Dynamic Frames Based Verification Method for Concurrent Java Programs*

Wojciech Mostowski

Formal Methods and Tools, University of Twente, The Netherlands
w.mostowski@utwente.nl

Abstract. In this paper we discuss a verification method for concurrent Java programs based on the concept of dynamic frames. We build on our earlier work that proposes a new, symbolic permission system for concurrent reasoning and we provide the following new contributions. First, we describe our approach for proving program specifications to be self-framed w.r.t. permissions, which is a necessary condition to maintain soundness in concurrent reasoning. Second, we show how we use predicates to provide modular and reusable specifications for program synchronisation points, like locks or forked threads. Our work primarily targets the KeY verification system with its specification language JML* and symbolic execution proving method. Hence, we also give the current status of the work on implementation and we discuss some examples that are verifiable with KeY.

1 Introduction

Permission-based verification of concurrent programs relies on specifications in an appropriate formalism enriched with permission annotations [1]. These annotations specify the read or write access rights to memory locations of the program to be verified. The verification is *thread local* and, when successful, shows the absence of race conditions in the verified program as well as some functional properties to hold. Many verification formalisms for permission-based reasoning are built on Separation Logic [2] or equivalent Implicit Dynamic Frames [3,4].

In the context of the VerCors project[1] [5], which is concerned with verification of concurrent data structures, we propose an approach to permission-based verification built on top of the more fundamental Dynamic Frames [6] verification method. We base our work on the Java Dynamic Logic [7] and its implementation in the KeY verifier[2] [8]. KeY is a symbolic execution-based interactive verification system for Java programs annotated with JML [9]. In addition to our automated VerCors toolset [10], KeY is meant to provide interactive verification capabilities in the VerCors project for more involved Java programs.

In our earlier work we developed a symbolic permission system that remedies some of the problems we identified with fractional permissions [11] and we also

* This work is supported by ERC grant 258405 for the VerCors project.

[1] http://fmt.cs.utwente.nl/research/projects/VerCors/.

[2] http://www.key-project.org/.

provided a base line for verification of concurrent Java programs in KeY based on Dynamic Frames and explicit use of two memory heaps in the verification logic and the specification language [12]. In this paper we extend this earlier work and describe our method for showing self-framing of specification w.r.t. permissions, and we discuss the use of JML model methods [13] for modular specification and verification of concurrent Java programs that make use of API methods that involve synchronisation. Throughout the paper, we relate our approach to the existing ones.

The rest of this paper is organised as follows. Section 2 recapitulates our symbolic permission system, and briefly explains verification of Java programs with Dynamic Frames as implemented in the KeY verifier. Sections 3 to 5 present the main contributions of the paper. Section 6 concludes the paper, discusses the current state of the implementation and future work.

2 Background

Symbolic Permissions. In an earlier paper [11] we proposed a symbolic permission system for concurrent reasoning as an alternative to classical fractional style permissions. Symbolic permissions address some of the issues we identified with fractional permissions, like inflexibility to handle complex synchronisation scenarios. Here we only give a brief description of the main idea behind symbolic permissions and we refer the reader to [11] for full account, including formal definitions and mechanically proved consistency properties.

A single symbolic permission p refers to one heap memory location of the program to be verified. From the point of view of the currently running thread, permission p maintains information about which other threads possibly hold access to the memory location and which threads are the permission's *originators*, i.e., threads that the permission should be returned to during synchronisation. As in Java, threads are identified by their corresponding object references and the currently running thread is uniquely identified by ct. On the top level, the permission expression assigned to p consists of a list of permission *slices*, and each slice defines one piece of ownership of the permission. Such a slice is again a list that holds the history of owners (threads) of this slice, with the current owner at the head of the list, and the tail containing previous owners that this slice is owed to (originators of the slice). Permission p grants read access to thread t when there is at least one slice in p that is owned by t, while the write access requires all of the slices in p to be owned by t. In principle, no empty permissions (with no slices) or empty slices (with no owners) are allowed and the defined permission operations guarantee this property.

For example, after acquiring a simple read lock the running thread might hold a permission of the form $[[ct, l], [l]]$ to some memory location, where l is the lock that provides read access to the threads that acquire it.[3] This permission contains two slices. The first slice $[ct, l]$ belongs to the current thread and consequently

[3] Although locks are actually not threads, classifying them as such allows us to suitably generalise the symbolic permission approach.

grants it a read permission while upon lock release it will be returned to the lock l, the originator of this slice. The second slice belongs to lock l only and allows further acquirings of the lock by other threads. When the current thread releases the lock the complete permission becomes $[[l], [l]]$ which is semantically equivalent to $[[l]]$ (and can be reduced to become so) meaning that the lock holds the full access to the associated memory location.

With symbolic permissions, the core difference compared to fractional style permissions is how permission transfers are specified. In our approach we state what *kind of transfer* is applied to a permission rather than saying *how much* of the permission is transferred. Using functional style expressions, we specify how a permission is changed w.r.t. its previous value upon a synchronisation point. For our lock example, when acquiring the lock, the specification would say $p = \mathit{transferPerm}(\texttt{true}, l, ct, p')$. It states that the old permission p' becomes permission p after splitting (indicated by parameter `true`) one (any) slice that belongs to l and transferring the ownership of one of the newly created slices to ct. That is if $p' = [[l]]$ it becomes p in two steps, first it becomes $[[l], [l]]$ and then $[[ct, l], [l]]$. For a write lock, by giving a `false` first parameter, no splitting of the permission would be applied and p would become $[[ct, l]]$, temporarily giving ct full access right to the associated resource.

Such functional style specifications are particularly suitable for dynamic frames with explicit heaps as we explain in the next section. However, in many situations it is not possible to operate on concrete permissions expressions that explicitly state all the threads that share the permission. In fact, in the example above the read lock would be passing the permission to other threads unknown to ct and it cannot be assumed that the slices we specified are the only ones that comprise the permission at any point in time. To cover situations like this, abstraction of the permission is necessary and possible, as we show later in Sect. 5. In particular, instead of spelling out concrete permission expressions, one simply uses *readPerm* or *writePerm* predicates that establish if a permission is sufficient to grant a read or write access, respectively, to a given thread.

Dynamic Frames in JML* Specifications. In Dynamic Frames specifications [6] memory locations are first class citizens, typically stored in ghost or model variables typed as location sets, which in turn are used to specify method frames or frame dependency relations, and mechanisms are provided that allow to specify dynamic changes of these frames (typically called memory footprints). In the KeY verification system, dynamic frames are added to the Java Modelling Language (JML) [9], a behavioural specification language for Java, to form a KeY-specific version of JML called JML* [7].

Listing 1 shows a simple example of a Java program specified with dynamic frames, purposely underspecified for clarity. It implements a simple array list based on an interface specification, which abstractly specifies a memory footprint that the implementations will be working with through declaring a model variable of a primitive type `\locset`. This footprint is made concrete in the implementing class with the `represents` clause that puts all the concrete locations

```
public interface List {
  //@ instance model \locset footprint;
  //@ accessible footprint : footprint;

  //@ ensures \result == size();
  //@ accessible footprint;
  public /*@ pure @*/ int size();

  //@ ensures size()==\old(size()) + 1;
  //@ assignable footprint;
  public void add(Object o);
}
```

```
public class ArrayList implements List {
  private Object[] contents;
  private int size;
  //@ represents footprint = size,
      ↪ contents, contents[*];

  public int size() { return size; }

  public void add(Object o) {
    contents[size++] = o;
  }
}
```

Lst. 1. Java program annotated with JML*

used by the ArrayList class into the **footprint** model field. This model field is in turn used in two frame specifications. Firstly, the **assignable** clause of method `add` states that these are the locations that may change when `add` is called. Secondly, through the **accessible** clause, the size method specifies that its result only depends on the locations contained in the footprint. Such specifications are commonly used to prove *independence* of pure expressions; If an expression is to be evaluated on two different heaps and it can be proved that the two heaps differ only on locations disjoint with the ones in the **accessible** clause, then it can be concluded that the two expressions are equal. This in turn enables abstract reasoning about expressions.

Note that in dynamic frames there is no implicit framing as found in approaches based on Separation Logic [2] or Implicit Dynamic Frames [4], hence the **assignable** and **accessible** clauses have to be stated explicitly. In particular, we also have to explicitly specify that the footprint is self-framed. However, there is no obligation to use model (or ghost) fields as in our example, it is also possible to state the locations explicitly in the corresponding clauses.

To prove a JML* annotated programs correct in KeY, the specifications are translated to the Java Dynamic Logic (JDL) in which the memory heap is modelled with an explicit program variable using the *theory of arrays* [7]. This program variable, simply called `heap`, is used in translating Java and JML* expressions to JDL and generating suitable proof obligations over this variable to show the correctness of method framing. For example, an object field access `o.f` is typically translated to *select*(`heap`, `o`, `f`) which reads the contents of the `heap` variable at the location mapped by `o` and `f`. Further, a part of the formula that establishes correct framing of a method usually reads:

$$\forall_{o:Object, f:Field}\ (o, f) \in \mathit{frame} \vee o.f@\texttt{heap} = o.f@\texttt{heapAtPre} \qquad (1)$$

where (i) *frame* is the methods frame, either concrete or abstract (in the latter case it can be concretely instantiated when the concrete instance of the object involved is known), (ii) @ is a shorthand notation for the *select* function, and (iii) `heapAtPre` is a snapshot of the heap taken before the method was called

(which is also used to translate JML* `\old` expressions). The actual Java programs are embedded in and treated with Dynamic Logic $[p]\phi$ and $\langle p\rangle\phi$ modalities for partial and total correctness, respectively, where p is a program and ϕ is a correctness formula. Modalities are actually in most part a orthogonal issue to the subject of this paper, however, what is important is that during correctness proofs programs in modalities are evaluated on a statement by statement basis using symbolic execution. During this evaluation the program heap is modified accordingly by updating the `heap` variable. For example, an object field assignment `o.f = v;` results in a modification of the `heap` variable expressed by $store(\texttt{heap}, \texttt{o}, \texttt{f}, \texttt{v})$, which gives a newly modified heap.

JML* Model Methods. Model methods [13] are specification only methods that extend the notion of model fields to fully fledged abstract predicates. When abstract, they do not have any method body, when instantiated (typically in a subclass), they contain a single return statement that gives the predicate its definition. Model methods are strictly pure, which means that they are not allowed to modify any of the heaps. The `accessible` clause attached to the method specifies memory locations that the method at most depends on, this is used to reason about their (in-)equality upon state changes (see Sect. 4). Finally, a model method can have a specification of its own, which essentially serves as a lemma mechanism for predicates to state additional properties. Model methods are particularly suitable to specify linked data structures [14], in this paper we use them to provide modular specification of Java API synchronisation points. In Sect. 5 we give the examples of the use of model methods for this.

3 Dynamic Frames with Permissions

The above described verification methodology works very well in a sequential setting. For the permission-based concurrent setting an appropriate extension is needed. Because the heap is a first class citizen in JDL, the extension is actually rather straightforward. The base heap stores the values of the memory locations that the program operates on, adding a second heap that stores our symbolic permissions in parallel to the values is in essence sufficient. Adding this *permission heap* means adding a second heap variable, which we simply call `permissions`, and extending the verification mechanisms of Java Dynamic Logic from one heap to two heaps. In fact, one can use more than two heaps in JDL easily, as long as the number of heaps is fixed, all mechanisms that work with the single `heap` variable extend naturally to multiple heaps [12].

For example, for proving the framing property (1) above, stemming from the `assignable` clauses, now two quantifiers, over the two heaps, are needed. The core semantics of the permission heap, i.e., granting of access permissions to heap locations, is encoded in the rules for heap location reading and assignment, these rules now operate on our two heaps. As before, the regular heap is read and modified to store the values of corresponding memory locations as briefly explained at the end of the previous section. In addition, each time a memory

location is read from or stored onto this heap, the access right is checked on the permission heap. The permission heap is read at the corresponding location and the resulting permission value is checked accordingly to establish that the current thread has the respective permission.

More concretely, when a location writing statement `o.f = v;` is symbolically executed, the value mapped to `o.f` on the heap is updated with *store*(`heap`, `o`, `f`, `v`) as before, but first the permission p is read from the permission heap with *select*(`permission`, `o`, `f`) and p is checked to be a write permission for the current thread, i.e., as explained in Sect. 2 all slices in p have to belong to the current thread object ct. Reading of locations from the heap is analogous, only the permission is checked to be a read permission instead. In both cases, the permission values stored on the permission heap are only read, but in two cases writing of permission values can also occur. First, when objects are created and permissions are initialised to full permissions for the current thread, i.e., when object `o` is created, for all fields `f` of this object a new full permission is stored on the permission heap with *store*(`permissions`, `o`, `f`, $[[ct]]$). Second, permissions are changed, and hence written on the permission heap, when they are subjected to permission transfers upon synchronisation points, in which case the current permission is first read from the permission heap and a modified one is then written back. For example, when a permission for location `o.f` is transferred from the lock l to the current thread as explained in Sect. 2, the permission heap becomes *store*(`permissions`, `o`, `f`, *transferPerm*(`true`, l, ct, *select*(`permissions`, `o`, `f`))).

For writing suitable user level specifications this extension of JDL to use two heaps has to be lifted to JML*. Following the explicit heap variable approach of JDL, we allow for the same explicit reference of the heap variables in JML* and provide operators to access permissions on the second heap and evaluate them. The following is a short example that illustrates this:

```
//@ requires \writePerm(\perm(this.o)); ensures this.o == p;
//@ assignable<heap> this.o; assignable<permissions> \nothing;
public void set(Object p) { this.o = p; }
```

First, we allow to explicitly state the heap variable that the **assignable** clause refers to (and similarly for the **accessible** clauses). This allows us to decouple the two heaps in the specifications. In the example, we state that the value of the **this**.o field is changed by pointing the **assignable** clause to the main `heap` variable, however, on the `permissions` heap the frame is empty, because the permission to the field `o` does not change when the `set` method is executed, it is only read to check the (write) access to the field **this**.o.

Furthermore, we provide operators to access the permission heap in the pre- and postconditions, and to evaluate the permission values. In the example, we use the **\perm** operator to access the **this**.o location on the permission heap. Thus, **\perm** is somewhat analogous to the **\old** operator, which redirects access from the current heap to the heap before the method was called. In our specifications the combination of the two operators is also possible, **\old(\perm(**·**))** reads the value of the permission before the method was called. Then, **\writePerm** is a predicate that abstracts checking the permission to be a write permission for the

current thread, an analogous predicate for checking the permission to be a read one is called **\readPerm**. Finally, operations **\transferPerm** and **\returnPerm** to modify permissions upon synchronisation points are also available. Typically, when this happens the corresponding location is also listed in the **assignable** clause for the permission heap, concrete examples of that are provided in Sect. 5.

4 Proof Obligations for Self-Framing

The above is sufficient to relate heap locations with their permissions and to perform basic permission-aware reasoning in the dynamic frames approach, i.e., permissions can be specified and are checked when locations are accessed in the verified program. However, permissions have also consequences for the specifications themselves, in terms of which specifications are actually sound and how they should be applied in modular reasoning. Namely, specifications themselves have to be self-framed w.r.t. permissions, i.e., specifications are only allowed to reference heap locations they have at least a read permission to. Locations with no permission can be modified by other threads that potentially hold a complete write permission, hence nothing can be said about them. The mechanism of applying method specifications in modular reasoning is also affected, i.e., when a permission to some memory location is lost, so should be the information about its current value on the heap. Unlike in Separation Logic (-like) approaches [4], in dynamic frames self-framing of expressions (even without permissions) is not given and has to be shown explicitly. In particular, explicit **assignable** clauses are required (not necessary in SL) and proof obligations have to be generated, like (1) above, to prove them correct.

In permission-based reasoning each thread is verified (on a per-method basis) in isolation under the assumption that it is the currently executing thread. The reasoning itself is very similar to the one for sequential programs, with the addition that if permission annotations are verified to be consistent for each thread then the threads are guaranteed to be non-interfering. In such a verification context, it is sufficient to abstract the permissions to be simply read, write, or no permission for the current thread, also when talking about soundness of specifications themselves. Hence, the actual permission system (symbolic or fractions-based) is irrelevant. What is relevant is how the memory and permissions are referred to in the logic, in our case through explicit heap variables.

Examples of Sound and Unsound Specifications. Suppose we have the following very simple method specified with JML*:

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable<heap> this.f; assignable<permissions> \nothing;
void setF(int v) { this.f = v; }
```

This specification is sound w.r.t. permission annotations. The preconditions establishes at least a read permission (here a full write one) for **this.f**, the permission is not changed by this method, hence the postcondition can freely specify the value of **this.f**. However, if we change the scenario slightly to become:

```
//@ requires \writePerm(\perm(this.f)); ensures this.f == v;
//@ assignable<heap> this.f; assignable<permissions> this.f;
void setFandUnlock(int v) { this.f = v; l.unlock(); }
```

then referencing `this.f` in the postcondition is no longer sound. Knowing that the `unlock` method modifies the permission to `this.f` we also have to put this location in the assignable permissions of `setFandUnlock` and consequently we cannot establish any permission to `this.f` in the postcondition. To fix this, if the `unlock` method leaves a read permission with the current thread then we can specify it:

```
//@ ensures \readPerm(\perm(this.f)) && this.f == v;
```

Or, if no permission to `this.f` is left after `unlock` the postcondition over the value of `this.f` has to be removed altogether, and the specification becomes:

```
//@ requires \writePerm(\perm(this.f));
//@ assignable<heap> this.f; assignable<permissions> this.f;
```

On top of that, when client code that calls `setAandUnlock` is verified, it is mandatory to loose all information about `this.f` after the call.[4] However, it is sound to leave this location in the assignable clause for the base heap, and in fact necessary. The presence of this location in the assignable clause actually causes erasure of information about this location from the current verification context upon a `setFandUnlock` method call, because no postcondition can be specified that would give the new value of this location. In other verification systems the mechanism of erasing information is typically called *havocing* [16], in the Java Dynamic Logic it is called *anonymisation*, and incidentally it also gives us the base for showing that specifications are self-framed w.r.t. permissions in JDL. We show how this is done for the preconditions in their basic form. With small technical alternations, the method scales correspondingly to other specification constructs, like postconditions, `measured_by` termination clauses, or model methods with their specifications (see Sect. 5).

Anonymisation. Locations on the heap are anonymised with the $anon(\texttt{heap1}, locs, \texttt{heap2})$ function that gives a new heap with the locations not appearing in *locs* copied from `heap1` and otherwise the locations are copied from `heap2`. For example, to anonymise locations `o.f` and `o.g` on the base heap one typically creates a new heap with $anon(\texttt{heap}, \{(o, f), (o, g)\}, \texttt{anonHeap})$, where `anonHeap` is a fresh unspecified heap. Such an operation is applied to the current heap during modular verification, when a method call is dispatched using its specification, in which case *locs* are the locations defined in the assignable clause.

This function can also be used in an *inverse* way, i.e., all locations outside of a certain set *locs* can be anonymised with $anon(\texttt{heap}, allLocs \setminus locs, \texttt{anonHeap})$.

[4] This problem is common in permission-based approaches and makes reasoning about functional behaviour of concurrent programs difficult. Solutions exist to enable to keep certain information about temporarily inaccessible locations [15], however, they are beyond the scope of this paper, here we concentrate on the basic soundness of dynamic frames enriched with permissions.

Now all locations in *locs* keep their values in the resulting heap w.r.t. the input `heap`, while all other locations are left undefined. This mechanism is commonly used in JDL to show data independence of expressions, in particular, to prove the **`accessible`** clauses of read-only methods. Suppose a method `getVal()` is specified with an **`accessible`** clause to only depend on `o.f`. To prove that this is indeed so, the following JDL proof obligation has to be discharged:

$$\texttt{getVal}() = \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \setminus \{(o, f)\}, \texttt{anonHeap})\}\texttt{getVal}() \quad (2)$$

The meaning of the right hand side of this equality is that `getVal` should be evaluated in a state with modified heap where all locations not in the set of locations *locs* are anonymised. Proving this equality means that changing the values of locations outside of the set *locs cannot* influence the valuation of `getVal` and indeed it depends at most on the values of locations in *locs*. In KeY, such a proof obligation is generated by default for every *state observing symbol* [7,13] with an accessible clause, in particular, for all read-only methods.

Proof Obligations for Self-Framing. To prove correct framing of specifications w.r.t. permissions a similar mechanism is used. The expression is simply our specification, e.g., a complete expression `pre` representing the method's precondition. However, there is no **`accessible`** clause to give the set of dependency locations of the expression, so we have to "extract" it from the expression instead. To this end, we introduce a fresh location set logic variable *readLocs* and we indirectly specify which locations it contains. Namely, locations that we can show at least a read permission for under the assumption that the expression `pre` itself holds. The complete proof obligation to show self-framing then reads:

$$\begin{aligned}\texttt{pre} \wedge \forall_{o:\mathit{Object}, f:\mathit{Field}}\ &\mathit{readPerm}(o.f@\texttt{permissions}) \rightarrow (o, f) \in \mathit{readLocs} \\ &\rightarrow \texttt{pre} = \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \setminus \mathit{readLocs}, \texttt{anonHeap})\}\texttt{pre}\end{aligned} \quad (3)$$

For a postcondition this construction has small additional complexity, which stems from the fact that the read permission might be specified in the postcondition itself, or, if the permission is not modified by the method, it might be kept from the precondition. To account for this, additional base and permission heap operations are required to "find" the permission in the method specification. Due to space restrictions, we do not quote the formula here, however, the main principle is exactly the same as in (3).

Proof obligation (3) shows that every location referenced on the base heap is accompanied by at least a read access on the permission heap. As explained above, we also have to show that all locations that the method may loose permissions to, i.e., locations for which at least a read permission cannot be established, are included in the assignable clause for the base heap. For this, the following formula has to be proved:

$$\begin{aligned}\texttt{post} \rightarrow \forall_{o:\mathit{Object}, f:\mathit{Field}}\ &(o, f) \in \mathit{permMod} \\ &\rightarrow \mathit{readPerm}(o.f@\texttt{permissions}) \vee (o, f) \in \mathit{heapMod}\end{aligned} \quad (4)$$

where *permMod* and *heapMod* are locations listed in the assignable clauses for the permission and base heap, respectively. Note, that (a) it is not necessary to

add locations to the base assignable clause for which the method did not have permissions for in the first place, only for the ones that are lost, (b) locations for which there is no permission can nevertheless remain in the base assignable clause without breaking the soundness. Locations without an initial permission (point (a)) cannot be used in method's specification or code. Hence, any information (or lack thereof) about such locations can remain in the verification context. For point (b) it is a simple case of over-approximation where the verification context will loose more information about locations than necessary.

Discussion. Enforcing the lost permission locations to be explicit in the assignable clause of the base heap puts unnecessary burden on the specifier. In (4) we name these locations directly and simply check that they are in the assignable clause. What is equivalently sound, but more practical, is to instead add these locations dynamically to the anonymisation set when the method contract is applied during a proof, in which case (4) does not have to be proved. In fact, this approach can be pushed even more to completely deduce assignable (and accessible) clauses from permission specifications. This exactly is the methodology used in Implicit Dynamic Frames (IDF) [4], where frames are inferred from permissions. A specified read permission implies that the corresponding location is in the accessible clause, and a specified write location puts the location in the assignable clause. The resulting reasoning system has the look-and-feel of permission-based Separation Logic [17,18]. It is also possible to achieve full IDF-style framing in our framework, however, we have chosen not to do so (yet) for two reasons. First, our explicit approach enables high specification and verification precision, in particular, explicit framing avoids frame over-approximation. For example, a write permission in the specification does not necessarily imply that the method assigns the corresponding location, in fact, it can still be a read-only method, in which case it can be used in specifications. For us, the query or mutator status of a method is indicated by the accessible, resp. assignable, clause independent of the permissions. Second, keeping the base and permission heaps explicit with separate framing enables decoupling permission-based reasoning from the classical sequential dynamic frames one while using the same specifications for both. To change from permission-based to sequential reasoning the permission heap is simply omitted during proof obligation generation, and our implementation in KeY provides a simple mechanism to do that.

5 Modular Specifications for Synchronisers

The most intricate part of permission-based reasoning are permission transfers that occur upon synchronisation points between threads, e.g., acquiring and realising of locks, thread forking and joining, etc. In approaches based on quantitative permissions the modelling of the synchronisation involve the use of so-called resource (or monitor) invariants [19]. Such invariant is essentially a quantitative amount of resource permissions that is passed to and from the current thread upon synchronisation. For example, in Chalice [3] every object can be used as a

lock (as in Java), and when an object is locked all permissions from the object's resource invariant are transferred to the currently running thread. Using Java and JML syntax, to use a shared counter one would specify and use it as follows:

```
class Counter { int val; /*@ monitor Perm(val, 1); @*/ }
class Client { void inc(Counter c) { synchronized(c) { c.val++; } } }
```

Here, in the scope of the `synchronized` block the method `increase` temporarily holds the permission to `c.val` specified in the monitor of the `Counter` class.

To make this method more modular and flexible one typically uses predicates to embed a set of permissions in one formula and use it as a single resource invariant. This way, concrete permissions are hidden behind the predicate and are only unfolded when required during verification. Such a predicate can be also passed between different classes. In particular, this is used when complex API synchronisation methods are considered [20]. API based synchronisation brings the challenge that several different use scenarios are possible for each mechanism (for locks, e.g., there are read locks, write locks, reentrant locks, etc.) and that they cannot be considered as primitive language constructs with a fixed notion of a resource invariant as above. Instead, their semantics is given with a generic API specification, which is external to the concrete use case. By passing a suitably defined resource predicate one makes such a generic specification concrete [21].

However, we cannot use resource invariants in our approach in the same way, because we specify permission *transformations* instead of permission *amounts*. Instead, we use a two stage mechanism. First, similarly to resource invariants, we give a formula that describes the state of permissions for the given synchroniser. But here, this specification contains a compound description of the symbolic permissions for both the state when the synchroniser is *engaged* and when it is not, both of which are described with the reference to the current thread and the synchroniser itself. Second, we make a connection between this state description and the methods that change the state, i.e., the actual synchronisation calls, like `lock` and `unlock`. We explain our method based on a simple example of a write lock used to protect a single counter variable, as above.

The abbreviated listing of our lock specification and sample client is given in Lst. 2. We compacted it for presentation, in particular we skipped all but one framing specifications to concentrate on the modular specification of the lock behaviour w.r.t. permissions. The full example that can be loaded and proved with KeY is available on-line [22]. The specification of a lock is delegated to a separate interface `LockSpec` that serves as a template and provides signatures of all predicates that clients have to instantiate. The lock itself, specified in the `Lock` interface, "receives" this specification through a binding of its ghost variable `spec` (l. 14). Then, the client code in the `Counter` class instantiates the specification and passes it to the lock object by specifying the binding in the invariant (l. 21).[5]

[5] This is not the most elegant way of passing specifications (predicates) around classes in JML*, however, a working one and currently the only one that the KeY implementation allows. In the future we plan to provide proper ghost and model parameters to classes and methods in the style of [21].

```
public class LockSpec {
  //@ model \locset fpPerm();
  //@ accessible<permissions> fpPerm(); model boolean state(boolean locked);
  //@ model boolean status(boolean locked);
  //@ model two_state boolean lockTr();
  //@ model two_state boolean unlockTr();
  /*@ ensures \result;
      model final two_state boolean consistent() { return
        (\old(state(false)) && \old(status(false)) && lockTr() ==>
          (state(true) && status(true))) &&
        (\old(state(true)) && \old(status(true)) && unlockTr() ==>
          (state(false) && status(false))); } @*/ }

public interface Lock { //@ public instance ghost LockSpec spec;
  //@ requires spec.status(false); ensures spec.status(true) && spec.lockTr();
  public void lock();
  //@ requires spec.status(true); ensures spec.status(false) && spec.unlockTr();
  public void unlock(); }

public class Counter extends LockSpec { private int val;
  private Lock lock; //@ invariant lock.spec == this && ...;
  /*@ model boolean state(boolean locked) { return \perm(val) ==
      locked ? [[ \ct, lock ]] : [[ lock ]]; } @*/
  /*@ model boolean status(boolean locked) { return locked ?
      \writePerm(\perm(val)) : !\readPerm(\perm(val)); } @*/
  /*@ model two_state boolean lockTr() { return \perm(val) ==
      \transferPerm(false, lock, \ct, \old(\perm(val))); } @*/
  /*@ model two_state boolean unlockTr() { return \perm(val) ==
      \returnPerm(\ct, lock, \old(\perm(val))); } @*/

  //@ requires status(false); ensures status(false);
  public void inc() { lock.lock(); val++; lock.unlock(); } }
```

Lst. 2. Modular specification for a lock in JML*.

To enable modularity, our predicates are specified with JML* model methods [13] briefly introduced in Sect. 2. The `state` predicate (l. 3) describes the state of the permissions in the locked and unlocked state. In the client (ls. 22–23) the lock is specified to protect the `val` field of the `Counter` object. In the unlocked state the permission to `val` is a single slice belonging to the lock – `[[ lock ]]`. When locked, the permission is also a single slice, but temporarily belonging to the current thread that acquired the lock and owing the slice to the lock – `[[ \ct, lock ]]`. The `status` predicate (l. 4) serves two purposes. First, it represents the binary state of actually holding the lock at any given point. Second, it provides an abstracted view of the permission to the protected resource, here the `val` field. By knowing the status the client can also deduce the actual access permission to the resource without having to evaluate the concrete sym-

bolic permission expression kept in the lock `state`. Our client code (ls. 24–25) specifies that in the locked state it holds a complete write permission to `val`, while in the unlocked state it holds no permission at all. Note that in this case these two are not the binary opposites of each other, hence the need for the `locked` parameter in `status`. The predicates `lockTr` and `unlockTr` (ls. 5 and 6) describe the permission change upon lock acquiring and releasing, respectively. They are **`two_state`** predicates, because they describe the state of permissions before and after the corresponding lock calls. Such two-state predicates can be used in an appropriate context, i.e., the method postcondition (e.g., l. 15). Upon locking (l. 27) all permission slices (the first **`false`** parameter denotes this) to `val` are transferred from the lock to the currently running thread (denoted with **`\ct`**). Upon unlocking (l. 29) all slices for `val` are returned from the current thread to the lock object.

Finally, the `consistent` predicate (ls. 7–12) binds the specification structure together. It establishes the relationship between the concrete and abstract view of permission for the lock, and that the two transfers correctly change the state of the lock. This predicate is defined directly in `LockSpec` – all clients instantiating this specification have to show this predicate to hold (its postcondition in l. 7 states so) to prove that their concrete lock specifications are consistent.

Following the same methodology we can develop similar generic specifications for other synchronisation triggering methods of the Java API, and in particular modular specification for asynchronous method calls invoked through the `start()` and `join()` methods of the `Thread` class [23]. In each such case a generic specification that would cover the typical usage scenarios is possible. Our `Lock` specification is not fully generic in this respect, in particular it does not cover Java re-entrant locks, but it can be extended to resemble the ones we developed before for Separation Logic [21] that cover all kinds of Java lock flavours. However, there will always be scenarios that would not fall within such a generic scheme. In particular, our version [11] of the motivating example from [24] that uses a primitive lock combined with a counter variable to effectively implement a semaphore-like read-write lock cannot be put in the frame of our `Lock` specification presented here without further extensions of this specification. Hence, we did not construct a complete generic specification solution for all API-based synchronisers, we only showed a methodology with a number of possible applications.

6 Conclusions

We presented an approach to the verification of concurrent Java programs based on Dynamic Frames extended with permissions. In particular, we showed how to treat the self-framing of specifications in Java Dynamic Logic and how to use JML* model methods to provide modular specifications for Java API synchronisation points.

Implementation Status. Our symbolic permission framework described in Sect. 2 is implemented in the current development version of the KeY verifier, and so is

the extension from Sect. 3 that incorporates permissions into the JML* dynamic frames. Furthermore, model methods that we used for modular specification in Sect. 5 are also implemented in KeY [13], and in fact did not require any particular extensions to work with permissions, apart from accounting for one additional heap. What is not yet implemented, is the generation of the additional proof obligations and checks for self-framing w.r.t. permissions described in Sect. 4. This is work in progress and we expect this to be finished soon.

Further Examples. The current state of the implementation allows for all the examples that we discussed or referred to in this paper to be verified. Technically speaking, however, the tool is not yet fully sound, in the sense that possibly unsound specifications can be admitted by KeY. Nevertheless, we developed several more non-trivial examples and verified them with KeY, while checking specification framing by hand. In particular, the KeY distribution contains modularly specified and fully verifiable example of a multi-threaded plotter that we developed earlier using Separation Logic [5]. In this example four different threads manipulate two shared buffers to process and "draw" some input data passing the permissions to these buffers in a non-trivial way. Few other examples are available in the KeY distribution, in particular fully specified and verified read-write lock example from [24] we mentioned above, and the examples from this and earlier paper on symbolic permissions are available on-line [22].

Related Work. To the best of our knowledge, our method so far is the only one that uses Dynamic Frames in the explicit form with permissions [1] and in this paper we have shown the necessary extensions and modifications to the Java Dynamic Logic (JDL) used in the KeY verifier to build a fully functional verification system for this combination. The existing approaches to (fractional) permission-based reasoning with functional tools are based on Separation Logic (SL) [2] or Implicit Dynamic Frames (IDF) [4], e.g., our own VerCors toolset [5,18], VeriFast [25], Silicon [26], or Chalice [3].

Compared to these existing approaches, ours is based on symbolic permissions we developed earlier to allow for more flexibility in permission flow specifications. Furthermore, we are more explicit in terms of exhibiting the underlying logic mechanism to the specifier, e.g., by allowing to refer to heaps directly in explicit JML* frame specifications. In comparison, e.g., in IDF memory and permission frames are calculated on the fly from pre- and postconditions. We stated two reasons for considering our explicit approach advantageous, namely very precise specifications and reasoning, as well as the possibility to decouple reasoning about functional and permission properties.

Future Work. Approaches based on SL and IDF have been shown to be practically equivalent [4]. On the verification end, the problems are translated to FOL formulas to be proved by an appropriate verifier, e.g., an SMT solver. In this respect our method is no different, symbolic execution of permission annotated program leads to pure FOL problems which are then discharged with FOL reasoning. However, our specification methodology is more explicit and closely

related to the actual reasoning logic, in our case Java Dynamic Logic implemented in the KeY verifier. In this respect, for future work we also consider a translation from permission-based SL to JDL with permissions making it an intermediate verification language, similarly to Silicon [26]. This translation would be a mixture of ideas presented in this paper and in [27] where a bridge between SL and Dafny – also DF-based – is described. Otherwise, we are finishing the implementation and working on more examples for our approach.

References

1. Boyland, J.: Checking interference with fractional permissions. In Cousot, R., ed.: Static Analysis Symposium. LNCS 2694, Springer (2003) 55–72
2. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science, IEEE Computer Society (2002) 55–74
3. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In Aldini, A., Barthe, G., Gorrieri, R., eds.: Foundations of Security Analysis and Design. Springer (2009) 195–222
4. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In Barthe, G., ed.: European Symposium on Programming. Volume 6602 of LNCS., Springer (2011) 439–458
5. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I., eds.: 14th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Executable Software Models. Volume 8483 of LNCS., Springer (2014) 172–216
6. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: Formal Methods. Volume 4085 of LNCS., Springer (2006)
7. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic. In Beckert, B., Marché, C., eds.: Formal Verification of Object-Oriented Software Conference. LNCS 6528, Springer (2011) 138–152
8. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In Giannakopoulou, D., Kroening, D., eds.: Verified Software: Theories, Tools, and Experiments (VSTTE). Volume 8471 of LNCS., Springer (2014) 1–17
9. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT **31**(3) (March 2006) 1–38
10. Blom, S., Huisman, M.: The VerCors Tool for verification of concurrent programs. In Jones, C., Pihlajasaari, P., Sun, J., eds.: FM 2014. Volume 8442 of LNCS., Springer (2014) 127–131
11. Huisman, M., Mostowski, W.: A symbolic approach to permission accounting for concurrent reasoning. In: 14th International Symposium on Parallel and Distributed Computing, IEEE Computer Society (2015) To appear.
12. Mostowski, W.: A case study in formal verification using multiple explicit heaps. In Beyer, D., Boreale, M., eds.: IFIP Joint International Conference on Formal Techniques for Distributed Systems. LNCS 7892, Springer (2013) 20–34

13. Mostowski, W., Ulbrich, M.: Dynamic dispatch for method contracts through abstract predicates. In: 15th International Conference on MODULARITY, ACM (2015) 109–116
14. Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. Software Tools for Technology Transfer (2013) On-line first.
15. Zaharieva-Stojanovski, M., Huisman, M., Blom, S.: Verifying functional behaviour of concurrent programs. In Pearce, D., ed.: FTfJP'14: Proceedings of 16th Workshop on Formal Techniques for Java-like Programs, Uppsala, Sweden, ACM (2014)
16. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: LPAR (Dakar). Volume 6355 of LNCS., Springer (2010) 348–370
17. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In Palsberg, J., Abadi, M., eds.: Principles of Programming Languages, ACM (2005) 259–270
18. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. Logical Methods in Computer Science **11** (2015)
19. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science **375**(1–3) (2007) 271–307
20. Blom, S., Huisman, M., Kiniry, J.: How do developers use APIs? A case study in concurrency. In: International Conference on Engineering of Complex Computer Systems, IEEE Computer Society (2013) 212–221
21. Amighi, A., Blom, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Formal specifications for Java's synchronisation classes. In Lafuente, A.L., Tuosto, E., eds.: Conference on Parallel, Distributed, and Network-Based Processing, IEEE Computer Society (2014) 725–733
22. Symbolic Permissions: `http://www.ewi.utwente.nl/~mostowskiwi/permissions/`.
23. Haack, C., Hurlin, C.: Separation logic contracts for a Java-like language with fork/join. In Meseguer, J., Rosu, G., eds.: Algebraic Methodology and Software Technology. LNCS 5140, Springer (2008) 199–215
24. Boyland, J., Müller, P., Schwerhoff, M., Summers, A.J.: Constraint semantics for abstract read permissions. In: Formal Techniques for Java-like Programs (FTfJP), ACM (2014)
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods. LNCS 6617, Springer (2011) 41–55
26. Juhasz, U., Kassios, I.T., Müller, P., Novacek, M., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zürich (2014)
27. Bao, Y., Leavens, G.T., Ernst, G.: Translating separation logic into dynamic frames using fine-grained region logic. Technical Report CS-TR-13-02a, Computer Science, University of Central Florida (March 2014)