Trace Specifications for Control- and Data-Flow 3

The formalisms described in the previous chapter for specifying object-oriented programs can be categorized in roughly two categories: those focussing on the control-flow of the program, and those focussing on the data-flow of the program. Formalisms focussing on the control-flow specify the allowed orderings between method calls, for example using regular expressions, context-free grammars or temporal logics. Formalisms for describing the data-flow generally use assertions to restrict the values of fields, parameters or local variables, possibly enhanced by constructs such as pre-post conditions and class invariants for supporting design by contract. But none of described specification languages were developed to *combine* the specification of the control-flow with the data-flow in a single formalism. In contrast, the behavior of almost all Java programs depends on both control-flow and data-flow: for example, the behavior of a stack is fully characterized by the sequence of method calls to `push` and `pop` it receives (the control-flow), together with the parameter and return values (the data-

flow). For Java programs that encapsulate their internal state[1] an execution can be represented by the *global communication history* of the program: the sequence of *messages* corresponding to the invocation and completion of (possibly static) methods, including actual parameters and return values. Similarly, the execution of a single object can be represented by its *local communication history*, which consists of all messages sent and received by that object. The behavior of a program (or object) can then be defined as the set of its allowed histories. Jeffrey and Rathke [52] develop a fully abstract semantics based on histories which coincides with the standard operational semantics.

Let us call the orderings between method-calls and returns the *control-flow* of a history, and the actual parameters and return values the *data-flow* of the history. In this chapter we develop a single formalism which allows combining data-oriented properties of the history with protocol-oriented properties. To be of practical use, such a formalism should be *user-friendly*, amenable to (at least) *automated run-time verification* and sufficiently *expressive*. Below we propose attribute grammars extended with assertions and conditional productions for the specification of histories, and compare several alternatives approaches with respect to expressiveness, usability and automation.

Specifications can be used in two different ways: as a description of how an API (in our case, a set of Java classes and interfaces) must be used by a client (this can be seen as a kind of formalized user manual), or as an internal specification for developers of a class to test the class which is being developed. In the first case, only methods visible to clients can be used in the specification (i.e. public methods and no self-calls, since the user has no control over private methods and self-calls), in the second case for internal use we must also monitor self-calls and calls to private methods.

[1] Encapsulation means that objects do not have direct access to the fields of other objects. If access to a field x is needed, the programmer instead adds two methods `T getx()` and `void setX(T val)`.

3.1 Modeling Framework

The modeling framework consists of three basic ingredients: communication views, grammars with conditional productions, and assertions. We use the interface of the Java `BufferedReader` (Figure 3.1) as a running example to explain these modeling concepts. In particular, we formalize the following property of the `BufferedReader`:

> The `BufferedReader` may only be closed by the same object which created it, and read actions may only occur between the creation and closing of the `BufferedReader`.

Note that the above property constrains the clients that use the `BufferedReader`; in other words, it is a kind of "user manual" for the reader, but does not guarantee that the reader itself works properly (since this property does not restrict the behavior of the reader itself). The property is a little unusual in that the reader actually cannot even detect whether a client uses it according to the above specification, since the reader has no way to detect whether the caller of `close` is the same object that constructed it. This last part can be seen as a form of dynamically checked ownership: the client which created the reader owns it, and the above property can serve as a first step to ensure that no information about the reader is leaked to other clients.

As a naive first step one might be tempted to define the behavior of `BufferedReader` objects simply in terms of 'call-m($\overline{T}$)' and 'return-m($\overline{T}$)' messages of all methods 'm' in its interface, where the parameter types $\overline{T}$ are included to distinguish between overloaded methods (such as `read`). However, interfaces in Java contain only signatures of provided methods: methods where the `BufferedReader` is the callee. Calls to these methods correspond to messages received by the object. In general the behavior of objects also depends on messages sent by that object (i.e. where the object is the caller), and on the particular constructor (with parameter values) that created the object. Moreover it is often useful to select a particular subset of method calls or returns, instead of using calls and returns

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

Figure 3.1: Methods of the BufferedReader Interface

```
local view BReaderView specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

Figure 3.2: Communication view of a BufferedReader

to all methods (a partial or incomplete specification). Finally in referring to messages it is cumbersome to explicitly list the parameter types. A *communication view* addresses these issues.

Communication View

A communication view is a partial mapping which associates a name to each message. Partiality makes it possible to filter irrelevant events and message names are convenient in referring to messages.

Suppose we wish to formally specify the property on page 23. This is a property which must hold for the local history of all instances of `java.util.BufferedReader`. The communication view in Figure 3.2 selects the relevant messages and associates them with intuitive names: *open*, *read* and *close*.

All return messages and call messages methods not listed in the view are filtered. Note how the view identifies two different messages (calls to the overloaded `read` methods) by giving them the same name *read*. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g. methods of other interfaces or classes) are also supported. Since such messages are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example consider the following message:

```
call void C.m() out
```

If we would additionally include the above message in the communication view, all call-messages to the method `m` of class `C` sent by a `BufferedReader` would be selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Besides normal methods, communication views can contain signatures of constructors (i.e. the messages named *open* in our example view). As such, the set of signatures that occur in a communication view is not necessarily a subset of the signatures in the interface it specifies (since Java interfaces do not contain constructors). In this case, the view selects all calls/returns to an object of a class that implements that interface.

Incoming calls to provided constructors raise an interesting question: what would happen if we select such a message in a local history? At the time of the call, the object has not even been created yet, so it is unclear which `BufferedReader` object receives the message. We therefore only allow return-messages of provided constructors (clearly constructors of other objects do not pose the same problem, consequently we allow selecting both calls and returns to required constructors), and for convenience omit `return`. Alternatively one could treat constructors like static methods, disallowing incoming call-messages to constructors in local histories altogether. However this makes it

impossible to express certain properties (including the desired property of the `BufferedReader`) and has no advantages over the approach we take.

Java programs can distinguish methods of the same name only if their parameter types are different. Communication views are more fine-grained: methods can be distinguished also based on their return type or their access modifiers (such as `public`). For instance, consider a scenario with suggestively named classes `Base` and three subclasses `Sub1`, `Sub2` and `Sub3`, all of which provide a method `m`. The return type of `m` in the `Base`, `Sub1` and `Sub2` classes is the class itself (i.e. `Sub1` for `m` provided by `Sub1`). In the `Sub3` class the return type is `Sub1`. To monitor calls to `m` only with return type `Sub1`, simply include the following event in the view:

```
call Sub1 C.m() messagename
```

One may ask: why allow private methods to appear in specifications? After all, private methods cannot be used by an outside client of the class. The same question arises when considering whether to monitor self-calls or not. By allowing to monitor private methods and self-calls, the modeling framework and corresponding tool support can also be used by developers of the class, to test the current implementation of the class in development. Communication views include an optional `excludeSelfCalls` keyword which indicates per event whether self-calls must be tracked (for self-calls, the caller and the callee are the same). While typically developers do not want to exclude self-calls for the purpose of internal tests, this keyword is especially useful in public specifications for other clients, that describe how the class must be used by the client.

Local communication views, such as 3.2, selects messages sent and received by *a single object* of a particular class, indicated by 'specifies `java.util.BufferedReader`'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views. Figure 3.3 shows a global view which selects all returns of the method `m` of a class or interface

```
global view PingPong {
  return void Ping.m() ping,
  call void Pong.m() pong
}
```

Figure 3.3: Global communication view

Constructors
Inheritance
Dynamic Binding
Overloading
Static Methods
Required Methods
Access Modifiers

Table 3.1: Supported Java features that require special care.

(or any of its subclasses) called `Ping`, and all calls to `m` on a subtype of a class or interface called `Pong`. Note that communication views do not distinguish instances of the same class (e.g. calls to 'Ping' on two different objects of class 'Ping' both get mapped to the same terminal 'ping'). Different instances *can* be distinguished in the grammar using the built-in attributes 'caller' or 'callee', see the next two sections.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. See table 3.1 for a list of supported features which require special care. For example, to support dynamic binding, the actual run-time type of the callee must be used, instead of the static type of the variable or field in which the callee is stored. This means that the correspondence between the messages named in the communication view, and actual method calls in the program source code must be made at runtime. The other features listed in the table have been discussed above.

Context-Free Grammars

Now that we have identified the basic messages using the communication view, the question arises how we can specify the valid orderings between these messages: *the protocol*. More specifically, we want to find a notation for the set of the valid histories (where a history is a finite sequence of messages). While the histories in this set will be finite (since at any point during execution, the then current history is finite), the set itself usually contains an infinite number of histories due to recursion or loops, so we cannot simply write it down explicitly. We can consider the set to be a language in which each history is a word, and each message is an alphabet symbol. This suggests we can use existing formalisms for defining languages, in particular the ones surveyed in Chapter 2. We use context-free grammars to specify the protocol behavior of histories.

Definition A history is valid with respect to a given context-free grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar.

The discussion in Section 3.2 provides a motivation for choosing grammars over the other formalisms, and a justification for our definition of a valid history.

The grammar below specifies the valid histories of the `BufferedReader`:

$$
\begin{array}{lcl}
S & ::= & \mathit{open}\ C \\
 & | & \epsilon \\
C & ::= & \mathit{read}\ C \\
 & | & \mathit{close}\ S \\
 & | & \epsilon
\end{array}
$$

Figure 3.4: Context-Free Grammar which specifies that 'read' may only be called in between 'open' and 'close'.

This grammar describes the prefix closure of sequences of the terminals 'open', 'read' and 'close' as given by the regular expression $((\mathit{open}\ \mathit{read}*\ \mathit{close})*)$. In general, the message names given by a communication view form the terminal symbols of the grammar, whereas the non-terminal symbols specify

the structure of valid sequences of messages (in particular, the start symbol S generates the valid histories).

Attribute Grammars and Assertions

While context-free grammars provide a convenient way to specify the *protocol structure* of the valid histories, they do not take data such as parameters and return values of method calls and returns into account. Thus the question arises how to specify the *data-flow* of the valid histories. To that end, we first extend the above context-free grammars with so-called attributes.

Definition Terminal Attributes. Given a terminal T, an attribute of T assigns a value to each instance[2] of T (i.e. to each token of T).

For example, consider a terminal INT_LITERAL, and suppose the string "33" is an instance of INT_LITERAL. One could define an attribute *val* for INT_LITERAL, which assigns the number 33 to the string "33". Note that terminal attributes can assign different values to different instances of the same terminal.

In the previous section we saw that (instances of) terminals correspond to call or return messages. The question arises: what are sensible attributes for such terminals? Several objects are involved in the sending of the messages: the *caller*, the *callee*, and the actual data being sent in the form of actual parameters or a return value *result*. We define *built-in* attributes (named *callee*, *caller*, and so on) to capture precisely those objects involved in the message. In summary, attributes of terminals are determined (i.e., built-in) from the method signatures given in the communication view.

Next we define attributes for non-terminals. Unlike attributes for terminals, they are defined by the user in the grammar. Given a context-free grammar G and a non-terminal V, let us denote by $L(V)$ the language generated from the non-terminal V by using the productions of G.

[2] A token is a string of symbols. A terminal can be seen as a token type, whose tokens are considered to be syntactically "similar"

Definition Non-terminal Attributes. Given a set of values D and a context-free grammar with a non-terminal V, an attribute for V is a function $f : L(V) \to D$.

Intuitively the above definition states that a non-terminal attribute assigns values to all of the words generated by that non-terminal. The value of non-terminal attributes is user-defined: the user must associate with each production, source code that computes the attribute values of all non-terminals involved in the production. There are two kinds of non-terminal attributes: synthesized attributes and inherited attributes. In each production the user defines the value of the synthesized attributes of the non-terminal on the left-hand side of the production, and the values of the inherited attributes of the non-terminals appearing on the right-hand side of the production. In general this does not rule out circular attribute definitions The seminal paper [59] in which Knuth first introduced attribute grammars contains an algorithm which detects circular definitions. Using actual source code for the attribute definitions ensures that all attribute values of non-terminals are computable. Of course this source code may not terminate, we rely on the user to make sure that it does.

In our setting, the grammar non-terminals generate sequences of call/return messages. Hence, a non-terminal attribute can be seen as a property of the data-flow of that sequence and hence, as an important special case, the attributes of the start symbol of the grammar can be considered as properties of the data-flow of the history. We are now ready to define attribute grammars:

Definition An attribute grammar is a pair (G, F), where G is a context-free grammar, and F is a set of attributes for G.

Note that the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the history. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Figure 3.5 a user-defined synthesized attribute 'c' for the non-terminal 'C' is defined to store the identity of the object which closed the

S	::=	*open* C_1	{`assert (open.caller == null ||` `open.caller ==`C_1`.c ||` C_1`.c == null);`}
	|	ϵ	
C	::=	*read* C_1	(C`.c =`C_1`.c;`)
	|	*close* S	(C`.c = close.caller;`)
	|	ϵ	(C`.c = null;`)

Figure 3.5: Attribute Grammar which specifies that 'read' may only be called in between 'open' and 'close', and the reader may only be closed by the object which opened it.

`BufferedReader` (and is `null` if the reader was not closed yet). Synthesized attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the 'c' attribute is not set in the productions of the start symbol 'S'. The extension of context-free grammars to attribute grammars with assertions and conditional productions (next called "extended attribute grammars") naturally gives rise to the following modification in the definition of a valid history.

Definition A history is valid with respect to a given extended attribute-grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar, and all assertions in the grammar were true for every prefix of the history.

The assertion in the attribute grammar of the `BufferedReader` allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the paper the start symbol in any grammar is named 'S'. For clarity, attribute definitions are written between parentheses '(' and ')' whereas assertions over these attributes are surrounded by braces '{' and '}'.

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a gener-

alization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal 'call-m' can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears. Traditional pre- and post-conditions are still useful if in every context, the same assertion must be used: in that case, the assertions in the grammar would be duplicated at every occurence of the appropriate terminal. In Section 5.1 we show an example which uses traditional pre- and post-conditions.

It is important to note that for a meaningful semantics we have to restrict the attribute grammars to those grammars which are side-effect free (with respect to the heap) so that they don't affect the flow of control of the tested program, and which do not involve dereferencing of the built-in attributes of the grammar terminal (the formal parameters of the corresponding methods as specified by the communication view) because these refer to the *current* heap (and not to the past one corresponding to the occurrence of the message). This latter restriction is a fairly natural requirement as the method call which generated the grammar terminal only passed the the object identities of the actual parameters, but not the values of the fields of these objects. Note also that this requirement is automatically satisfied by using encapsulation.

Attribute grammars in combination with assertions cannot express *protocol that depend on data*. To express such protocols we consider attribute grammars enriched by *conditional productions* [72]. In such grammars, a production is chosen only when the given condition (a `boolean` expression over the inherited attributes) for that production is true. Hence conditions are evaluated before any of the symbols in the production are parsed, before synthesized attributes of the non-terminals appearing in the production are set and before assertions are evaluated. In contrast to assertions, conditions in productions affect the parsing process. The Worker grammar in Figure 5.17 in the case study contains a conditional production for the 'T' non-terminal.

In summary, a communication view selects and names the relevant messages. Selection allows to focus just on the relevant messages while names allow the identification of different messages, and enable the user to refer to the messages in a user-friendly manner. Context-free grammars specify the allowed orderings of the messages. The terminals of the grammars are the names as introduced by the communication view. These names are not just simple strings, but also contain various attributes such as the sender, receiver and the data sent in the message. The non-terminals are user-defined and generate sets of sequences of messages (i.e. histories), as given by the grammar productions. The start symbol of the grammar generates the valid histories. A context-free grammar can thus be seen as specifying a kind of invariant of the control-flow. Attribute grammars allow defining data properties of sequences of terminals, and in particular of the whole history. To this end, the user defines attributes of the grammar non-terminals in terms of the attributes of the grammar terminals. The values of non-terminal attributes are defined by Java code, which ensures that the attribute definitions are computable. The extension of attribute grammars with assertions makes it possible to specify data-oriented properties of the history, by constraining the value of the non-terminal attributes.

Finally, conditional productions can be used for protocols that *depend* on data. In general, it is possible to specify a single interface or class with multiple communication views (and corresponding grammars). This increases expressiveness: it makes it possible to specify the intersection of two context-free languages (if the user specifies two grammars, the history must satisfy both), and context-free languages are not closed under intersection. Furthermore multiple communication views and grammars can be used as partial specifications for the class or interface, to focussing on a particular behavioral aspect. If it is possible to decompose a single complete specification into multiple partial specifications, the resulting specifications are often simpler. This stems from the fact that a complete specification formalizes various properties, and care must be taken to avoid unwanted interference between these properties. In contrast, partial specifications can be used to formalize each property individually.

3.2 Discussion

We now briefly motivate our choice of attribute grammars extended by assertions as specifications and discuss its advantages over alternative formalisms.

Instead of context-free grammars, we could have selected push-down automata to specify protocol properties (formally these have the same expressive power). Unfortunately push-down automata cannot handle attributes. An extension of push-down automata with attributes results in a kind of Turing machine. From a user perspective, the declarative nature and higher abstraction level of grammars (compared to the imperative and low-level nature of automata) makes them much more suitable than automata as a *specification* language. In fact, a push-down automaton which recognizes the same language as a given grammar is an *implementation* of a parser for that grammar.

Both the BufferedReader above and the case study use only regular grammars. Since regular grammars simplify parsing compared to context-free grammars, the question arises if we can reasonably restrict to regular grammars. Unfortunately this rules out many real-life use cases. For instance, the following grammar in EBNF[3] specifies the valid protocol behavior of a stack:

$$S \quad ::= \quad (push\ S\ pop\ ?)^*$$

It is well-known that the language generated by the above grammar is not regular (apply the pumping lemma for regular languages [81]), so regular grammars (without attributes) cannot be used to enforce the safe use of a stack. It is possible to specify the stack using an attribute which counts the number of pushes and pops:

[3] EBNF is an extension of the usual BNF notation for context-free grammars which allows using the operators on regular expressions (such as the Kleene star '*' and the '?' operator standing for an optional occurrence, i.e., 'r?' stands for 'r + ϵ') directly inside grammars.

S	::=	S_1 *push*	(S.cnt = S_1.cnt+1;)
	|	S_1 *pop*	(S.cnt = S_1.cnt-1;)
			{assert S.cnt >=0;}
	|	ϵ	(S.cnt = 0;)

The resulting grammar is clearly less elegant and less readable: essentially it encodes (instead of directly expresses, as in the grammar above) a protocol-oriented property as a data-oriented one. The same problem arises when using regular grammars to specify programs with recursive methods. Thus, although theoretically possible, we do not restrict to regular grammars for practical purposes.

Ultimately the goal of run-time checking safety properties is to prevent unsafe ongoing behavior. To do so, errors must be detected as soon as they occur; this is known as *fail-fast*, and the monitor must *immediately* terminate the system: it cannot wait until the program ends to detect errors. In other words, the monitor must decide *after every event* whether the current history is still valid. The simplest notion of a valid history (one which should not generate any error) is that of a word generated by the grammar. One way of fulfilling the above requirement, assuming this notion of validity, is to restrict to prefix-closed grammars. Unfortunately it's not possible to decide whether a context-free grammar is prefix-closed. The following lemmas formalize this result:

Lemma 3.2.1 *Let L_M be the set of all accepting computation histories*[4] *of a Turing Machine M. Then the complement $\overline{L_M}$ is a context-free language.*

Proof See [81].

Lemma 3.2.2 *It is undecidable whether a context-free language is prefix-closed.*

[4] A computation history of a Turing Machine is a sequence $C_0\#C_1\#C_2\#\ldots$ of configurations C_i. Each configuration is a triple consisting of the current tape contents, state and position of the read/write head. Due to a technicality, the configurations with an odd index must actually be encoded in reverse.

Proof We show how the halting problem for M (which is undecidable) can be reduced to deciding prefix-closure of $\overline{L_M}$. To that end, we distinguish two cases:

1. M does not halt. Then L_M is empty so $\overline{L_M}$ is universal and hence prefix-closed.

2. M halts. Then there is an accepting history $h \in L_M$ (and $h \notin \overline{L_M}$). Extend h with an illegal move (one not permitted by M) to the configuration C, resulting in the history $h\#C$. Clearly $h\#C$ is not a valid accepting history, so $h\#C \in \overline{L_M}$. But since $h \notin \overline{L_M}$, $\overline{L_M}$ is not prefix-closed.

Summarizing, M halts if and only if $\overline{L_M}$ is not prefix-closed. Thus if we could decide prefix-closure of the context-free language (lemma 3.2.1) $\overline{L_M}$, we could decide whether M halts.

Since prefix-closure is not a decidable property of grammars (not even if they don't contain attributes) we propose the following alternative definition for the valid histories. A communication history is valid if and only if all its prefixes are generated by the grammar. Note that this new definition naturally fulfills the above requirement of detecing errors after every event. And furthermore this notion of validity is decidable assuming the assertions used in the grammar are decidable. As an example of this new notion of validity, consider the following modification of the above grammar:

T	::=	S	{`assert S.cnt >=0;`}
S	::=	S_1 *push*	(`S.cnt = S_1.cnt+1;`)
	|	S_1 *pop*	(`S.cnt = S_1.cnt-1;`)
	|	ϵ	(`S.cnt = 0;`)

Note that the history *push pop* is a word generated by this grammar, but not its prefix *pop*, which as such will generate an error (as required). Note that thus in general invalid histories are guaranteed to generate errors. On the other hand, if a history generates an error all its extensions are therefore also invalid.

Observe that our approach monitors only safety properties ('prevent bad behavior'), not liveness ('something good eventually happens'). This restriction is not specific to our approach: liveness properties in general cannot be rejected on any finite prefix of an execution, and monitoring only checks finite prefixes for violations of the specification. Most liveness properties fall in the class of the non-monitorable properties [75, 9]. However it *is* possible to ensure liveness properties for terminating programs: they can then be reformulated as safety properties. For instance, suppose we want to guarantee that a method `void m()` is called before the program ends. Introduce the following global view

```
global view livenessM {
  call void C.m() m,
  return static void C.main(String[]) main
}
```

The occurence of the 'main' event (i.e. a return of the main method of the program) signifies the program is about to terminate. Define the EBNF grammar

$$\begin{array}{lll} S & ::= & \epsilon \\ & | & m \\ & | & m+\ main \end{array}$$

(where '+' stands for one or more repetitions). This grammar achieves the desired effect since the only terminating executions allowed are those containing `m`. In local views a similar effect is obtained by including the method `finalize` (which is called once the object will be detroyed) instead of `main`.