From Explicit to Implicit Dynamic Frames in Concurrent Reasoning for Java (Extended Abstract)*

Wojciech Mostowski

Center for Research on Embedded Systems, Halmstad University, Sweden
wojciech.mostowski@hh.se

1 Introduction

In [9] we presented an approach to permission-based reasoning about concurrent Java programs in the context of the interactive program verifier KeY [1] which is based on Dynamic Logic and explicit dynamic frames [6, 13]. We argued for the *explicit* approach advocating the modular use (w.r.t. sequential vs. concurrent) and overall preciseness. It was noted, however, that changing our specification and verification approach to an established one of *implicit dynamic frames* (IDF) [11] should be also possible. In consequence, this would allow us to translate Separation Logic (SL) specifications [12, 2] into our framework to provide a powerful interactive theorem prover support for SL-like formalisms. In this context, we present some of the challenges associated with transition to implicit frames in KeY and possible solutions.

2 Permission-based Reasoning with Explicit Frames

```
public class ArrayList {
 Object[] cnt; int s;
 //@ model \locset fp = s, cnt, cnt[*];

 //@ ensures \result == s;
 //@ accessible fp;
 /*@ pure @*/ int size() { return s; }

 //@ ensures size()==\old(size()) + 1;
 //@ assignable fp;
 void add(Object o) { cnt[s++] = o; } }
```

Figure 1: A simple array list specified (incompletely) with explicit frames.

As originally proposed in [6], the essence of specifying and reasoning about programs using explicit dynamic frames is the introduction of locations sets into the specification language as first class citizens and allowing them to be embedded within abstract predicates. In the KeY verification system which uses specially crafted Dynamic Logic for Java, this gives rise to JML* specification language that introduces locations sets to the classic JML syntax [7] and the Dynamic Logic is equipped with means to reason about them. A classical, albeit minimalistic example of a Java program annotated with JML* specification is shown in Fig. 1. The essential parts of this specification are frames, here expressed using abstraction through the JML model field `fp`. A read frame is specified with the **accessible** clause, and a write frame is specified with the **assignable** clause. As all other specifications, both clauses are effectively lemmas. Consequently, one is obliged to show that the lemma holds by showing that the corresponding method adheres to the limits of the frame, and then the lemma can be used to support proofs that involve the use of the method. For mutator methods the **assignable** clause is used to anonymise/havoc the corresponding locations of the program when the method is modularly applied in the proof to discharge a method call, while the **accessible** clause is used to establish the equivalence of two expressions under two different states when the expression is known,

*This work is supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project.

according to the read frame, not to depend on the locations changed between the two states. In our example, size() is guaranteed to always evaluate to the same value in all states in which s is not changed. In the Dynamic Logic for Java used by the KeY verifier suitable mechanisms are devised to both show the framing lemmas to hold and to use them in the proofs [13].

To verify concurrent behaviour, a convenient approach is to annotate programs with permission expressions, typically based on fractions [4], to guard every memory location access. A full permission grants a write access, while a partial permission grants only a read access. The construction of the verification method and the permission manipulation system guarantee that verified programs are data-race free. Typically, an SL-like framework is used for verification, among them IDF method of the Chalice verifier [8].

```
public class ArrayList {
  Object[] cnt; int s;
  //@ model \locset fp = s, cnt, cnt[*];

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ accessible<heap> fp;
  //@ accessible<perms> \nothing;
  /*@ pure @*/ int size() { return s; }

  //@ requires \readPerm(\perm(cnt));
  //@ requires \writePerm(\perm(s));
  //@ requires \writePerm(\perm(cnt[s]));
  //@ ensures size() == \old(size()) + 1;
  //@ assignable<heap> fp;
  //@ assignable<perms> \nothing;
  void add(Object o) { cnt[s++] = o; } }
```

Figure 2: Explicit frames specification with permissions.

To enable permission-based reasoning in KeY without going too far away from its existing explicit frames framework, we add a second heap to track permissions and extend framing to that heap. In Fig. 2 a specification extended this way is shown. Most notably, heaps are now named explicitly in the specification (heap and perms) and are both given separate framing specifications. In most cases permission frames are actually empty, but not for methods and programming constructs that transfer permissions, e.g., mutual exclusion or sharing locks used to access another thread's data. Additionally, in KeY we opted for developing a symbolic permission framework[1] [5] as an alternative to fractional permissions. The verification logic of KeY extends naturally to deal with the extra permission heap and methods are provided to enable fully modular and abstract specifications for the whole framework [9].

3 Transformation to Implicit Dynamic Frames

In [11] it has been shown that IDF and SL are essentially equivalent w.r.t. expressiveness of specifications. Hence, here we concentrate on the task of treating IDF-style specifications in our framework.

The key observation in our explicit methodology is that because of the the use of permission annotations in specifications, particularly in preconditions, and verifying the code against these annotations, the **assignable** and **accessible** clauses are in essence obsolete. That is, the precondition provides the complete framing information for a given method – a read permission indicates that a method might be accessing a heap location (previously indicated in the **accessible** clause) and the write permission indicates that a method might be modifying a heap location (previously indicated in the **assignable** clause). More importantly, no heap location access (read or write) in the code would be allowed without a corresponding permission annotation, hence the permission annotations provide complete and sound framing specification. Dropping the frame specifications for the regular heap means two things. First, the frames do not have to be shown to hold in a separate proof obligation, the checking of the program w.r.t. the specified access

[1]Here the details of this permission system are not really relevant, the important part is that we can specify a permission to be a read or write permission, and that permissions can be transferred.

rights establishes the adherence to frame specification given by permission annotations. Second, it becomes a bit more difficult to apply modular method dispatch based on framing information, as the frames are not specified directly. The solution to this is to build the frame dynamically on demand using almost the same mechanism as we have presented in [9] to show self-framing of specifications w.r.t. permissions. For the assignable frame, a formula of the following shape is constructed: $\texttt{pre} \wedge \forall_{o:Object,f:Field}\ (\mathit{writePerm}(o.f@\texttt{perms}) \rightarrow (o,f) \in \mathit{writeFrame})$, where *writeFrame* is a fresh function symbol that collects all the heap locations for which we can show a write permission assuming that the method's precondition `pre` holds. The *writeFrame* can then be used in modular method dispatch.

However, the frames to the permission heap `perms` cannot be simply dropped in the same way without consequences for the specification method and patterns. The main reason is that the presence of a given permission in the specification does not, in general, imply that the permission heap `perms` is accessed or modified within the body of the method. In fact, the most common case is that a permission is present in the specification to allow a corresponding access on the regular heap, while the permission heap itself stays unchanged. Yet, assuming a frame for the `perms` heap as described above for the regular heap is the minimal sound approach if the frame is not to be stated explicitly. The resulting over-approximation of the permission frame can be mitigated on the specification level by specifying for each permission whether it is changed (and how) or not. In the latter case we propose to use a new keyword **`\samePerm`**. Figure 3 shows the specification of the program in Fig. 2 modified to suit the implicit frame specification approach following the ideas just described. The need to specify all permissions in postconditions to enable precise reasoning is not surprising – all SL-like specifications are required to do so.

```
public class ArrayList {
  Object[] cnt; int s;

  //@ requires \readPerm(\perm(s));
  //@ ensures \result == s;
  //@ ensures \samePerm(\perm(s));
  /*@ pure @*/ int size() { return s; }

  //@ requires \readPerm(\perm(cnt));
  //@ requires \writePerm(\perm(s));
  //@ requires \writePerm(\perm(cnt[s]));
  //@ ensures size() == \old(size()) + 1;
  //@ ensures \samePerm(\perm(s));
  //@ ensures \samePerm(\perm(cnt));
  //@ ensures \samePerm(\perm(cnt[s]));
  void add(Object o) { cnt[s++] = o; } }
```

Figure 3: IDF specification in JML*.

The implicit framing brings another small over-approximation issue. A write permission in the method's precondition implicates a corresponding location to be in the assignable clause of the method, while in reality the method might be only reading the location. Methods under-specified like this cannot be considered pure, despite being so. To check that this situation does not occur, an additional proof obligation in Java Dynamic Logic could be devised.

4 Conclusion

We presented the preliminary ideas for supporting IDF-style specifications in JML* and the KeY program verifier for Java. As the explicit frames approach is deeply embedded in the KeY philosophy, the implementation considerations in KeY for IDF might bring further challenges. Moreover, we have not covered here interactions with JML* *model methods* that we use for very flexible abstract and modular specifications in the context of inheritance [10]. Finally, despite the mentioned equivalence of IDF and SL, the ideas that we have discussed here are not sufficient for full and proper translation of SL specifications to JML* and KeY logic. In particular, to fully support SL we also have to deal the separating conjunction operator `*` and the magic-wand operator `-*`, the latter being known for requiring non-trivial encodings [3].

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 8471 of *LNCS*, pages 1–17. Springer, 2014.

[2] Afshin Amighi, Christian Haack, Marieke Huisman, and Clément Hurlin. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11, 2015.

[3] Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. *International Journal on Software Tools for Technology Transfer*, pages 1–25, 2015.

[4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[5] Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *14th International Symposium on Parallel and Distributed Computing (ISPDC 2015)*, pages 165–174. IEEE Computer Society, 2015.

[6] Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23:267–288, 2011.

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT*, 31(3):1–38, March 2006.

[8] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 195–222. Springer, 2009.

[9] Wojciech Mostowski. Dynamic frames based verification method for concurrent Java programs. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, LNCS. Springer, 2015. To appear.

[10] Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. In *15th International Conference on MODULARITY*, pages 109–116. ACM, 2015.

[11] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *European Symposium on Programming*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.

[12] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[13] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software Conference*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.