Specifying Object-Oriented Programs: Formalisms and Tools 2

In this chapter we give an overview of existing specification languages for object-oriented programs. The specification languages can be roughly partitioned into those which focus on formalizing protocol-oriented properties (all but the last three categories listed below), and those focussing on data. All specification languages for protocol properties are based on some form of histories (also known as traces): sequences of method calls or returns. Languages focussing on data restrict the values of variables and fields in a program by means of logical formulas. We describe whether the specification languages are used in actual tools for static verification or run-time checking.

Sequence Diagrams A sequence diagram[1] shows how multiple objects interact with each other over time. The diagram depicts the messages exchanged between the objects, and the order in which they are sent. In the context of object-oriented programs, the messages in a sequence diagram correspond to method calls. Since sequence diagrams visualize a single in-

[1] See `http://www.omg.org/spec/UML/` for the latest UML specification of sequence diagrams.

teraction, one could select a set of sequence diagrams as a specification of the behaviour of an object-oriented program, by requiring that the methods in the program are executed in the order specified by one of the sequence diagrams in the set. The resulting specification language describes properties of the protocol of the program.

While sequence diagrams have been used in theoretical studies for verification purposes [28, 62], to the best of our knowledge, sequence diagrams as a specification language have not been used in actual tools for static or run-time verification. There are several reasons for this. First, any specification based on visualization tends to become unclear and even infeasible for describing large interactions. Second, the number of interactions exhibited in programs are often unbounded due to loops and recursion. Thus one would need an additional language for characterizing infinite sets of sequence diagrams.

Regular Expressions A regular expression [56] is a declarative notation for a regular language. A language is a set of words. The words are usually (finite) strings of characters, though more complex objects can be used as well. The regular languages are those that can be obtained from a finite language by union, concatenation and Kleene star (an infinite union of finite concatenations of a language). If r_1 and r_2 are regular expressions, the notation for these three operations is respectively $r_1 + r_2$ (union), $r_1 r_2$ (concatenation) and r_1* (Kleene star). As an example, the regular expression $(ab)*$ denote the language of all words starting with “a” in which “a” and “b” alternate. The formal properties of regular languages have been widely studied in the field of formal languages and theory of computation, see for example the books [81, 63].

As a specification language for object-oriented programs, regular expressions can be used to denote valid histories [21]. In this setting, the alphabet symbols correspond to method names, histories are represented as sequences of such alphabet symbols, and the valid histories are the words of the regular language. Note that in contrast to the previous sequence diagrams, regular expressions support a convenient notation for an infinite set of histories with the Kleene star.

There are various tools for run-time checking which support regular expressions: JmSeq [70], Tracematches [3] and JavaMOP [20]. The run-time check corresponds to solving the word problem (or parsing problem): decide whether the history is a word of the language denoted by a given regular expression. This can be done efficiently. In particular, if a history is valid according to a given regular expression, then parsing algorithms exist that decide in constant time whether the history resulting from appending a single call is also valid according to the regular expression (for the full history, this leads to parsing algorithms which are linear in the size of the history), see [41]. Moreover one does not need to store the full history, only the "state" of the parser for the previous history, and the method call which is added to the previous history are needed to determine validity of the new history.

Context-Free Grammars A context-free grammar G is a quadruple $G = \langle V, \Sigma, P, S \rangle$ where V is a set of non-terminals, Σ is a set of terminal symbols, S is the start-symbol of the grammar (a non-terminal), and P is a set of production rules. The production rules specify how each non-terminal (independent of the context in which that non-terminal occurs, hence the name context-free) is allowed to be rewritten into a sequence of terminals and non-terminals. The grammar generates a context-free language, namely the set of all strings of terminal symbols that can be obtained by repeatedly applying the production rules of the grammar, starting from the start symbol of the grammar. For example, the grammar below (the used notation for the grammar is BNF [6]) with the non-terminal S as its start symbol, and "a" and "b" as terminal symbols generates all words of the form $\mathrm{a}^k\mathrm{b}^k$, $k \geq 0$ (in words: k a's, followed by k b's). The symbol ϵ denotes the empty word.

$$\boxed{\begin{array}{lcl} S & ::= & \mathrm{a}\ S\ \mathrm{b} \\ & \mid & \epsilon \end{array}}$$

Context-free grammars are strictly more expressive than regular expressions. Using the so-called pumping lemma [81], one can prove that there is no regular expression which denotes the same language as the grammar above. However it is more complex to parse a string in a given context-free grammar, than in a regular expression. The currently best known practical algorithms can parse a string of length n in (worst case) $\mathcal{O}(n^3)$ time.

When used as a specification language for object-oriented programs, the terminal symbols are the method names, and the grammar specifies the valid orderings in which these methods are allowed to be called (in other words, the context-free grammar generates the valid histories). The run-time check which decides whether a history is valid consists of parsing the current history in the given grammar. PQL [64] and JavaMOP [20] are examples of tools that support run-time checking based on context-free grammars.

Automata There are too many kinds of automata too list them here exhaustively, but all of them contain at least two things: a notion of a state, and a transition function between states. A finite automaton, one of the simplest automata, contains additionally a set of accepting states and a start state, with the requirement that the set of states must be finite. Finite automata are equivalent in expressive power to regular expressions. A push-down automaton is an extension of a finite automaton with a stack of infinite size. Push-down automata are equivalent in expressive power to context-free grammars.

In general, automata can be seen as a representation of a formal language: it takes a string as input, and accepts or rejects it based on an acceptance condition (the specific acceptance condition varies greatly between the different kinds of automata). However, unlike the above declarative formalisms of regular expressions and context-free grammars, automata tend to have an imperative flavor, focussing on *how* to parse a formal language, as opposed to directly specifying the language itself.

As a specification language for object-oriented programs, JavaMOP [20] supports finite automata. LARVA [26] supports a kind of automata called timed automata with stopwatches.

Temporal Logics Temporal logic [74] is a variant of *Modal Logic* [39]. As the name indicates, the basis for temporal logics is a notion of time on which the truth of a formula may depend. In particular, as the system described a temporal logic formula evolves from one state to the next, the truth value of the formula *can* change. There are many kinds of temporal logics, but they can roughly be classified as being linear-time or branching-time. In linear-time logics, time is viewed as a set of paths (the paths being sequences of "time instances"). LTL [74] is a widely used linear-time logic. Branching-time logics represent time as a tree in which the current time is the root, and the branches are considered as "possible futures". CTL [24] is the main branching-time logic.

Temporal logics have been used extensively in model checking [25], for example in the tools (there are too many others to fully list here): BLAST [46] Java Pathfinder [87] NuSMV [22] PRISM [60] SPIN [49] UPPAAL [11]. Temporal logics have also been used in run-time checking, even for the functional language Haskell [83]. Examples of run-time checkers of temporal logic formulas for Java are JavaMOP [20] and Java Pathfinder [5].

Process Algebras Process algebras [7, 45] have been used to formally model concurrent systems. There exist a wide variety of process algebras (or process calculi), but all approaches share some basic characteristics.

Each approach has a notion of a basic process from which larger processes are built using various operators (for example, for parallel composition, sequential composition and recursion). Message passing is used as the only way two different actors or processes can interact (instead of for example, shared variable concurrency). Finally, all approaches come with a set of algebraic laws (hence the name "process algebra") which for example can be used to show that syntactically different processes are semantically equal (i.e. have the same behavior).

For reference we list some of the most used process algebras here: CSP [48, 1], LOTOS [86], CCS [68], ACP [14] and the more recent π-calculus [69, 78]. CSP has been used in the tool Jass [8] for run-time checking object-oriented programs.

First-Order Logic First-order logic is a formal system for specifying and reasoning about formulas about objects (or values) that range over some domain of discourse. All variables and terms in a first-order formula range over objects of the domain of discourse.

First-order logic can be used to specify programs by means of assertions: a logical formula in which the free variables (i.e. all variables not bound by $\forall$ and $\exists$) are program variables. Assertions are written in the source code of the program and must be true whenever control passes over them. Floyd describes in [38] a method for proving properties using first-order assertions. His work was extended by Hoare in [47]. First-order logic also forms the basis for dynamic logic and second- and higher-order logic described below.

The popular tool-suite for JML [17] supports first-order assertions for both static verification and run-time checking of Java programs. The run-time checker for JML only checks formulas involving bounded quantifiers: quantified variables that range over a finite set of values. Validity of formulas involving unbounded quantifiers is in general undecidable, as already noted in the previous chapter.

Dynamic Logic Like temporal logic, Dynamic Logic (DL) [76, 42] is a variant of *modal logic* [39] which allows the direct expression of program equivalence and weakest preconditions. DL extends full first-order logic with two additional (mix-fix) relations: $< . > .$ (diamond) and $[.] .$ (box). In both cases, the first argument is a *statement*, whereas the second argument is another DL formula. A formula $< s > p$ is true if there exists a terminating execution of s after which the formula p is true. A formula $[s]p$ is true after all terminating executions of s, the formula p is true. For example, the formula <`x=x-1;`> $(x == 0)$ is equivalent to $x = 1$. Dynamic logic has been used as a specification language in the static verifiers KeY [10] and KIV [44].

Second- and Higher-Order Logic Second-Order logic is a highly expressive formalism which allows quantification over predicates and functions over the values of the underlying domain. This contrasts with first-order logic, in which only quantification over values of the domain is allowed. The expressiveness comes at a price: no sound and complete proof systems (with decidable proof rules and axioms) can exist for full second-order logic. Higher-Order logic is a generalization of second-order and first-order logic which allows quantification over objects of an arbitrary higher type (i.e. quantification over predicates of predicates, and so on). There exist various theorem provers for programs that support higher-order logic: Isabelle/HOL [57], Why3 [36], PVS [85] and Coq [15].

Another relatively recent approach is Separation Logic [77], which extensively uses inductively defined predicates (i.e. second-order logic), but adds several non-standard logical connectives to reason about heap properties, such as the separating conjunction and the points-to predicate. These connectives support modularity, though they complicate proof theory (they cannot be axiomatized [18]). Tools that support separation logic for static verification of programs include: VeriFAST [51], jStar [35], Slayer [13] and Smallfoot [12].