Subobject Transactional Memory

Marko Dooren, Dave Clarke

Subobject Transactional Memory

Marko van Dooren and Dave Clarke

IBBT-DistriNet, KU Leuven, Leuven, Belgium.
{firstname.lastname}@cs.kuleuven.be

Abstract. Concurrent object-oriented programs are hard to write because of the frequent use of state in objects. In a concurrent program, this state must be protected against race-conditions and deadlocks, which costs a lot of effort and is error-prone. Software transactional memory is a mechanism for concurrency control that is similar to mechanisms used in databases. The programmer does not deal with low-level locks, but instead uses transaction demarcation to protect shared memory.

We show that in a statically typed subobject-oriented programming language, a transactional program requires less effort than writing a regular object-oriented programming. In addition, we show how transactionality can be added to existing classes without performing code transformations or using a meta-object protocol.

1 Introduction

With the rise of multi-core processors, there is growing demand for multi-threaded applications. But to ensure proper functioning of the program, data that is shared between threads must be guarded to avoid problems such as lost updates and dirty reads. With lock-based approaches, the programmer must place locks in the appropriate places in the code to prevent race conditions. Placing all locks correctly, however, is very hard and requires a lot of effort. Software transactional memory [17] (STM) is popular mechanism to support transactional behavior of a program that avoids the problems associated with lock-based approaches. A programmer must only demarcate transactions, and the STM ensures that the code in a transaction is executed atomically and isolated.

STM implementations can be divided into two categories: language implementations and library implementations. Language implementations add dedicated language constructs to provide STM functionality and/or modify the language run-time to support transactional semantics [2, 5, 12, 13, 16] The advantages of language implementations are that they allow low-level optimizations and impose a minimal syntactic overhead on the programmer. The disadvantages are that using non-standard language implementation is usually not an option in an industrial setting, and that the implementation of a customized transaction mechanism usually is difficult.

Library STM implementations in static languages [6, 9–11] provide an API to use the STM. The advantage of this approach is that neither the language nor the run-time must be adapted. The disadvantage is that the programmer must use reified memory locations instead of the variables that are normally used, which results in more boilerplate code. In addition, existing classes cannot be made transactional.

Library STM implementations in dynamic languages work by dynamically rewriting the program [15], or modifying the language semantics via a meta-object protocol [7]. In these approaches, the language semantics are changed without using modified language run-times or external code generation tools. Therefore, the programming overhead is limited and the standard language run-time can be used. The disadvantage of these approaches is that the required language features are not available in static programming languages.

The contribution of this paper is to show that an STM library in a statically typed subobject-oriented programming language [19, 18] can offer the same ease of use as dynamic STM libraries and dedicated language implementations. We show that a transactional subobject-oriented program contains even less boilerplate code than a non-transactional object-oriented program. In addition, transactional behavior can be added to existing non-transactional classes. We present a proof-of-concept implementation of a multi-version concurrency control mechanism.

Overview

Section 2 gives a short introduction to subobject-oriented programming. Section 3 discusses how subobjects can be used to write transactional applications. Section 4 present our proof-of-concept implementation. Section 5 discusses related work, and Section 6 concludes.

2 A Subobject-Oriented Approach

The focus of this paper is on improving the ease of use of an STM library in a static language. Our proof-of-concept implementation is not optimized for performance or memory footprint.

The context of our approach is a development process that uses a statically typed programming language. We do not allow modifications to the compiler or the language run-time for two reasons. First, such modifications are typically not allowed in an industrial setting. Second, such modifications make it harder to use develop transaction mechanisms that are better suited for the read/write pattern of a particular application.

In this paper we use *subobject-oriented programming* to make an application transactional. Subobject-oriented programming, which was developed by the first author [19, 18], augments object-oriented programming with a mechanism to compose classes from other classes. While the composition mechanism is relatively recent, and thus not supported in mainstream programming languages, it is important to note that it is a *general purpose* language construct. As such, we treat our prototype language JLo as a standard programming language. The remainder of this section gives an introduction to subobject-oriented programming.

2.1 An Introduction to Subobject-Oriented Programming

Subobject-oriented programming augments object-oriented programming with subobjects. A subobject can be seen as a combination of inheritance and delegation, and

allows a developer to easily create classes using other classes as configurable building blocks. Subobjects allow high-level concepts such as associations, bounded values, and graph nodes to be encapsulated in regular classes and reused to build applications. Subobjects avoid the name conflicts of regular multiple inheritance but still allow repeated inheritance, unlike traits and mixins.

Fig. 1 shows how subobjects can be used to create a class of elevators. An elevator is positioned on floor between the ground floor and the highest floor in the building, and carries a load between 0kg and the maximum capacity.

```
class Elevator {
  subobject floor BoundedValue<Int> {
    export getValue() as getFloor,
           setValue(Int) as selectFloor;
  }
  subobject currentLoad BoundedValue<Int> {
    export getValue() as getLoad,
           increaseValue(Int) as load,
           decreaseValue(Int) as unload;
  }
  Elevator(Int nbFloors, Float capacity) {
    subobject.floor(0,0,nbFloors);
    subobject.currentLoad(0,0,capacity);
  }
}
// Client code                    // Equivalent client code
Elevator elevator=...;
elevator.selectFloor(1);          // elevator.floor.setValue(1);
elevator.load(100);               // elevator.currentLoad.add(100);
elevator.selectFloor(0);          // elevator.floor.setValue(0);
```

Fig. 1. A subobject-oriented class of elevators.

Instead of duplicating the code to keep a value within certain bounds, the concept of a bounded value is captured in class *BoundedValue*. Class *Elevator* uses subobjects of type *BoundedValue* to model its floor and its current load. By default, the interface of *Elevator* does not contain any methods of the *floor* subobject. To add such methods to the interface of *Elevator*, they are exported in the body of the subobject. This avoids an explosion of name conflicts when a class uses multiple subobjects of the same type, as is the case for class *Elevator*.

An *export* clause creates an *alias* for a subobject member. For example, subobject *floor* exports the getter and setter methods of its value under the respective names *getFloor* and *selectFloor*. A client can therefore change the floor of the elevator by invoking either *elevator.selectFloor(...)* or *elevator.floor.setValue(...)*. The alias relation both methods cannot be broken in any way. If a subclass of *Elevator* overrides *setFloor*, the new definition also overrides the *setValue* method of its *floor* subobject.

Subobject methods that are not exported can still be accessed by clients. A client can access subobject *floor* as a real object of type *BoundedValue<Int>* through the expression *elevator.floor*. She can then use the resulting reference to increase the current floor by invoking *elevator.floor.increaseValue(...)*..

Subobject *currentLoad* models the current load of the elevator, and has the same type as the *floor* subobject. Contrary to the semantics of traditional repeated inheritance, however, both subobjects are completely isolated by default. Invoking *selectFloor* on an elevator will only change the *value* field of the *floor* subobject. Similarly, internal calls in the *floor* subobject are bound within the *floor* subobject. The subobject behaves as if *this* is replaced with *this.floor* in the subobject code in the context of class *Elevator*. A subobject can invoke methods on another subobject, but only if they it is explicitly given a reference to such a subobject, or if its methods are overridden in the composing class to do this. Parts of subobjects can be joined by overriding members of both subobjects in the composed class. In this paper, however, we do not need this functionality.

```
class BoundedValue<T extends Number> {
  subobject max Property<T> {...}
  subobject value Property<T> {
    export getValue, setValue;
    def isValid(T t) =
      outer.min.getValue <= t && t <= outer.max.getValue;
  }
  subobject min Property<T> {...}
  ...
}
```

Fig. 2. Enforcing the bounds of a bounded value.

The composed class can override subobject members by redefining them in the body of the subobject. Fig. 2 shows how class *BoundedValue* ensures that its value remains between its bounds. Class *BoundedValue* uses three subobjects of type *Property<T>* for its value and its bounds. The *setValue* method of *Property* invokes *isValid* to verify if the given value can be set. Subobject *value* overrides *isValid* to check if the given value exceeds the bounds. The ***outer*** expression is used to access the getter methods of the min and max subobjects to obtain the bounds. The value of the *outer* expression is the same as the value of *this* in the directly enclosing context. Similar to the *this* expression, calls on *outer* are bound dynamically.

```
class EventElevator {
  subobject floor EventBoundedValue<Int>;
}
```

Fig. 3. Refining a subobject.

A subobject can be *refined* in a subclass, which can customize the subobject by overriding its methods and changing its super class. The class of the new subobject is a subclass of the class of the refined subobject and the new superclass. A *rule of dominance* is used to resolve conflicts, similar to C++ and Eiffel. Suppose that *EventBoundedValue* is a subclass of *BoundedValue* that sends events if its value is changed. Fig. 3 shows how subobject refinement is used for elevators that send events when changing floors. The export clauses are not redefined, as they are inherited from *Elevator.floor*. With manual delegation it would not be possible to modify the bounded value unless *Elevator* would have contained additional boilerplate code to change the delegation object.

More details on subobject-oriented programming can be found in earlier work [19], but note that the paper uses the term *component* instead of *subobject*.

3 Subobject Transactional Memory

The key to implementing software transactional memory with subobjects is that subobject-oriented programs use subobjects to store the state of an object instead of fields. The class library of JLo contains a class *Property* that models an encapsulated field. Instead of using fields to store the state of object, a programmer can use subobjects of type *Property*. To maintain backward compatibility with Java, there are additional property classes for encapsulated lists, sets, and maps. After all, if a list of objects of type *T* is stored in a *Property*<*List*<*T*>> subobject, it is impossible to encapsulate the list because the client can directly access list. Fig. 4 shows a part of class *Property*, along with an example of how to use it. In an object-oriented style, the code in class *Property* would have been duplicated for every field in the application.

```
class Property<T> {
  T _value;
  def getValue() = _value;
  def setValue(T t) {
    if(isValid(t)) _value = t
    else throw new IllegalArgumentException();
  }
  def isValid(T t) = true; // can be overridden in subobjects
}

class Person {
  subobject name Property<String> {
    export getValue() as getName, setValue(String) as setName;
  }
  subobject children ListProperty<Person> {
    export add(Person) as addChild, values() as getChildren;
  }
}
```

Fig. 4. Implementing state with subobjects.

```
class Person {
  subobject TProperty<String> name {
    export getValue as getName, setValue as setName;
  }
  subobject TListProperty<Person> children {
    export add(Person) as addChild, values as getChildren();
  }
}
```

Fig. 5. A JLo implementation of a transactional person.

Using subobjects to store the state of an object provides the opportunity to intercept all read and write operations in an application. Suppose for example that *TProperty*, *TListProperty*, and so forth are subclasses of *Property*, *ListProperty*, and so forth that override all mutators and inspectors to add transactional behavior. We can then use a *TProperty<String>* subobject in *Person* to make the state transactional. The code in Fig. 5 and Fig. 6 show the JLo and Java implementations of a transactional class of persons with a name and a list of children. The Java implementation uses versioned boxes. Two things are noteworthy. First, the transactional JLo implementation is almost identical to the non-transactional JLo implementation. Second, the JLo implementation is not only simpler than the Java implementation, but also simpler than a non-transactional Java implementation. In addition, the more functionality the properties offer, the bigger the difference becomes. For example, methods such as *addAll()* are still accessible in the JLo version as *person.children.addAll()*, whereas the object-oriented implementation version would need an additional delegation method.

```
class Person {
  VBox<String> name;

  String getName() {
    return name.get();
  }
  void setName(String name) {
    name.set(name);
  }
  List<Person> getChildren() {
    return new ArrayList<Person>(children.get());
  }
  void addChild(Person person) {
    children.get().add(person);
  }
}
```

Fig. 6. A Java implementation of a transactional person.

3.1 Making Existing Classes Transactional

To be practical, the STM should be able to work with non-transactional third-party code. Remember from Fig. 1 and Fig. 2 that *Elevator* uses *BoundedValue* subobjects, and that *BoundedValue* uses three *Property* subobjects. Suppose that *BoundedValue* is a class from a third-party library, and uses regular *Property* subobjects for its bounds. To create a transactional elevator class, we need to create a class of transactional bounded values without modifying (or reimplementing) *BoundedValue*.

Remember from Sect. 2 that the type of a subobject can be changed in a subclass through subobject refinement. Fig. 7 shows the definition of *TBoundedValue*, which refines the *min*, *max*, and *value* subobjects such that the value and the bounds are store in *TProperty* subobjects. No conflicts resolution is required because *BoundedValue* only overrides the *isValid* methods of its *Property* subobjects while *TProperty* does not override them. The resulting subobjects in *TBoundedValue* uses the validation methods defined in *BoundedValue* and the inspector and mutator methods defined in *TProperty*.

```
class TBoundedValue<T> extends BoundedValue<T> {
  subobject max TProperty<Int>;
  subobject value TProperty<Int>;
  subobject min TProperty<Int>;

  TBoundedValue(T min, T val, T max) {
    super(min, val, max);
    subobject.min(min);
    subobject.value(val);
    subobject.max(max);
  }
}
```

Fig. 7. Creating a transactional bounded value through subobject refinement.

The constructors for the subobjects must be called explicitly in *TBoundedValue* because the sub object types have changed. These subobject constructor calls *replace* the corresponding subobject constructor calls in *BoundedValue*, and are executed when the original subobject constructor calls would have been executed. This prevents the construction of a subobject of the wrong type in the constructor of *BoundedValue*, but still guarantees that the subobjects are initialized at the correct time.

Similar to *TBoundedValue*, class *TElevator* can also be implemented as a subclass of *Elevator* that refines the *floor* and *currentLoad* subobjects, as shown in Fig. 8. It is of course also possible to create a transactional elevator from scratch by directly using *TBoundedValue* subobjects instead of *BoundedValue* subobjects.

The application logic of the program is not affected by the STM. Only the types of the subobjects that store data are different. Other than the types of the subobjects that store data, the interfaces of transactional classes such as *TElevator* and *TBoundedValue* are the same as the interfaces of their non-transactional versions. Therefore, code that

```
class TElevator extends Elevator{
  subobject floor TBoundedValue<Int>;
  subobject currentLoad TBoundedValue<Int>;

  TElevator(Int nbFloors, Float capacity) {
    super(nbFloors, capacity);
    subobject.floor(0,0,nbFloors);
    subobject.currentLoad(0,0,capacity);
  }
}
```

Fig. 8. Adding transactional behavior to a non-transactional elevator class.

uses a the transactional class looks no different than code that uses the non-transactional class. For example, the *isValid* methods, which are written in *BoundedValue* for non-transactional *Property* subobjects do not have to be modified in *TBoundedValue*, where they work with *TProperty* subobjects.

In an object-oriented style, is not always possible to add transactional behavior to a class by overriding the individual getter and setter methods because fields are often read and modified directly within a class. But even if the data is stored in reified memory locations, anticipation and additional boilerplate code for the initialization is required to be able to replace the delegation objects with transactional objects. Fig. 9 illustrates the problem. Suppose that class *Elevator* uses *Box* objects instead of regular fields to store its state. Without introducing additional boilerplate code to allow a subclass to initialize the boxes, the state cannot be replaced with *VBox* objects to add transactionality.

In a subobject-oriented programming, no anticipation is required because it requires less effort to store state in subobjects than to use fields. JLo still provides support for fields due to backward compatibility with Java, but we plan to remove this feature and use "native" code in the few core library classes that use fields.

3.2 Transaction Demarcation

Transactions are demarcated by writing the transactional code in the body of the *execute* method of a subclass of *Transaction*. The advantage over using separate *start* and *stop* calls is that the *stop* call could accidentally be forgotten. The *execute* method, which is protected, is invoked by the *commit* method of *Transaction*. If the code in *execute* throws an exception or if the transaction manager detects a conflict, the default policy is to abort the transaction and propagate the exception. Custom retry policies can be defined by overriding the *retry* method of *Transaction*. The code in Fig. 10 illustrates how two threads can use the elevator without running the risk of overloading the elevator or trying to load the elevator when it is on the wrong floor.

Adding transaction demarcation to an existing program can be done by overriding the methods that must be executed as a transaction and performing a *super* call in the *execute* method of a *Transaction*. Methods that create new threads may have to be reimplemented to ensure that all threads run in a separate transaction.

```
class Elevator {
  private Box<Int> nbFloors = new Box<Int>(0);
  private Box<Int> floor = new Box<Int>(0);
  private Box<Int> capacity = new Box<Int>(0);
  private Box<Int> load = new Box<Int>(0);

  Int nbFloors() {return nbFloors.get();}
  Int floor() {return floor.get();}
  Int capacity() {return capacity.get();}
  Int load() {return load.get();}

  Elevator(Int nbFloors, Float capacity) {
    this.nBfloors.set(nbFloors);
    this.capacity.set(capacity);
  }
}
class TElevator extends Elevator {
  // Impossible to change Box objects to VBox objects.
}
```

Fig. 9. Object-oriented delegation requires anticipation and additional boilerplate code.

```
TElevator elevator = new TElevator(3,150);
new Thread() {
  void run() {for(int i=0;i<100;i++) {
              new Transaction() {
                void execute() {
                  elevator.setFloor(2);
                  elevator.load(100);
                  elevator.setFloor(0);
                }
              }.commit();
            }
}.start();
new Thread() {
  void run() {for(int i=0;i<100;i++) {
              new Transaction() {
                void execute() {
                  elevator.setFloor(1);
                  elevator.load(100);
                  elevator.setFloor(0);
                }
              }.commit();}
            }
}.start();
```

Fig. 10. Demarcating transactions with the Command pattern.

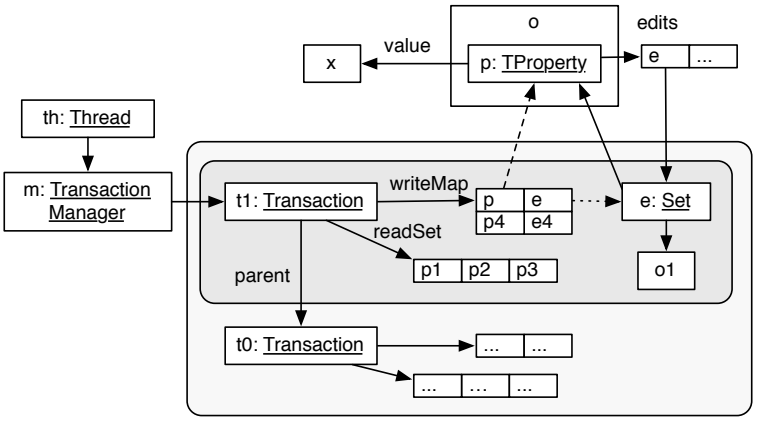

Fig. 11. Thread-local transaction managers provide a unique view per transaction.

4 Example Implementation

In this section, we discuss our STM implementation. The current implementation uses multi-version concurrency control [14, 3], but other mechanisms can be used as well.

Classes *TProperty*, *TListProperty*, and so forth are subclasses of the regular property classes *Property*, *ListProperty*, and so forth that add transactional behavior. The transactional classes override the mutator and inspector methods to log all reads and writes to provide transactional behavior. Write operations are reified as subclasses of *Edit*. Transactions are modeled by the *Transaction* class.

Figure 11 illustrates the run-time object layout. The solid arrows represent normal object references. The striped arrows represent weak object references, which are implemented with *WeakReference* object in Java. Weak references do not prevent an object from being garbage collected. The heap contains an object o with a transactional single valued property *p*. Transaction t_1 is nested in transaction t_0, and has modified the value of *p* to a reference to o_1 via the *Set* object *e*. Class *Set* is a subclass of *Edit* that represents a write operation to a single valued property. In addition, the transaction has performed read operations on transactional properties p_1, p_2, and p_3.

Class *TransactionManager* has a static thread-local variable *manager* that stores a reference to a *TransactionManager* object. This gives each thread its own transaction manager which it can access via *TransactionManager.manager*. A *TransactionManager* keeps a reference to the transaction in which it is currently running. Nesting of transactions is reflected in the object structure of the *CompositeTransaction* objects, which keep a reference to their parent transaction. In Fig. 11, transaction t_1 is nested in transaction t_0, but it is not running in a separate thread.

To give a transaction its own unique version of the state of an object, it keeps track of all reads and writes that are performed during its execution. The reads are stored as a set of property subobjects that were read. The writes are stored as a map that stores the latest *Edit* that was performed on a property subobject within the transaction. In the example in Fig. 11, single valued property *p* points to o_1 within transaction t_1, whereas it points to x in every transaction that has not modified *p*.

Transactions in Action

The diagram in Fig. 12 illustrates the process of setting the value of *p* to *v*. The dotted arrows represent temporary references via local variables. Instead of directly modifying a its field, subobject *p* creates an object *s* of class *Set* that references the new value *v*. Subobject *p* then and passes *s* to the transaction manager, which tells its transaction *t* of the current thread to absorb *s*. Transaction *t* first checks whether its write map already contains an *Edit* for sub object *p*. If that is the case, it tells the current *Edit* for *p* to absorb *s*; otherwise, it registers *s* in its write map and lets *s* register itself in the edit list of *p* to keep it from being garbage collected. For a *Set* object, the *absorb* method simply replaces the referenced value.

The diagram in Fig. 13 illustrates the process of reading the value of a single valued property. Subobject *p* asks the transaction manager to search for an *Edit* object that is associated with *p* in the current transaction. If an *Edit* object *e* is found, it is returned and *p* uses *e* to determine the current value. If no *Edit* object is found, the value that is stored in *p* is returned. In either case, transaction *t* adds *p* to its read set.

The *Edit* objects for the other property classes are similar, but their implementation is more complex because the data structures are more complex. For example, the *Edit* objects for a transactional list property become part of a linked list when being absorbed. In addition, setting the *i-th* item in a list also implies a read operation. Otherwise, there would be no conflict with a concurrent transaction that reduces the size of the list below *i*.

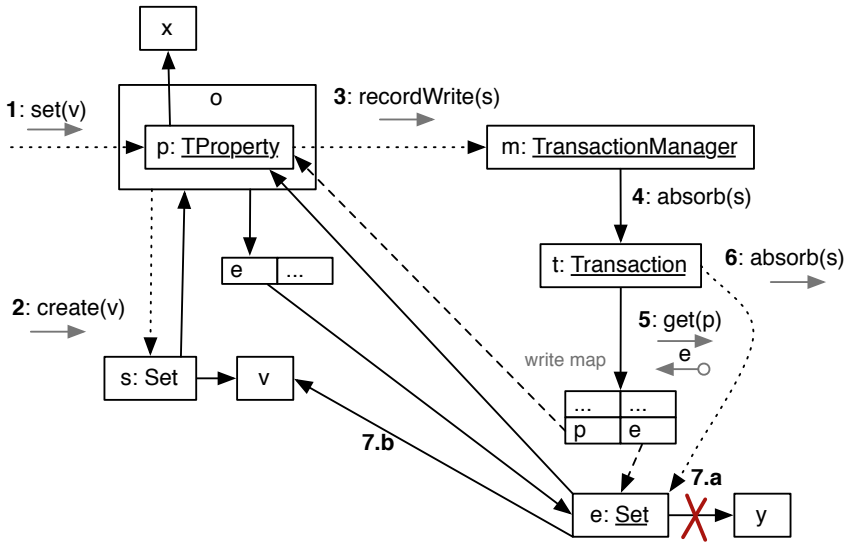

Fig. 12. Setting the value of a transactional property.

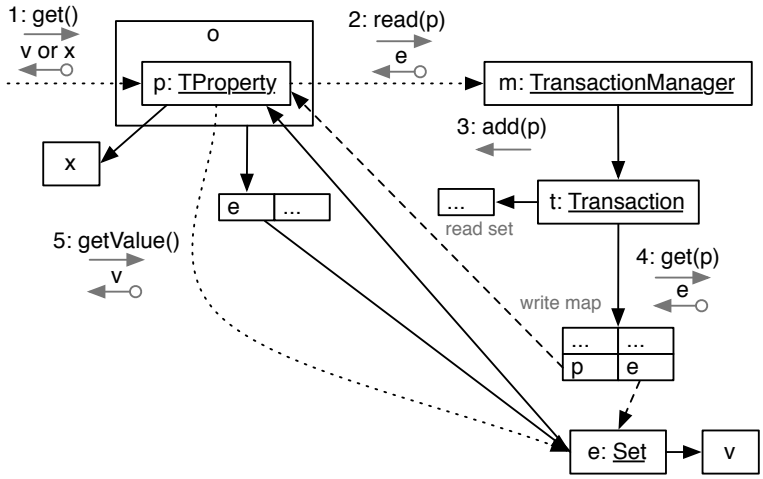

Fig. 13. Getting the value of a transactional property.

When a transaction is committed, conflict resolution is performed, and the transaction is aborted when a cycle in the waits-for graph is detected. When a nested transactions commits, the read set and write map are merged with those of its parent transaction. When a root transaction commits, the *Edit* objects apply their modifications to the fields of the corresponding property subobjects. Read and write operations that are performed outside of transactions are applied directly to the fields of the property subobjects.

Memory management

To ensure proper memory management, both the key and value references in the map are weak references, each property subobject stores strong references to all basic transactions that are applied to it. As a result, objects that were created and modified within a transaction can be freed by the garbage collector when they are no longer reachable in the view of any transaction.

Because of the weak references in the write map, however, we must prevent garbage collection of objects that are created within a transaction and are referenced by a reachable object through an *Edit* object. The *Edit* object has a strong reference to the newly created object, but there is no strong reference to the *Edit* object. Therefore, each property subobject keeps a list of strong references (list *edits* in Fig. 11) to the *Edit* objects that have modified it, and are part of living transactions. The *Edit* objects register themselves when they become part of the object structure in the write map.

Suppose for example that object v of Fig. 13 was created within the current transaction and is referenced only by the transaction-local version of p. In this case, v cannot be garbage collected because of the strong reference of e, which in turn is referenced by p via the *edits* list. But if p is set to a new value within the transaction, the strong reference from e is replaced with a reference to the new value. Object v then becomes unreachable and will be garbage collected.

5 Related Work

Approaches for software transactional memory can be divided into two categories: language approaches and library approaches. Language implementations add dedicated language constructs to provide STM functionality and/or modify the language runtime to support transactional semantics. Library approaches provide STM functionality through an API or by using the metaprogramming facilities of languages. For reasons of space, we only discuss the approaches that are most closely related to our approach.

Language Approaches Isolation types [4, 5] provide a language approach that is similar to versioned boxes. Instead of implementing an STM, isolation types implement a revision control system that is similar to revision control systems used for managing source code. Every asynchronous task runs isolated from the others and has its own revision of the shared state. When tasks are joined, the state revisions are merged. Conflict resolution is defined by the isolation type, is deterministic, and can never fail. The execution of concurrent programs is therefore deterministic, but isolation is not guaranteed. Isolation type require the addition of dedicated language constructs to C#.

Static Library Approaches TBoost.STM [9], DSTM2 [10], Versioned Boxes [6], and SAW are library approach for static programming languages.

TBoost.STM (formerly known as DracoSTM [9]) is a C++ library for software transactional memory. Memory locations are reified as objects of the *native_trans* class. Fields and local variables are wrapped in *native_trans* objects and reads and writes are performed by invoking methods on the current transaction object. As a result, algorithms must be adapted to work with the transaction mechanism. The authors provide a list class that can be used without knowing about the transaction mechanism.

DSTM2 [10] is a Java STM library with a customizable transaction mechanism. Transactional classes are written as interfaces with getter and setter methods. To construct an object whose class implements that interface, the *Class* object of the interface is given to a transactional factory. The factory dynamically generates code that implements the getters and setters in a transactional manner. The transaction mechanism can be changed by changing the factory. DSTM2 cannot work with existing code since standard Java classes do not use the transactional factories.

Versioned Boxes [6] are reified memory locations that are used to write transactional applications. Mutator and inspector methods for single values, lists, and so forth are implemented by delegating the calls to the versioned box. Existing classes written in an object-oriented style cannot be made transactional because such classes may access their fields directly.

SAW [20] is a Java library that adds synchronization to existing classes through aspect weaving. Classes and transactional methods are marked with *@shared* and *@atomic* annotations. The authors implement both an STM mechanism and a lock-based mechanism. The active mechanism is chosen by selecting a particular aspect. This choice, however, is not transparent because the programmer must manually prevent dead-locks when the lock-based mechanism is used. As a result, SAW is more difficult to use than a pure STM.

Dynamic Library Approaches SSTM [7] and the Smalltalk library of Renggli and Nierstrasz [15] are library approaches for dynamic programming languages. These approaches provide transactional functionality by using the metaprogramming facilities of the host language.

CSTM [7] is an STM framework based on context-oriented programming. The framework is implemented in ContextL, a context-oriented extension of CLOS. Slots in an object do not store values directly, but instead store a reified memory location object. The default behavior of these memory locations is to get and set the memory value directly. ContextL provides a layered slot access protocol that allows context layers to modify the behavior of slot accesses. A mode layer defines the semantics of regular slot accesses and defines the transaction demarcation. The transaction layer defines the semantics of transactional slot accesses. The transaction mechanism can be change at run-time by enabling a different transactional layer. The enable the STM for a class, the class must be annotated with the *define-transactional-class* function. Transactions are demarcated by wrapping the code in a call to the *atomic* function. The use of memory location objects is similar to the use of property subobjects.

Renggli and Nierstrasz implement a Smalltalk STM library that exploits the dynamic nature of Smalltalk [15]. Their implementation lazily rewrites the Smalltalk program while it runs to insert the transactional behavior. State accesses are rewritten to redirect the control flow to the transaction mechanism. Primitive operations, such as *#at:* and *#put:* are annotated with the name of an equivalent non-primitive method that must be used instead in transactional code. When an object is accessed in a transaction, two copies are made. One object represents the initial object, while the other object represents the transaction-local object. When a transaction is committed, the initial object is compared to the current version of the original object to detect conflicts. A transaction is created by sending the *#atomic* to a block.

6 Conclusion and Future Work

Existing STM libraries for static object-oriented programming languages require additional boilerplate code compared to STM librarires for dynamic languages, or dedicated language extensions.

We have defined an STM library for a static subobject-oriented programming programming language. We have shown that this library not only requires less boilerplate code than static object-oriented STM libraries, but also requires less boilerplate code than a regular object-oriented program. In addition, transactional behavior can be added to existing subobject-oriented classes. We have implemented a multiversion concurrent control mechanism as a proof-of-concept.

We plan to combine subobjects with classboxes [1] or higher-order hierarchies [8], which are generic modularization techniques, to simplify adding transactionality to an application. The transactional property classes can then be placed in a separate classbox or hierarchy. Similarly, a subclass of *Thread* would create a transaction when a new thread is started. To make an application transactional, a programmer would then extend both the original application and the transactional classbox or hierarchy.

References

1. A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in Java. In *OOPSLA*, pages 177–189, 2005.
2. E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, pages 81–96, 2009.
3. P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.
4. S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. OOPSLA, pages 691–707, 2010.
5. S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *OOPSLA*, pages 427–444, 2011.
6. J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.
7. P. Costanza, C. Herzeel, and T. D'Hondt. Context-oriented software transactional memory in common lisp. In *DLS*, pages 59–68, 2009.
8. E. Ernst. Higher-order hierarchies. In *ECOOP*, pages 303–328, 2003.
9. J. E. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memroy. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD). In conjunction with OOPSLA*. Oct 2007.
10. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. OOPSLA, pages 253–262, 2006.
11. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. PODC, pages 92–101, 2003.
12. A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. OOPSLA, pages 671–690, 2010.
13. R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA*, pages 885–902, 2011.
14. D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Cambridge, MA, USA, 1978.
15. L. Renggli and O. Nierstrasz. Transactional memory in a dynamic language. *Comput. Lang. Syst. Struct.*, 35:21–30, April 2009.
16. B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. PPoPP, pages 187–197, 2006.
17. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.
18. M. van Dooren and B. Jacobs. Implementations of subobject-oriented programming, 2012. `http://people.cs.kuleuven.be/marko.vandooren/subobjects.html`.
19. M. van Dooren and E. Steegmans. A higher abstraction level using first-class inheritance relations. In *ECOOP*, pages 425–449, Berlin, Germany, 2007. Springer.
20. Y. Yamada, H. Iwasaki, and T. Ugawa. SAW: Java synchronization selection from lock or software transactional memory. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 104–111. IEEE, 2011.