An Update on the Method Combinations MOP

Didier Verna
EPITA Research Laboratory
Le Kremlin-Bicêtre, France
didier@didierverna.net

Abstract

We report on our progress implementing a metaobject protocol for Common Lisp method combinations. This work was started in 2018 with a first implementation in SBCL, and then refined in 2023. Since then, we have two additional working implementations: one for ABCL and another for ECL. This paper describes the aforementioned ports, as well as a couple of refinements that were added since 2023.

CCS Concepts

• **Software and its engineering → Object oriented languages**; **Extensible languages**; **Polymorphism**; *Inheritance*; *Classes and objects*; *Object oriented architectures*; *Abstraction, modeling and modularity*; *Incremental compilers*; *Runtime environments*.

Keywords

Object-Oriented Programming, Common Lisp Object System, Metaobject Protocol, Generic Functions, Dynamic Dispatch, Polymorphism, Multi-Methods, Multiple Dispatch, Method Combinations, Orthogonality

ACM Reference Format:
Didier Verna. 2026. An Update on the Method Combinations MOP. In *Proceedings of the 19th European Lisp Symposium (ELS'26)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.20024295

1 Introduction

Method combinations are a very powerful, yet somewhat obscure feature of CLOS [1–4], the Common Lisp [7] Object System. Very little is actually standardized about them (merely a base class and a macro). Unfortunately, the CLOS MOP [5] does not improve the situation (it sometimes does the contrary).

Between 2018 and 2023, we developed a metaobject protocol for method combinations and originally implemented it in SBCL. The idea behind this work is to offer a portable specification for method combinations, in terms of structure as well as behavior. Since 2023, we have added a couple of refinements to the system, and we have adapted the proposed architecture to two new implementations: ABCL and ECL. In this paper, we report on these recent developments.

The paper is organized as follows. Out of a concern for self-containment, Section 2 summarizes again the proposed implementation. Sections 3 and 4 describe the current state of ABCL (ECL respectively) vis-à-vis method combinations, and provide some details on porting their CLOS/MOP implementation to the new infrastructure. Section 5 analyzes the performance impact of the new implementation. Sections 6 and 7 address two marginal issues, and finally section 8 compares our approach to some related work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'26, Kraków, Poland
© 2026 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.20024295

2 Overview

Our rationale and motivation for this work have been originally described in [8], and then precised in [9]. Out of a concern for self-containment, we briefly summarize the proposition here, including a couple of small refinements that were added since 2023. Essentially, our design fills the gap between the existence of the `method-combination` base class, and the `define-method-combination` macro.

2.1 Structure

In terms of structure, our proposition establishes one regular class hierarchy from which new method combinations inherit, and one metaclass hierarchy in terms of which new method combinations are implemented (Figure 1).

A new method combination type is reified as a new class, inheriting from either `short-` or `long-method-combination`, and implemented as either `short-` or `long-method-combination-type`. Conceptually, all method combination types have a `type-name` and a `lambda-list` (`()` for the standard method combination, `&optional (order :most-specific-first)` for the short ones, and arbitrary for the long ones). Additionally, short method combination types are associated with a specific `operator`, and behave according to `identity-with-one-argument`.

A method combination, as used by a generic function, is reified as an instance of a method combination type for a specific set of `options`. The options in question must conform to the method combination type's lambda list (again, `nil` for the standard method combination, just one kind of `order` for short ones, and arbitrary for long ones).

2.2 Behavior

Method combination types are created by calling `define-method-combination` as usual. However, the macro is extended with two options allowing to specify exactly which class to inherit from, and which metaclass to use for implementation. The syntax and default values for these options depend on whether the short or long form of `define-method-combination` is used. They also make the macro still usable if the method combination / method combination type hierarchies are extended. The validation protocol is updated to check for consistency: short (resp. long) method combinations can only be superclasses of short (resp. long) method combination type classes. Finally, new method combination types may be globally accessed with the following new function:

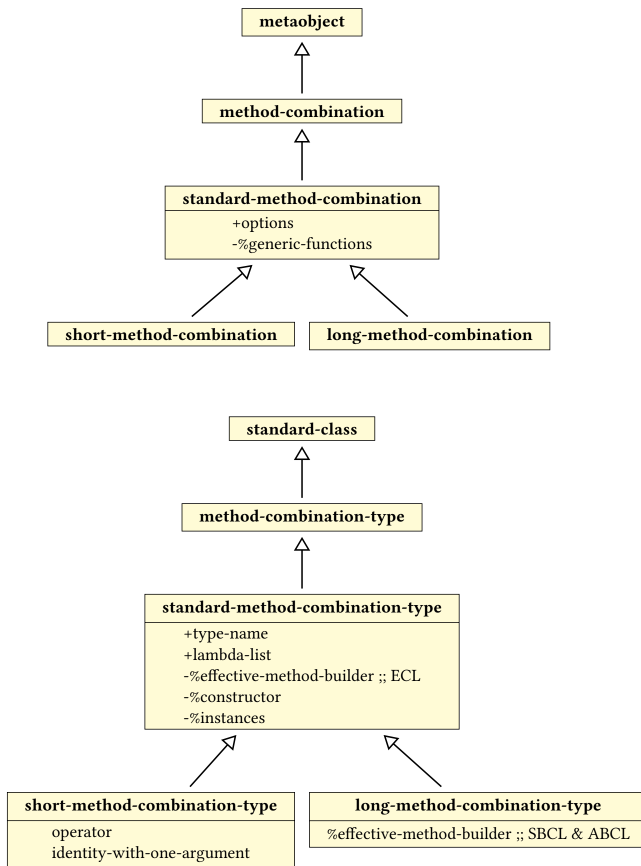

Figure 1: Method Combination Hierarchies

```
find-method-combination-type
  (name &optional (errorp t))
"Find a NAMEd method combination type.
If ERRORP (the default), signal an error if no such
method combination type is found.
Otherwise, return NIL."
```

In general, method combination objects (as used by a generic function with a specific set of options) are created as a result of calling the standard function `find-method-combination`. This process is specified more precisely in our proposal however. The instantiation of a method combination type is done by calling the `%constructor` function memoized in the method combination type itself. Additionally, method combination types maintain a cache of `%instances` mapping a specific set of options to a unique method combination object, shared across all generic functions using the same set of options (Figure 1). Because the specification of `find-method-combination` is odd (the generic function argument is useless), we also provide an alternative (ordinary) function doing essentially the same thing:

```
find-method-combination-instance
  (name &optional options (errorp t))
"Find a method combination object of type NAME,
for OPTIONS. If ERRORP (the default), signal an error
if no NAMEd method combination type is found.
Otherwise, return NIL.

Note that when a NAMEd method combination type exists,
asking for a new set of (conformant) OPTIONS will
always instantiate the combination again, regardless
of the value of ERRORP."
```

Consistency between method combinations and generic functions using them is ensured in the following way. Method combination objects maintain a cache of `%generic-functions` using them (Figure 1). Generic functions handle their own addition or removal from these caches when they are created, or when their method combination change. This is done via the standard (re)initialization protocols.

When a method combination type is changed on the other hand (typically by calling `define-method-combination` again), existing method combination instances are updated via a call to `change-class`. This has the effect of calling `u-i-f-d-c`[1]. A method is installed on this function to inform every client generic function of the change, through a new protocol: `(u-g-f-f-r-m-c`[2] `gf old-mc new-mc)`.

Note that in our proposed architecture, redefining a method combination type may lead to change its metaclass. However, the metaobject protocol stipulates that it is not possible to change the class of a class metaobject. Because of that, method combination types redefinition is done by creating a brand new class rather than updating the previous one. This explains why, as described earlier, existing instances are explicitly notified via `change-class`, rather than implicitly, as the result of a call chain eventually leading to a call to `make-instances-obsolete`.

3 Abcl Port

Abcl[3] is a hybrid Lisp implementation which runs on the JVM. Its implementation is a mixture of core functionality written in Java, and the rest in Lisp directly. In particular, some elementary Clos components are grounded in the Java world. This notably includes the implementation of `standard-class` and `standard-object`, as well as the slot definition and funcallable objects machineries.

3.1 Status

Abcl's Clos/Mop implementation is originally based on Closette, the simplified version described in the "Art of the Metaobject Protocol" (Amop) [5]. As such, it is similar to the Pcl implementation, and consequently also to pre-2018 Sbcl, as described in [8] and [6].

Abcl defines three method combination classes: the one required by the standard (`method-combination`), plus two subclasses corresponding to the short and long forms of `define-method-combination`. These classes provide slots for storing information, not only about the method combinations themselves (such as a short method combination's operator), but also about their *usage*, via an `options` slot (the options specified by a generic function using the method combination). Thus, this hierarchy fails to make the distinction between method combinations definition, and use.

[1] `update-instance-for-different-class`
[2] update-generic-function-for-redefined-method-combination
[3] https://armedbear.common-lisp.dev/

Note that even though it is not a standard requirement, there is no intermediate `standard-method-combination` class in this hierarchy. The standard method combination is created as a *direct instance* of the class `method-combination`, which is non-conformant.

When a new method combination is defined, one of the aforementioned classes is instantiated, and the resulting object is stored as a `method-combination-object` property on the method combination's name. As in the case of pre-2018 SBCL, `find-method-combination` creates new method combination objects every time it is called, even for the same method combination with the same set of options. It does so essentially by creating a copy of the original object, and updating the `options` slot with the requested options. As a consequence, there is no consistency between method combinations and generic functions: identical method combination uses are duplicated, and method combination updates are not propagated to the generic functions using them.

3.2 Upgrade

Because ABCL's CLOS/MOP implementation is close to PCL, upgrading to our proposed infrastructure is very similar to what we did for SBCL [8, 9]. The general principle is to establish as little as needed during bootstrap, and inject the full-blown machinery afterwards.

ABCL originally stores the standard method combination in a constant called `+the-standard-method-combination+`. The CLOS/MOP machinery short-circuits a lot of the general code paths to provide versions optimized for the standard method combination. Besides, the standard method combination is the only one needed during bootstrap (note that SBCL needs an `or` method combination as well).

During bootstrap, we provide an ephemeral class called `early-method-combination`, and create a half-baked standard method combination object out of it. The only useful thing this class does is to let the early standard method combination keep track of all the generic functions using it via a `%generic-functions` cache similar to the one which will be installed later on (Figure 1).

The `+the-standard-method-combination+` constant is turned into a variable and initialized with the early standard method combination. After bootstrap, the full-blown infrastructure is installed, the real standard method combination is created and all existing generic functions are updated to use it.

The fundamental use for method combinations is in `compute-effective-method`. ABCL originally provides a hard-wired code path for the standard method combination, and another one for all short method combinations. Each long method combinations possess a "long method combination function" (again, similarly to SBCL) which is used to compute effective methods. This implementation is retained, with the small difference that long method combination functions are now stored in the method combination type objects directly, in the `%effective-method-builder` slot (Figure 1).

4 ECL Port

ECL[4] is a Common Lisp implementation with the ability to compile to C/C++, making it in turn easy to create standalone executables. The CLOS/MOP implementation in ECL is *not* based on PCL, and is in fact very different from it. An overview of the state of ECL relative to method combinations has already been given in [6]. We provide a slightly different one here.

4.1 Status

ECL defines its hierarchy of classes in a very declarative way. A constant called `+class-hierarchy+` provides the list of all standardized classes (including MOP ones), and each class definition is accompanied with another constant describing its slots (for example, `+method-combination-slots+`). Note that this hierarchy description is "flattened", in the sense that each class must explicitly provide its complete list of slots, including the inherited ones.

The hierarchy reveals that there is only one class for method combinations, namely `method-combination`. This class has three slots for storing the combination's `name`, a set of `options`, and a so-called `compiler`, which, in fact, is a procedure for computing effective methods (this is exactly what we call an `%effective-method-builder` in our infrastructure). Defining a new method combination does not create any object. Instead, such a "compiler function" is created and stored in a hash table, the keys of which are method combination names.

The standard method combination is represented by a built-in compiler function called `standard-compute-effective-method`. Contrary to PCL-based implementations however, there is no distinction between short and long method combinations: short method combinations are implemented in terms of long ones. One notable consequence is that every short method combination gets its own compiler function. Contrast this with PCL-based implementations in which there is one effective method builder for the standard method combination, a single one for *all* short method combinations, and one for *each* long method combination.

Every time `find-method-combination` is called, a new instance of `method-combination` is created, and filled with the method combination's name and retrieved compiler function, plus the set of options specifically required with this call. It thus follows that this behavior is non-conformant, as the `method-combination` class is supposed to be abstract. Additionally, and as in the case of pre-2018 SBCL and current ABCL, there is no consistency between generic functions and method combinations: identical method combination uses are duplicated, and method combination updates are not propagated to the generic functions using them.

4.2 Upgrade

In spite of the important differences between ECL's implementation of the CLOS MOP and the PCL tradition, porting our new infrastructure to it has proven easier. Our complete classes hierarchies can be installed without worrying about bootstrapping issues (that is, without any particular injection technique), in the same declarative fashion as the rest of the system.

One notable difference with SBCL or ABCL, visible in Figure 1, is that the `%effective-method-builder` slot appears in the class `standard-method-combination-type` for ECL, instead of in `long-method-combination-type` for the other implementations. This, again, is due to the fact short method combinations are expressed in terms of long ones.

Early in the process, the standard method combination is created by hand and its `%effective-method-builder` slot is set to

[4] https://ecl.common-lisp.dev/

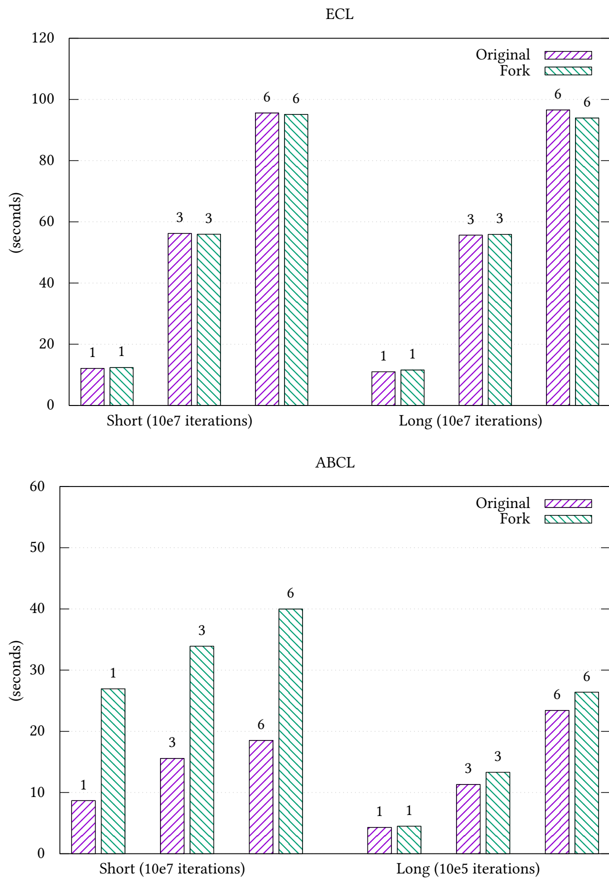

Figure 2: `compute-effective-method` Performance

`standard-compute-effective-method`. The creation of the standard short method combinations is deferred until after bootstrap.

5 Performance

Since the creation or modification of method combinations is bound to happen very rarely, we are *not* interested in benchmarking that. Instead, we want to know the impact of our infrastructure change on the *usage* of method combinations, which typically occurs when `compute-effective-method` is called.

To that aim, we conducted the same experiments as for SBCL in 2023 [9]: we timed the execution of `compute-effective-method` (ten million calls in a row, except in one case; see below) on one generic function using a short method combination, and another one using a long method combination, each time with one, three, and six applicable methods . Those experiments were run on the original implementations of ABCL and ECL, as well as on our forked versions. The results of these experiments are show in Figure 2.

Note that since the way effective methods are computed by each implementation is not affected by our architectural changes, the only possible impact comes from the way method combination information is accessed (in other words, slot access).

5.1 ECL

First of all, ECL exhibits identical performance for short or long method combinations. This is explained by the fact that short method combinations are implemented in terms of long ones, so the same code paths apply. With our new infrastructure in place, the performance remains practically identical, to the point that the small differences observed remain inconclusive. It is safe to say that the new infrastructure has no impact performance-wise.

5.2 ABCL

The case of ABCL is more difficult to analyze. Even though the point is *not* to compare it with ECL, two observations are still worth making. First of all, short method combinations are handled in a better way (note that the upper time limits in the charts are different: 120s for ECL against 60 for ABCL). This, again, is explained by the fact that they are special case'd in ABCL as opposed to being implemented as long ones in ECL. On the other hand, long method combinations are so much slower that we had to reduce the number of iterations to remain within a manageable time frame (we ended up using 10^5 iterations instead of 10^7).

The second important point is that our experiments are not very stable. The same benchmarks give very different timings the first time (much faster), than on subsequent runs within the same runtime environment. As a consequence, we ran the benchmarks several time before collecting the results. Even then, we observed some variations across the different runs that may be considered not so insignificant. In other words, the reliability of these experiments is questionable.

Still, there is one observation that we can make with enough confidence: the performance impact in the case of short method combinations is important. Our current hypothesis is that slot access is a costly operation in ABCL. In the original version, method combination instances carry all the required information, whereas in our infrastructure, the same information is stored in the method combination type. This means that `compute-effective-method` requires at least an additional call to `class-of` before proceeding.

6 Alternative Method Combinations

Alternative method combinations is an experimental feature which has been successfully implemented in SBCL [8, 9]. In short, the idea is to be able to call the same generic function with different method combinations efficiently (meaning, without having to reinitialize it at every call), simply by maintaining a cache of discriminating functions. A typical usage example would be to have a generic function concatenate the results of its primary methods, either via `append` of `nconc`, depending on the situation.

In our previous writings, we mentioned the fact that this feature is highly experimental, since neither Common Lisp nor the CLOS/MOP offer the necessary guarantees. More specifically, this feature can only be successfully implemented under the following conditions.

(1) The generic function's discriminating function can be retrieved and stored for later use. Note that the MOP specifies this existence of a discriminating function writer (`set-funcallable-instance-function`), but no reader.

(2) Effective methods are not cached, or if they are, the cache depends on the discriminating function (for example, the discriminating function closes over it) or is retrievable (as the discriminating function), for storage and later reuse.

Unfortunately, this feature cannot currently be made available in either Abcl or Ecl. In the case of Ecl, there is no way to access the discriminating function at the Lisp level (it belongs to the C level). In the case of Abcl, the discriminating function is readable, but generic functions maintain an independent global effective methods cache, which is not. This cache is in fact implemented at the Java level, and the only possible action from Lisp is to reinitialize it (obviously required when a generic function is reinitialized). In either case, implementing alternative method combinations efficiently will require substantial modifications to both of these compilers.

7 Documentation

Documentation is a corner case that we took the opportunity to refine in this work. The Common Lisp standard defines three documentation accessors for method combinations:

```
;; Both readers and SETF writers as follows.
(documentation symbol 'method-combination)
(documentation mc-object t)
(documentation mc-object 'method-combination)
```

It follows from the specification that documentation may diverge. Indeed, it is possible to independently read and write a documentation string on a method combination's name (a symbol), and on method combination objects. The standard does not mandate any consistency between those, and indeed experience shows a variety of divergent behaviors.

7.1 Status

In Sbcl, the original method combination's documentation is available via the three possible readers. Subsequently, different documentation strings set on a method combination's name or object become independent. Note that when accessing the documentation of a method combination object, `eql`-specializing on `t` or `'method-combination` is equivalent in Sbcl.

In Abcl and Ecl, there is no `documentation` method specialized on method combination objects. Thus, there is no divergence, but these implementations are not conforming.

7.2 Upgrade

The Common Lisp standard's wording about documentation leaves some room for interpretation. The three cases related to method combinations are as follows.

```
(documentation symbol 'method-combination)
```

Access the documentation string of the method combination whose name is `symbol`*.*

```
(documentation mc-object t)
```

Access a documentation string specialized on the class of the argument `mc-object` *itself.*

```
(documentation mc-object 'method-combination)
```

Access the documentation string associated with `mc-object`*.*

In this last case, the exact meaning of "associated with" is not clearly specified. Our proposition is that the the two accessors `eql`-specialized on `'method-combination` get the same behavior, namely to access the method combination name's documentation. Note that this is different from what Sbcl currently does, since in that implementation, it is the two accessors specialized on method combination objects that get the same behavior.

When a new method combination is defined, we install the provided documentation on its name as well as in the newly created class (the method combination type). Remember that method combination types are in fact extended standard classes (Figure 1) so the class documentation mechanism works out of the box. But since we now have both method combinations and method combination types, we can define two more documentation accessors. The behavior of all accessors can thus be summarized as follows.

```
(documentation symbol 'method-combination)
(documentation mc-object 'method-combination)
(documentation mc-type 'method-combination)
```

Access the documentation string attached to `symbol`*, a method combination type name, where* `symbol = (type-name mc-type)`*, and* `mc-type = (class-of mc-object)`.

```
(documentation mc-object t)
(documentation mc-type t)
```

Access the documentation string of `mc-type`*, where* `mc-type = (class-of mc-object)`.

In other words, it is still possible for the documentation to diverge, but the divergence may now only occur between the documentation attached to the name, and the one attached to the method combination type class. This situation is very unlikely to happen, but if intentional, a possible use could be to document a method combination's usage (by attaching a documentation string to the name), and implementation (by attaching a documentation string to the type class).

8 Related Work

The underspecification and ambiguities around method combinations that we address in this work get a different treatment in Sicl[5]. The adopted solution in that implementation has been published in 2020 [6].

This work differs from ours in two important ways. First of all, method combination objects are reified by instantiating a *single* class called `standard-method-combination`, whereas we choose to remain closer to the Pcl tradition, by providing two additional subclasses: `short-` and `long-method-combination`. Second, method combination types are reified by instantiating a *single* class called `method-combination-template`, whereas we choose to dynamically create new classes for each new method combination type.

In Sicl, each method combination object maintains a reference to its original template. In our solution, each method combination object is associated with its type by means of its metaclass. In other words, Sicl's solution is not architected around a metalevel,

[5] https://github.com/robert-strandh/SICL

but around mere aggregation. In both cases, accessing a method combination type's property requires two indirections:

```
(property (template mc-object)) ;; SICL
(property (class-of mc-object)) ;; this work
```

We tend to favor our solution because of its tighter integration with the Mop, and because of its proximity to Pcl upon which many implementations are based. As stated in 2023 [9], a tighter integration with the Mop makes the architecture extensible, and thus facilitates further experimentation (for example, with new forms of method combination besides short and long ones, and specific optimizations on `compute-effective-methods`).

9 Conclusion and Perspectives

In this paper, we have described our continued work implementing a fully specified Mop for method combinations. The benefits of this approach have already been described in detail [9]. In addition to some refinements over the original design, we now have three working implementations, namely for Sbcl, Abcl, and Ecl. Each version can currently be found in our forks of the original compilers, in branches that are regularly updated with the main ones[6].

In addition to providing a fully specified and extensible design for method combinations across multiple platforms, we have seen that this work also has the benefit of fixing several non-conformance issues as well as some behavioral inconsistencies similar to the ones originally observed in Sbcl in 2018. Incidentally, the porting to Abcl and Ecl led to the discovery of two important bugs that should be fixed mainstream, or will soon be. In Abcl, `subtypep` was working properly on symbolic type specifiers for numbers (*e.g.* `'single-float`) but not on the corresponding built-in classes. In Ecl, it was possible to allocate early objects (during bootstrap) of the wrong size.

In the case of Abcl, we still have a glitch in the post-bootstrap injection phase preventing one generic function to be upgraded to the full-blown standard method combination. This problem is likely to be due to a metacircularity of some kind. We are still trying to figure it out, but otherwise, the system is fully functional. Our testing repository[7] has been updated with the support of the two new compilers, and now features 144 unit tests that all pass for each supported vendor.

In addition to the perspectives and future work described in [9], several additional points need to be made here. First of all, the performance impact on short method combinations in Abcl needs to be further investigated, and its acceptability asserted. If it is indeed the case that slot access is very costly, then we can always compensate by replicating the required method combination type properties directly into the method combination objects, without sacrificing the design. Another area of improvement (for all implementations) lies in handling potential consistency issues between method combinations lambda lists, and the options actually requested by generic functions. This is actually a central concern in the Sicl solution [6] and we can certainly get inspiration from it. Finally, we also have the intention of looking into porting our infrastructure to other vendors (Ccl and Cmu-CL notably).

[6]https://github.com/didierverna/sbcl/tree/method-combination-types
https://github.com/didierverna/abcl/tree/method-combination-types
https://gitlab.com/didierverna/ecl/-/tree/method-combination-types?ref_type=heads
[7]https://github.com/didierverna/els2023-method-combinations

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988. ISSN 0362-1340.

[2] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In *European Conference on Object Oriented Programming*, pages 151–170, 1987.

[3] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991. ISSN 0001-0782.

[4] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to Clos.* Addison-Wesley, 1989. ISBN 0-20117-589-4.

[5] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[6] Robert Strandh. Representing method combinations. In *ELS 2020, the 13th European Lisp Symposium*, Zurich, Switzerland, April 2020. ISBN 9782955747445. doi: 10.5281/zenodo.3747553.

[7] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[8] Didier Verna. Method combinators. In *11th European Lisp Symposium*, pages 32–41, Marbella, Spain, April 2018. ISBN 9782955747421. doi: 10.5281/zenodo.3247610.

[9] Didier Verna. A MOP-based implementation for method combinations. In *ELS 2023, the 16th European Lisp Symposium*, pages –, Amsterdam, Netherlands, April 2023. ISBN 9782955747476. doi: 10.5281/zenodo.7818680.