Specifying and Checking Protocols of Multithreaded Classes

Clément Hurlin

Specifying and Checking Protocols of Multithreaded Classes

Clément Hurlin
INRIA Sophia Antipolis - Méditerranée
2004 route des lucioles - BP 93
06902 Sophia Antipolis Cedex, France
0031 3489 4661
clement.hurlin@inria.fr

ABSTRACT

In the Design By Contract (DBC) approach, programmers specify methods with *pre and postconditions* (also called *contracts*). Earlier work added *protocols* to the DBC approach to describe allowed method call sequences for classes. We extend this work to deal with a variant of generic classes and multithreaded classes. We present the semantical foundations of our extension. We describe a new technique to check that method contracts are correct w.r.t. to protocols. We show how to generate programs that must be proven to show that method contracts are correct w.r.t. to protocols. Because little support currently exists to help writing method contracts, our technique helps programmers to check their contracts early in the development process.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Protocols, Multithreading, Object-Orientation, Design By Contract.

1. Introduction

Over the last years, major work has been done towards software verification. Among the variety of methods to verify software, a method of major importance is *Design By Contract* (DBC) [1]. In the DBC approach, programmers specify methods with *pre and postconditions* (also called *contracts*). A precondition specifies what the client must provide at method entry, while a postcondition specifies what is ensured to the client at method exit. Tools for DBC include the Eiffel programming language [2], the Java Modeling Language (JML) [3] for Java, and the Spec# project for C# [4].

SAC'09 March 8-12, 2009, Honolulu, Hawaii,U.S.A.

The aforementioned techniques for DBC both provide tool support to *(1)* dynamically check contracts (with Eiffel's built-in facilities or with JML's *runtime assertion checker* [5]) and *(2)* statically check contracts (with ESC/Java2 [6] for JML or with Boogie [7] for Spec#).

Typically, the DBC approach recommends to write method contracts before implementing the methods. The tools mentioned above, however, are useful to check implementation of methods against their contracts: In the typical development process outlined above, the feedback from static checking tools comes very late in the development process. In this paper, we propose a technique to use static checking earlier in the development process. To do this, we extend earlier work on specifying method call sequences in JML [8].

A method call sequence is a regular expression indicating what methods can be called on an object and in which order methods must be called. As explained earlier [8], many classes must be used in a specific way by clients: for these classes it is important to be able to concisely express allowed method call sequences. For example, clients of Java's interface `StreamBuffer` must call method `read()` zero or more time and then call method `close()` *once* [9]. This can be concisely specified with the following method call sequence:

```
read()*, close()                    (1)
```

In this paper, we extend [8] to deal with a variant of generic classes and multithreaded classes (i.e., classes such that multiple methods can safely execute in parallel on instances of these classes). We extend the specification language and the semantical foundations of the work mentioned above. We maintain [8]'s spirit by providing a concise and intuitive regular expression-like notation to write protocols of multithreaded programs.

In addition, we provide a technique to check if method contracts are correct w.r.t. to protocols. For example, given the protocol (1) above, a programmer has to make sure that *1)* `read()`'s postcondition implies `reads`'s precondition (because `read` can be called multiple times successively) and *(2)* `read()`'s postcondition implies `close()`'s precondition (because `close` is called after `read`). If one of the conditions above does not hold, some programs, even if they obey `StreamBuffer`'s protocols, will fail to verify. We formalize this intuition by saying that *method contracts should adhere to protocols*. We describe how to generate programs that must be proven to show that method contracts adhere to protocols.

This technique can be applied early in the development

process because it only require methods to be specified with contracts (not necessarily to be implemented), providing feedback to programmers early in the development process.

This paper is organized as follows: Section 2 informally introduces our model language, Section 3 formally defines our language for specifying protocols, Section 4 presents an example of writing protocols in our language, Section 5 defines the semantics of multithreaded programs and the semantics of protocols, Section 6 shows how we generate programs that must be proven to show that method contracts adhere to protocols, Section 7 evaluates our approach and discusses its limitations, Section 8 describes related work and Section 9 concludes.

2. The Model Language

Model language similar to ours can be found in other works [10, 11].Our model language is a Java-like language where programmers specify methods with pre and postconditions (also called contracts) à la JML [3]. A precondition specifies what the client must provide at method entry, while a postcondition specifies what is ensured to the client at method exit. Contracts are specified using permission-accounting separation logic [12, 13].

Our model language features class parametrized by specification values. Specifications values include client-defined objects, permissions and built-in Java values (objects, integers, booleans, etc.). This allows classes to express different behaviors without code duplication.

3. The Protocol Language

Protocols are specified with the following grammar:

$n \in \mathbb{N}$		integers
$q \in \mathbb{Q}$		rationals
$v \in$ ProtVar		protocol variables
$\alpha \in$ LogVar		logical variables (generic parameters)
$op \in$ Operator	::=	{ `==` , `!` , `<` , `<=` , `+` , . . . }
$e \in$ Expr	::=	`true` | `false` | n | q
		e op e | π | v | α
$m \in$ Method	$T \in$ Type	
$s \in$ Spec	::=	$m(\bar{T})$ | v `=` $m(\bar{T})$ |
		s, s | s `|` s |
		s`?` | s`*` | s`+` |
		e `?` s : s | s `||` s | `!<`n`>` s

Our specification language consists in specifications for methods, sequential composition of specifications, composition of specifications with regular expression-like notations, conditional specifications, and parallel composition of specifications.

Now, we describe the meaning of specifications, $m(\bar{T})$ denotes m's execution, v `=` $m(\bar{T})$ denotes m's execution where m's return value is stored in v. Types $\bar{T}$ are used to choose an implementation of m in the case where m is overloaded. s, s' denotes the sequential composition of s and s', s `|` s' denotes s or s', e `?` s : s' denotes s if e is true (s' otherwise), s`?` denotes s zero or one time, s`*` denotes s zero or many times, s`+` denotes s one or many times, s `||` s' denotes s in parallel with s' (heterogeneous parallelism), and `!<`n`>` s denotes one to n s in parallel (homogeneous parallelism).

Protocols can depend on class parameters in the case e `?` s : s. This permits to adapt protocols to the different behaviors of a class. For simplicity, we do not include a conditional without "else" branch even if we use it in later examples. We could include it without any complication.

Compared to [8], we do not allow to specify nested call sequences. This forbids to specify that, for example, given two methods m and n, m should call n. We do not consider this as a limitation since our goal is different from the cited work. We focus on public protocols (protocols that are used by clients), because checking adherence of method contracts to protocols (see Section 6) is useful only for public protocols, whereas nested call sequences are useful only for private protocols (protocols that are used by class implementers). Contrary to [8], we introduce protocol variables (case v `=` $m(\bar{T})$), we allow to specify optional protocols (case s`?`), to specify conditionals (case e `?` s : s), and to specify parallelism (cases s `||` s, and `!<`n`>` s).

4. Examples

4.1 `Iterator` Example:

To illustrate our approach of specifying protocols, we use Java's `Iterator` interface:

```
interface Iterator {
 boolean hasNext();
 Object next();
 void remove();
}
```

Clients of interface `Iterator` must follow a precise protocol. In Java's documentation this protocol's description is spread among the documentation of the three methods of interface `Iterator` [9].

Intuitively, Java's documentation specifies that `hasNext()` must be called first, then `next()` should be called if `hasNext()` returned true before, and then `remove()` may be called if the iterator is read-write (it can read and write to the iteratee), and so on. To express that iterators may have write-access to the iteratee, we parametrize interface `Iterator` by a permission `p` [11]. If `p` is instantiated by `1`, one obtains a read-write iterator, otherwise a read-only iterator. Now, one can can precisely and concisely express `Iterator`'s protocol as follows:

```
(v=hasNext(), v?(next(), p==1?(remove()?)))*;
```

We believe that such a formal specification (compared to Java's informal documentation) would help clients to use `Iterator`s in a disciplined way.

4.2 `Roster` Example

Our second example is a `Roster` interface that collects student identifiers and associates them with grades which we borrow from [11].

```
interface Roster {
 void updateGrade(int id, int grade);
 boolean contains(int id);
}
```

For performance issues, we want implementers of the `Roster` interface to allow *(1)* multiple threads to concurrently read a roster and *(2)* a thread to update the grades while another threads concurrently read the student identifiers. This can be specified with the following protocol (where the unparametrized `!` is desugared to `!<`2^{32}`>`):

```
(updateGrade(int,int) || (!contains(int)))*;
```

5. Semantics

5.1 Semantics of Multithreaded Programs

The semantics of multithreaded programs is defined in terms of *actions*, *ghost stores*, and *traces*. An action is either method entering or method exiting. Ghost stores keep track of generic parameters (encoded as final ghost fields) and protocol variables (encoded as normal ghost fields). Generic parameters are final: they are assigned only once (at object creation) while protocol variables are assigned each time the corresponding method is called. For example, in the `Iterator`'s protocol, the protocol variable `v` (represented by the ghost field `v`) is assigned each time `hasNext()` is called. Ghost stores map logical variables (generic parameters) and protocol variables to specification values (permissions, client-defined values and Java built-in values). Finally, a trace is a sequence of ghost stores and actions.

$$\begin{array}{rcl} a \in \mathsf{Action} & ::= & m.\texttt{enter}(\bar{T}) \mid m.\texttt{exit}(\bar{T}) \\ \sigma \in \mathsf{GhostStore} & ::= & (\mathsf{LogVar} \cup \mathsf{ProtVar}) \rightharpoonup \mathsf{SpecVal} \\ \tau \in \mathsf{Trace} & ::= & \overline{\mathsf{GhostStore} \times \mathsf{Action}} \end{array}$$

We write $a\,\tau$ for the concatenation of action a and trace τ, $\tau\,\tau'$ for the concatenation of trace τ and trace τ', and ϵ for the empty sequence or trace. Note that, given the execution of a multithreaded program, we do not distinguish between actions from different threads. All actions of all threads form a single trace. This suffices to express the semantics of protocols.

We omit the operational semantics of our model language. It is completely standard except that *1)* we extend object stores with ghost stores *(2)* ghost stores are updated when methods that assign protocol variables are called. Because the operational semantics does not depend on ghost stores (ghost stores are written but never read), there is a trivial erasure to Java's operational semantics.

5.2 Semantics of Protocols

The semantics of protocols is given by $[\![]\!] : \mathsf{Spec} \to \overline{\mathsf{GhostStore}} \to 2^{\overline{\mathsf{Action}}}$. Intuitively, $[\![s]\!](\overline{\sigma})$ returns the set of all possible traces that satisfy s w.r.t. $\overline{\sigma}$. For the cases different from $e\ \texttt{?}\ s : s$, $s\ \texttt{||}\ s$, and $\texttt{!<}n\texttt{>}\ s$, $[\![]\!]$'s definition is standard:

$$\begin{array}{rcl} [\![m(\bar{T})]\!](\sigma\,\sigma') & \triangleq & \{m.\texttt{enter}(\bar{T})\ m.\texttt{exit}(\bar{T})\} \\ [\![v = m(\bar{T})]\!](\sigma\,\sigma') & \triangleq & \{m.\texttt{enter}(\bar{T})\ m.\texttt{exit}(\bar{T})\} \\ [\![s,\ s']\!](\overline{\sigma}) & \triangleq & \{\overline{a}\,\overline{a}' \mid \begin{array}{l} \overline{a} \in [\![s]\!](\overline{\sigma_0}) \\ \overline{a}' \in [\![s']\!](\overline{\sigma_1}) \\ \overline{\sigma_0}\,\overline{\sigma_1} = \overline{\sigma} \end{array}\} \\ [\![s \mid s']\!](\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \cup [\![s']\!](\overline{\sigma}) \\ [\![s\texttt{?}]\!](\overline{\sigma}) & \triangleq & \{\epsilon\} \cup [\![s]\!](\overline{\sigma}) \\ [\![s\texttt{*}]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \mathbb{N}} [\![s]\!]^i(\overline{\sigma}) \\ [\![s\texttt{+}]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \mathbb{N}^+} [\![s]\!]^i(\overline{\sigma}) \end{array}$$

and $[\![s]\!]^n(\overline{\sigma})$ is defined as follows:

$$\begin{array}{rcl} [\![s]\!]^0(\epsilon) & \triangleq & \{\epsilon\} \\ [\![s]\!]^i(\overline{\sigma}) & \triangleq & \{\overline{a}\,\overline{a}' \mid \begin{array}{l} \overline{a} \in [\![s]\!](\overline{\sigma_0}) \\ \overline{a}' \in [\![s]\!]^{i-1}(\overline{\sigma_1}) \\ \overline{\sigma_0}\,\overline{\sigma_1} = \overline{\sigma} \end{array}\} \end{array}$$

Below, we write $(\!|\,|\!) : \mathsf{Expr} \to \mathsf{State} \to \{\bot, \top\}$ to denote the (standard and omitted) semantics of expressions and we write $\mathsf{fst}(\overline{\sigma})$ to denote the first ghost store of a sequence of ghost stores:

$$[\![e\ \texttt{?}\ s : s']\!](\overline{\sigma}) \triangleq \begin{cases} [\![s]\!](\overline{\sigma}) & \text{iff} \quad (\!|e|\!)(\mathsf{fst}(\overline{\sigma})) = \top \\ [\![s']\!](\overline{\sigma}) & \text{otherwise} \end{cases}$$

To define the cases $s\ \texttt{||}\ s$ and $\texttt{!<}n\texttt{>}\ s$ of the semantics of specifications, we define the interleaving of two sequences of actions with $\between : \overline{\mathsf{Action}} \times \overline{\mathsf{Action}} \to 2^{\overline{\mathsf{Action}}}$:

$$\begin{array}{rcl} \epsilon \between a & \triangleq & \{a\} \\ a \between \epsilon & \triangleq & \{a\} \\ a\,\overline{a} \between a'\,\overline{a}' & \triangleq & \begin{array}{c} \{a\,\overline{a}'' \mid \overline{a}'' \in \overline{a} \between a'\,\overline{a}'\} \\ \cup \\ \{a'\,\overline{a}'' \mid \overline{a}'' \in a\,\overline{a} \between \overline{a}'\} \end{array} \end{array}$$

The $\between$ operator is extended to sets of sequences ($\between : 2^{\overline{\mathsf{Action}}} \times 2^{\overline{\mathsf{Action}}} \to 2^{\overline{\mathsf{Action}}}$) in the straightforward way. Then, we can define:

$$\begin{array}{rcl} [\![s\ \texttt{||}\ s']\!](\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \between [\![s']\!](\overline{\sigma}) \\ [\![\texttt{!<}n\texttt{>}\ s]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \{1,2,\ldots,n\}} \texttt{||}^i\ s(\overline{\sigma}) \end{array}$$

where $\texttt{||}^i : \mathsf{Spec} \to 2^{\overline{\mathsf{Action}}}$ is defined as follows:

$$\begin{array}{rcl} \texttt{||}^1\ s(\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \\ \texttt{||}^i\ s(\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \between (\texttt{||}^{i-1}\ s(\overline{\sigma})) \quad (\text{for } i \geq 2) \end{array}$$

5.3 Satisfaction of Traces w.r.t. Protocols

A trace satisfies a protocol if its underlying sequence of actions is the prefix of one of the sequences denoted by the protocol. We use the *prefix* of one of the sequences denoted by the protocol because we consider that not terminating a protocol is harmless. Formally:

$$(\sigma_0, a_0) \ldots (\sigma_n, a_n) \vdash s \quad \text{iff} \quad a_0 \ldots a_n \models [\![s]\!](\sigma_0 \ldots \sigma_n)$$

$$\begin{array}{c} a_0 \ldots a_n \models [\![s]\!](\sigma_0 \ldots \sigma_n) \\ \text{iff} \\ (\exists\, \overline{a}\ \overline{\sigma})(a_0 \ldots a_n\,\overline{a} \in [\![s]\!](\sigma_0 \ldots \sigma_n\,\overline{\sigma})) \end{array}$$

6. Adherence of Method Contracts to Protocols

To help programmers check that method contracts adhere to protocols, we generate programs. Then, generated programs must be proven to show that method contracts adhere to protocols. If generated programs cannot be proven, method contracts are incorrect w.r.t. to protocols: some (client-provided) programs will fail to verify.

When generating programs, we cannot (easily) provide method parameters fulfilling the part of method preconditions relevant to method parameters. Therefore, our technique is restricted to methods whose precondition has the form $F\ op\ G$ (where $op = \{\texttt{*}, \texttt{\&}\}$) and no method parameter (except `this` and `result`) occurs in F. Later, we refer to this restriction by saying that method preconditions must be *receiver splittable*. Because of this restriction, we can syntactically split method contracts into a part that concerns the receiver `this` (F) and a part that concerns the parameters (G). Then, when checking adherence of protocols, the part of the preconditions relevant to parameters (G) is dropped. We believe that, in practice, most method preconditions are *receiver splittable*. Our belief is supported by the fact that

```
requires i.init * Perm(i.v,1);
ensures  true;
void checkAdherence(Iterator<p> i){
 ...  // initialization (not shown in the interface)
 boolean b = havoc(boolean);
 while(b){
  i.v = hasNext();
  boolean b0 = havoc(boolean);
  if(b0){
   assume(i.v); Object o = i.next();
   boolean b1 = havoc(boolean);
    if(b1){ assume(i.p==1); i.remove(); }
  }}
}
```

Figure 2: Checking `Iterator`'s Protocol

all examples from [10] and [14](i.e., 14 classes) are *receiver spittable* [15] (it should be noted though, that our study is limited since the literature on object oriented separation logic is scarce).

Figure 1 shows the rules for generating programs. Function $\mathsf{gen}(r, s, T)$ generates the program to be proven to show that method contracts of class T adhere to protocol s given that the object considered is r. Intuitively, $\mathsf{gen}(r, s, T)$ generates the automaton corresponding to protocol s. In the cases $m(\bar{T}')$ and $v = m(\bar{T}')$, "unknown parameters" (parameters about which nothing is known) are passed to m. Unknown objects are axiomatized by using `havoc()` methods whose preconditions and postconditions are simply `true`. In practice, we use one `havoc()` method per type. Because we restrict to methods whose preconditions are *receiver splittable*, it is correct to use unknown parameters (recall that the parameter-relevant part of preconditions is discarded when checking adherence). We use `havoc(boolean)` statements to model non-deterministic choice. À la JML, we use `assume` statements to give hints to the verification system when generating the program corresponding to a conditional protocol (case e `?` $s : s$). We cannot use Java's `if` statement directly because expressions occurring in conditional protocols are not valid Java expressions (they can refer to ghost fields).

Because protocols can model multithreaded programs, $\mathsf{gen}(r, s, T)$ can also generate custom classes extending `Thread` (see cases s `||` s' and `!<`n`>` s and the function $\mathsf{classgen}(C, s, T)$). Generated classes are called from the generated program. In generated classes the receiver is stored in the field `rec`. Note that pre and postconditions of the `run` method of generated classes should be fulfilled manually.

By a proof by induction over s's structure, we showed that our technique is sound in the sense that it does not produce false positives. If a generated program cannot be proven, then either the verification technique is too weak or method contracts do not adhere to the protocol considered. In both cases, some (client-provided) programs will fail to verify (even if they obey the order on method calls induced by the protocol).

Example of the `Iterator` interface.

To check that method contracts of the `Iterator` interface adhere to interface `Iterator`'s protocol (shown in Section 4), one has to verify the method shown in Figure 2. The method's precondition is `i.init` because our model language does not have constructors and `init` is a special predicate that is obtained after object creation. The predicate `Perm(i.v,1)` represents the permissions to read/write the extra ghost field added for expressing `Iterator`'s protocol. Note that ghost fields are usually not allowed in interfaces [3]. For simplicity, we allow to use ghost fields in interfaces because, when checking adherence, we can consider interfaces as classes (we only use method contracts, not method implementations).

The program shown in Figure 2 can be verified with standard (pen and paper) techniques for verifying object-oriented programs annotated with our flavor of separation logic [10, 11].

Implementation.

We implemented the set of rules shown in Figure 1. Our implementation, examples of protocols and proofs of the corresponding generated programs are available [15].

7. Evaluation and Limitations

Previous work [8] already showed the usefulness of specifying protocols of sequential classes. Because we extend [8] to deal with protocol variables and a variant of generic classes, some sequential classes whose protocol cannot be precisely expressed with [8]'s language can now be expressed (as the `Iterator` example). We evaluated the usefulness of our extension for multithreaded classes: we found that that immutable classes, temporarily immutable classes (classes whose instances alternate between being written by a single thread and read by multiple threads), and classes which allow multiple methods to execute in parallel on single instances of these classes (like the `Roster` example) can be specified with our protocols [15].

Among other examples, we evaluated our technique of checking adherence of contracts while we were trying to prove an implementation of the `Iterator` interface [11]. We found that, without the systematic approach of checking the adherence of method contracts to protocols, it was time-consuming to find the right way to write specifications. Other examples are available [15].

8. Related Work and Future Work

Design by Contract (DBC) was first introduced by Meyer[2] in the Eiffel programming language. Eiffel features method contracts, invariants, and built-in facilities to dynamically check contracts. Eiffel does not support protocols and there is no tool to statically check properties of Eiffel programs.

Support for DBC in Java is provided by several tools including JML [3], Jass [16], and ESC/Java2 [6]. The earlier work of Cheon and Perumandla [8] to provide method call specifications in JML greatly inspired our work. Jass permits to specify protocols in the style of CSP. Both the work of Cheon and Perumandla and Jass support dynamic checking of protocols. We do not provide an implementation for dynamically checking protocols but our goal is different: we use protocols to statically check adherence of method contracts to protocols. ESC/Java2 permits to statically check many properties of Java programs. It does not support protocols.

Support for DBC in C# is provided by Spec# [4] and Boogie [7]. There is no support for protocols.

Separation logic [12] has been adapted to a sequential Java-like setting by Parkinson [10]. Later Parkinson's work has been extended to support fork/join parallelism [11]. The

$\mathsf{gen}(r, m(T_0, \ldots, T_n), T) \triangleq T''o = r.m(\texttt{havoc}(T_0), \ldots, \texttt{havoc}(T_n))$

$\mathsf{gen}(r, v = m(T_0, \ldots, T_n), T) \triangleq T''o = r.v = m(\texttt{havoc}(T_0), \ldots, \texttt{havoc}(T_n))$

$\mathsf{gen}(r, s,\ s', T) \triangleq \mathsf{gen}(r, s, T); \mathsf{gen}(r, s', T)$

$\mathsf{gen}(r, s \mid s', T) \triangleq$ `boolean b = havoc(boolean); if(b){`$\mathsf{gen}(r, s, T)$`}else{`$\mathsf{gen}(r, s', T)$`}`

$\mathsf{gen}(r, s?, T) \triangleq$ `boolean b = havoc(boolean); if(b){`$\mathsf{gen}(r, s, T)$`}`

$\mathsf{gen}(r, s*, T) \triangleq$ `boolean b = havoc(boolean); while(b){`$\mathsf{gen}(r, s, T)$`}`

$\mathsf{gen}(r, s+, T) \triangleq \mathsf{gen}(r, s, T);$ `boolean b = havoc(boolean); while(b){`$\mathsf{gen}(r, s, T)$`}`

$\mathsf{gen}(r, e\ ?\ s : s', T) \triangleq$ `boolean b = havoc(boolean);`
`if(b){assume(e);`$\mathsf{gen}(r, s, T)$`}else{assume(!e);`$\mathsf{gen}(r, s', T)$`}`

$\mathsf{gen}(r, s \mid\mid s', T) \triangleq$ `ThreadS tS=new ThreadS(`r`); ThreadSp tSp=new ThreadSp(`r`);`
`tS.start(); tSp.start(); tS.join(); tSp.join()`

Class `ThreadS` is $\mathsf{classgen}(\texttt{ThreadS}, s, T)$ and Class `ThreadSp` is $\mathsf{classgen}(\texttt{ThreadSp}, s', T)$

$\mathsf{gen}(r, \texttt{!<}n\texttt{>}\ s, T) \triangleq$ `ThreadS[] at=new ThreadS[n]; int i; while(i<n){at[i]=new ThreadS(`r`);i++;}`
`i=0; while(i<n){at[i].start();i++;}; i=0; while(i<n){at[i].join();i++;}`

Class `ThreadS` is $\mathsf{classgen}(\texttt{ThreadS}, s, T)$

$\mathsf{classgen}(C, s, T) \triangleq$ `class` C `extends Thread{`T `rec; ... // initialization`
`requires ?;ensures ?; void run(){gen(rec,`s, T`)}}`

Convention: (1) In the first two cases, T'' is m's return type (2) All introduced names are fresh.

Figure 1: Program Generation to Check Adherence of Contracts to Protocol s of receiver r of Class T

works cited focused on foundational issues and no support for protocols is provided.

Future work includes the implementation of a runtime checker to check (dynamically) that programs respect their protocols. For this, we need to adapt the technique outlined in [8]. In a nutshell, it would be necessary to *(1)* add the appropriate ghost fields to classes, *(2)* synchronize access to the executable representation of specifications (i.e., automatons) and *(3)* extend automatons generation with the new cases of our specification language.

We did not mentioned inheritance earlier in the paper because it is unproblematic. As in the previous work on method call sequences [8] a straightforward interpretation of protocols inheritance is to conjoin inherited protocols to the protocols declared in the inheriting class. This means that a class has to respect inherited protocols (possibly ignoring methods that are declared in this class) and has to respect its own protocols. In addition, an inheriting class does not need to check adherence of its contract w.r.t. to inherited protocols: checking adherence of the class's protocol against the class's own method contracts is sufficient.

Finally, future work includes studying how to weaken the *receiver splittable* restriction mentioned in Section 6 for checking adherence.

9. Conclusion

We provide a concise and intuitive regular expression-like notation to specify protocols of multithreaded Java-like programs that use a variant of generic classes. We present the semantical foundations of our specification language.

We show a new technique to check that method contracts are correct w.r.t. to protocols. For this, we generate programs that must be proven to show the method contracts are correct w.r.t. to protocols. The program generator has been implemented.

Acknowledgments We thank Christian Haack for interesting discussions about separation logic and Marieke Huisman and Gustavo Petri for their feedback on earlier versions of this paper. We thank Paul Brauner for his remarkable assistance in reviewing later versions of this paper.

This work was supported in part by the European Commission IST-FET-2005-015905 Mobius project and by the Agence Nationale de la Recherche ANR-06-SETIN-010 ParSec project.

10. References

[1] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[2] B. Meyer. Eiffel: applying the principles of object-oriented design. *Computer Language*, 5(5):81–87, 1988.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 151–171. Springer-Verlag, 2004.

[5] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

[6] P. Chalin, J. R. Kiniry, G. T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Springer-Verlag, editor, *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 342–363, 2006.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*. Springer-Verlag, 2005.

[8] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, 2007.

[9] Java's documentation: http://java.sun.com/.

[10] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[11] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *Algebraic Methodology and Software Technology*, number 5140 in LNCS, pages 199–215. Springer-Verlag, 2008.

[12] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Press.

[13] R. Bornat, P. W. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, pages 259–270, New York, NY, USA, 2005. ACM Press.

[14] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, January 2008.

[15] Additional material (examples, detailed statistics, and implementation), http://tinyurl.com/6n4ma5.

[16] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification*, volume 55 of *ENTCS*. Elsevier, 2001.