Synchronization Component Verification

Vlad Gheorghe

2nd May 2008

1 Introduction

Multi-threaded programming has become a mainstream practice in the last years because of its ease of expressing simultaneous processes that many applications contain. Examples range from complex graphic user interfaces to powerful servers that are running on multiprocessor machines.

Although its merits in naturally expressing parallelism are incontestable, there is a well established agreement among programmers that working with multiple threads is by far more error-prone than sequential programming. Race conditions and deadlocks are known to occur as consequences of most subtle bugs.

This kind of programming errors have been studied and formalized in [3].

Race condition is a situation that may appear when many threads that execute concurrently access a shared resource without explicit synchronization. There are two different types of races.

General races can cause undeterministic execution and are failures of programs intended to be deterministic.

Data races cause non-atomic execution of critical sections. The effect of a data race is that the resource may be accessed by a thread when in an inconsistent state, hence being possible to lose its integrity (this can be formalized by the violation of some integrity invariants). In order to avoid data races programmers use synchronization mechanisms that allow the restriction of possible concurrent thread interleaving (with regard to the common shared resource) such that only executions that leave the resource in a consistent state are allowed. In particular, shared data is accesed from critical sections implemented by some synchronization mechanism.

The most classic example of synchronization mechanism is the *mutex* . By using this mechanism a thread can be given exclusive access to a certain shared resource, i.e. the guarantee that no other thread accesses the resource at the same time, thus being eliminated the possibility of interference between the concurrent threads.

In order to reason about concurrent programs one needs an execution model. Such a model, so-called *of interleaving atomic actions,* is described in [1]. A program state σ associates a value to each variable of the program. Execution of a sequential program results in a sequence of *atomic actions* that modify the program state. Execution of multiple concurrent sequential programs results in an interleaving sequence of the atomic actions of the component processes that can be described as a *history* or *computation sequence*

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_i} \sigma_i \xrightarrow{\alpha_{i+1}} \ldots$$

where σ_i are states and α_i are atomic transitions (actions). The sequence $\alpha_1\alpha_2\alpha_3\ldots$ is an interleaving of the sequences of atomic action resulting from the execution of concurrent processes.

The *behavior* of a concurrent program is defined by the set of all possible histories it can exhibit during execution, each history corresponding to a particular interleaving of those sequence of atomic actions of the component processes.

In order to interact, processes must communicate and synchronize. *Communication* allows one process to influence execution of another one, and can be accomplished in two fundamentally different ways : *shared variables* and *message passing*. When shared variables are used, a process may write to a variable that can be accesed later by another process. When message passing is used, one process sends a message to another process.

As we want to analyze Java programs, we are placed on the shared variable side of inter-process communication.

To communicate, one thread changes the state of an object and the other reads it. This will work only if the second thread performs the read *after* the first thread has finised with the writing. If the second thread reads the shared object state before the modification it can get a meaningful, but incorrect state from the communication point of view. Even worse, if the read is performed concurrently with the write (in the case that at least one of these two operations consists of more than one atomic actions) the former may result in an meaningless object state.

The ability of forcing a thread to delay execution in order to prevent communication errors is called *synchronization*.

In a shared memory framework, as the Java platform is, two kinds of synchronization appear and are supported natively by the language through explicit synchronization primitives :

- *mutual exclusion* groups actions of a thread into *critical sections* that are never interleaved with actions of other threads. Java language supports critical sections by the **synchronized** keyword

- *conditional synchronization* allows a thread to be suspended until some

specified condition becomes true. This is implemented in Java by the primitive methods **wait**, **notify** and **notifyAll** of the Object class.

A more detailed insight to Java synchronization primitives will be given in 5.1.

Parallel programming is known to be difficult and error-prone. The history of parallel algorithms has many examples of incorrect and published algorithms for different problems of various complexity, ranging from mutual exclusion to the difficult task of garbage collection. In many of such cases the authors were outstanding computer scientists. The bottom line is that the task of programming concurrent threads that communicate and are correctly synchronized is too complex in its general form to be easily comprehensible by humans.

However, once a parallel program is written, there are methods that can verify that it is correct. That is, it really does what is supposed to do, according to the *specification* of problem it is intended to solve. Thus, a primary requirement for the verification is the presence of a specification for the problem, along with the implementation. Usually, in the program development process, the specification is informal, whereas a formal verification requires a *formal* representation. To this end, specification languages have been developed.

However, there are significant differences in style between languages for fromulating properties and specifications of a computational task and programming languages which describe precise, unambiguous and efficent solutions that are reasonably easy for a computer to execute. The abstraction levelof the former is usually much higher than that of the latter. Consequently, the key issue in formal verification is establishing the relation between these two widely different levels of description of a problem.

** In general, the natural questions one could ask about .....

This work aims to develop an effective method that can be used to perform verification of programs for detecting synchronization errors.

The rest of this paper is organized as follows. In Section 2 we review the motives that make formal verification desirable. After that Section 3 gives an overview of other work regarding formal software verification that contains many base concepts of the domain. Section 4 presents our original approach to race condition detection. Before describing in Section 6 the automated tool implemented in this project, the more important concepts for the implementation are described in Section 5. Some practical results that were obtained using our verification methodology are presented in Section 7. Next, we discuss some of the issues that arised during the development of our methodology, drawbacks and improvement opportunities in Section 8, before concluding with Section 9.

2 Motivation

In an attempt to tame the complexity of parallel programming different idioms have been developed. These consist of sets of "programming rules" that, when correctly applied, guarantee freedom of synchronization errors. For example, "when accessing a shared variable the protecting lock must be held". While this approach of programming discipline is susceptible of errors because consistent implementation of the rules is only be verified by hand, it also incurs loss of performance, as for a particular problem may require more synchronization than sufficient. It is known that synchronization causes performance loss because of two reasons. First, it involves an inherent overhead that results from global communication between threads. As a typical configuration runs threads on multiple processors, sharing a common main memory, each processor must flush the local memory (cache) and, before continuing execution, it must refresh all the local copies that have changed, in order to maintain consistency. Second, synchronization results in the *serialization* of operations on the different processors in the system. This leads to lesser speed-up factors, that also have a negative effect on performance, as the potential parallelism of the system is not properly used.

Another approach is to design higher level synchronization mechanisms that will hopefully help to implement more naturally a broader class of synchronization problems. The *monitor* concept, introduced by C.A.R. Hoare in the 1970's, is an example of such a construct. However, this approach may introduce unnecessary synchronization as well, because each synchronization problem has its own particularities that only can be addressed best with custom synchronization.

Designing synchronization mechanisms specific to each application would issue better performance because the minimum synchronization required to maintain data consistency over every possible execution of the parallel program is implemented. The complexity of such an implementation would make synchronization errors to get easily unnoticed, hence making it more error-prone. This is the moment when formal verification comes into play.

Formal verification is a set of techniques that can *increase one's confidence* in the correctness of some implementation of a problem. This definition stresses the fact that absolute correctness of a program cannot be determined. In general, the verification of a certain implementation is performed *against* a specification of the problem. Although the verification process ends with success, the specification may contain errors. On the other hand the verification process itself may be faulty, whether it is performed by hand, in the form of a mathematical demonstration, or it is automated by using software verification tools. However, after successfuly passing a verification process, an implementation is improbable still to have errors.

There still remains the problem of how to write a correct specification. This task is as hard as the verification problem itself. A program which is proved to be correct relative to a faulty specification remains incorrect w.r.t. its intended

purpose. For this matter, the specification method should be as concise and human comprehensible as possible because at this point human good sense alone can decide whether it is correct or not.

The need of formal verification has become more acute lately with the emergence of embedded systems. In this field that involves mass production a fault of a system that escaped the testing phase to distribution on market can cause massive financial losses. Hardware producers are facing similar problems, too.

Safety critical systems is another category of applications where verification is necessary. The failure of such systems involve in general important losses (expensive equipment needed in space missions, large amounts of money in financial transaction systems or even human lifes in traffic control systems) that justify the high overall cost. In fact, this class of applications has been the playground of fromal verification methods since the beginning.

One problem with classic methods of formal verification is the cost. It takes to have highly qualified proffesionals in mathematics, logic, theorem proving or model checking in order to perform such correctness proofs. While this is acceptable for costly safety critical systems, it is no longer feasible with the proliferation of low cost embedded applications. It follows that an automated, simple to understand and to use verification method is highly desirable. This would allow a dramatic quality improvment of software at a significantly lower cost.

Another difficulty is the integration of the verification process in the software development process. In general, verification is performed after the entire implementation is completed and operates at the implementation level. In case that an error is found in the verification stage it can be difficult to realize whether the source of the bug is at logical level, dating from the design stage, or at implementation level. If the former is found, the project must re-iterate the development process from the early design phase. To avoid this kind of unpleasant situations, the developers should be able to detect such errors from the early stages of the project. One solution is to use a component-based approach toghether with a component verification technique.

If, instead of performing the verification at a late stage of the development, the system is paritioned in independent components that are verified independently then the potential errors will be detected much sooner and it will be more easy to track down and fix, as they are already localized to a ceratin component.

The main challenge of practically every formal verification method is to state what correctness really means. That is, to give a formal specification for the verified system. This can become a daunting task for fair large systems.A common approach to system verification is to create, using a special purpose representation, a *model* of the system that has the same execution semantics as the source language and then *specify* the desired properties using the model. If the translation to the language supported by the verification process is done manually errors

can be introduced. An automated translation is preferrable in this case. Another issue is that, in most of the approaces, the specification is given in terms of the source program or its model (e.g. in the form of predicates over the original variables). That means that a specification can not be reused for another component implementing he same logical behaviour, but it has to b entirely rewritten in terms of the new component's variables. The independence of the specification from the specified component would enable reusability.

The aim of this work is to devise an effective, component oriented, static verification methodology, based on a comprehensible specification method, that is targeted at detecting synchronization errors in multithreaded, shared memory programs.

3 Related work

Given that race conditions are amongst the most difficult to detect, reproduce and eliminate programming errors, there has been a continuous interest in developing tools that help programmers in these matters.

There are two main categories of methods in this domain: static (compile-time) methods and dynamic (run-time) methods.

In run-time methods the synchronization primitives are instrumented with instructions that generate a usage log at run-time. This log is inspected (run-time or post-mortem) in order to find synchronization errors such as race conditions. A classic example of such tool is Eraser [2].

This kind of detection methods are mainly used in the testing phase of program development, as there is of not much help detecting errors in a program already in use (aside from stopping the program such that it does not yield erroneous results). Another major drawback is that race conditions may occur only on very particular executions (nondeterminism being scheduling dependent) and thus infrequent. To improve this situation mixed approaches of dynamic and static detection have been devised.

Static methods only use the program source code in the attempt to find synchronization errors without running the program. This is a major advantage over run-time methods, beacause if an error exists it is guaranteed to be detected. The downside is that the definition for errors is not always accurate, so only a particular class of errors are detected, or even 'spurious' errors (it is the case in that due to the lack of information a certain offending but infesasible execution is taken as feasible).

Several approaches lay in the static verification methods category: theorem proving, type checking and model checking.

The theorem proving approach uses predicates over program variables to specify the set of correct program behaviors.

-safety /liveness

There are also several recent papers on static, *type checking*-based methods [4, 5]. The basic idea is to add locking semantics to the type system of the original language. Thus, when declaring a variable, one can specify the "guarding" lock aside the variable's datatype (or, as an equivalent approach, the "owning" object). Then the typesystem ensures that a thread holds the respective lock whenever the variable is accessed. The result is that the programmer is forced to obey a programming discipline, otherwise the type checking will not succeed.

The major problem with this approach is that it is only able to verify local properties, while the correcness of general synchronization is a global property. The target language only contains mutexes as synchronization primitives, thus making impossible the implementation of more complex synchronization behavior such as read-write locks. This method can only by applied to the simplest (and also the most unefficent) idiom of data race avoidance : accesses to all shared data shall be made only within critical sections. For many problems this requirement is too conservative and results in performance degradation.

-MC

4 Approach

Our goal is to find a way to statically detect race conditions that may occur in a program. In particular, the focus is on data races, but, as we will see further on, general races can also detected.

The Java programming language has been chosen source language for its established qualities such as being easily analyzable (mainly due to the absence of pointers) and native multiprogramming support.

On the route to our goal, we must first define program corectness for our particular purpose.

Data races occur as a direct consequence of failing to insure *atomic execution* of the actions of a single thread that access shared varibles. The actions in question are said to form a *critical section.* The intention is of altering the shared object (that contains many variables) form a consistent state to another consistent state without interference from another concurrent threads. Note that, in this context, atomicity is only relative to the set of shared variables accessed inside the critical section; another thread can still interleave its actions as long as these do not refer to variables from this set. So, from data race freedom point of view, a program is correct if all critical sections are correctly implemented.

Any multi-threaded programming environment must provide some explicit synchronization primitives. The Java language has native support for such primitives in the form of *monitors* (details are given in 5.1).

Critical section is a conceptual construct that can be implemented in general using the synchronization primitives at hand. The same effect can be obtained by exploiting the dependencies between actions of different threads induced by accesses to the same shared variables. However, this leads to inefficient implementations, thus making the former the preferred alternative.

The simplest way of implementing a critical section is by using *locks.* Before entering the critical section a thread first acquires the protecting lock, and releases it after leaving the critical section. If every thread conforms to this rule then the critical section property is preserved.

Note that the Java language only supports locks in the form of **synchronized** code blocks with the semantics that the specified lock is acquired before entering the block and released immediately afterwards. This means that a thread can use a lock only to explicitly implement a critical section of code, and not for inter-thread signalling (the other primitives are reserved for that purpose). This synchronization mechanism is actually called *mutex* (for *mut*ual *ex*clusion).

However, the mutex is not powerfull enough to express more advanced synchonization idioms. An widely used such idiom is the *read-write lock.* When multiple threads share a resource, read operations can be executed concurrently without any danger of data corruption, but write operations must be executed in an exclusive fashion. This funcionality is of great importance for the performance of concurrent database systems, to give one example.

Efficient synchronization can be achieved only if custom designed for each problem, using the full range of synchronization primitives provided by the language, i.e. both *locks* and *condition variables.*

To summarize, the programs that we want to analyze for race freedom implement critical sections in terms of locks and condition variables.

Now, we come to the difficult task of *specifying* the problem that such programs implement, focusing on synchronization. After all, how can we specify that critical sections are correctly implemented, i.e. the actions that they consist of are atomically executed by every possible computation sequence ? A simple observation is that, before we can get to the verification of such properties, we have to know exactly which actions belong to the respective critical section. Assuming that the source code is mapped to sequences of actions, this amounts to designating the particular block of source code that is intended to be executed as critical section.

Suppose that we have to verify a program that uses the read-write lock idiom. A decent Java implementation would contain a ReadWriteLock class that provides the threads with methods that implement the desired semantics, i.e. readLock(), readUnLock(), writeLock() and writeUnLock(). In order to give a specification, the user has to designate in the program source which sections of

code must execute atomically. While in the case of a write section this amounts to marking the section of the program between writeLock and writeUnLock (as this as to execute atomically), for the reads the critical sections are inside the ReadWriteLock class.

In the quest for a concise and comprehensible specification method, we conclude that critical sections are situated at a too low level to be used as means of specifying corectness. Therefore we abandon this abstraction level for the specification, but move one level up and continue.

We introduce a verification method that complements the type-checking approach to data race detection. In that approach is checked that a 'user' program makes correct use of a primitive synchronization mechanism - Java's **synchronized**, i.e. mutex. We generalize the class of synchronization mechanisms to more complex (and powerful) ones and argue these to be correct w.r.t their specifications. The verification process is greatly simplified by the use of a specification model devised for the special purpose of easily stating the correctness of synchronization components.

The approach to verification of multi-threaded programs presented in this paper is in line with the component orientation of present software development practices, that are guided by concepts as modularity and compositionality, both of these being captured by component-based programming.

When a complex program is being designed it is desirable functionality to be partitioned in modules as independent as possible. For a multi-threaded program, the inter-thread communication and synchronization functionality is a perfect candidate for such a module. For object-oriented languages such functionality would be implemented in classes whose instances we will call *synchronization objects.* Threads are being synchronized by using synchronization objects as high level components. Data exchange through synchronization objects may also be permitted (e.g. bounded buffer).

This approach to programming not only isolates in a component some independent behavior, but also the perils of multi-threading programming. If every inter-thread communication is made through synchronization objects any synchronization error will occur inside these components. It follows that in order to answer whether a program is free of synchronization errors it suffices to consider the synchronization components.

A major difficulty in formal verification is identifying what needs to be proved, i.e. the specification that states the expected (correct) behavior of a component.

The purpose of the specification is, beside to state correctness, to separate the implementation of a component from its usage. From the viewpoint of the component, the specification represents the functional requirements of the potential clients that must be implemented accordingly. From the point of view of the "user", the specification describes the valid usage patterns of the component.

From an external point of view such synchronization components have sequential behavior in the sense that the collection of threads that use them are only able to observe the operation's invocations as atomic actions, given that data inside synchronization components is totally encapsulated. This leads to the idea that a specification for this kind of components may be formulated in a sequential fashion, for instance using *abstract data types* (ADTs).

Methods of a synchronization object exhibit the following functions :

- *blocking* methods are able to suspend a thread's execution until the system reaches a particular condition (resource is available, awaited event occurred, etc.)
- *notification* - called by a thread to signal an event to the syn-object. The syn-object can take appropriate action (waking-up other threads, for instance)
- *observation* - inquiries on the syn-object's state.

We argue that the specification method we introduce is able to express the blocking-notification functionality of these methods.

The specification of a synchronization component describes an abstract model of a virtual implementation. The model has an abstract internal state described as an ADT in the specification language.

The model given in the specification is implicitly bound to a certain implementation through the method's names. Every method in the implementation contains two outstanding locations, labeled *on_ entry* and *on_ exit.* We will give characterizing properties for them later.

Each of these outstanding locations in the implementation are conceptually present in the specification, too. Actually, they represent the *only* relation between implementation and specification, and are therefore called *joinpoints.*

At each joinpoint the specification contains two predicates and an atomic action that operate on the abstract (specification's) state.

The basic intuition of our model is that the specification and the implementation of a synchronization object "execute" simultaneously, the *only* link between them being the *joinpoints*, *on_ entry* and *on_ exit*, for each method, where a conceptual rendez-vous between the two takes place (Figure 2 on page 12).

Correctness is expressed by the predicates of the specification's joinpoints, that must hold for every execution of the corresponding joinpoint in the implementation.

For each joinpoint there are two predicates over the abstract state :

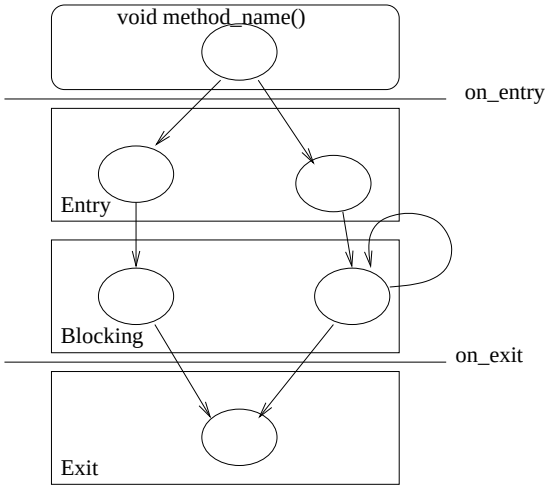


Figure 1: A method has three sections

- an *assertion* - ensures that the implementation only allows correct executions of the methods. It captures the correctness of the implementation of the synchronization component
- an *assumption* - ensures that the component is used in a valid manner, i.e. the methods are called in a correct pattern according to the semantics of the component. It captures the correct usage of the implemented component.

Correctness of an implementation against a specification is expressed in general by one of the assertions of the *on_ exit* actions from the model. The implementation drives the control flow and the model monitors through assertions that at certain points in the execution the model state is valid.

Note that the actions of the model operate on the 'abstract' state, not the 'real' state (made up of the implementation's variables).

The *on_ entry* joinpoint is placed right at the first location of the method.

The meaning of *on_ exit* is that when reaching this point a thread cannot be blocked anymore by the implementation. An characterizing property is that when calling a method in an single-thread environment, for certain valid states of the synchronization object, the thread would block just before the method's *on_ exit* point.

However, for simplification reasons we assume every method **synchronized** and then the *on_ exit* point coincides with the last location of the method.

Using this simple model and the power of expressing abstract the state as and ADT term (also being able to use ready to use structures defined in libraries) the user can easily specify complex behavior of a synchronization object.

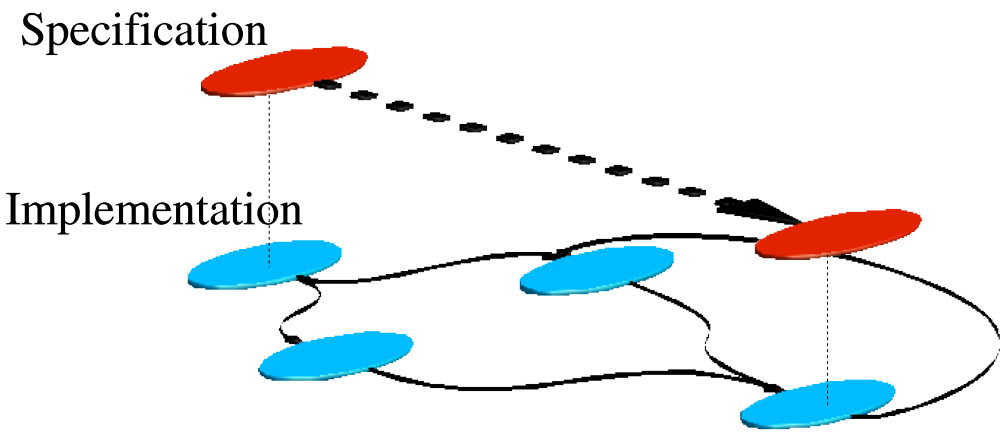


Figure 2: Specification and implementation are linked through joinpoints

A major advantage is the independence of the specification from the implementation. This allows the separation of concerns in the matters of specification and implementation. Also, once a specification written, it can be used to verify different implementations, thus promoting reusability. On the other hand, a single implementation can be verified against different specifications, with different levels of abstraction. We will elaborate more on this idea in Section 8.

Once we have both the implementation and the specification all we need in a platform that supports the semantics of both and that will perform the real verification. After many failed attempts with different high level languages, theorem proving verification methods and tools, all the answers were given by a specification language created by Leslie Lamport [8], Temporal Logic of Actions (TLA+). It has been designed as a specification language for general systems, with the ability of expression at practically any abstraction level. This made it possible to implement an automated tool that translates the original Java code of the implementation and merges it with the specification's internal actions and asssertions/assumptions, resulting in a TLA+ specification that describes the whole assembly. The tool is named *FUSION*.

TLA+ is described in more detail in 5.3.

The determinant factor in chosing TLA+ as a target language is the tool support. The Temporal Logic Checker (TLC) uses a form of model checking in

order to verify TLA+ specifications.

The language and the model-checking tool support both safety (the case of assertions/assumptions that we use) and liveness properties. This means that eventual liveness properties can be later added to the component specification, these being already supported by the target language.

4.1 Verification process overview

The user's point of view of a Java component's verification is depicted in Figure 3 on page 14.

The FUSION tool takes as arguments the names of three input files :

- Java source code - a valid Java class that constitutes an implementation of the synchroniation object being verified
- Specification - a valid TLA+ module that defines special identifiers designating the specification state, initial predicate, and for every joinpoint of every method in the component the assumptions, assertions and actions over the specification state. Being a TLA+ module, the EXTENDS clause can be used to access standard or user libraries. This results in code reuse and improved power of expression in writing specifications.
- Composition - a text file describing a closed program that contains multiple concurrent threads making calls on the component's methods.

The detailed syntax of these files is described later in 6.1.

Two files are the result of the fusion process : out.tla and out.cfg. These are ready to be fed into the TLA+ checker, TLC, by a single shell command :

```
$ tlc out
```

As the invariant's verification is done by TLC as exhaustive state space exploration, this process can take some time for large (or complex) components as well as compositions. Although, in general the synchronization components contain a large part of critical section code, this having the effect of drastically reducing the allowed interleavings.

The outcome of the automated TLC checking has the following interpretations:

- Success - the *implementation* conforms to the *specification* for the given *composition*.
- An assumption is violated - according to the *specification*, the *composition* uses the component (i.e. calls its methods) in an invalid pattern.

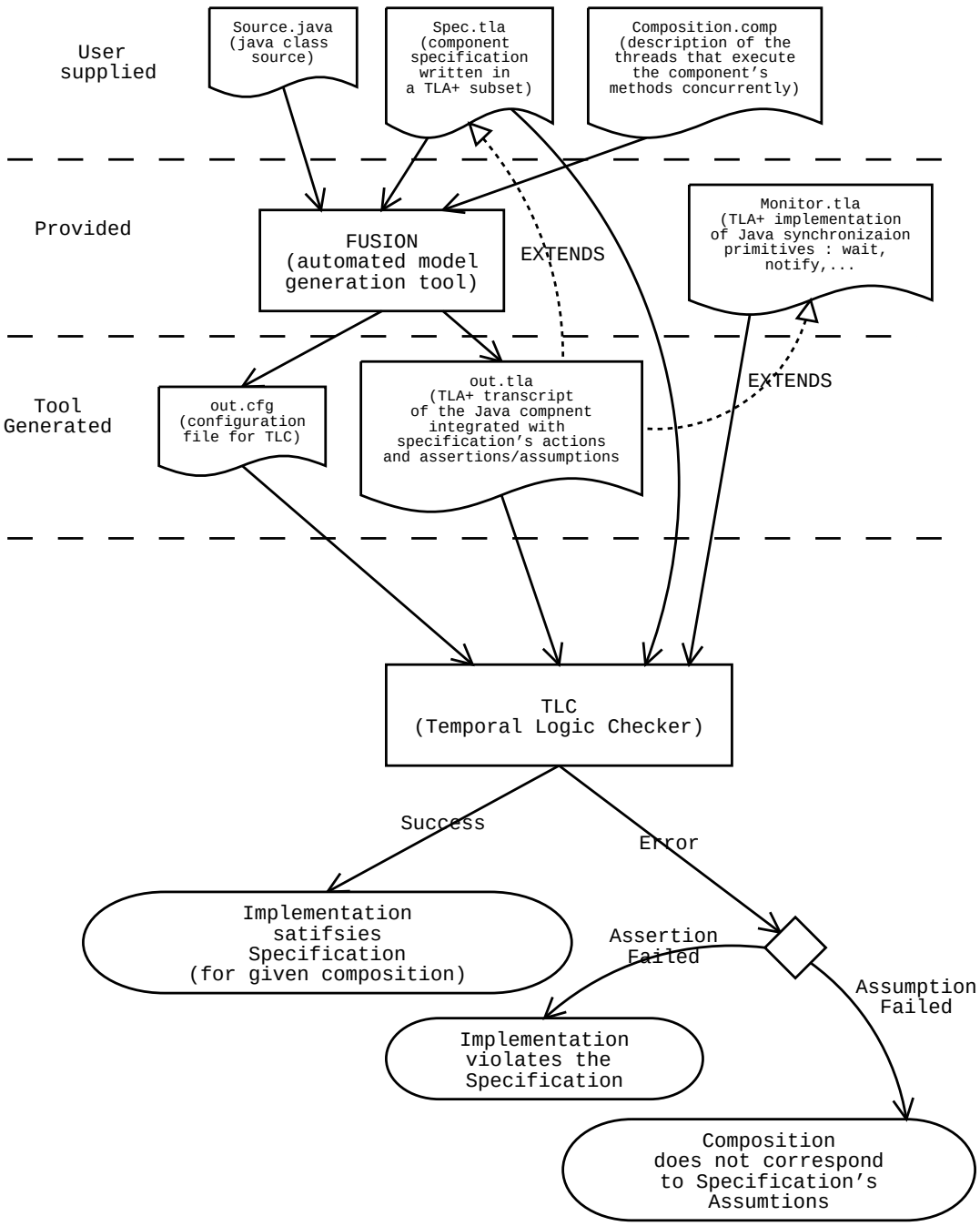


Figure 3: Information flow in the automated verification process

- An assertion is violated - according to the *specification*, the *implementation* allows executions that are invalid.

In case of failure, TLC also generates a trace of the offending execution, that is of great help in locating the actual fault in the implementation.

4.2 Example

- semaphore

5 Prerequisites

In this section we will overview the base concepts that this project is built upon.

5.1 The Java language semantics

As we chose Java as the source language, we have to implement the Java language semantics. The Java Language Specification (JLS) [6] defines a set of rules that every JVM implementation must obey.

Of particular interest is the Java Memory Model (JMM), that defines the rules that govern inter-thread communication through the main shared memory.

Referring to the Java programming language, only a few explicit synchronization mechanisms are provided :

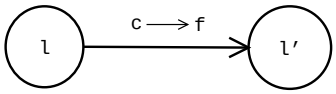
- *locks* - used by **synchronyzed** code blocks
- *condition variables* - accessed by **wait notify notifyAll**

5.2 Transition Diagrams as an execution model

We will use transition diagrams as described in [1] as the support of the Java language semantics. These are simple objects that are still powerfull enough to express thread concurrency, atomic actions and, most important, can be used to implement Java primitive synchronization mechanisms (monitor locks, **wait**, **notify**, **notifyAll**).

Transition diagrams describe the control structure of a program in terms of *locations* and *transitions*. Locations represent the program counter that points to the next instruction in the program to be executed. Transitions model the effect of acutal execution of program instructions in terms of a new value for the program counter. The execution itself is modeled by a state transformation, where state represents the contents of the memory. A state transformation consists of assigning to some memory cells the results of some operation performed with values read from the memory.

A transition diagram can be represented as a labelled directed graph. The nodes of the graph are referred to as locations. There is a distinguished node called the *entry* node, where the computation starts. Similarly, the distinguished *exit* node has no outgoing edges and is the node where every terminating computation ends. Each directed edge is labelled by an *instruction* of the form $c \rightarrow f$ where c denotes a total boolean function over the state space, also called *condition*, and f denotes a total state transformation:



- interleaving of actions
- advance assumptions

5.3 Temporal Logic of Actions as a multi-level language

- math based - easy to understand & use
  - modules, library of primitive types
  - highly expressive -> n/p with lts
  - safety & liveness properties
  - tool support

5.4 The Temporal Logic Checker (TLC)

- BFS, difftrace, config, performance

6 Implementation

The *FUSION* tool has been implemented to automate the process of translation from Java source code to TLA+ and integration of specification's actions in the result.

The language used in this implementation is Haskell [12].

The transformation process of the original Java code passes several levels :

1. Java source code
2. LTS - for each method or constructor a LTS with equivalent semantics can be generated.
3. The specification actions and assertions/assumptions are merged in the LTSs at the corresponding joinpoints.
4. TLA+ implementations of each LTS are generated

5. A closed program is created as a TLA+ module by assembling the constructor and methods in concurrent threads according to the description given in the *composition* file.

The specification is already a valid TLA+ module, so the generated module only needs to EXTEND it in order to access the specification initial predicates, actions, assertions and assumptions.

6.1 Input files

This section describes the various files that must be provided in order to perform component verrification.

6.1.1 Java source code

This is intended to be a real, compilable Java class source code, that implements the synchronization component that needs to be verified.

However, as the tool has been implemented for demonstration purposes only, just a subset of the Java language is supported.

The implemented gramar is given below in a BNF-like notation.

```
Class ::= { Modifier } class Identifier [ super Identifier]
            { { FieldsOrMethod } }
Modifier ::= static | public | private | protected | synchronized | abstract
FieldsOrMethod ::=
  { Modifier }
  ( Identifier MethodCont //constructor
    | Type Identifier
        ( { , Identifier } ; //fields declaration
          | MethodCont   // method definition
        )
  )
Type ::= void | int | boolean
MethodCont ::=
  ( [ Formal { , Formal } ] )
  [ throws Identifier { ,  Identifier } ]
  Block
Formal ::= { Modifier } Type Identifier
Block ::=
  { { Locals } {Statement} }
Locals ::= { Modifier } Type Identifier { ,  Identifier } ;
Statement ::=
```

```
    Block
  | synchronized ParExpr Block
  | if  ParExpr Statement [ else Statement ]
  | while ParExpr Statement
  | return Expr ;
  | Expr ;
  | ;
ParExpr ::= ( Expr )
Expr ::= ! Primary | - Primary | + Primary |
  Primary * Primary | Primary / Primary | Primary % Primary |
  Primary + Primary | Primary - Primary |
  Primary < Primary | Primary > Primary | Primary <= Primary | Primary >= Primary
  Primary == Primary | Primary != Primary |
  Primary && Primary |
  Primary || Primary |
  Primary = Primary
Primary ::=
    ParExpr
  | new Identifier
  | this
  | Access
  | Literal
Access ::= Identifier [ Arguments ] //access to a field or method
Arguments ::= ( [ Expr { ,  Expr } ] )
Literal ::= IntLiteral | BoolLiteral
IntLiteral ::= Integer
BoolLiteral ::= false | true
```

The grammar has several omissions, but proved to be sufficient to easily accomodate real-world examples.

As this grammar is a strict subset of the Java Language grammar, we can assume that the source code has already been compiled using a Java compiler, thus being correct both syntactically and semantically. However, at this level some checks are still performed.

Note that there is no use of Java object references, as the present method does not support these. This is a major restriction that, as discussed in Section 8, could be lifted by the assimilation of an interthread alias analysis in the method.

6.1.2 The Specification File

6.1.3 The Composition File

The target language and accompanying tool (TLA+ and TLC respectively) only directly support *closed* systems, i.e. systems with all components known at the

time of verification. The synchronization component that we intent to verify constitutes an *open* system, as it can be used in virtually any environment.

Throughout the verification process, the composition file has the role of the "unknown" environment. The user is free (and even encouraged) to perform the component's verification repeatedly, with several compositions. Knowing the complexity of the component, one should be able to devise an usage (composition of component methods) that will generate a sufficiently large set of executions to expose all potential synchronization errors.

The composition file describes a set of threads concurrently executing calls on the same shared component. Each thread is a sequential composition of such calls.

The syntax is very simple :

```
Composition ::= Thread { || Thread }
Thread ::= MethodName { ; MethodName }
```

6.2 Java frontend

The Java grammar subset used (as defined in 6.1.1) has been implemented using the monadic parser combinator library Parsec[11].

The high degree of reusability and polymorphism specific to Haskell allowed the implementation of the parser to be very close to the initial grammar.

In the monadic setting, a parser is an action over an input string (consisting of characters or tokens, depending on whether one has chosen to use basic parsers or to implement the lexers separately, by hand) that results in a representation, according to the implemented grammar, of a *part* of the input string (this is generally an *abstract syntax tree*, or AST in short) and the remaining, unparsed string.

The library provides very useful basic character parsers (such as **integer** or **identifier**) that actually are lexeme parsers and can be configured according to the particularity of the source language. These basic parsers can then be "assembled" into more complex parsers, by using *combinators.*

Parser combinators are functions that have parsers both as arguments and results. Examples of such parsers are :

- **many** p - is the parser that accepts a string of repeated strings accepted by the argument p
- p1 <|> p2 - first try p1, and if it fails report no error and parse p2

Combinator parsing is very well suited to funcional languages and provides an elegant solution with several advantages over classic parser generators such as YACC :

- Combinator parsers are written and used within the same programming language as the rest of the program. There is no gap between the grammar formalism (yacc) and the host programming language (C). This also implies that the development takes place in a single environment, with the same rules for type checking, modularisation, and so on.

- Parsers are first-class values within the language. They can be put into lists, passed as parameters and returned as values. It is easy to extend the existing set of parsers to custom made parsers specific to the problem.

A very powerful feature that has been extensively used is the expression parser generator, **buildExpressionParser**. It supports prefix, postfix and infix operations and priority levels.

The parser is implemented in the module *Java.hs.* The internal representation of Java code, the abstract syntax tree, along with the corresponding accesor and utility methods, is defined in a separate module, *JavaAS.hs*, as it is also used by the next stages of the generation process.

The verification system supports local block variables, so nested namespaced had to be used to implement this functionality at frontend level.

6.3 Java to Transition Diagrams translation

6.4 Transition Diagrams to TLA+ translation

6.5 Monitor library

The Java Virtual Machine provides instructions that can be used to implement the synchronization primitives of Java language. As the source code that uses them is translated to TLA+, we must also implement the synchronyzation primitives, with the semantics described in [7], in TLA+.

Java semantics make no assumption whatsoever about the order of threads

The Java synchronization primitives were designed as transition diagrams and implemented in TLA+ by hand. The diagrams are given in figure Figure 4 on page 21.

The monitor is implemented using the following shared variables :

- *lock* is the state of the lock protecting the monitor. Must be positive. A thread may acquire a lock multiple times.

- *owner* is the unique ID of the thread that is holding the lock (if any). It helps implement the lock re-acquiring capability.

- *CV* (Condition Variable) is used by a thread that calls **notify**/**notifyAll** to signal the waiting threads.

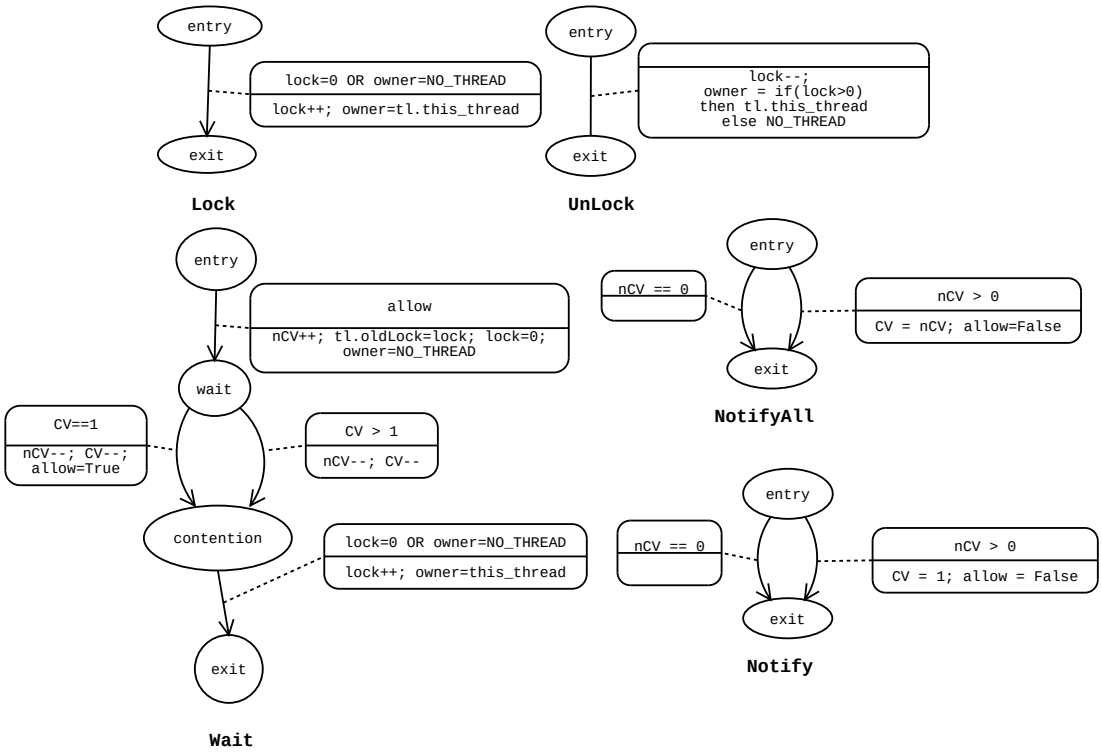


Figure 4: LTS diagrams for the synchronization primitives

- *nCV* keeps track of the number of threads waiting on the condition variable
- *allow* is an auxiliary boolean variable that helps preventing other threads to enter the waiting queue while this is in the flushing phase following a call to **notifyAll**.

The Lock/UnLock primitives are straigh-forward to implement.

When a thread calls **wait** or **notify(All)**, we can assume that it also owns the corresponding lock, as the Java specification states that an exception must be thrown otherwise.

Notify and **notifyAll** have no effect when no thread is waiting for the condition (nCV == 0).

If there are some threads waiting, **notify** assigns CV:=1 and

7 Experience

This section presents some practical results obtained using our verification methodology.

7.1 Semaphore

A semaphore is a shared object with two operations: P() and V(). V() increments the value of the semaphore P() decrements the value of the semaphore, only if that value would not become negative (i.e. the value is not zero). An implementation of a semaphore will block the thread calling P if the condition does not hold, until some future moment that another thread modifies (by calling V) the value s.t. the condition becomes true. The V operation may be called and complete any number of times. It just increments the semaphore's value, not violating any invariant of the model. It follows that any precondition on this method is inappropriate.

Algorithm 1 Simple semaphore specification

```
specification Semaphore {

    np, nv : Nat;
    on_exit P() {
      assertion : np < nv
      action : np' = np + 1
    }
    on_entry V()
    {
      action : nv' = nv + 1
    }
  }
```

And a Java implementation is given on the following page

The corresponding on_entry and on_exit points are marked in the implementation.

Note how the specification's state and implementation's state are independent and, necessary, equivalent. At this low level of complexity it is not hard to imagine an implementation straight from the semaphore specification. Nevertheless, for more complex synchronization components, the specification designer can use the full power of abstraction offered by TLA+ in order to write a concise and hence credible component specification.

7.2 Read-Write Lock

In order to demonstrate the expressiveness of the specification language we present a more elaborated example. This will also show that an object can be specified at different levels of refinement, thus a same implementation will comply to two different specifications, related by the refinement relation

Algorithm 2 A simple semaphore implementation in Java

```
class Semaphore {

      int value=0 ; // the implementation's state
      synchronized void P() {
       //ON_ENTRY
       while ( value == 0 ) { wait(); }
       //ON_EXIT
       value = value - 1 ;
      }
      synchronized void V() {
        //ON_ENTRY
        //ON_EXIT
        value = value + 1;
        notify();
      }
    }
```

Read-Write Lock is a synchronization object used to coordinate threads that access a resource in two modes : read and write. Before accessing the shared resource a thread must acquire a lock corresponding to the type of operation it will perform on the resource. After finishing the operation it must release the previously acquired lock. Related to the lock, a thread is characterized by the type of lock it holds : reader or writer. The lock can be in one of three states :

- unlocked
- read-locked
- write-locked

Multiple threads may hold the lock in its read-locked state. Only one thread may hold the lock in the write-lock state.

The example presented below complies to this semantics, and more : it includes a fairness policy that prevents the participating threads from starving when waiting to acquire a certain kind of lock. The policy states that :

- when in the *read-locked* state readers shall not be granted the lock if there are also writers waiting
- when both readers and writers request the lock it will be alternately granted to both threads categories.

This behavior is achieved by recording the kind of the last holder in the unlocked state. The lock will be granted to the threads of the opposite kind.

An implementation follows. Note that this implementation was taken from a quite respectable textbook.

If we drop the fairness in requirements the following specification can be written. This is a weaker specification than the first one, so the above implementation complies to it, also.

7.3 The checking : surprise, surprise...

As the semaphore example is pretty simple we expected that it is also correctly implemented, and so it is. Verification with many different configurations did not reveal any errors.

The surprises came from the Readers & Writers test, which was assumed to be error-free, given that it comes from a textbook. In fact it doesn't even prevent both readers and writers accessing the database concurrently, not to speak of enforcing fairness.

A check against the weak RWLock specification revealed that the OnExit assertion for *startWrite* is violated by an 83 steps long behavior. An inspection of the trace generated by the checker allowed the scenario diagram to be created.

If we take a second (advised) look at the code, it is not hard to notice that once a waiting reader is signaled nothing can stop it from entering the critical region, as soon it regains the monitor's lock, because the waiting condition will always be false from that point on (startWaitingReadersTime can only increase). It follows that if another writer (Writer1) gains the lock released by Writer0 in the contention with Reader0 it finds that the critical region is empty (Reader0 has not entered yet) so it proceeds. When Reader0 gets at last the monitor's lock it just finishes the startRead method and enters the critical region too, at the same time with Writer1.

When we tried to patch the bug (added a condition so that this situation would not happen) the checking process found another execution that violates an assertion.

8 Discussion

-Specification is closely related to OO interface + contract, specific to synchronization

An interesting note is that an implementation (defined by a Java class) can be seen as an instance of a specification; the instantiation relation between classes an objects is lifted to specifications and implementations (classes). Also, the

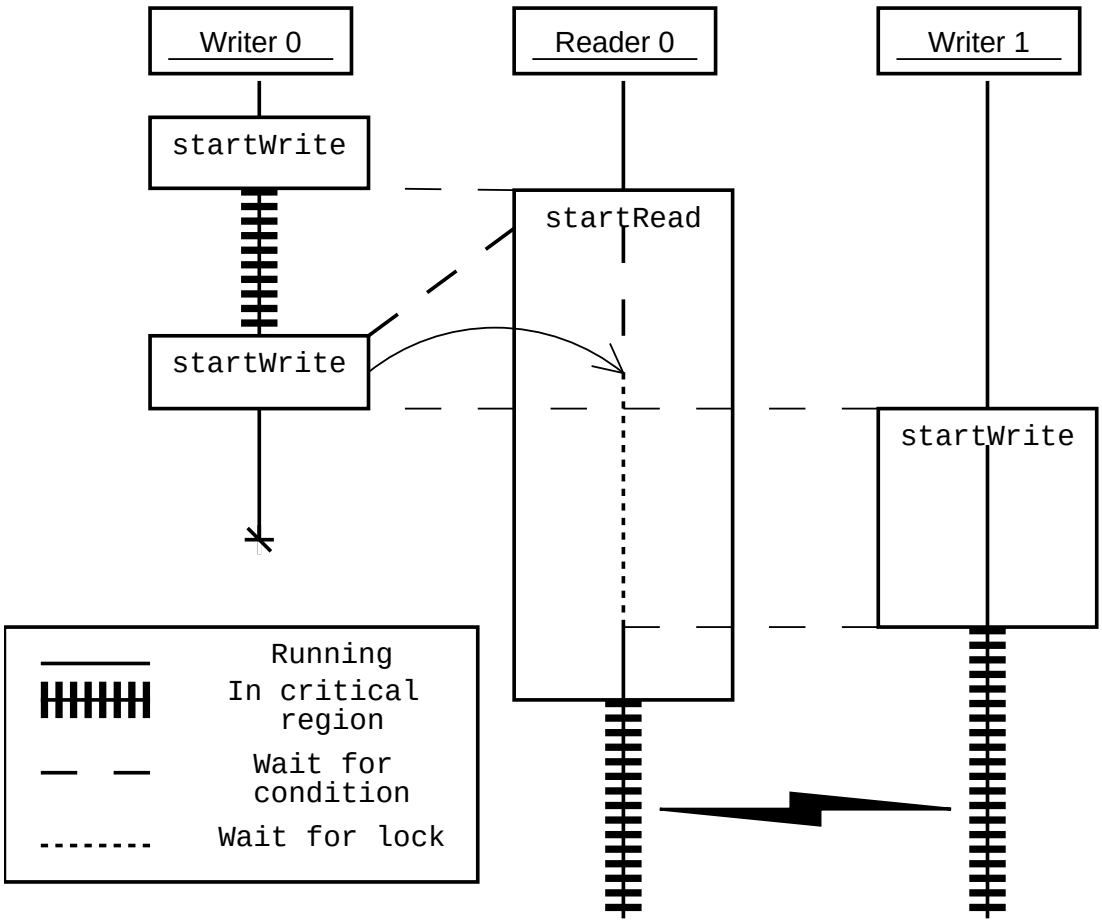


Figure 5: The read-write lock implementation has bugs...

generalization and specialization relations on classes lift over to specifications as a *refinement* relation.

We consider the denotation D of a specification S as the set of classes that comply to it .

$D = denot(S)$

A specification S1 *is refined* by a specification S2 iff the denotation of S1 includes the denotation of S2 .

$S1 \succ S2 \leftrightarrow denot(S1) \supseteq denot(S2)$

- JVM semantics, differences - abhick ?

- grammar omissions

- restrictions on components - no data ,just control. -> improvement : handle data transfer as well , eg bounded buffer; alias analysis

- alias analysis would allow use of object creation and manipulation inside the Java component, broadening the area of aplicability. Note that TLA+ already

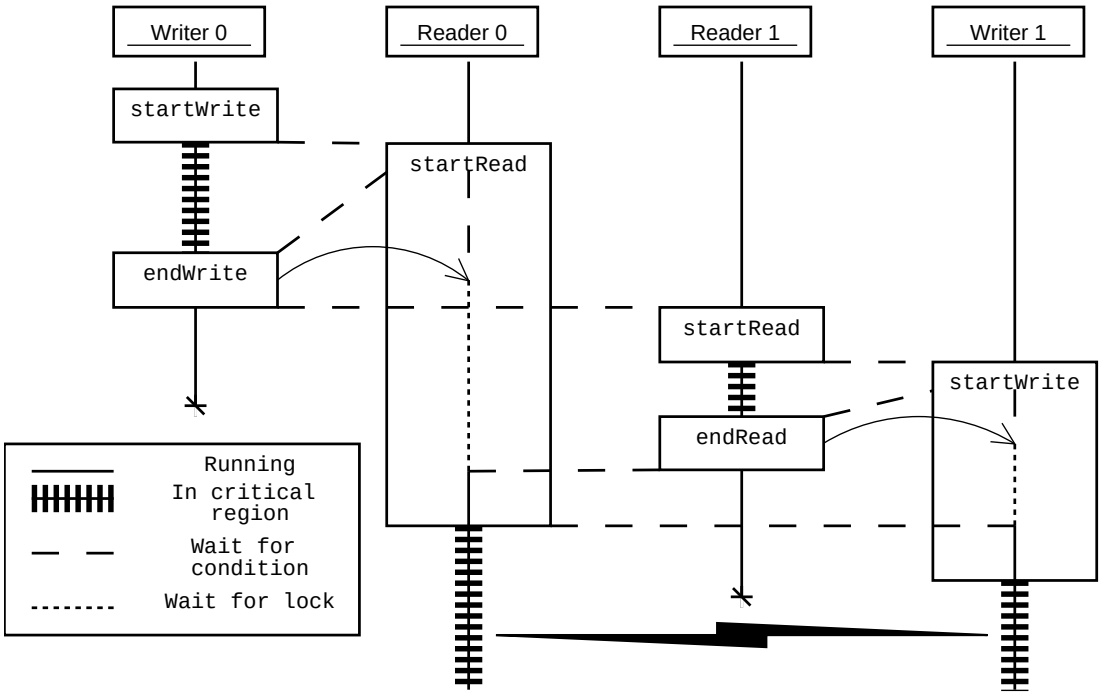


Figure 6: The patched version of fair read-write lock showed another bug

has the support for describing specifications of such components (the libraries contain Naturals,Sets, Sequences and practically any datatype can be defined)

- improvement of execution trace (code line) -> locating bugs easier

9 Conclusion

In this paper we introduced a simple and effective verification methodology for synchronization components.

10 ideas

- The modules verified are small enough for model checking being a viable alternative. As the use of mutual exclusion is of common occurence, the state space is reduced compared to regular program sections.
- We can assume that wait/notify are called within a synchronized block of the same object, as otherwise an IllegalMonitorStateException will be raised at runtime. (the JVM spec says that this is true only for notify, but most implementations do it for wait, also).
- Bugs in TLC (implementation experience)

- BEGIN with an example (- semaphore) to illustrate the whole generation process - explain techniques used...
- data race = failure to enforce atomicity. problem : how to specify where actions should be atomic. -one option=to indicate the critical regions. but if we knew that we could just insert mutexes at the specified points and the problem is solved. => we need a more high level method of specifying what "correct" really means.
- **independence** of spec from implementation, as opposed to classic methods
- composition - || includes ;

References

[1] Willem-Paul de Roever et al. Concurrency Verification. Introduction to Compositioanl and Noncompositional Methods. Cambridge University Press 2001

[2] Stefan Savage et al. Eraser : A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, Vol 15, No. 4, November 1997

[3] Robert H.B. Netzer, Barton P. Miller What Are Race Conditions ? Some issues and formalizations. in ACM Letters on Programming Languages and Systems, Vol. 1, No. 1, March 1992, Pages 74-88

[4] Martin Rinard, Chandrasekhar Boyapati. A Parametrized Type System for Race-Free Java Programs, OOPSLA 2001

[5] Cormac Flanagan, Stephen N. Freund. Type-Based Race Detection for Java

[6] J. Gosling, B. Joy, G. Steele. The Java Language Specification. Chapter 17, Addison Wesley, 1996

[7] Tim Lindholm, Frank Yellin. The JavaTM Virtual Machine Specification Copyright © 1999 Sun Microsystems, Inc. All rights reserved

[8] Leslie Lamport. Specifying Systems 04 Mar 2002 (Preliminary Draft)

[9] Stephen J. Hartley, Concurrent Programming: The Java Programming Language, Oxford University Press, 1998

[10] M. Ben-Ari, Principles of Concurrent and Distributed Programming, Prentice-Hall, 1990

[11] Daan Leijen. Parsec, a fast combinator parser. http://www.cs.uu.nl/~daan/parsec.html

[12] Haskell, a general purpose, purely functional programming language. http://www.haskell.org

Algorithm 3 A fair read-write lock specification

```
specification  FairRWLock {
import Nat
type LockKind = RLock | WLock
type RWLockState = RLocked Nat | WLocked | UnLocked LockKind
var
        lock : RWLockState = UnLocked RLock;
        nReadersWaiting : Nat = 0;
        nWritersWaiting : Nat = 0;
// readers
on_entry startRead() {
        action : nReadersWaiting' = nReadersWaiting + 1
}
on_exit startRead() {
        assert : (lock /= WLocked) /\
                (nWritersWaiting=0 \/ lock = UnLocked WLock)
        action :
          (nReadersWaiting' = nReadersWaiting -1) /\
        (lock' = case lock of {
                        UnLocked _ -> RLocked 1
                        RLocked n -> RLocked (n+1)
                 })
}
on_entry endRead() {
  assume : lock = RLocked _ // describes the usage
  action : lock' = case lock of {
                        RLocked 1 -> UnLocked RLock
                        RLocked n -> RLocked (n-1)
                }
}
// writers
on_entry startWrite() {
  action : nWritersWaiting' = nWritersWaiting + 1
}
on_exit startWrite() {
  assert : (lock = UnLocked RLock) \/
                (lock= UnLocked WLock /\ nReadersWaiting=0)
  action : (nWritersWaiting' = nWritersWaiting - 1) /\ (lock' = WLocked)
}
on_entry endWrite() {
  assume : lock = WLocked // describes the usage
  action : lock' = UnLocked WLock
}
}//spec
```

Algorithm 4 Fair read-write lock implementation in Java

```
class Database specification FairRWLock {
   private int numReaders = 0;
   private int numWriters = 0;
   private int numWaitingReaders = 0;
   private int numWaitingWriters = 0;
   private boolean okToWrite = true;
   private long startWaitingReadersTime = 0;
   private long age=0;
   public Database() {   }
   public synchronized void startRead() {
      //ON_ENTRY
      long readerArrivalTime = 0;
      if (numWaitingWriters > 0 || numWriters > 0) {
         numWaitingReaders++;
         readerArrivalTime = age++ ;
         while (readerArrivalTime >= startWaitingReadersTime)
            wait();
         //ON_EXIT1
         numWaitingReaders--;
      } else {
        //ON_EXIT2
      }
      numReaders++;
   }
   public synchronized void endRead() {
      //ON_ENTRY
      numReaders--;
      okToWrite = numReaders == 0;
      if (okToWrite) notifyAll();
   }
   public synchronized void startWrite() {
      //ON_ENTRY
      if (numReaders > 0 || numWriters > 0) {
         numWaitingWriters++;
         okToWrite = false;
         while (!okToWrite)
            wait();
         //ON_EXIT1
         numWaitingWriters--;
      } else {
         //ON_EXIT2
      }
      okToWrite = false;
      numWriters++;
   }
   public synchronized void endWrite() {
```

Algorithm 5 Read-write lock specification without any fairness requirements

```
specification  BasicRWLock {
import Nat
type RWLockState = RLocked Nat | WLocked | UnLocked
var
        lock : RWLockState = UnLocked RLock;
// readers
on_exit startRead() {
        precondition (lock != WLocked)
        lock' = case lock of {
                        UnLocked -> RLocked 1
                        RLocked n -> RLocked (n+1)
                 }
}
on_entry endRead() {
        assume lock = RLocked _ // describes the usage
        lock' = case lock of {
                        RLocked 1 -> UnLocked RLock
                        RLocked n -> RLocked (n-1)
                }
}
// writers
on_exit startWrite() {
        precondition (lock = UnLocked )
        lock' = WLocked
}
on_entry endWrite() {
        assume lock = WLocked // describes the usage
        lock' = UnLocked
}
}//spec
```