properties of specifications, which basically amounts to solving the OCL constraints. UML2Alloy performs a translation to the Alloy language, UMLtoCSP is based on the Eclipse constraint programming system, and USE employs, among other techniques, the kodkod constraint solver. All of these tools require the user to specify bounds to restrict the scope of analysis. Moreover, they do not support modifies only clauses or objective functions. HOL-OCL [33] is an embedding of OCL into Isabelle/HOL and also relies on substantial guidance from the user for most kinds of analysis. Another tool along these lines is the SQL query explorer Qex [150] that automatically constructs test cases for databases and is based on a powerful SMT solver.

Chapter 9

Generating Tests from Object-Oriented Specifications

In this chapter we present a technique for generating tests from object-oriented specifications. Its cornerstones are the logical manipulation of OCL expressions that are embedded into HOL, the systematic exploration of object-oriented datastructures, the analysis of recursive query operations and a representation of object graph classes by an equivalence relation.

9.1 Running Example: Linked Lists

We present a small OCL specification that will serve as a running example for our test generation technique. We will also discuss the translation of OCL into HOL and discuss the implicit invariants of this example. We use the notation introduced in Chapter 7. As additional notation, we define $\partial\, X \equiv \texttt{not}\ (X \triangleq \bot)$ for marking OCL expressions that are not undefined (`not _.oclIsInvalid()`).

9.1.1 Singly-Linked Lists

Figure 9.1 illustrates our running example of a singly-linked list: the list stores integers as data and links between nodes are modeled by an association. As a node does not necessarily need to have a successor, the association end `next` has multiplicity `0..1`. An invariant of the class states that the integers are stored in a descending order in the list. We specify an operation `insert` that adds an integer to the list. The postcondition of the `insert` operation states that the set of integers stored in the list in the post-state is the set of stored integers in the pre-state extended by the argument. For defining the set of integers stored in the list, we separately specify the recursive query operation `contents()`.

In the sequel, we will describe how to build the context Γ_τ from this OCL specification. We will add semantic presentations of the specification to Γ_τ which

```
context Node
inv: next <> null implies i > next.i

context Node::contents():Set(Integer)
  post: result = if next = null
                 then Set{i}
                 else next.contents()->including(i)

context Node::insert(x:Integer)
  post: contents() = contents@pre()->including(x)
```

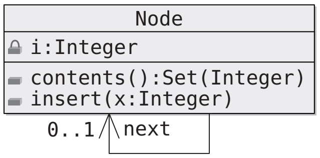


Figure 9.1: A Singly-linked list specified in OCL (excerpt)

are already in a "massaged format" suitable for test generation. Since the transition is not changing in the rest of this paper, we will assume one global transition τ (understood to be relative to the specification of this example); we will drop the index and abbreviate $\tau \vDash \phi$ to just $\vDash \phi$. In our test generation approach, we assume that all diagrammatic constraints over the class model are represented as OCL expressions (for details, see [74]). For example, associations are represented by collection-valued class attributes together with OCL constraints expressing the multiplicity.

9.1.2 Translating Invariants into Recursive HOL Predicates

The example in Figure 9.1 only includes one explicit invariant. The multiplicity constraints in the class model constitute invariants semantically. For our example, the multiplicity constraints could be expressed as follows in OCL:

```
inv: (next = null or next <> null) and i <> null
```

In the sequel, we will assume that attributes and arguments that have a primitive type (e. g., `Integer`) have a multiplicity of `1..1`, i. e., they cannot be `null`. Thus we can simplify the invariant representing the multiplicity constraints to:

```
inv: (next = null or next <> null)
```

This simplification improves the readability of the formulas and is not a fundamental restriction of our approach.

For our purposes it will be convenient to convert invariants to recursive predicates and add them to Γ, paving the way for the exploration of input parameters by simply unfolding them rather than making them, based on the definition (7.1), lengthy arguments over `.allInstances()`. Of course, not any recursive predicate is consistent; however *these* recursive predicates can be derived from the invariants by using a greatest fixed-point construction and proving that the body of the invariant is monotone—the reader interested in the details is referred to

HOL-OCL [32] where this is done automatically (albeit for OCL 2.0, i. e., without `null`). The invariant of the `Node` class can be expressed as follows in HOL.

$$\begin{array}{l} \forall\ \texttt{self.}\ \ \models \partial\ \texttt{self} \land \models \texttt{self} \not\doteq \texttt{null} \to\ \ \models \mathrm{inv}_{\mathrm{Node}}\texttt{(self)} \\ \quad \iff \models \texttt{self.next} \doteq \texttt{null} \lor \texttt{(}\models \texttt{self.next} \not\doteq \texttt{null} \\ \qquad\quad \land \models \texttt{self.i > self.next.i} \land \models \mathrm{inv}_{\mathrm{Node}}\texttt{(self.next))} \end{array}$$

Additionally to this recursive predicate, we add to Γ the fact that any non-null object that is not undefined will satisfy this invariant:

$$\forall\ \texttt{self.}\ \ \models \partial\ \texttt{self} \land \models \texttt{self} \not\doteq \texttt{null} \to\ \ \models \mathrm{inv}_{\mathrm{Node}}\texttt{(self)}$$

Our recursive definitions are a conjunction of the explicit invariant and the multiplicity constraints of our example. The invariant $\mathrm{inv}_{\mathrm{Node}}$ `@pre` expresses well-formedness in a pre-state:

$$\begin{array}{l} \forall\ \texttt{self.}\ \ \models \partial\ \texttt{self} \land \models \texttt{self} \not\doteq \texttt{null} \\ \qquad \to\ \ \models \mathrm{inv}_{\mathrm{Node}}\texttt{@pre(self)} \iff \models \texttt{self.next@pre} \doteq \texttt{null} \\ \quad \lor \texttt{(}\models \texttt{self.next@pre} \not\doteq \texttt{null} \land \models \mathrm{inv}_{\mathrm{Node}}\texttt{@pre(self.next@pre)} \\ \qquad\qquad \land \models \texttt{self.i@pre > self.next@pre.i@pre)} \end{array}$$

9.1.3 Translating Contracts into HOL

Given the fact that $\models (\texttt{true})_{pre}$ just collapses to `true`, the formulas that we add to Γ is the straight-forward simplification of the semantics rule (7.2):

$$\begin{array}{l} \forall\ \texttt{self.}\ \Delta\texttt{(self)} \to \models \texttt{self.contents()} \triangleq \\ \qquad\qquad\qquad \texttt{if self.next} \doteq \texttt{null then Set\{i\}} \\ \qquad\qquad\qquad \texttt{else self.next.contents()->including(i)} \\ \qquad \land \neg\Delta\texttt{(self)} \to \models \texttt{self.contents()} \triangleq \bot \end{array}$$

where Δ`(self)` is a short-cut for $\models \partial$ `self` $\land \models$ `self` $\not\doteq$ `null`. The variant for `contents@pre()` reads as follows:

$$\begin{array}{l} \forall\ \texttt{self.}\ \Delta\texttt{(self)} \to \models \texttt{self.contents@pre()} \triangleq \\ \qquad\quad \texttt{if self.next@pre} \doteq \texttt{null then Set\{i\}} \\ \qquad\quad \texttt{else self.next@pre.contents@pre()->including(i)} \\ \qquad \land \neg\Delta\texttt{(self)} \to \models \texttt{self.contents@pre()} \triangleq \bot \end{array}$$

9.2 Test Generation

We follow the classical approach of transforming the test specification into a DNF, extended by the treatment of invariants and recursive definitions, which corresponds to a case distinction that partitions the input space of the operation(s).

A particular class of case distinctions arises from *aliasing*; i. e., the fact that two object references can designate the same object, e. g., s`.next.next` may in fact be identical to s due to a cycle in the object graph. Aliasing is a crucial phenomenon in object-oriented systems. It is likely that a system behaves differently depending on the aliasing relationships among the objects it handles. Therefore we will add further case distinctions to the specification under analysis that distinguish different aliasing relationships. We will refer to this transformation as *alias closure*.

9.2.1 Unfolding

For generating a set of test cases, we start with the test specification stating that the operation call `s.insert(x)` is executed successfully. This test specification amounts to the translation of the postcondition to HOL, restricted to the part where s is not undefined and not `null`:

$$\Delta\texttt{(s,x)} \wedge\ \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}$$

This test specification does not show any explicit case distinctions. Rather, the case distinctions are hidden in the recursive specification of `contents()`.

The invariants over the different arguments of the operation (including `s`) must be taken into account for the generation of relevant test cases. In our example, only ordered lists can occur in pre-states and post-states of the `insert` operation. Adding these invariants as constraints over the pre-states or post-states reduces the number of test cases derived from the test specification by removing as many non-satisfiable clauses as possible before the test data selection. Because of the facts contained in Γ, we obtain:

$$\forall\ \texttt{self.} \models \partial\ \texttt{self} \wedge\ \models \texttt{self} \not\doteq \texttt{null} \rightarrow\ \models \mathrm{inv}_{\mathsf{Node}}\texttt{(self)}$$

These invariants can be inserted at any time during the unfolding process.

For instance, we can already insert the invariant for the pre-states and post-states of the `insert` operation, knowing that `s` is not undefined and not `null`:

$$\begin{aligned}&\Delta\texttt{(s,x)} \wedge \models \mathrm{inv}_{\mathsf{Node}}\texttt{@pre(s)} \wedge \models \mathrm{inv}_{\mathsf{Node}}\texttt{(s)}\\ &\wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}\end{aligned}$$

To enrich this condition with explicit case distinctions, we unfold the operation calls and invariants by replacing them with their specification: an operation call will be replaced with its contract and an invariant with its definition, which is allowed here since we have $\Delta(s,x)$. For the sake of readability, we do not replace the `contents` operation calls directly with their contract but rather conjoin the contract with the existing formulas. We obtain the following conditions:

$$\begin{aligned}&\quad\Delta\texttt{(s,x)}\\ &\wedge\ (\models \texttt{s.next@pre} \doteq \texttt{null}\\ &\quad\vee\ (\models \texttt{s.next@pre} \not\doteq \texttt{null}\\ &\quad\ \wedge \models \texttt{s.i@pre > s.next@pre.i@pre} \wedge \models \mathrm{inv}_{\mathsf{Node}}\texttt{@pre(s.next@pre)))}\\ &\wedge\ (\models \texttt{s.next} \doteq \texttt{null}\\ &\quad\vee\ (\models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \mathrm{inv}_{\mathsf{Node}}\texttt{(s.next)))}\end{aligned}$$

$$\begin{array}{l}
\wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)} \\
\wedge \models \texttt{s.contents()} \triangleq \texttt{if s.next} \doteq \texttt{null then Set\{s.i\}} \\
\qquad\qquad\qquad\qquad \texttt{else s.next.contents()->including(s.i)} \\
\wedge \models \texttt{s.contents@pre()} \triangleq \\
\qquad \texttt{if s.next@pre} \doteq \texttt{null then Set\{s.i@pre\}} \\
\qquad\quad \texttt{else s.next@pre.contents@pre()->including(s.i@pre)}
\end{array}$$

A second refinement step could be performed by unfolding the invariants and the operation calls a second time: we could insert the invariant definitions again and instantiate the operation contract for the `contents` operation with `s.next` (correspondingly for the pre-state).

The unfolding process and invariant insertion can be stopped at any time, once the refinement is sufficient according to the tester's needs. Then, the DNF of the obtained formula is generated to enumerate the different test cases coming from case distinction. The DNF obtained for the previous formula is the following, leading to four clauses distinguishing whether `s.next` and `s.next@pre` are `null`.

$$\begin{array}{l}
\quad (\Delta(s,x) \\
\quad \wedge \models \texttt{s.next} \doteq \texttt{null} \\
\quad \wedge \models \texttt{s.next@pre} \doteq \texttt{null} \\
\quad \wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)} \\
\quad \wedge \models \texttt{s.contents()} \triangleq \texttt{Set\{s.i\}} \\
\quad \wedge \models \texttt{s.contents@pre()} \triangleq \texttt{Set\{s.i@pre\})} \\
\vee \ (\Delta(s,x) \\
\quad \wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \mathrm{inv}_{\mathrm{Node}}\texttt{(s.next)} \\
\quad \wedge \models \texttt{s.next@pre} \doteq \texttt{null} \\
\quad \wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)} \\
\quad \wedge \models \texttt{s.contents()} \triangleq \texttt{s.next.contents()->including(s.i)} \\
\quad \wedge \models \texttt{s.contents@pre()} \triangleq \texttt{Set\{s.i@pre\})} \\
\vee \ (\Delta(s,x) \\
\quad \wedge \models \texttt{s.next} \doteq \texttt{null} \\
\quad \wedge \models \texttt{s.next@pre} \not\doteq \texttt{null} \wedge \models \texttt{s.i@pre > s.next@pre.i@pre} \\
\quad \wedge \models \mathrm{inv}_{\mathrm{Node}}\texttt{@pre(s.next@pre)} \\
\quad \wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)} \\
\quad \wedge \models \texttt{s.contents()} \triangleq \texttt{Set\{s.i\}} \\
\quad \wedge \models \texttt{s.contents@pre()} \triangleq \texttt{s.next@pre.contents@pre()} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{->including(s.i@pre))} \\
\vee \ (\Delta(s,x) \\
\quad \wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \mathrm{inv}_{\mathrm{Node}}\texttt{(s.next)} \\
\quad \wedge \models \texttt{s.next@pre} \not\doteq \texttt{null} \wedge \models \texttt{s.i@pre > s.next@pre.i@pre} \\
\quad \wedge \models \mathrm{inv}_{\mathrm{Node}}\texttt{@pre(s.next@pre)} \\
\quad \wedge \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)} \\
\quad \wedge \models \texttt{s.contents()} \triangleq \texttt{s.next.contents()->including(s.i)} \\
\quad \wedge \models \texttt{s.contents@pre()} \triangleq \texttt{s.next@pre.contents@pre()}
\end{array}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\texttt{->including(s.i@pre))}$$

The first case boils down (due to constant propagation and set reasoning) to:

$$\begin{array}{l}
\quad\Delta(s,x) \wedge \models \texttt{s.next} \doteq \texttt{null} \wedge \models \texttt{s.next@pre} \doteq \texttt{null}\\
\wedge \models \texttt{s.i} \triangleq \texttt{s.i@pre} \wedge \models \texttt{s.i} \triangleq \texttt{x}
\end{array}$$

All other cases still contain invariant applications like $\models \mathrm{inv}_{\texttt{Node}}\texttt{(s.next)}$. The derivation

$$\begin{array}{l}
\quad\Delta(s,x)\\
\wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \mathrm{inv}_{\texttt{Node}}\texttt{(s.next)}\\
\wedge \models \texttt{s.next@pre} \doteq \texttt{null}\\
\wedge \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{->including(x)}
\end{array}$$

for the second case expands to:

$$\begin{array}{l}
\texttt{(}\Delta(s,x)\\
\wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \texttt{s.next.next} \doteq \texttt{null}\\
\wedge \models \texttt{s.next@pre} \doteq \texttt{null}\\
\wedge \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{->including(x))}\\
\vee\ \texttt{(}\Delta(s,x)\\
\wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i}\\
\wedge \models \texttt{s.next.i > s.next.next.i} \wedge \models \texttt{s.next.next} \not\doteq \texttt{null}\\
\wedge \models \mathrm{inv}_{\texttt{Node}}\texttt{(s.next.next)}\\
\wedge \models \texttt{s.next@pre} \doteq \texttt{null}\\
\wedge \models \texttt{s.next.contents()->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{->including(x))}
\end{array}$$

While the second sub-case is unsatisfiable since it asserts that the insertion increases the list length by two, the first sub-case reduces to:

$$\begin{array}{l}
\quad\Delta(s,x)\\
\wedge \models \texttt{s.next} \not\doteq \texttt{null} \wedge \models \texttt{s.i > s.next.i} \wedge \models \texttt{s.next.next} \doteq \texttt{null}\\
\wedge \models \texttt{s.next@pre} \doteq \texttt{null}\\
\wedge \models \texttt{Set\{s.next.i\}->including(s.i)} \doteq \texttt{Set\{s.i@pre\}}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{->including(x)}
\end{array}$$

which, due to set reasoning, corresponds to a test case in which the inserted element `x` is not already in the list. The test cases still containing an occurrence of the invariance predicate correspond to the class of "yet to be tested" test cases.

9.2.2 Alias Closure

Unfolding and invariant insertion represent only a first step of the exploration of the specification by case distinction. There is another implicit case distinction that needs to be considered, since the two references `s` and `s.next` could actually refer to the same object, due to a cycle in the object graph. We should thus distinguish the cases $\texttt{s.next} \triangleq \texttt{s}$ and $\texttt{s.next} \not\triangleq \texttt{s}$. Analogously, we should distinguish the cases $\texttt{s.next@pre} \triangleq \texttt{s}$ and $\texttt{s.next@pre} \not\triangleq \texttt{s}$.

To handle these four cases during test generation, we add the following tautology, called *alias distinction*, to the unfolding of our test specification:

$$
\begin{aligned}
&(\models \texttt{s.next} \triangleq \texttt{s} \lor \models \texttt{s.next} \not\triangleq \texttt{s})\\
\land\ &(\models \texttt{s.next@pre} \triangleq \texttt{s} \lor \models \texttt{s.next@pre} \not\triangleq \texttt{s})
\end{aligned}
$$

In the cases $\texttt{s.next} \triangleq \texttt{s}$ and $\texttt{s.next@pre} \triangleq \texttt{s}$, the invariants evaluate to false due to the strict inequality, thus only the cases $\texttt{s.next} \not\triangleq \texttt{s}$ and $\texttt{s.next@pre} \not\triangleq \texttt{s}$ remain. Computing the DNF in our example leads to almost the same formula as in the previous subsection, where $\models \texttt{s.next} \not\triangleq \texttt{s} \land \models \texttt{s.next@pre} \not\triangleq \texttt{s}$ is added to each conjoint.

In the general case, the alias closure of a formula is the conjoint of the tautologies $\texttt{p} \triangleq \texttt{q} \lor \texttt{p} \not\triangleq \texttt{q}$ for all the references $\texttt{p}$ and $\texttt{q}$ occurring in the formula. Formally, let $\text{Path}(\varphi)$ be the set of path-expressions (references) occurring in a formula φ. We define $\text{AliasClosure}(\varphi)$ as the set of formulas

$$\{\, \texttt{p} \triangleq \texttt{q} \lor \texttt{p} \not\triangleq \texttt{q} \mid \texttt{p}, \texttt{q} \in \text{Path}(\varphi) \land \texttt{p} \text{ non-identical to } \texttt{q} \,\}$$

This produces all possible objects graphs.

9.3 Implementation in HOL-TestGen

Using the interface for test generation rules presented in Chapter 3, we defined test generation rules for alias closure as well as for unfolding invariants and recursive functions. We do not show their definition here since it is very technical. For carrying out alias closure, references can be identified by means of their type.

To prepare our test specification

$$\Delta\texttt{(s,x)} \land \models \texttt{s.contents()} \doteq \texttt{s.contents@pre()->including(x)}$$

for HOL-TestGen, we define functions *contents* and *contents_at_pre* that take the respective state as an explicit argument. The insert operation can be modeled as a function *post_state* that takes a pre-state as well as the argument to the operation and yields the post-state returned by the operation. This results in the following test specification that can be processed by HOL-TestGen:

$$
\begin{aligned}
&\mathit{inv\ pre\_state\ s} \longrightarrow \mathit{contents}\ (\mathit{post\_state\ pre\_state\ x})\ (\mathit{Some\ s})\\
&\qquad\qquad\qquad\qquad = \mathit{contents\_at\_pre\ pre\_state}\ (\mathit{Some\ s}) \cup \{x\}
\end{aligned}
$$

The precondition *inv pre_state s* stands for the recursive predicate inv_{Node} from Section 9.1.2 applied to $\texttt{s}$ in the pre-state. The function *post_state* is designated to be treated by HOL-TestGen as the program under test.

We executed HOL-TestGen with an unfolding depth of two to this test specification. Applying the SMT solving techniques presented in Section 4.3 yielded concrete pre-states, object identifiers for `s` and integer arguments `x` to `insert` that constitute test cases. For example, one such test case was described by a pre-state given as the lambda expression

$$((\lambda\_.\ \mathit{Some}\ (\mathit{Node}\ 0\ \mathit{None}))(2 \mapsto \mathit{Node}\ 1\ (\mathit{Some}\ 3))$$

together with the object identifier 2 for the `s` reference. As can be seen from evaluating the lambda expression on this object identifier and the successor identifier 3, this test input corresponds to the list $[1, 0]$.

9.4 Related Work

UML models and OCL constraints can be used for testing in various ways [131]. There have been several proposals for test generation from UML/OCL models, however, none of them is based on the three-valuedness of OCL. The most closely related publications [148, 2, 153] are all inspired by the seminal work of [61] and, thus, share the idea of using symbolic DNF computation for partitioning the input space. Moreover, sequence diagrams have been used an input for test generation, e. g., [108]. Pairwise testing of OCL contracts has been proposed in e. g., [121]. Finally, [72] applies random testing techniques to the analysis of OCL specifications.

A well known tool for generating tests from object-oriented specifications is Spec Explorer [149]. Spec Explorer processes Spec# specifications and is based on the traversal of finite state machines. TestEra [93] is another tool that generates object-oriented tests from preconditions in the Alloy language. The precondition is mapped to a Boolean satisfiability problem, and a SAT solver is used to enumerate all solutions. Each solution is interpreted as a test case. A difficulty inherent in this approach is that in general, the same test case may be represented by different solutions to the satisfiability problem, depending on the Boolean encoding used. Thus, a particular Boolean encoding has to be chosen, and additional constraints that enforce the uniqueness of test cases may be necessary. In contrast, in our approach test cases are identified by monomes of the DNF, independent from any encoding possibly used for constraint solving.

Chapter 10

Conclusion

10.1 Summary

Both test generation and animation are valuable techniques for software development. We presented novel approaches to these tasks. We described new fundamental techniques for constraint solving and test derivation as well as their application to concrete testing and animation scenarios.

Constraint solving is essential to both test generation and animation. Random solving techniques have been used for a long time for generating tests for functional programs. However, SMT solving can be substantially more efficient for a large class of constraints. We showed how counterexamples returned from SMT solvers can be used for solving constraints in the test generation tool HOL-TestGen that is based on the Isabelle theorem prover. A counterexample is interpreted and transformed to an Isabelle theorem that expresses a solution to the constraint. Recursive functions in constraints are unavoidable since recursive functions are an essential feature of HOL, the language processed by HOL-TestGen. Although constraints with recursive functions are in general undecidable, we show how under-approximation can yield decidable constraints. We demonstrated how Isabelle's support for user interaction can be used to further extend the class of constraints that can be handled with SMT solvers within Isabelle.

For solving constraints that arise when animating OCL specifications, we propose to represent the constraints as arithmetic formulas with bounded quantifiers. The language of arithmetic formulas with bounded quantifiers is very expressive, and most OCL language constructs can be mapped conveniently to it. As further benefit, constraints expressed as arithmetic formulas with bounded quantifiers can be solved using an eager SMT approach in a straightforward and efficient manner. We also showed how an objective function can be taken into account when solving this kind of constraints.

For test derivation, we presented an approach based on modular test derivation rules. We showed how the common steps for test derivation within a theorem prover can be united in a kernel, while test derivation rules that are tailored to specific types of data and specifications are provided by pluggable modules. We outlined an implementation of this architecture in tactical code.

Furthermore, we presented applications of these novel fundamental techniques to concrete testing and animation scenarios. We defined tailored test derivation rules for generating tests from object-oriented specifications with recursive functions. These rules unfold recursive functions and enumerate object graphs systematically.

For animation, we presented an efficient as well as fully automatic approach to the animation of OCL operation contracts. It is implemented in the tool OCLexec that generates from OCL operation contracts corresponding Java implementations which call a constraint solver at runtime. We showed how most OCL constraints can be expressed by arithmetic formulas with bounded quantifiers. The addition of objective functions to OCL operation contracts makes the animation of OCL operation contracts much more useful. Case studies demonstrate that our animation approach can handle problem instances of considerable size. For implementing tool support for OCL, a precise semantics of OCL is essential. We clarified the semantics of undefined values in OCL by extending an embedding of OCL into HOL that was initiated by the “HOL-OCL” project.

10.2 Future Work

The implementation of tools is a time-consuming activity, and thus it is natural that many desirable enhancements remain to be carried out. Here we list some of these possible targets for future work on test generation and animation:

- An asset of HOL-TestGen is the explicit representation of test hypotheses. However, for all but the smallest test suites, the generated test hypotheses are too numerous to be comprehensible when examined manually. It would be interesting to seek well-formedness rules for test hypotheses which give further assurance that a good test coverage has been achieved.

- An integration of the HOL-OCL [33] system with HOL-TestGen could provide further support for test generation from OCL specifications. The translation of OCL constraints to HOL presented in Chapter 9 was carried out manually, although the necessary semantic theory is available.

- It is plausible that symmetry breaking could improve the efficiency of animation. Symmetries in post-states arise when animation creates new class instances, for example when animating the contracts in Chapter 6. There is a large amount of literature on symmetry breaking for constraint solving, see [144] for an approach that is targeted at object-oriented structures.

- In order to make tool support for the animation of OCL operation contracts useful in practice, it must be integrated into a complete modeling toolchain. Such a toolchain must in particular provide support for editing operation contracts, including invariability clauses and objective functions.

Bibliography

[1] General algebraic modeling system (GAMS). http://www.gams.com.

[2] B. K. Aichernig and P. A. P. Salas. Test case generation by OCL mutation and constraint solving. In *QSIC*, pages 64–71. IEEE Computer Society, 2005.

[3] D. H. Akehurst, S. Zschaler, and W. G. J. Howells. OCL: Modularising the language. *ECEASST*, 9, 2008.

[4] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In Ramakrishnan and Rehof [129], pages 109–123.

[5] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In Grumberg and Huth [79], pages 134–138.

[6] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2007.

[7] D. Aspinall. Proof general: A generic tool for proof development. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[8] J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.*, 9(1), 2007.

[9] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005.

[10] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[11] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2000.

[12] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

[13] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6:387–405, November 1991.

[14] D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–64, 2010.

[15] J. Blanchette. Relational analysis of (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. *Software Quality Journal*.

[16] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Kaufmann and Paulson [92], pages 131–146.

[17] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In Kaufmann and Paulson [92], pages 179–194.

[18] M. Boshernitsan, R.-K. Doong, and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In L. L. Pollock and M. Pezzè, editors, *ISSTA*, pages 169–180. ACM, 2006.

[19] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005.

[20] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B — a constraint solver to animate a B specification. *Int. J. Softw. Tools Tech. Trans.*, 6(2):143–157, 2004.

[21] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[22] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient theory combination via boolean search. *Inf. Comput.*, 204(10):1493–1525, 2006.

[23] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In *ICST*, pages 345–354. IEEE Computer Society, 2010.

[24] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In R. Breu, J. Crampton, and J. Lobo, editors, *SACMAT*, pages 133–142. ACM, 2011.

[25] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2008.

[26] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, volume 6627 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2010.

[27] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2009.

[28] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2004.

[29] A. D. Brucker and B. Wolff. Interactive testing with HOL-TestGen. In W. Grieskamp and C. Weise, editors, *FATES*, volume 3997 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2005.

[30] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[31] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen with an application to firewall testing. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2007.

[32] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in hol. *J. Autom. Reasoning*, 41(3-4):219–249, 2008.

[33] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Inf.*, 46(4):255–284, 2009.

[34] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2011. To appear.

[35] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors,

TACAS, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[36] R. Brummayer and A. Biere. Effective bit-width and under-approximation. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, volume 5717 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2009.

[37] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[38] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In Damm and Hermanns [55], pages 547–560.

[39] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *STTT*, 11(2):95–104, 2009.

[40] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL operation contracts. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2009.

[41] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artif. Intell.*, 162(1-2):89–120, 2005.

[42] M. Carlier and C. Dubois. Functional testing in the Focal environment. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

[43] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FocalTest. In J. A. M. Cordeiro, M. Virvou, and B. Shishkov, editors, *ICSOFT (2)*, pages 82–91. SciTePress, 2010.

[44] N. Cataño and T. Wahls. Executing JML specifications of Java card applications: a case study. In Shin and Ossowski [137], pages 404–408.

[45] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reasoning*, 41(1):33–59, 2008.

[46] P. Chalin and F. Rioux. Non-null references by default in the Java modeling language. In *Proceedings of the 2005 conference on Specification and verification of component-based systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.

[47] J. Christiansen and S. Fischer. Easycheck - test data for free. In J. Garrigue and M. V. Hermenegildo, editors, *FLOPS*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[48] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[49] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[50] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proc. Wsh. Model Computation — Principles, Algorithms, Applications*, Miami, Florida, 2003.

[51] T. Clark and J. Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer, 2002.

[52] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[53] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. In Clark and Warmer [51], pages 115–149.

[54] S. Cook and P. Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[55] W. Damm and H. Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[56] M. Davis. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly*, 80, 1973.

[57] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[58] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In Ramakrishnan and Rehof [129], pages 337–340.

[59] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[60] A. Dick, P. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J. Nicholls, editor, *Proc. 4^{th} Z Users Workshop*, Workshops in Computing, pages 71–85, Oxford, 1989. Springer.

[61] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. G. Larsen, editors, *FME*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 1993.

[62] V. Doma and R. A. Nicholl. EZ: A system for automatic prototyping of Z specifications. In S. Prehn and W. J. Toetenel, editors, *VDM Europe (1)*, volume 551 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1991.

[63] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[64] S. Flake. Towards the completion of the formal semantics of OCL 2.0. In V. Estivill-Castro, editor, *ACSC*, volume 26 of *CRPIT*, pages 73–82. Australian Computer Society, 2004.

[65] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In W. A. H. Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2003.

[66] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002.

[67] A. M. Frisch and P. J. Stuckey. The proper treatment of undefinedness in constraint languages. In I. P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2009.

[68] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[69] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In Damm and Hermanns [55], pages 519–531.

[70] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[71] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comp. Prog.*, 69(1–3):27–34, 2007.

[72] M. Gogolla, L. Hamann, and M. Kuhlmann. Proving and visualizing OCL invariant independence by automatically generated test cases. In G. Fraser

and A. Gargantini, editors, *TAP*, volume 6143 of *Lecture Notes in Computer Science*, pages 38–54. Springer, 2010.

[73] M. Gogolla, M. Kuhlmann, and F. Büttner. A benchmark for OCL engine accuracy, determinateness, and efficiency. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 446–459. Springer, 2008.

[74] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In Clark and Warmer [51], pages 85–114.

[75] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106. Morgan Kaufmann, 2002.

[76] J. Gray and S. Schach. Constraint animation using an object-oriented declarative language. In A. J. Turner, editor, *ACM Southeast Regional Conference*, pages 1–10. ACM, 2000.

[77] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB*, volume 1878 of *Lecture Notes in Computer Science*, pages 414–432. Springer, 2000.

[78] H. Grönniger, J. O. Ringert, and B. Rumpe. System model-based definition of modeling language semantics. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2009.

[79] O. Grumberg and M. Huth, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*. Springer, 2007.

[80] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *Operating Systems Review*, 41(4):3–11, 2007.

[81] R. Hennicker, A. Knapp, and H. Baumeister. Semantics of OCL operation specifications. *Electr. Notes Theor. Comput. Sci.*, 102:111–132, 2004.

[82] A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[83] H. Hußmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. *Sci. Comp. Prog.*, 44(1):51–69, 2002.

[84] ISO/IEC 13568. *Information technology — Z formal specification notation — Syntax, type system and semantics.*

[85] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[86] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[87] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *ISSTA*, pages 239–249, 1996.

[88] E. Jaffuel and B. Legeard. LEIRIOS test generator: Automated test generation from B models. In Julliand and Kouchnarenko [90], pages 277–280.

[89] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[90] J. Julliand and O. Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.

[91] JUnit. http://www.junit.org.

[92] M. Kaufmann and L. C. Paulson, editors. *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010.

[93] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[94] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *OOPSLA*, pages 231–245, 2002.

[95] P. W. M. Koopman, A. Alimarine, J. Tretmans, and M. J. Plasmeijer. Gast: Generic automated software testing. In R. Pena and T. Arts, editors, *IFL*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2002.

[96] P. Kosiuczenko. Specification of invariability in OCL. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 676–691. Springer, 2006.

[97] R. A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, 1988.

[98] J. Krajíček. *Bounded arithmetic, propositional logic, and complexity theory.* Cambridge University Press, New York, NY, USA, 1995.

[99] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, B. R. Haverkort, M. Leucker, and J. van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296. Springer, 2006.

[100] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* PhD thesis, Technische Universität München, 2009.

[101] M. P. Krieger and A. D. Brucker. Extending OCL operation contracts with objective functions. *ECEASST*, 44, 2011.

[102] M. P. Krieger, A. Knapp, and B. Wolff. Automatic and efficient simulation of operation contracts. In E. Visser and J. Järvi, editors, *GPCE*, pages 53–62. ACM, 2010.

[103] M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In J. Bishop and A. Vallecillo, editors, *TOOLS (49)*, volume 6705 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2011.

[104] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[105] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual (revision 1.2), Feb. 2007. Available from `http://www.jmlspecs.org`.

[106] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In Shin and Ossowski [137], pages 615–622.

[107] M. Leuschel and M. Butler. ProB: An automated analysis toolset for the B method. *Int. J. Softw. Tools Tech. Trans.*, 10(2):185–203, 2008.

[108] B.-L. Li, Z. shu Li, L. Qing, and Y.-H. Chen. Test case automate generation from UML sequence diagram and OCL expression. In *CIS*, pages 1048–1052. IEEE Computer Society, 2007.

[109] S. Markovic and T. Baar. Semantics of OCL specified with QVT. *Software and System Modeling*, 7(4):399–422, 2008.

[110] B. Marre. LOFT: A tool for assisting selection of test data sets from algebraic specifications. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 799–800. Springer, 1995.

[111] D. C. J. Matthews and M. Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In L. Petersen and E. Pontelli, editors, *DAMP*, pages 53–62. ACM, 2010.

[112] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 511–520. ACM, 2011.

[113] M. Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 20–29, New York, NY, USA, 2009. ACM.

[114] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

[115] G. Myers. *The art of software testing.* Business data processing. Wiley, 1979.

[116] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[117] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[118] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.

[119] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2006.

[120] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[121] S. Noikajana and T. Suwannasart. An improved test case generation method for web service testing from WSDL-S and OCL with pair-wise testing technique. In S. I. Ahamed, E. Bertino, C. K. Chang, V. Getov, L. Liu, H. Ming, and R. Subramanyan, editors, *COMPSAC (1)*, pages 115–123. IEEE Computer Society, 2009.

[122] I. Oliver and S. Kent. Validation of object oriented models using animation. In *EUROMICRO*, pages 2237–. IEEE Computer Society, 1999.

[123] UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.

[124] UML 2.0 OCL specification, Apr. 2006. Available as OMG document formal/06-05-01.

[125] UML 2.2 OCL specification, Feb. 2010. Available as OMG document formal/2010-02-01.

[126] L. Paulson. *ML for the working programmer*. Cambridge University Press, 1996.

[127] L. C. Paulson. The relative consistency of the axiom of choice – mechanized using Isabelle/ZF. *LMS Journal of Computation and Mathematics*, 6:2003, 1999.

[128] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007.

[129] C. R. Ramakrishnan and J. Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[130] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[131] B. Rumpe. Model-based testing of object-oriented systems. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 2852 of *Lecture Notes in Computer Science*, pages 380–402. Springer, 2002.

[132] R. Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[133] R. Sedgewick. *Algorithms*. Addison-Wesley, Second edition, 1988.

[134] T. Servat. BRAMA: A new graphic animation tool for B models. In Julliand and Kouchnarenko [90], pages 274–276.

[135] S. A. Seshia and R. E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. *Logical Methods in Computer Science*, 1(2), 2005.

[136] L. Shan and H. Zhu. A formal descriptive semantics of UML. In S. Liu, T. S. E. Maibaum, and K. Araki, editors, *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 375–396. Springer, 2008.

[137] S. Y. Shin and S. Ossowski, editors. *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*. ACM, 2009.

[138] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. In E. Giunchiglia and A. Tacchella, editors, *Sel. Rev. Papers 6th Int. Conf. Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*. Springer, May 2004.

[139] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In P. Barahona and J. J. Alferes, editors, *EPIA*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[140] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[141] N. Tamura, T. Tanjo, and M. Banbara. Solving constraint satisfaction problems with SAT technology. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS*, volume 6009 of *Lecture Notes in Computer Science*, pages 19–23. Springer, 2010.

[142] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg, 2008.

[143] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In *Frontiers of Combining Systems (FroCos)*, pages 103–119, 1996.

[144] E. Torlak and D. Jackson. Kodkod: A relational model finder. In Grumberg and Huth [79], pages 632–647.

[145] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[146] G. Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983.

[147] M. Utting. Data structures for Z testing tools. In G. Schellhorn and W. Reif, editors, *Proc. 4^{th} Wsh. Tools for System Design and Verification (FM-TOOLS'00)*. Technical Report 2000-07, Universität Ulm, 2000.

[148] L. van Aertryck and T. Jensen. UML-CASTING: Test synthesis from UML models using constraint resolution. In J.-M. Jézéquel, editor, *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*. INRIA, 2003.

[149] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[150] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[151] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[152] T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.

[153] S. Weißleder and B.-H. Schlingloff. Quality of automatically generated test cases based on OCL expressions. In *ICST*, pages 517–520. IEEE Computer Society, 2008.

[154] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

[155] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.