Second-Order Principles in Specification Languages for Object-Oriented Programs

Bernhard Beckert[1] and Kerry Trentelman[2]

[1] Department of Computer Science, University of Koblenz-Landau
beckert@uni-koblenz.de
[2] Automated Reasoning Group, Australian National University
Kerry.Trentelman@anu.edu.au

Abstract. Within the setting of object-oriented program specification and verification, pointers and object references can be considered as relations between the elements of a data structure. When we specify properties of these data structures, we often describe properties of relations. Hence it is important to be able to talk about relations and their properties when specifying object-oriented programs or programs with pointers. Many interesting properties of relations such as transitive closure, finiteness, and generatedness are not expressible in first-order logic (FOL); hence neither are they expressible in first-order fragments of specification languages. In this paper we give an overview of the different ways such properties can be expressed in various logics, with a particular emphasis on extensions of FOL, *i.e.* transitive closure logic, fixed-point logic, and first-order dynamic logic. Within the paper we also discuss which of these extensions already are – or in fact should be – implemented within specification languages. We feel that such a discussion is necessary since it is often the case that when an extension of FOL is implemented within a specification language it is done so in an *ad hoc* manner or the underpinning logical concepts are not well documented..

1 Introduction

When it comes to specifying object-oriented programs, we need to be able to: (a) refer to a set of particular objects in an object structure; and (b) talk about the properties of the relation between the objects. As an example, consider the definition of sets of related objects which are used in modifies clauses (a modifies clause allows one to specify those parts of a program state that are exclusively allowed to change [28, 6]). To illustrate, suppose we have a linked list with objects of class `Node` having a `next` field. For a method say, `sortInPlace`, it would be useful to be able to write `list.next`* in the method's modifies clause, where * denotes some form of transitive closure. Its semantic intention would then be that the set of locations that are reachable from `list` using the field `next` may be modified during the method's execution. One may also wish to specify that the list is not cyclic; assuming that this is the case, a field such as `position()` may be introduced such that it returns a reference to a node at a given position. If the position is less than or greater than 1, then the field returns `null`.

All specification languages have some form of modification which allows them to extend beyond the limitations of first-order logic. For example the query language SQL implements fixed-point logic, the Object Constraint Language OCL uses the `iterate` and `let` constructs, the Common Algebraic Specification Language CASL uses the notion of freeness, and the Java Modeling Language, JML, incorporates built-in recursion. However it is often the case that the modifications made to specification languages are done in a "make-do" fashion and their designers are unaware of the logic underpinning their decisions. In this paper we attempt to clarify what is really going on within these specification languages.

Our work is carried out in the framework of the KeY project. The KeY system is a commercial CASE tool augmented with specification and deductive verification functionalities [1] (see website at `www.key-project.org`). KeY uses the Unified Modeling Language UML for visual modelling of designs and specifications, along with OCL for specifying constraints and other expressions attached to the models [29]. The target language for program verification is Java. Both the specification language OCL and the verification language of the KeY tool – namely, dynamic logic – have second-order elements (as described in Section 4). Our case study experience has shown that often there is a need for expressing second-order principles in a more usable and/or flexible way; this need provides the motivation behind our investigations. In particular, a modifies clause has been recently implemented within the KeY system [6]. As the above example demonstrates, it would be advantageous to be able to express transitive closure in OCL in an easier fashion than the current method – which is by using the OCL `iterate` construct – described in Section 4.

The paper is organised as follows: in Section 2 we look at how one goes about expressing properties of relations and composing relations. We discuss various properties which may or may not be expressed in first-order logic. This logic's lack of expressiveness leads us to an examination of a number of extensions of first-order logic in Section 3. In Section 4 we discuss several specification languages and the approaches they take in determining properties of relations. Finally, we draw conclusions in Section 5.

2 Relations and Relational Formulae in a FOL Setting

We are interested in both expressing properties of relations and composing relations in relational formulae. In this section we provide the basic definitions for these notions and briefly discuss relational algebra. We conclude by describing a number of properties which can or cannot be expressed in first-order logic. However, before we begin, we need to stipulate what we mean by a relation within an object-oriented language.

Following [30] we say that a relation expresses (the symmetric form of) those associations which are represented in a programming language as pointers or object references. Hence we model both object references and pointers as first-order functions on objects.

A property P of a relation R (a formula with two free variables) is said to be expressible if there is a closed formula $\phi_P(R)$ such that, for all models M, the interpretation R^M has property P if and only if $\phi_P(R)$ is true in M. Here R^M is the (single) interpretation of relation R in M. The formula $\phi_P(R)$ must be effectively constructible from any given R in a uniform way. This notion is extended to properties of tuples of relations. Formally, a property is a relation on relations.

A composition C of relations $R_1, \ldots, R_k$ is expressible if there is a formula $\psi_P(R_1, \ldots, R_k)(x, y)$ with free variables x and y such that $(\psi_P(R_1, \ldots, R_k))^M$ is the relation composed from $R_1^M, \ldots, R_k^M$. Here $(\psi_P(R_1, \ldots, R_k))^M$ is the (single) interpretation of $\psi_P(R_1, \ldots, R_k)(x, y)$ in M. The formula $\psi_P(R_1, \ldots, R_k)$ must be effectively constructible from any given $R_1, \ldots, R_k$ in a uniform way. Formally, a composition is a function on relations.

Note that the constructibility of ϕ and ψ neither implies the decidability of P, nor respectively the computability of C. This is because the validity of the constructed formula is in general undecidable. Moreover, the composition of relations may be iterated, but the properties themselves cannot be iterated.

Relational algebra is a formal system used for manipulating relations. The set of its operations may vary per definition, but it usually includes set operations – since relations are sets of tuples – and special operators defined for relations such as select, project, and join. The select operator selects tuples from a relation whose attributes meet the selection criteria (which is normally expressed as a predicate). The project operator selects certain attributes from a single relation, discarding the rest. The join operator composes two relations. Relational algebra forms the basis of a multitude of relational query languages; these are used in order to manipulate the data of a relational database. We discuss aspects of one of the standard languages, SQL, in Section 4.

Examples of properties expressible in FOL are reflexivity and transitivity; concatenation is an expressible composition: We say that R is reflexive if $\forall x.\, xRx$ and R is transitive if $\forall x \forall y \forall z.\, (xRy \wedge yRz \rightarrow xRz)$. The concatenation of two relations R and S is expressible by $R \circ S \equiv \{(x, z) \mid \exists y.\, xRy \wedge ySz\}$. Note that we use the notation xRy for $(x, y) \in R$ and $R(x, y)$ respectively.

On the other hand, properties that demand the finiteness of certain sets of elements are not expressible. For example: "all elements are at most related to a finite number of other elements". Furthermore, many properties that demand the existence of a finite but unknown number of elements which are related in a certain way are not expressible. For example quantifications such as $\exists n.\, \exists x_1 \ldots x_n$ (which are routinely used in mathematical notation) do not exist in FOL and often cannot be expressed by any other means.

Another typical but important example is transitive closure. The transitive closure of a relation R is the relation $TC(R)$ such that for all elements x and y the relation $TC(R)(x, y)$ holds if and only if there is a finite number of intermediate points $z_0, \ldots, z_n$ where $n \in \mathbb{N}$ with $x = z_0$, $y = z_n$ and $z_{i-1} R z_i$ for $1 \leq i \leq n$. Accordingly, one cannot express in FOL that some point b is R-reachable from some other point a, *i.e.* $TC(R)(a, b)$. An alternative – yet equivalent – definition

of transitive closure $TC(R)$ is: (1) $TC(R)$ is transitive; (2) $R \subseteq TC(R)$ and; (3) if R' is transitive and $R \subseteq R'$ then $TC(R) \subseteq R'$. The latter condition is not expressible in FOL as it implicitly quantifies over R'.

It is important to note, however, that the transitive closure of a structure can be expressed in a FOL setting if the structure is both finite and acyclic (see Section 4).

3 Extensions of FOL

In this section we investigate a number of extensions of first-order logic including transitive closure logic, fixed-point logic, and first-order dynamic logic. These extensions allow us to express various properties and compositions of relations that cannot be expressed using first-order logic alone.

Transitive Closure Logic. First-order logic extended by a transitive closure operator – written $FO(TC)$ and called transitive closure logic – was first introduced by Immerman [16]. If we let the formula $\phi(\bar{x}, \bar{y})$ represent a binary relation on two n-tuples of domain variables – which range over the universe of a Kripke structure – then the reflexive transitive closure of this relation is expressed by $TC_{\bar{x},\bar{y}}\phi(\bar{x}, \bar{y})$, or more succinctly $TC\phi$. Strict transitive closure is denoted $TC^s\phi$. This represents the transitive closure of ϕ as opposed to the reflexive transitive closure of ϕ. The restriction $FO^2(TC)$ is such that only two variables x and y may appear in a formula ϕ. For example, the formula $\exists y.\ ((TC_{x,y}R_a(x, y))(x, y) \wedge p(y))$ expresses "there is a path of a-edges from x to a vertex where p holds".

Reachability Logic $\mathcal{RL}$ is a fragment of $FO^2(TC)$ with an unbounded number of boolean variables in addition to the two domain variables x and y [3]. Boolean variables are first-order variables restricted to range over 0 and 1. Formulae of the logic are constructed using an adjacency formula $\delta(x, \bar{b}, y, \bar{b'})$ which is a binary relation between two n-tuples $(x, b_1, \ldots, b_{n-1})$ and $(y, b'_1, \ldots, b'_{n-1})$. This is in fact a disjunction of conjunctions where each conjunction contains at least one of the following: $x = y$, $R_a(x, y)$, or $R_a(y, x)$ for some binary relation R_a. Hence the adjacency formula necessarily implies that there is an edge from x to y, or an edge from y to x, or that x is equal to y. Conjuncts may also contain expressions of the form $\neg(b_i = b_j)$, $b_i = 0$, or $b_i = 1$. For $\phi \in \mathcal{RL}$ the formulae $NEXT(\delta)\phi$ (denoting $\exists y.\ (\delta(x, \bar{0}, y, \bar{1}) \wedge \phi[y/x])$), $REACH(\delta)\phi$ (*i.e.*, $\exists y.\ (TC\delta)(\delta(x, \bar{0}, y, \bar{1}) \wedge \phi[y/x])$), and $CYCLE(\delta)$ (*i.e.*, $(TC^s\delta)(\delta(x, \bar{0}, x, \bar{0}))$) are also formulae of $\mathcal{RL}$. Hence it is possible to describe in this logic: steps out of the current vertex x, paths out of x, and cycles from x back to itself.

Importantly, the boolean variables allow Propositional Dynamic Logic (PDL) and the variation of Computational Tree Logic, CTL*, to be embedded in $\mathcal{RL}$. Consider the PDL formula $\langle\alpha\rangle p$, which is a true property of a state s whenever there is some state t in which p holds that is reachable from s by execution of α. The regular expression α can be translated into an non-deterministic finite automaton N_α with n states. Within the framework of $\mathcal{RL}$ the adjacency formula

of α is a translation of the transition relation of N_α, whereby each state of the automaton is represented by $k \equiv 1 + \log n$ bits with $\bar{0}$ and $\bar{1}$ representing the initial and final states respectively. For example, if α is the sequential composition $\pi_0;\pi_1$ then a transition from state s to state t in $N_{\pi_0;\pi_1}$ is represented by the adjacency formula $R_{\pi_0}(x,y) \wedge b_1 \ldots b_k = s \vee R_{\pi_1}(x,y) \wedge b'_1 \ldots b'_k = t$ where $b_1 \ldots b_k$ is the initial state and $b'_1 \ldots b'_k$ is the final state. Hence an example of a formula in $\mathcal{RL}$ is $REACH(\delta)p$ where $\delta(x, b_1, b_2, y, b'_1, b'_2)$ is $(R_{\pi_0}(x,y) \wedge b_1 b_2 = 00 \wedge b'_1 b'_2 = 01) \vee (R_{\pi_1}(x,y) \wedge b_1 b_2 = 01 \wedge b'_1 b'_2 = 11)$. This has the meaning that it is possible to take the path of a π_0-edge followed by a π_1-edge to a point where p holds; this is just $\langle \pi_0;\pi_1 \rangle p$ in PDL.

Regular Expressions Over Relations. Kleene algebras are algebraic structures that generalise the operations of regular expressions. A Kleene algebra consists of a set K with binary $+$ and $\cdot$ operations, a unary operation *, and constants 0 and 1. In general the algebra's operational semantics depends on the model, but typically * involves some notion of finite iteration. A Kleene algebra gives rise to a relational algebra extended with reflexive transitive closure when the following interpretations of the operations are made: operation $\cdot$ as join; element 0 as the null/empty relation; element 1 as the identity relation; and * as the reflexive transitive closure of a relation.

As mentioned previously, an extension of first-order logic with the ability to write `list.next`* – or even more generally, to be able to use regular expressions to describe terms or term sets – would be very useful. There exist approaches which allow an extended syntax for terms in first-order logic. For example in [10] recursive term definitions are added to first-order logic.

Rather than using regular expressions and Kleene algebras to extend FOL, it is possible to manipulate FOL formulae such that they fulfill a purpose similar to that of regular expressions.

Two ways to define words and/or formal languages are by using: (1) predicate logics, such that each model corresponds to a word in the language; and (2) modal logics, such that each path in a Kripke structure corresponds to a word. There is a large amount of literature on the latter. For (1), we fix a family of signatures Σ_A. They contain the binary relation symbol $<$, a constant symbol first, a unary postfix function $+1$, and for every a in the alphabet A, we have the unary relation symbol Q_a. The set of words over A is denoted A^*. For $w \in A^* \backslash \{\Lambda\}$, where Λ is the empty word, the associated Σ_A-structure is denoted $\mathcal{M}_w$ (the empty model is not possible). The formula $\mathcal{M}_w \models Q_a(\mathsf{first})$ holds true if and only if the first letter of w is a. The formula $\mathcal{M}_w \models Q_b(+1)$ holds true if and only if the second letter of w is b, *etc.* For (2), we express information about semi-structured data – represented as a graph – by imposing constraints on the possible paths through the graph. Such a constraint might be "all objects reachable by a path p are also reachable *via* a path q", where p and q are sequences of labels possibly involving regular expressions. In order to check that the constraints hold, we recast them as model or satisfiability checking tasks in some logic (usually modal).

For example, see [2] where this is done using propositional dynamic logic, and [12] where this is done using monadic second-order logic.

Fixed-Point Logic. Fixed-point logics are particularly well-suited for modelling recursion and have consequently found applications in various areas of computer science such as database theory, finite model theory and, formal verification. Following [22, 13], for a set A and a function $F : \wp(A) \to \wp(A)$, a fixed-point P of F is any set $P \subseteq A$ such that $F(P) = P$. A fixed point Q is called the least (greatest) fixed-point of F if and only if $Q \subseteq P$ ($P \subseteq Q$) holds for all fixed points P of F. The function F is said to be monotone if $F(X) \subseteq F(Y)$ for all $X \subseteq Y \subseteq A$. A well-known theorem by Knaster and Tarski states that every monotone function has a least and a greatest fixed-point [33]. For limit ordinals λ and the monotone function F, consider the sequence $(X^\alpha)_{\alpha \in Ord}$ of sets $X^\alpha \subseteq A$ defined by (i) $X^0 = \emptyset$, (ii) $X^{\alpha+1} = F(X^\alpha)$, and (iii) $X^\lambda = \bigcup_{\xi<\lambda} X^\xi$. A fixed-point X^∞ is reached in this sequence whereby $X^\infty = X^\alpha$ for the least ordinal α such that $X^\alpha = X^{\alpha+1}$. This fixed-point X^∞ is called the inductive fixed-point of F. A second theorem by Knaster and Tarski states that the least and inductive fixed-points coincide, hence any least fixed-point of a monotone function can be defined inductively by a sequence of sets as described above. Dually, the greatest fixed-point of a monotone function F can be defined inductively using the sequence $(X^\alpha)_{\alpha \in Ord}$ of sets $X^\alpha \subseteq A$ defined by (i) $X^0 = A$, (ii) $X^{\alpha+1} = F(X^\alpha)$, and (iii) $X^\lambda = \bigcap_{\xi<\lambda} X^\xi$. Note that if F is inflationary (*i.e.* $X \subseteq F(X)$ for all $X \subseteq A$) rather than monotone, then X^∞ is called the inflationary fixed-point of F. Next let τ be a signature, *i.e.* a finite set of relation symbols, and let $\mathcal{A}$ be a structure consisting of a universe A and interpretations for each relation symbol in τ. Consider a first-order formula $\varphi(R, \bar{x})$ with R a k-ary free relation symbol not occurring in τ and $\bar{x}$ a k-tuple of free variables. On $\mathcal{A}$ the formula φ induces a fixed-point operator $F_\varphi : \wp(A^k) \to \wp(A^k)$ such that $F_\varphi(R) = \{\bar{a} \mid (\mathcal{A}, R) \models \varphi(\bar{a})\}$. Here $(\mathcal{A}, R) \models \varphi(\bar{a})$ means that formula φ is satisfied by the interpretation that assigns to each variable x_i of $\bar{x}$ the element a_i of $\bar{a} \in A^k$.

Below we investigate three fundamental fixed-point logics: monotone, least, and inflationary fixed-point logics. First of all we discuss monotone fixed-point logic. Using this logic we can nest inductive definitions; from one fixed-point built-up from a formula we can define another.

Monotone Fixed-Point Logic. Monotone Fixed-Point Logic *MFP* is the extension of FOL by the following rule: if R is a k-ary free relation variable, $\bar{x}$ is a k-tuple of free first-order variables, $\bar{t}$ is a k-tuple of terms and $\varphi(R, \bar{x})$ is a formula such that the corresponding operator F_φ is monotone on all structures, then $[lfp_{R,\bar{x}}\varphi](\bar{t})$ is also a formula. For any structure $\mathcal{A}$ that provides an interpretation of the free variables of φ except for $\bar{x}$, $\mathcal{A} \models [lfp_{R,\bar{x}}\varphi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the least fixed-point of the operator defined by $\varphi(R, \bar{x})$. As we have mentioned previously, the least and greatest fixed-point of any monotone operator always exists. However it is undecidable as to whether a formula induces

a monotone operator. In order to guarantee monotonicity on the operator one can restrict the formulae such that they are positive in the relation variable R. This leads us to the definition of least fixed-point logic.

Least Fixed-Point Logic. Least Fixed-Point Logic *LFP* is the extension of FOL by the following rule: if R is a k-ary free relation variable, $\bar{x}$ is a k-tuple of free first-order variables, $\bar{t}$ is a k-tuple of terms and $\varphi(R, \bar{x})$ is a formula in which R occurs only positively, then $[\mathit{lfp}_{R,\bar{x}}\varphi](\bar{t})$ is also a formula. For any structure $\mathcal{A}$ that provides an interpretation of the free variables of φ except for $\bar{x}$, $\mathcal{A} \models [\mathit{lfp}_{R,\bar{x}}\varphi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the least fixed-point of the operator defined by $\varphi(R, \bar{x})$. Consider, for example, the directed graph (V, E), where V is a set of n vertices and $E \subseteq V \times V$ is a set of ordered pairs, *i.e.* edges. Then the transitive closure of E is defined as $[\mathit{lfp}_{R,x,y}\,(xEy \vee \exists z.\,(xRz \wedge zRy))](x, y)$.

Inflationary Fixed-Point Logic. Inflationary Fixed-Point Logic *IFP* can be considered the simplest non-monotone fixed-point logic. It is the extension of first-order logic by the following rule: if R is a k-ary free relation variable, $\bar{x}$ is a k-tuple of free first order variables, $\bar{t}$ is a k-tuple of terms and $\varphi(R, \bar{x})$ is a formula, then $[\mathit{ifp}_{R,\bar{x}}\varphi](\bar{t})$ is also a formula. Let $\mathcal{A}$ be a structure which provides an interpretation of the free variables of φ except for $\bar{x}$. The operator $I_\varphi(R) = \{\bar{a} \mid \bar{a} \in R \text{ or } (\mathcal{A}, R) \models \varphi(\bar{a})\}$ is inflationary and therefore has an inflationary fixed-point R^∞. Hence $\mathcal{A} \models [\mathit{ifp}_{R,\bar{x}}\varphi](\bar{t})$ if and only if the interpretation of $\bar{t}$ in $\mathcal{A}$ is in the inflationary fixed-point. An interesting result is that both least and inflationary fixed-point logics are equally expressive on arbitrary structures [21].

First-Order Dynamic Logic. The principle of dynamic logic (DL) is to facilitate the formulation of statements about program behaviour by integrating programs and formulas within a single language (see *e.g.* [15, 20] for general expositions of DL). By permitting arbitrary programs α as actions of a labelled multi-modal logic, dynamic logic provides formulas of the form $[\alpha]\phi$ and $\langle\alpha\rangle\phi$.

When considering states during program execution as worlds of modal logic, $[\alpha]\phi$ expresses that all (terminating) executions of program α lead to states in which ϕ holds; whereas $\langle\alpha\rangle\phi$ is a true property of a state s whenever there is some state t reachable from s by execution of program α in which ϕ holds. A Hoare-style specification $\{\phi\}\alpha\{\psi\}$ of partial correctness can be expressed as $\phi \to [\alpha]\psi$. In contrast to Hoare logic and temporal logic approaches to program verification, dynamic logic permits the expression of structural relationships between different programs by using multiple modalities. For example relative correctness statements like $\langle\alpha\rangle\phi \to \langle\alpha'\rangle\phi$ as well as nesting are possible, as in the formula $[\alpha](c \geq 0 \to \langle\alpha'\rangle c \leq d \cdot d)$.

Provided that they are computable, dynamic logic can express properties of relations that are ordinarily not expressible in pure first-order logic. For example to express that y is reachable from x *via* applications of the function *next* (*i.e.* x and y are related in the transitive closure of the relation p defined by $p(u, v)$ iff $v = \mathit{next}(u)$) can be expressed by $\langle\text{while } (x \neq y)\ x := \mathit{next}(x)\rangle \mathit{true}$.

4 Specification Languages

In this section we look at the approaches that specification languages take in defining transitive closure and similar properties of relations. Most require "hacks" to force a model's finiteness and acyclicity before transitive closure can be determined (an interesting and unique approach is taken by the Java Modeling Language JML).

Alloy. The Alloy Analyzer implements an automatic analysis method for formulae of relational logic [17, 18]. This logic acts as an intermediate language for the object modelling notation Alloy. It is a first-order logic with sets and relations whereby each formula is accompanied by a declaration that associates variables to their types. The combination of formula and declaration is called a problem. There are three kinds of types: set, relation, and function. Scalar variables are treated as singleton sets and sets are encoded as "degenerate" relations. For example, a scalar variable v of set type T can also be represented as the relational type $T \rightarrow \mathit{Unit}$, where Unit is a special type designed for this purpose.

A "navigation" expression $s.r$ denotes the image of a set s under a relation r. The encoding of sets as degenerate relations allows a uniform syntax to be given to such expressions, *i.e.* if p is a person then $p.\mathit{mother}$ will denote p's mother, whereas $p.\mathit{parents}$ will denote the set of p's parents. A transitive closure operator $+$ is also included in Alloy. For example, the formula $(p+) \cap \mathit{Id} = 0$ expresses that p is acyclic. Here Id is the identity relation and 0 is the empty relation.

Because relational logic is undecidable, it is in general impossible to prove that a formula is either consistent or valid. To determine for a given formula whether a model exists (within a particular scope), the Alloy Analyzer places restrictions on the size of the sets of the basic types. A model is said to be within a scope of k if it assigns to each type a set consisting of no more than k elements.

SQL. In order to manipulate the data of a relational database, relational query languages – based on relational algebra – are used. The database query language SQL was adopted as an industry standard in 1986 [32]. Having undergone two major revisions, SQL3 is now the current version.

```
WITH
RECURSIVE AncestorDescendant(ancestor, descendant) AS
    ((SELECT * FROM ParentChild)
    UNION
    (SELECT ad1.ancestor, ad2.descendant
    FROM AncestorDescendant ad1, AncestorDescendant ad2
    WHERE ad1.descendant = ad2.ancestor))
SELECT ancestor FROM AncestorDescendant WHERE descendant = "Mary";
```

Fig. 1. SQL specification.

Unlike its predecessors, SQL3 supports linear recursion; a recursive query has the form "`WITH RECURSIVE` R `AS` r Q`;`", where r is the expression that you want to recurse and R is its name that can then be used in the associated query expression Q. If we consider a query as a function on tables, then a recursive query computes the "fixed-point table" [34]. Essentially, we start with R as an empty table. We then evaluate r using the (temporary) contents of R and replace R with this new value. As long as $R^{new} \neq R$, we continue to evaluate r and replace R by its new value. Once $R^{new} = R$, we compute Q using the current contents of R and output the result. The example shown in Figure 1 outlines how we find `Mary`'s ancestors from the schema `ParentChild(parent, child)`. The first part of the recursive definition – utilising * – is the base case. Its meaning is that all "`parent/child`" pairs are also "`ancestor/descendant`" pairs. Although initially we know nothing about ancestor-descendant relationships, after the first round we deduce that parents are ancestors and children are descendants. In each subsequent round we use the facts deduced in previous rounds to get more ancestor-descendant relationships. We eventually stop when no new facts can be proven.

When the query Q is non-monotone, *i.e.* adding tuples to R might cause some tuple to be removed from the result of Q, then the fixed-point iteration may not converge. A way to circumvent this is to construct a dependency graph whereby: (1) each table R_i is a node; (2) there is a directed arc from R_i to R_j if R_i is defined in terms of R_j; and (3) the arc is labelled "`-`" if the query defining R_i is non-monotone with respect to R_j, *i.e.* by adding something to R_j we may cause something to be removed from R_i. The maximum number of `-` arcs on any path from R in the dependency graph is called the stratum of node R. A recursive query statement is said to be stratified if every node has a finite stratum, *i.e.* there are no cycles containing `-` arcs. Hence legal SQL3 recursive queries are required to be stratified. Note that this technique can also be used in other languages using fixed-point definitions in order to exclude non-monotonicity cases that lead to fixed-points being undefined.

CASL. The Common Algebraic Specification Language (CASL), has been developed by CoFI, the international Common Framework Initiative for algebraic specification and development (see website at `http://www.cofi.info`). The algebraic approach to software specification was conceived in the early 1970s, see for example [35]. Programs are considered as algebras consisting of datatypes and operations; the intended behaviour of a program is specified by formulae involving these operations. The development of dozens of languages, all with slight variations in syntax and semantics, demanded the need for a common framework, hence CoFI was formed. The resulting specification language CASL features partial functions, subsorts, sort generation constraints, first-order logic, and structural and architectural specifications [27].

In CASL datatypes are specified using the keyword **type** and are given in terms of sorts (*i.e.* the types of values) and constructors. Datatypes may be declared to be either **generated** or **free**. When a generated datatype is declared,

spec GENERATED_CONTAINER [**sort** *Elem*] =
generated type *Container* ::= *empty* | *insert*(*Elem*; *Container*)
pred $\_is\_in\_ : Elem \times Container$
$\forall e, e' : Elem;\ C : Container$
$\bullet\ \neg(e\ is\_in\ empty)$
$\bullet\ e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
end

spec TRANSITIVE_CLOSURE [**sort** *Elem* **pred** $\_R\_ : Elem \times Elem$] =
free { **pred** $\_R^+\_ : Elem \times Elem$
$\forall x, y, z : Elem\ \bullet\ x\ R\ y \rightarrow x\ R^+\ y$
$\bullet\ x\ R^+\ y \wedge y\ R^+\ z \rightarrow x\ R^+\ z$ }
end

Fig. 2. CASL specifications.

then the corresponding sort is constrained to be generated only by the declared constructors. For example in the specification of GENERATED_CONTAINER taken from the CASL User Manual [7] (see Figure 2), the generatedness constraint is such that any value of sort *Container* is denoted by a term built only with operators *empty*, *insert* and variables of sort *Elem*.

Note that within this specification, the pairs of underscores "_" indicate place-holders for the binary predicate *is_in* and the bulleted list features "axioms" which constrain the predicate. Essentially, the generatedness constraint allows one to prove – by induction on the declared constructors – properties of values of the sort *Container*. A **free** datatype declaration has the same interpretation as the **generated** datatype declaration with the additional property that all distinct constructor terms of the same sort denote distinct values.

In CASL a "freeness" constraint – using the keyword **free** – can be imposed on a predicate declaration. This has the effect that a predicate that is consistent with the given axioms but not a consequence of the axioms will be false; predicates hold minimally. We can see this in the specification of TRANSITIVE_CLOSURE shown in Figure 2 (also taken from [7]). Here the transitive closure of a binary relation R on some sort *Elem* is specified. Since predicates hold minimally in models of free specifications, R^+ is actually the smallest transitive relation including R.

OCL. The Object Constraint Language (OCL) [19] is a part of the Unified Modeling Language (UML) [14]. Currently the industry standard, UML allows software developers to graphically specify, visualise and document models of software systems. OCL can be used to augment UML object models with additional textual information which cannot otherwise be expressed by UML diagrams. This additional information takes the form of side-effect-free expressions and constraints. An expression is a specification of a value. A constraint is a restriction of one or more values in (part of) the object-oriented model. The semantics of OCL constraints is defined by an evaluation function which maps – in a given

object diagram – any constraint to one of the logical constants `true`, `false`, and `undefined`. Admissible diagrams are those whereby all constraints of the corresponding class diagram evaluate to `true`.

The type of an OCL expression is either pre-defined (`Boolean`, `Integer`, *etc.*) or it is the type of a class in the corresponding class diagram. Dot notation is used for accessing the attributes of objects. The basic data structures of OCL are the collections `Set`, `Bag` and `Sequence`.

OCL does not have a primitive operator for transitive closure, but it does allow recursion. Consider the following OCL invariant in the context `Person`, where `ancestors` are recursively defined in order to represent the transitive closure of the relation defined by `parents` (note that both `ancestors` and `parents` are of type `Set(Person)`): `ancestors = parents -> union(parents.ancestors)`. The expression `parents.ancestors` computes the set of all ancestors of a set of parents and returns a value of type `Set(Person)`.

Now suppose `A` is a parent of `B`, who in turn is a parent of `C`. Then the minimal object structure which solves the constraint is such that the parent of `B` is `A` and the ancestors of `C` include both `B` and `A`. However, additional solutions involve situations where `B` and `A` are both ancestors of each other and themselves. In our case we would prefer to use the minimal solution (corresponding to the minimal fixpoint), but this cannot always be found: there may be more than one equivalent solution, or it may not even exist. A suggestion to uniquely characterise the minimal solution is given in [11]. This paper suggests mimicking induction over a natural number `n`. This is exhibited in the following OCL specification.

```
ancestors_up_to(n) = if (n==1) then parents
                     else parents -> union(parents.ancestors_up_to(n-1))
Nat -> forall(n | ancestors_up_to(n) = ancestors_up_to(n+1)
                                  implies ancestors = ancestors_up_to(n))
```

Of course this makes the assumption that the models are finite. Alternatively, as done in [9], we can use the OCL `let` construct to stipulate that the inheritance relationship must be acyclic. Note that `self` refers to any instance of the class in which it is specified.

```
let parents = self.parents
let ancestors = self.parents -> union(self.parents.ancestors)
    in <some_expression_using_definition_of _ancestor>
```

The `let` construct is a new addition to OCL, introduced in version 2.0. The expression `let x = e1 in e2` evaluates expression `e2` with each occurrence of `x` replaced by the value of `e1`. Its use avoids evaluating the same expression multiple times. However the construct's semantics within OCL is not entirely clear [9]. Whether arbitrary recursively defined expressions are allowed is uncertain. Thus, using `let` to define transitive closure is not advised.

In [26] the transitive closure of a relation is computed by coding the well-known Warshall's algorithm in OCL. This coding makes use of the OCL `iterate` construct which iterates through all items of a collection, verifying a given condition and possibly updating the value of a variable returned at the end of

the iteration. The algorithm itself calculates the transitive closure of a directed graph (V, E), where V is a set of n vertices and $E \subseteq V \times V$ is a set of ordered pairs, *i.e.* edges. A path from vertex v_0 to v_k is denoted $v_0 \xrightarrow{*} v_k$ and is a sequence of edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The intuition behind Warshall's algorithm is this: if the graph contains paths $v \xrightarrow{*} w$ and $w \xrightarrow{*} u$ whose intermediate vertices belong to the set S, then the graph also contain a path $v \xrightarrow{*} u$ such that the intermediate vertices belong to $S \cup \{w\}$. The algorithm iterates from 1 to n. At the k^{th} iteration it selects paths whose intermediate vertices come from $\{v_1, \ldots, v_{k-1}\}$. Unfortunately the resulting OCL code of this algorithm is about one and a half pages in length; it is neither intuitive nor easy to read, and furthermore it requires the directed graph to be finite.

A transitive closure construct for OCL is proposed by Schürr in [31]. This is based on features of the path expression sublanguage – similar to OCL – of PROGRES, a graph transformation language. The transitive closure operator * is implemented to keep track of already visited objects and therefore avoids any cyclic problems. Schürr defines it as follows:

```
self.ancestors* = self.ancestorsClosure(self)
self.ancestorsClosure(visitedObj) =
    let S : ... = self.ancestors -> excludeAll(visitedObj) in
    S -> collect(ancestorsClosure(S -> union(visitedObj))) -> asSet
```

This definition will suffer from the unclear semantics of the `let` construct.

As mentioned in Section 2, it is possible to define the transitive closure of relations known to be finite and acyclic. To illustrate this, Baar [4] defines `ancestors` by $APar(x) = Par(x) \cup \{y \mid \exists z.\ z \in Par(x) \wedge y \in APar(z)\}$, where $Par(x)$ and $APar(x)$ are the translations of `x.parents` and `x.ancestors`, respectively. Correspondingly, in first-order logic, this definition can be expressed by the formula $r^*(x, y) \Leftrightarrow (r(x, y) \vee \exists z.\ r(x, z) \wedge r^*(z, y))$, where the relation symbols r and r^* are substituted for Par and $APar$, with $r(x, y)$ meaning $y \in Par(x)$ and $r^*(x, y)$ meaning $y \in APar(x)$. This formula is interpreted by the structure (U, R, R^*) where U is a universe of variables, and R and R^* are interpretations of the relations r and r^*, respectively. Countermodels for this formula are presented whereby R^* does not coincide with the transitive closure of R. However if the model (U, R, R^*) is finite and the axiom $\neg r^*(x, x)$ holds – enforcing R^*s acyclicity – then R^* is a correct definition of transitive closure (however, in general finiteness is not expressible).

JML and SPEC#. The Java Modeling Language (JML) was originally designed by Leavens *et al.* at Iowa State University in 1998. Having spawned a much larger community of users and tool developers who are now actively involved in its development, JML has since become the standard specification language used for verification of Java programs. JML is used to specify Java classes and interfaces [23, 24].

The Spec# system [5] has been developed as a specification language for .Net. The recent developments in the JML community have been influenced and some

ideas have been adopted that originated from the Spec# project. The treatment of second-order concepts is similar in both languages (we concentrate on JML in the following).

Specifications in JML are formulated by making use of (side-effect-free) boolean Java expressions; they are written as Java comments. The original JML tool is a pre-compiler designed to translate specified programs into Java programs that explicitly monitor assertions at run-time. Specification violations that are found throw Java exceptions. Since JML's conception, many more tools have been developed using JML as an input specification language. For a more extensive overview of JML tools and applications, see [8].

When specifying transitive closure, JML manages to avoid the whole issue of acyclicity by defining recursive datagroups [28]. These have been designed primarily with frame-condition issues in mind. To solve the information hiding problem (*i.e.* that protected or private fields of a class should remain hidden from their clients) the `represents` clause was introduced to JML, allowing one to specify the representation of concrete fields by particular abstract fields. Hence protected or private fields in an implementation can be changed without changing the specification visible to its clients. Unfortunately, the use of abstract fields generated problems with the `modifies` clause. (A method's `modifies` clause specifies those locations that are permitted to be changed by execution of the method.) This was fixed by a `depends` clause which relates those locations used to determine an abstract location's values. A datagroup can be modelled by an abstract location whose value contains no information. By using a `depends` clause, a location can be declared to be in a datagroup, therefore membership in a datagroup allows the locations in the datagroup to be modified whenever the datagroup is mentioned in the `modifies` clause. The license to modify a datagroup implies the license to modify the members of the datagroup as defined by a downward closure rule [25]. For any set of datagroups S, the downward closure of this set is the smallest superset of S such that for any group G in the closure of S, all nested datagroup members of G also belong in the closure of S. For example, consider the following Java linked list with `Node` objects having `next` and `value` fields:

```
class Node { Integer value; Node next; }
```

The datagroups `nodeValues` and `nodeLinks`are are defined recursively using clauses such as "`maps next.nodeValues \into nodeValues`". Hence the clause "`modifies list.nodeLinks;`", when it is added to the JML specification of a method `sortInPlace(Node list)`, says that all node objects reachable from `list` may be changed whenever `sortInPlace` is executed.

Such specifications rely on a smallest-fixed-point semantics for recursive definitions built into JML. Gleaned from mailing list discussions, Leavens *et al.* have considered introducing regular expressions, (*i.e.* writing `list.next`* in order to specify the `JMLObjectSet` of all objects reachable from `list` using the field name `next`) but have rejected this as not particularly beneficial since using datagroups seems to be an adequate enough solution.

5 Conclusions

Although important properties of relations are not expressible in classical first-order logic, it is possible to extend first-order logic (*e.g.* with fixed-point and transitive closure operators) in order to describe such properties. We find that all specification languages feature modifications which allow them to extend beyond the limitations of first-order logic. For example SQL implements fixed-point logic, OCL uses the `iterate` and `let` constructs, CASL implements the notion of freeness, whereas JML incorporates built-in recursion. However, the logical concepts underpinning these modifications are often not well documented. This paper has attempted to clarify what is going on regarding these extensions.

Generally we have found that once integers are "available" in a specification language, it is possible to define transitive closure and other properties of relations in the language. Otherwise this is possible only for finite relations (which is mostly adequate). In our opinion the best solution is that which is taken by CASL and JML, namely by building freeness or minimal fixed-points either explicitly or implicitly into the language. It still seems desirable to add regular expressions to specification languages. It is not clear yet how this should be done; this is the subject of future work.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. N. Alechina, S. Demri, and M. de Rijke. A modal perspective on path constraints. *Journal of Logic and Computation*, 13:1–18, 2003.
3. N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337, 2000.
4. T. Baar. The definition of transitive closure with OCL: Limitations and applications. In *Proceedings of the Fifth Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS 2890, pages 358–365. Springer, 2003.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, Revised Selected Papers*, LNCS 3362. Springer, 2005.
6. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, SEFM 2003*, pages 91–99. IEEE Press, 2003.
7. M. Bidoit and P. Mosses. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*. LNCS 2900. Springer, 2004.
8. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. In *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.
9. M. V. Cengarle and A. Knapp. A formal semantics for OCL 1.4. In *Proceedings, The Unified Modeling Language (UML 2001)*, LNCS 2185. Springer, 2001.
10. H. Chen, J. Hsiang, and H. Kong. On finite representation of infinite sequences of terms. In *Proceedings of 2nd International Workshop on Conditional and Typed Rewriting Systems*, number 516 in LNCS, pages 100–114. Springer, 1990.

11. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL, 1999. Available at `http://www.trireme.com/whitepapers/design/components/OCL_manifesto.PDF`.
12. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, 1997.
13. A. Dawar and Y. Gurevich. Fixed point logics. In *The Bulletin of Symbolic Logic*, volume 8, pages 65–88. Association for Symbolic Logic, 2002.
14. M. Fowler and K. Scott. *UML Distilled, 2nd ed.* Addison-Wesley, 2000.
15. D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, chapter 10, pages 497–604. Reidel, 1984.
16. N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.
17. D. Jackson. Automating first-order relational logic. In *Foundations of Software Engineering*, pages 130–139, 2000.
18. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings,ICSE 2000*, pages 730–733. IEEE, 2000.
19. Klasse Objecten. OCL center, 1999. At `http://www.klasse.nl/ocl`.
20. D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 14. The MIT Press, 1990.
21. S. Kreutzer. Expressive equivalence of least and inflationary fixed-point logic. In *Proceedings, Symposium on Logic in Computer Science (LICS)*. IEEE, 2000.
22. S. Kreutzer. *Pure and Applied Fixed-Point Logics*. PhD thesis, Aachen University of Technology, 2002.
23. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, Iowa State Univ., 2000. Available at `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz`.
24. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual. At `http://www.cs.iastate.edu/~leavens/JML/jmlrefman`.
25. K. R. M. Leino. Specifying the modification of extended state. Technical Report 1997-026, Digital Systems Research Center, 1997.
26. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *Proceedings, FM 1999*, LNCS 1708, pages 854–874. Springer, 1999.
27. P. D. Mosses. CASL: A guided tour of its design, 1999. Available at `http://www.brics.dk/Projects/CoFI/Documents/CASL/GuidedTour/index.html`.
28. P. Müller, A. Poetzsch-Heffter, and G. Leavens. Modular specification of frame properties in JML. Technical Report 02-02, Iowa State University, 1997.
29. Object Management Group. UML resource page, 1999. At `http://www.uml.org`.
30. J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings, OOPSLA 1987*, pages 466–481, 1987.
31. A. Schürr. Adding graph transformation concepts to UML's constraint language OCL. In *Proceedings, First Workshop on Language Descriptions, Tools and Applications (LDTA)*, ENTCS 44. Elsevier, 2001.
32. JCC's SQL std. page. At `http://www.jcc.com/SQLPages/jccs_sql.htm`.
33. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
34. J. Yang. SQL3 recursion. Lecture notes, Stanford University, 1999.
35. S. Zilles. Algebraic specification of data types. Technical Report XI, MIT Laboratory for Computer Science, 1974.