Identifying A Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-Language Virtual Machines

Stefan Marr and Theo D'Hondt

Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Elsene, Belgium
{stefan.marr, tjdhondt}@vub.ac.be

Abstract. Supporting all known abstractions for concurrent and parallel programming in a virtual machines (VM) is a futile undertaking, but it is required to give programmers appropriate tools and performance. Instead of supporting all abstractions directly, VMs need a unifying mechanism similar to `INVOKEDYNAMIC` for JVMs.
Our survey of parallel and concurrent programming concepts identifies concurrency abstractions as the ones benefiting most from support in a VM. Currently, their semantics is often weakened, reducing their engineering benefits. They require a mechanism to define flexible language guarantees.
Based on this survey, we define an ownership-based meta-object protocol as candidate for VM support. We demonstrate its expressiveness by implementing actor semantics, software transactional memory, agents, CSP, and active objects. While the performance of our prototype confirms the need for VM support, it also shows that the chosen mechanism is appropriate to express a wide range of concurrency abstractions in a unified way.

Keywords: Virtual Machines, Language Support, Abstraction, Parallelism, Concurrency

1 The Right Tool for the Job

Implementing parallel and concurrent systems has been argued to be a complex undertaking that requires the right tools for the job, perhaps more than other problems software engineering encountered so far. Instead of searching for a non-existing *silver bullet* approach, we argue that language designers need to be supported in building domain-specific concurrency abstractions.

Let us consider the implementation of a typical desktop application. A mail application combines several components that interact and have different potentials to utilize computational resources. The user interface component is traditionally implemented with an event-loop to react to user input. In a concurrent setting, it is also desirable to enforce encapsulation like in an actor model, since encapsulation simplifies reasoning about the interaction with other components.

Another part of the application is the data storage for emails and address book information. This part traditionally interacts with with a database. The natural way to implement this component is to use a software transactional memory (STM) system that extends the transaction semantics of the database into the application. This allow a unified reasoning when for instance a new mail is received from the network component and needs to be stored in the database.

A third part is a search engine that allows the user to find emails and address book entries. Such an engine can typically exploit data-parallel approaches like map/reduce or parallel collection operations for performance.

However, supporting the various different approaches to parallel and concurrent programming on top of the same platform comes with the challenge to identify basic commonalities that allow to abstract from the particularities of specific constructs and languages. Today's high-level language virtual machines (VMs) do not provide intrinsic support for more than one specific approach [17]. While some approaches like Fork/Join [14], Concurrent Collections [3] or PLINQ can be implemented as libraries without losing any semantics or performance, approaches like the actor model are typically implemented as an approximation losing for instance the engineering benefits of encapsulation [11].

We approach this problem with a survey of the various concepts of parallel and concurrent programming to identify concepts that are relevant for a multi-language virtual machine (short: VM). Based on this survey, we define an ownership-based meta-object protocol and evaluate its suitability to implemented the identified concepts. Furthermore, we briefly evaluate the performance properties of our prototype and discuss related work which could be used to realize an implementation with optimal performance characteristics.

2 A Survey of Parallel and Concurrent Programming Concepts

The goal of this survey is to *identify concepts* that are relevant for a multi-language VM. To that end, we first select questions that enable us to categorize the concepts by relevance. Afterwards, we detail our approach to identify the concepts and finally, we present the findings and discuss our conclusions.

2.1 Survey Questions

When concepts are considered for inclusion in a VM, one of the main goals is to avoid unnecessary complexity. From that follows, that a new concept only needs to be added to a VM if it cannot be implemented reasonably in terms of a library on top of the VM. Thus, our first question is:

Lib Can this concept be implemented in terms of a library?

Interpreting the question very broadly, we consider whether some variation of the concept can be implemented. Typically, such a library implementation can either suffer from losing semantic guarantees, or it has to take performance drawbacks into account. Common examples are implementations of the actor model on top of the JVM or CLR [11].

To account for that variation, we need the following two questions:

Sem Does this concept require runtime support to guarantee its semantics?
Perf Would runtime support enable significant performance improvements compared with a pure library solution?

To answer Sem, we also consider interactions of different languages on top of a VM. This is relevant since common language guarantees are enforced by a compiler but do not carry over to the level of the VM. One example is the semantics of single-assignment variables, which is typically not transferred to the bytecode level of a VM. Similarly, we considered for Perf that knowledge about full language semantics often enables better optimizations. For instance, the knowledge about immutability enables constant folding, and taking the semantics of critical sections into account enables optimizations like lock elision.

The last categorization criterion is whether the concept is prior art:

PA Is the concept already supported by a VM like the JVM or CLR?

2.2 Selecting Subjects and Identifying Concepts

To identify concepts, we rely foremost on the overview given by two surveys [2, 25] as our main subjects. They give a broad foundation but are dated. To ensure that the most common concepts are included, we survey also a number of languages used in research or industry and select research papers from recent years to cover current trends. The full list of *subjects* is given in Tab. 1.

Table 1. Survey Subjects: Languages and Papers

Active Objects [13]	Charm++	Fortress	Occam-pi	Simple Java
Ada	Cilk	Go	OpenCL	Skillicorn&Talia [25]
Aida [15]	Clojure	Io	OpenMP	Sly
Alice	CoBoxes [23]	JCSP	Orleans [4]	StreamIT
AmbientTalk	Concurrent Haskell	Java Views [5]	Oz	Swing
Ateji PX	Concurrent ML	Join Java	PAM [22]	UPC
Axum	Concurrent Objects	Linda [7]	Reactive Objects [20]	X10
Briot et al. [2]	Concurrent Pascal	MPI	SCOOP [19]	XC
C#	Erlang	MapReduce [16]	STM [24]	
Chapel	Fortran 2008	MultiLisp [9]	Simple C/C++	

Starting with the two surveys, we *identify* for each subject the basic concepts and mechanisms introduced in the paper or provided by the language. For languages, we regard the language-level as well as possible implementation-level concepts. Note that the identified concepts necessarily abstract from specific details that vary between the different subjects. Thus, we do not regard every minor variation of a concept separately. However, this leaves room for different interpretations of our survey questions. For subjects like C/C++ and Java, we regard the *simple* core language and standard libraries. Interesting libraries or extensions available for their eco systems are considered as separate subjects.

2.3 Results

The analysis of the subjects given in Tab. 1 resulted in 82 identified concepts. Since most of them are accepted concepts in the literature, we will only discuss the results with regard to our questions in this paper. As mentioned earlier, some concept variations have been considered together as a single concept. For example, the distinct concepts of monitors and semaphores, have been regarded as part of *locks* in this survey. Similarly, *parallel bulk operations* is included and also covers *parallel loops* because of their similarity and closely related implementation strategies. Thus, Tab. 2a and 2b include 60 concepts and their respective survey results.

Table 2a. *Survey Results:* Prior Art and Library Solutions

Prior Art	PA	Lib	Sem	Perf		PA	Lib	Sem	Perf
Atomic Primitives	X	-	-	X	Co-routines	X	-	-	X
Condition Variables	X	X	-	X	Critical Sections	X	X	-	X
Global Address Spaces	X	X	-	X	Green Threads	X	-	-	-
Immutability	X	-	X	X	Join	X	-	-	-
Locks	X	X	-	X	Memory Model	X	-	X	X
Method Invocation	X	-	-	X	Race-And-Repair	X	X	-	-
Thread Pools	X	X	-	-	Thread-local Variables	X	X	-	X
Threads	X	X	-	-	Volatiles	X	-	X	-
Wrapper Objects	X	X	-	X					

Library Solutions	PA	Lib	Sem	Perf		PA	Lib	Sem	Perf
APGAS	-	X	-	-	Agents	-	X	-	-
Atoms	-	X	-	-	Concurrent Objects	-	X	-	-
Event-Loop	-	X	-	-	Events	-	X	-	-
Far-References	-	X	-	-	Fork/Join	-	X	-	-
Futures	-	X	-	-	Guards	-	X	-	-
Message Queue	-	X	-	-	One-sided Communication	-	X	-	-
PGAS	-	X	-	-	Parallel Bulk Operations	-	X	-	-
Reducers	-	X	-	-	Single Blocks	-	X	-	-
State Reconciliation	-	X	-	-					

As Tab. 2a shows, about half of the concepts are either already available in JVM and CLR or can be implemented in terms of a library without sacrificing semantics or performance aspects. We will discuss only the remaining 26.

With 18, the majority of the concepts requiring runtime support (Tab. 2b) suffer from weaker semantics. Most of these concepts are usually realized either with enforcement on a compiler level or require correct construction by the programmer. However, a compiler cannot enforce guarantees on a VM if they

Table 2b. *Survey Results:* Runtime Support Required

Runtime Support Required	PA	Lib	Sem	Perf		PA	Lib	Sem	Perf
Active Objects	-	X	X	-	Actors	-	X	X	X
Asynchronous Invocation	-	X	X	X	Axum-Domains	-	X	X	-
Barriers	-	X	-	X	By-Value	-	X	X	X
Channels	-	X	X	X	Clocks	-	X	-	X
Data Movement	-	-	-	X	Data Streams	-	X	X	X
Implicit Parallelism	-	X	-	X	Isolation	-	X	X	X
Locality	-	-	-	X	Map/Reduce	-	X	X	-
Message sends	-	X	X	X	Mirrors	-	X	-	X
No-Intercession	-	X	X	X	Ownership	-	-	-	X
Persistent Data Structures	-	X	X	-	Replication	-	X	X	-
Side-Effect Free	-	-	X	X	Speculative Execution	-	-	X	X
Transactions	-	X	X	X	Tuple Spaces	-	X	X	-
Vats	-	X	X	X	Vector Operations	-	X	-	X

are not present in the bytecode intermediate language. Thus, a Java program can mutate a supposedly immutable object of another language. An example is the semantics of final fields in Java, which can be changed via reflection, and thus, are not truly constant. Persistent data structures and tuple spaces are examples that rely on the notion of immutable values to provide a consistent framework for reasoning. Similarly, E's vats, AmbientTalk's actors, and Axum's domains restrict mutation to an owner. While a reference can be obtained to an object owned by another actor or vat, they can only be mutated within the owner's context. These concepts also share the property that method invocation on such objects need to be done asynchronously in the context of the owning entity and under the scheduling regime of the entity. This applies to active objects, too.

The other concepts, like barriers, clocks, data movement, locality, and vector operations, will benefit from adaptive optimizations of a just-in-time compiler, which is aware of their semantics, or require information of the underlying hardware that is normally not exposed by the VM. Implicit parallelism and speculative execution can be considered as adaptive optimizations, too. However, they imply likely a significantly higher complexity.

2.4 Conclusions and Requirements

We conclude from our survey that approaching the semantics of concurrency constructs is the most promising angle to take when improving support for parallel and concurrent languages. Performance is another important but to specific problem. The concepts discussed here do not lend themselves towards more generally applicable optimizations. Instead, such optimizations would likely be specific to a single concept. Thus, from the set of concepts that will benefit from semantic enforcement, we distill the following requirements for a VM:

Managed Mutation Many concept impose rules for when and how state can be modified. Thus mutation must be manageable in a flexible manner.

Managed Execution Similarly, the activation of methods on an object is typically also regulated and needs to be adaptable.

Ownership One recurring notion is that mutation and execution are regulated based and relative to an owning entity. Thus, ownership needs to be supported in a manner that enables adaptable mutation and execution rules.

Leveled Reflection Many use cases of reflective meta-programming still need to follow the concurrency-related language semantics to be safe. Thus, there is a need to distinguish between restricted language-level reflection, and unrestricted meta-level reflection.

Enforceability These rules need to be enforceable across different concurrency models. Thus, if a reference to an object belonging to an actor is obtained, everything done with it must obey the rules of the actor language.

3 An Ownership-based MOP to Express Concurrency Abstractions

Based on the described requirements we define a meta-object protocol (MOP) [12] that is based on the notion of ownership. First, we describe its semantics, then given an example how it can enforce immutability, and finally, we are going to detail the implementation approach for our Smalltalk-based prototype.

3.1 Design of the MOP

Following the stated requirements, we base our approach on the notions of object ownership, state access, and execution. The owner of an object, here called *domain*, defines the semantics of operations on all objects it owns. The semantics it defines regard *reading* of object fields, *writing* of object fields, and *invocation* of methods on objects. A thread of execution is executing in a domain, but as objects may change their owners, threads can change the domains they execute in. In addition, the thread has a flag that defines whether it is executing on the base level, where the domain semantics are enforced, or on the meta level without enforcement. See Fig. 1 for an overview.

Depending on the VM, a domain also needs to regard globally accessible resources that may lie beyond its scope but that can have impact on the execution. That typically includes *lexical* globals and primitive operations of a VM that cannot be regarded otherwise. Thus, the following conceptual semantics are associate with the MOP.

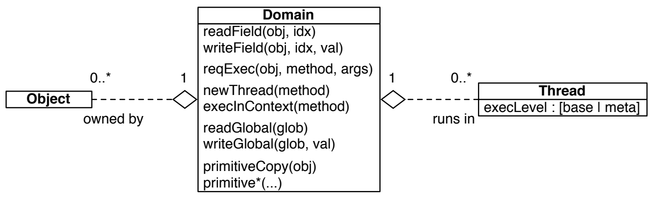

Fig. 1. Ownership-based Meta-object Protocol Supported by the Domain Object

A **Domain** owns objects, and every object has an owner. It defines the concurrency semantics for owned objects. This satisfies the *ownership* requirement.

A **Thread** is the unit of execution. It executes either in the base level, enforcing the semantics of domains (incl. reflective operations), or it executes on the meta level without enforcement (`execLevel`). This satisfies the *leveled reflection* requirement. Furthermore, a thread *runs in* the context of a domain. The `newThread` operation enables the domain to control the number of threads executing at the same time. `execInContext` enables an existing thread to change the execution domain for the duration of the execution of `method`. This is necessary for the *managed execution* requirement.

All **Read/Write** operations of object fields are delegated to `readField` and `writeField` of the owner. The domain can then decide based on the given object and the slot index, as well as other execution state, what action needs to be taken. This satisfies the *managed mutation* requirement.

Request Execution (`reqExec`) is used for all method invocations enabling the domain to decide based on the given object, the method to be executed, its arguments, and other execution state, how the invocation is to be handled. This satisfies, together with the execution context of a thread, the *managed execution* requirement.

External Resources, i. e., globally shared variables and primitives need to be handled by the domain if they otherwise break semantics. To that end, the domain includes `readGlobal`/`writeGlobal` which enables for instance to give globals a semantic local to the domain. Furthermore, it includes `primitive*` operations, as for instance `primitiveCopy` to override the semantics of VM primitives. The direct use of `primitiveCopy` would allow to copy arbitrary objects without regarding domain semantics. This and all of the above allow us to satisfy the *enforceability* requirement.

3.2 Example: Enforcing Immutability

Fig. 2 gives a sequence diagram of how immutability could be enforced based on our approach. The `JavaThread` starts running in the meta level and then directly starts to execute application code in the base level. At some point, it invokes `setFoo` on an immutable object. In our model, this invocation goes first as a request for execution to the domain owning the immutable object. The domain

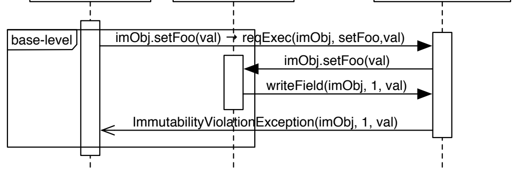
JavaThread
Immutable Obj
Immutable Domain
base-level
imObj.setFoo(val) → reqExec(imObj, setFoo,val)
imObj.setFoo(val)
writeField(imObj, 1, val)
ImmutabilityViolationException(imObj, 1, val)

Fig. 2. Example of Immutability Enforcement based on the MOP

code executes itself on the meta level. Since immutability does not interfere with method execution semantics, the request is granted, and `setFoo` is invoked on the object. The invocation is executed in the base level to enforce the desired semantics, which then results in a request to the domain to write a field in the object. For this immutable object, the request is denied and instead an exception is raised to notify the `JavaThread` which executes the code. The mutation would also be denied if `JavaThread` would use reflection while executing in the base level, since the reflective operations also pass by the domain.

Section 4.1 discusses a longer example, showing how to implement Clojure's agents based on the MOP.

3.3 Implementation Strategy

Our prototype is implemented in Smalltalk applying the implementation strategy presented by Renggli and Nierstrasz for an STM [21]. Similar to their solution, we enforce the use of our MOP by transforming Smalltalk bytecode. Thereby, we abstract from a particular language that compiles to bytecode. Our transformations change reads and writes of instance variables as well as globals to the corresponding MOP operations discussed earlier. Message sends are adapted similarly to request execution on the owning domain.

To keep meta and base level apart, selectors in base-level message sends are prefixed. This prefixing also separates the actual compiled methods for meta and base level. The unmodified version of the bytecode executes on the meta level, while the transformed code executes on the base level. Instead of relying on the conceptual `execLevel` flag in a thread, we explicitly enter and exit the base level. Entry points are marked by sending `#enter:` to a block. The compiler transforms all blocks that statically receive the enter message, i. e., lexically in the form of `[foo doSomething] enter: domain`. To exit the base-level code, certain methods are not transformed. To mark such exit points methods can be annotated with `<doNotTransform>`. VM primitives are handled by annotating their Smalltalk representation with `<replacement: #selector>`.

The owner of an object is expressed by a new slot for the domain in all classes that support it. For some classes the VM make special assumptions and does not allow adding slots. One example is the `Array` class. Here we provide an adapted subclass with the slot and ensure it is used instead of `Array`.

4 Evaluation

To evaluate our approach, we present Clojure's agents[1] as a detailed example and then discuss the expressiveness and performance of our approach. The expressiveness is assessed by demonstrating that a number of concurrency models can be implemented straightforwardly. Furthermore, we comparing how our abstraction fairs compared to ad-hoc implementations. For the performance evaluation,

[1] `http://clojure.org/agents`

we use an actor implementation as well as an STM system. Both have been implemented in an ad-hoc version and in a version based on our MOP to compare the performance of the two approaches. Note that Sec. 3.1 already evaluated how the MOP satisfies the requirements derived from our survey.

4.1 By Example: Clojure's Agents

Since the discussion was so-far theoretical, we will look into one concurrency construct more closely. Clojure's agents provide an abstraction for *event-loop* concurrency. An agent represents a resource with a mutable state. However, the state is modified only by the agent itself. The agent receives *update functions* asynchronously. An update function takes the old state and produces a new state. The execution is done in a dedicated thread, so that at most one update function can be active for a given agent at any time. Furthermore, other threads will always read a consistent state of the agent at any time. However, while Clojure encourages the use of immutable data structures, it is not enforcing it. Thus, in practice the assumed guarantees can be violated. See Lst. 1.1 for a simplified implementation. The complete implementation is slightly longer and takes 8 methods with a total of 31 lines of code (LOC) (cf. Tab. 3).

```
Object < #Agent instanceVariables: 'mailbox state'.

Agent >> await [ mailbox waitUntilEmpty ]
Agent >> read  [ ^ state ]

Agent >> send: anUpdateBlock [ mailbox nextPut: anUpdateBlock ]

Agent >> send: anUpdateBlock with: args [
        self send: [:old | anUpdateBlock value: old value: args ] ]

Agent >> initialize [
        mailbox := SharedQueue new.
        [true whileTrue: [ self processIncomingMessages ]] fork ]

Agent >> processIncomingMessages [
        | updateBlock |
        updateBlock := mailbox waitForFirst.
        state       := updateBlock value: state.
        mailbox removeFirst ]
```

Listing 1.1. Agent implementation in Smalltalk

Like in Clojure, Lst. 1.1 does not guarantee any execution semantics. Since Smalltalk does not have private methods, `#processIncomingMessages` could even be called from another thread and violate the assumption that only one update function is executed at a time.

To enforce the expected guarantee, we now define `AgentDomain` in Lst. 1.2. Since `Agent` and `AgentDomain` implement the concurrency semantics, all methods need to be annotated with `<doNotTransform>`, including the ones in Lst. 1.1. With this annotation, we make sure that our implementation code is executed on the meta level. The domain then defines the `#requestExecutionOf:on:...` methods to ensure that the main constraint of having a single thread of execution for agent methods is obeyed.

Table 3. Agent Implementation Metrics

Class	#M	LOC	#BC
Agent	8	31	77
With Guarantees			
Agent	8	39	85
AgentDomain	4	34	132

With Immutability	#M	LOC	#BC
Agent	8	41	100
AgentDomain	4	34	132
ImmutableDomain	6	17	25

#M: number of methods
#BC: number of bytecodes

```
Domain < #AgentDomain instanceVariables: 'agent'

AgentDomain >> agent: anAgent [ agent := anAgent ]

AgentDomain >> requestExecutionOf: aSelector on: anObject [
    <doNotTransform>
    "Rules are only enforced on the agent itself"
    anObject = agent             ifFalse: [ ^anObject perform: aSelector].
    (aSelector = #read or:   [ "White-listed methods"
     aSelector = #await or:  [
     aSelector = #shutdown   ]]) ifTrue:  [ ^agent perform: normSel     ].
    Error signal: 'Access denied'. "Everything else is an error"]

AgentDomain >> requestExecutionOf: aSel on: obj with: par1 [
    <doNotTransform>
    obj = agent     ifFalse: [ ^obj perform: aSel with: par1. ].
    (aSel = #send:) ifTrue:  [ ^agent send: par1              ].
    Error signal: 'Access denied'. "Else: an error"]

AgentDomain >> requestExecutionOf: aSel on: obj with: p1 with: p2 [
    <doNotTransform>
    obj = agent          ifFalse: [ ^obj perform: aSel with: p1 with: p2].
    (aSel = #send:with:) ifTrue:  [ ^agent send: p1 with: p2            ].
    Error signal: 'Access denied'. "Else: an error"]
```

Listing 1.2. AgentDomain to enforce desired guarantees

As already demonstrated in Sec. 3.2, it becomes also simple to add the guarantee that an agent state only refers to immutable data structures. The implementation of `ImmutableDomain` changes the semantics of all operations that write to object fields. Thus, for instance `writeField` throws an error as in Fig. 2. The agent itself will ask the `ImmutableDomain` to adopt the new state after an update function is completed. This guarantees that the immutability cannot be violated while executing code on the base/language level. Tab. 3 shows that the necessary adaptations are minimal to provide this extra guarantee.

4.2 Subjects

LRSTM is the STM implementation by Renggli and Nierstrasz [21]. We reimplemented the compiler transformations to use the same Smalltalk and libraries as for our MOP to allow a comparison of the systems. Since the MOP is implemented using the ideas of the LRSTM implementation, the resulting systems are very similar. The STM algorithm tracks read and write operations, by keeping read- and write-logs. It uses the read-log to detect conflicts during the commit phase and when no conflicts are detected, it will apply the writes atomically.

AmbientTalkST is a framework to build applications using actor semantics similar to E [18] and AmbientTalk [28]. We call it a framework, since it requires care to set up the actors correctly to enforce the desired semantics. We have not implemented a full language with its own syntax, parser, and compiler, which would take care of these details implicitly. However, the framework uses stratified proxies [1, 27] to guarantee actor semantics. Since the proxies are stratified, the guarantees are also given for reflective operations.

Actors refer to objects owned by other objects only via far-references which in return enforce that all messages sent to them will be reified and put into the inbox of an actor. The actor will process the messages one at a time. The far-reference implementation makes sure that parameters and return values are encapsulated in far-references as necessary to avoid introducing shared state. This implementation approach is different from our MOP-based one, but reflects more closely how AmbientTalk enforces its language guarantees.

Additional Concurrency Abstractions. To demonstrate the expressiveness of our abstraction, we implemented also as already discussed Clojure's agents. Furthermore, we implemented the Active Object pattern [13] and CSP+π, a minimal version of occam-π's semantics.

4.3 Expressiveness

Appropriate abstractions allow a concise description of a problem. Thus, we compare implementation metrics to assess the impact of using our abstraction instead of ad-hoc approaches. The used metrics are *number of classes*, *number of methods*, *LOC*, and *number of bytecodes*. Number of methods includes necessary extensions and changed methods in system classes. LOC refers to the length of a method including comments but excluding blank lines. Since LOC varies based on coding conventions and comments, we also list the number of bytecodes of all methods.

Table 4. Metrics for Ad-hoc and MOP-based Implementations

	#Classes	#Methods	LOC	#Bytecodes
Agents (ad-hoc, without enforcement)	1	8	31	77
Agents (MOP, with enforcement)	2	12	73	217
LRSTM (ad-hoc)	8	151	886	2411
LRSTM (MOP)	7	69	167	452
AmbientTalkST (ad-hoc)	6	37	163	390
AmbientTalkST (MOP)	2	26	115	213
Active Objects (MOP)	3	15	73	148
CSP+π (MOP)	5	16	39	61
MOP base system	5	170	1068	2767
AmbientTalkST (MOP*)	2	38	213	638
MOP base system*	5	206	1163	3016

* including duplicated code for variadic argument emulation

As the results in Tab 4 show, the concurrency constructs and their guarantees can be expressed concisely. As pointed out in Sec. 4.1, the 31 LOC of the ad-hoc agent implementation do not include any enforcement, while the additional 42 LOC of the MOP-based one include the domain and its guarantee enforcement. The more than 80% reduction of LOC for the MOP-based LRSTM comes mostly from avoiding the need for a custom bytecode transformation, which is already included in the MOP base system. The MOP-based AmbientTalkST implementation is with 115 LOC also slightly more concise than the ad-hoc version with 163 LOC. The MOP further enables the implementation of active objects in 73 LOC, and a minimal CSP in 39 LOC, both enforcing their full semantics.

Note that Smalltalk does not support variadic methods, which currently results in replicating code for the handler of method execution requests. We duplicate the code for 0 to n parameters manually, which could be avoided with a template or macro mechanism. For completeness, we also give the numbers including the duplicated code for variadic methods.

The MOP base system is with 170 methods and 1068 LOC still manageable. This core provides the main mechanisms for the unified reusable abstraction of our MOP and simplifies the implementation of concurrency constructs.

4.4 Performance

We assess the overhead of our prototype by comparing the performance of the ad-hoc with the MOP-based implementations by using AmbientTalkST and LRSTM. We concentrate on AmbientTalkST and LRSTM, because these two provide in both implementations the same semantics, while the other concurrency constructs do not enforce their semantics in the ad-hoc implementation. Furthermore, since our prototype is meant to demonstrate the expressiveness of our MOP-based approach, we will concentrate on kernel benchmarks to get a first impression of the performance impact. Thus, we use adapted versions of four kernel benchmarks from the *Computer Language Benchmarks Game*[2] for general assessment. Additional microbenchmarks then allow to assess which part of the MOP influences performance most.

Our methodology is derived from the advice of Georges et al. [8]. All benchmarks are executed 100 times on the CogVM. We measure steady-state performance to account for the just-in-time compiler. The used machine runs OS X 10.6 with Intel Xeon E5520 processors. The benchmarks use only a single core to avoid noise in the measurements.

Fig. 3 depicts the results as a box plot. It shows the performance ratio of ad-hoc/MOP-based. Ideally, the implementations would perform on-par, i. e., would be at the dashed ideal line with a value of 1. However, the benchmarks show that the ad-hoc implementation of AmbientTalkST outperforms MOP-based one for all but one kernel benchmark. The microbenchmarks point out that the MOP has an impact on all method invocations and array as well as instance variable

[2] `http://shootout.alioth.debian.org/`

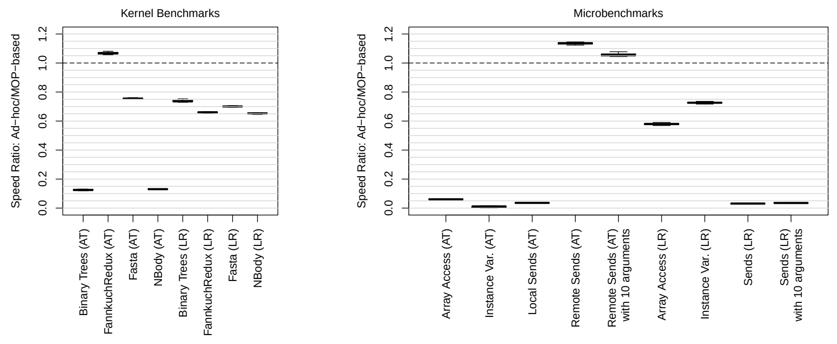

Fig. 3. Boxplot of Speed Ratio Ad-hoc/MOP-based for AmbientTalkST (AT) and LRSTM (LR): The ideal line (dashed) is at 1. While higher means better, everything below indicates that the MOP-based implementations are slower than the ad-hoc ones.

accesses. The implementation of asynchronous (remote) message sends to other actors is however more efficient. This explains also the behavior observed for the FannkuchRedux kernel, which has almost no instance variable accesses, but a high amount of inter-actor message sends. While the proxy-based solution of the ad-hoc implementation has a higher overhead on remote sends to other actors, it does not incur any cost for local sends and variable accesses.

The LRSTM kernel benchmarks show an overhead of 26–34% for the MOP-based solution. Here the impact of the message sends overhead is smaller since the performance impact of array and instance variable access is not as high as for the AmbientTalkST implementation, since LRSTM already modifies them.

While our prototypical implementation is able to enforce the desired semantics, it comes with a high performance cost. Therefore, we will discuss in the next section how our approach could be implemented in a VM.

5 Discussion and Performance Perspectives

While performance is an issue, the current MOP also has limitations since it only regards the owner of an object as the entity defining the semantics for interaction. This approach does not offer any mechanism to control the interaction of different semantics. Thus, they need to be defined directly as part of the concurrency model for all possible combination with other concurrency models. However, for instance the possible interaction between actor-like models and STM systems cover a wide design space that needs to be carefully considered to achieve appropriate semantics.

Other types of guarantees, for instance deadlock freedom, are also problematic. Deadlock freedom is usually guaranteed by providing only non-blocking operations in a language. Thus, it is not clear how such a guarantee could be given in a system that provides arbitrary blocking mechanisms to other languages.

Our prototypical implementation does not yet rely on runtime support. However, we expect to be able to reduce the overhead to an acceptable level by applying the following techniques. Most promising seems to be the implementation techniques used by Hoffman et al. [10]. They use the hardware memory protection support to enable the isolation of program components in an otherwise shared memory model. It could be used to reduce the performance overhead for a wide set of concurrency models, too. Especially actor-like and CSP-like models could benefit from a protection model where local operations do not impose any overhead. While memory protection is relevant for field accesses, the overhead of customized semantics of method invocation mechanisms is also significant. This could be solved with the INVOKEDYNAMIC infrastructure [26] included in current JVMs. Furthermore, tracing compilers [6] are known to enable the elimination of expensive guarding checks to ensure semantics effectively.

6 Related Work

Classic MOPs were an inspiration for our approach. However, the CLOS [12] and also current Smalltalk MOPs [29] are based on the notion of classes or metaclasses that define the semantics of their subclasses. In contrast to that, our model is based on ownership. Thus, a domain defines the concurrency semantics for its objects and is orthogonal to the classic classification-based schemes.

Our survey is based on two surveys [2, 25] and thus closely related to them. However, we are not aware on any survey or approach that enables a runtime system to directly support the semantics of multiple concurrency models.

As mentioned in Sec. 5 the work of Hoffmann et al.[10] is closely related in the field of VMs. They enable the isolation of components enabling the enforcement of memory access constraints inside an application. As argued, this is a promising technique to approach the performance implications of our approach.

The prototype implementation of our approach is closely related, and inspired by the work of Renggli and Nierstrasz [21]. While they use it to implement an STM, we use the same ideas to enable our MOP which in return allows use to enforce different language semantics. A similar transformation based approach was also used to implement an STM for Java [30].

7 Conclusion

Our survey showed that most parallel constructs can be provided in terms of libraries without sacrificing neither performance nor semantics. Concurrency constructs however, often suffer from either the loss of semantic integrity or a high performance penalty when implemented in terms of libraries. Based on this survey we identified the requirements for the support of such concurrency mechanisms in a multi-language virtual machine. A VM needs a mechanism to managed mutation/execution with regard to ownership, as well as support for leveled reflection to guarantee enforceability of language semantics.

Based on these requirements, we designed an ownership-based MOP and demonstrated that it is a *unifying mechanism* to enforce the semantics for a wide range of concurrency models in a concise manner. The main concepts of the MOP are reification of field accesses, message sends, and object ownership. The distinction between language-level and meta-level reflection enables us further to guarantee concurrency semantics even when meta-programming is used.

The performance of our bytecode-transformation-based prototype shows that the performance impact is significant and actual VM support is required to achieve acceptable performance. However, it also shows the unifying potential of the MOP. It enabled us to implement active objects, actors, agents, CSP, and STM in less than 500 LOC in total. With the example of agents, we were also able to demonstrate how a concurrency abstraction can be easily extended to provide desirable engineering properties.

For our future work, we identified a number of promising techniques that can be used to implement our MOP more efficiently as part of a VM.

Acknowledgments. Stefan Marr is supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

References

1. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proc. of OOPSLA'04. pp. 331–344. ACM (2004)
2. Briot, J.P., Guerraoui, R., Lohr, K.P.: Concurrency and distribution in object-oriented programming. ACM Computing Surveys 30(3), 291–329 (September 1998)
3. Budimlic, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Multi-core implementations of the concurrent collections programming model. In: he 14th Workshop on Compilers for Parallel Computing (January 2009)
4. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: Cloud computing for everyone. In: Proc of SOCC'11. pp. 16:1–16:14. ACM (2011)
5. Demsky, B., Lam, P.: Views: object-inspired concurrency control. In: Proc. of ICSE'10 (2010)
6. Gal, A., Probst, C.W., Franz, M.: Hotpathvm: An effective jit compiler for resource-constrained devices. In: Proc. of VEE'06. pp. 144–153. ACM (2006)
7. Gelernter, D.: Generative communication in linda. ACM TOPLAS 7, 80–112 (January 1985)
8. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. SIGPLAN Not. 42(10), 57–76 (2007)
9. Halstead, Jr., R.H.: Multilisp: a language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. 7, 501–538 (October 1985)
10. Hoffman, K.J., Metzger, H., Eugster, P.: Ribbons: A partially shared memory programming model. SIGPLAN Not. 46, 289–306 (Oct 2011)
11. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: A comparative analysis. In: Proc. of PPPJ'09. pp. 11–20. ACM (2009)

12. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT (1991)
13. Lavender, R.G., Schmidt, D.C.: Active object: An object behavioral pattern for concurrent programming. In: Pattern Languages of Program Design 2, pp. 483–499. Addison-Wesley Longman Publishing Co., Inc. (1996)
14. Lea, D.: A java fork/join framework. In: JAVA '00: Proceedings of the ACM 2000 conference on Java Grande. pp. 36–43. ACM (2000)
15. Lublinerman, R., Zhao, J., Budimlić, Z., Chaudhuri, S., Sarkar, V.: Delegated isolation. SIGPLAN Not. 46, 885–902 (Oct 2011)
16. Lämmel, R.: Google's mapreduce programming model - revisited. SCP 70(1), 1 – 30 (2008)
17. Marr, S., Haupt, M., D'Hondt, T.: Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support. In: Proc. VMIL'09 Workshop. pp. 3:1–3:2. ACM (October 2009), (extended abstract)
18. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in e as plan coordination. In: Nicola, R.D., Sangiorgi, D. (eds.) Symposium on Trustworthy Global Computing. LNCS, vol. 3705, pp. 195–229. Springer (April 2005)
19. Morandi, B., Bauer, S.S., Meyer, B.: Scoop - a contract-based concurrent object-oriented programming model. In: Müller, P. (ed.) Advanced Lectures on Software Engineering, LASER Summer School 2007/2008. LNCS, vol. 6029, pp. 41–90. Springer (2008)
20. Nordlander, J., Jones, M.P., Carlsson, M., Kieburtz, R.B., Black, A.P.: Reactive objects. In: Symposium on Object-Oriented Real-Time Distributed Computing. pp. 155–158 (2002)
21. Renggli, L., Nierstrasz, O.: Transactional memory for smalltalk. In: ICDL '07: Proceedings of the 2007 international conference on Dynamic languages. pp. 207–221. ACM (2007)
22. Scholliers, C., Tanter, E., Meuter, W.D.: Parallel actor monitors. In: 14th Brazilian Symposium on Programming Languages (2010)
23. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) Proc. of ECOOP'10, LNCS, vol. 6183, pp. 275–299. Springer (2010)
24. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. of PODC'95. ACM (1995)
25. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM CSUR 30, 123–169 (June 1998)
26. Thalinger, C., Rose, J.: Optimizing invokedynamic. In: Proc. of PPPJ'10. pp. 1–9. ACM (2010)
27. Van Cutsem, T., Miller, M.S.: Proxies: Design principles for robust object-oriented intercession apis. In: Proc. of DLS'10. pp. 59–72. ACM (October 2010)
28. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In: Proc. of SCCC'07. pp. 3–12. IEEE CS (2007)
29. Verwaest, T., Bruni, C., Lungu, M., Nierstrasz, O.: Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In: Proc. of OOPSLA'11. pp. 959–972 (2011)
30. Ziarek, L., Welc, A., Adl-Tabatabai, A.R., Menon, V., Shpeisman, T., Jagannathan, S.: A uniform transactional execution environment for java. In: Proc. of ECOOP'08. pp. 129–154 (2008)