Abstract. Run-time MetaObject Protocols (MOPs) are reflective systems that allow objects to be controlled at run time by one or many metaobjects. These metaobjects can then alter the semantics of the execution for the objects they control. Jumping to the meta level to alter the semantics is powerful but indeed fairly costly. It is therefore desirable to let users fine-tune their run-time MOP according to their specific needs. This is exactly what we are investigating in this PhD thesis: how to design and build an open run-time MOP.

But although our intuition (or faith) never fails, it remains hard to definitely answer the realistic question of “what is this really good for?” Therefore, we are also investigating applications of run-time MOPs that differentiate them from other program transformation approaches. Since we are worried about the applicability of run-time MOPs, all our work is done with the Java programming language, in a fully portable way.

1 What are run-time MOPs?

Run-time MetaObject Protocols (MOPs) are reflective systems that allow objects to be controlled at run time by one or many metaobjects. These metaobjects can then alter the semantics of the execution for the objects they control. A classical example is that of a metaobject that controls method invocations on an object to trace them on a debug output. An object controlled by one or more metaobjects is called a reflective object. A reflective class is a class whose instances are reflective objects. From the point of view of the metaobject, the object it controls is called the base object.

Characterizing a run-time MOP starts by specifying which language mechanisms are reified and therefore can be controlled at the metalevel (its expressiveness). Some simple run-time MOPs only offer the possibility to control method invocation on the receiver side, and some more powerful ones make it possible to control field accesses and object creation. A more advanced expressiveness criterion concerns the way a same mechanism can be handled (e.g., there might be
several ways to handle method invocations, for instance by introducing method categories).

In run-time MOPs, the link between a base object and its metaobject(s) is called the *causal connection link*. This link exists at run time, as illustrated in figure 1. The causal connection link can mainly be characterized upon three criteria: its cardinality (can a metaobject control several base objects and can a base object be controlled by several metaobjects), its implementation technique (is it event- or invocation-based, implemented by a unique aggregation link or several, etc.), and its reification degree, that is, to what extent the controlled mechanism is reified. This last point greatly influences what metaobjects can do.

Finally, another criterion to characterize run-time MOPs is their *implementation approach*. That is, how the hooks are introduced into the original code so that an occurrence of a language mechanism implies a jump at the metalevel and a reification of the occurrence. There are three kinds of run-time MOPs: some are based on a specialized virtual machine, some are based on source code transformation and others based on bytecode transformations. The implementation approach influences the portability of the MOP, and its applicability (e.g., if source code is required, then the MOP is not applicable in systems where binary code is uploaded dynamically).

### 2 The Appropriate Candidate

Since we are interested in concretely validating the usefulness of run-time MOPs, we have to discard what we consider *unrealistic* proposals of such MOPs. Amongst the different implementation approaches mentioned aboved, we consider VM-based MOPs and source-code-based MOP unrealistic, because (i) you cannot expect people to use a dedicated VM and (ii) source code is not always available.
Within the bytecode-based MOPs, we need to have at hand a “perfect” run-time MOP. Existing run-time MOPs are limited in two ways:

- **They are closed software systems.** Though they allow the implementation of open and adaptable systems, they are themselves closed. This non-openness of run-time MOPs is annoying: indeed, jumping to the meta level to alter the semantics of execution is powerful but costly. Consequently, jumping to the meta level should be done only when it is really needed. It is therefore required to allow the user of a MOP to fine-tune it.

- **Their expressiveness is limited.** Though theoretically run-time MOPs should make it possible to alter any language mechanism, they are usually limited to altering method invocations on the receiver side, and possibly fields accesses and object creations. However there is a lot of other mechanisms that could be altered: casts, array accesses/creations, local variables, synchronization, serialization, etc.

In this PhD thesis, we are investigating on the definition and implementation of an open run-time MOP that enables fine-tuning and that offers an almost complete expressiveness. We are working on a prototype of such MOP, Reflex [4, 3]. When achieved, Reflex will allow its users to fine-tune:

- **Which types/instances are made reflective.** Reflex can transform a whole type or sub-hierarchy of types at load time or can generate implicit reflective subclasses on demand at run time, making it possible to have reflective and non-reflective objects of a same type coexist in a running application.

- **Which base events are controlled at the meta level.** To achieve a high expressiveness, we are working on a semantic inliner, Jinline [6], that we will use in Reflex to insert hooks into the bytecode for any language mechanism occurrence, based on possibly fine-grained criteria. Furthermore, the level of granularity, usually limited to types, can be that of instances: there might be non-reflective instances of a given type coexisting with reflective ones, and among the reflective ones, the controlled mechanisms can be different.

- **How the delegation from the base level to the meta level is done.** Users are provided with a fairly simple way to specialize or specify hooks (the piece of code that does the delegation). This makes it easy to handle the same mechanism in different ways (e.g., to introduce method categories based on a specification file).

The Reflex framework defines abstract interfaces for user extension and includes implementation of generic components that are able to take into account user extensions. Reflex relies on a generic MOP and a generic class builder into which transformers can be plugged-in. Once the framework is finalized, we plan to work on a domain-specific language or aspect language that will be used to specify the reflective aspect of an application.

Due to its openness and high expressiveness, Reflex is, to our opinion, the perfect candidate to evaluate the usefulness of run-time MOPs. This is why we are working on concrete applications of Reflex.
3 The Quest for the Holy Application

3.1 The issue of the necessity

According to the literature, applications of run-time MOPs are many. However, as Gilad Bracha pointed out during the ECOOP Workshop on reflection and metalevel architecture in 2000 (see [1]), the reflection community is still looking for the killer application that will at last demonstrate to the world the necessity of reflection. This is the point: until now, almost every concrete application that can be done with reflection could be done without it. Of course, with reflection the solution is cleaner, or more beautiful, but these arguments are usually not convincing for realistic computer scientists.

In particular, we want to first raise –and hopefully address– the issue of the necessity of run-time MOPs. Is there something useful that can not be done without a run-time MOP? To start the Quest for the Holy Application of run-time MOPs, we have to start from their very definition and what makes them different from the other proposals.

3.2 Distinctive characteristics of run-time MOPs

Static transformation systems (compile-time MOPs, AOP systems, macros, etc.) operate by merging the altering code into the original code. We refer to the result of this transformation as the altered code. For instance, an aspect weaver will weave the definition of some aspects (altering code) into the original code before execution, resulting in a tangled code ready for execution (altered code). At execution, the altered code is fixed: it is not possible to change the definition of the altering code dynamically. Furthermore, since only class definitions exist at compile time, static transformation systems can only reason about classes, not about instances.

The main advantage of run-time MOPs over these systems relies in the existence at run time of a causal connection link, as mentioned in section 1 and illustrated by figure 1. The base object code is the original one, and the metaobjects encapsulate the altering code.

The cost of using a run-time MOP is two-fold: first, we need to introduce the hooks into the base code (at compile or load time) and second, at run time the causal connection link is traversed, usually entailing some reification of entities. The first part of the cost is common to all transformation systems, whereas the second part is specific to run-time MOPs. This link is therefore a drawback from a performance viewpoint, but from a functional viewpoint it has the following advantages:

1. It is possible to change the metaobjects of an object during execution,
2. It is possible to assign metaobjects on a per-instance basis.
Therefore, run-time MOPs are particularly adequate in situations where:

1. dynamic adaptability is required, and/or
2. instance-specific alterations are needed.

If these advantages do not match with concrete needs, then it is better to use a more static and efficient approach.

3.3 On the way to applications

Distributed and mobile object systems are areas domains where dynamic adaptability is needed. As Katz pointed out in [2], mobility requires adaptability, which means that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion. Run-time MOPs are good candidates to achieve adaptability in a distributed setting, for the following reasons:

– Being able to change metaobjects at run time makes it possible to adapt the behavior of objects based on reasoning about the context of execution, therefore achieving context-awareness. For instance, depending on the current state of the network, an application might optimize the way data is compressed before transmission.

– The possibility to have instance-specific alterations may help for another dimension of awareness: self-awareness. This is useful to adapt the behavior of an object based on its own state (e.g. depending on the size of this instance, update the migration strategy).

– Furthermore, since awarenesses are implemented at the meta level, the base level (the application code) does not need to be updated: they can be achieved transparently.

All in all, context- and self-awareness are the bases of intelligence. It is therefore very likely to find the Holy Application of run-time MOPs in such demanding application domains.

In this area, we are investigating adaptability in the issue of reference management upon object migration. Our first work [3] was an achievement of a clean separation of concerns, but without adaptability. We are now exploring an extension of this work to enable dynamic adaptability of reference management policies, in the particular setting of ubiquitous computing and PIM applications [4]. The idea is to be able to plug onto an application an adaptability feature that allows migration of the object graph to take place in an appropriate manner, depending on destination host characteristics, network state, and instance-specific criteria. For instance, if a user transfers his agenda from his personal computer to his handheld device, only data for the current might be migrated, the rest being accessed remotely. The interface module might also be a lightweight version of the standard one. Promising results are on the way.
4 Conclusion

The aim of this position paper was first to briefly present our PhD work on open run-time MOPs. We have mentioned the motivation behind Reflex, our prototype, trying to design and implement a state-of-the-art, open, and concretely applicable run-time MOP for Java.

We have then raised the issue of validating the partially-shared intuition according to which run-time MOPs are useful systems. We start from the assets of such MOPs to identify which kind of requirements may lead to using a run-time MOP. We then explain why we think mobile code systems is a field where the Holy Application of run-time MOPs could be found and we mentioned our current research work in this regard.

Acknowledgments. This work is partially funded by Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References