UML Model Analysis and Checking with MACH

Harald Störrle

Department of Applied Mathematics and Computer Science (DTU COMPUTE), Technical University of Denmark (DTU), Matematiktorvet, 2800 Kongens Lyngby, Denmark

Abstract

Many research articles need to provide an implementation as proof-of-concept. Often such implementations are created as exploratory prototypes, with no intention of ever making them accessible. However, if such a tool turns out to be successful in addressing a real need, it may be useful to make it accessible to a larger audience.

The MACH tool is a case study in unifying a set of prototypes implementing advanced functions for analyzing and checking UML models with the least possible effort. MACH provides a common user interface and shared infrastructure, integrating several independent prototypes without any changes. In this paper, we discuss the requirements for MACH and show how they impact the architectural design decisions.

1. Motivation

Many research articles need to provide an implementation as proof-of-concept. For the purposes of research and publication, an exploratory prototype is often sufficient, that is, an implementation that realizes only the bare minimum, with little or no effort is being spent on issues such as usability, stability, portability, extensibility, and performance. For the purpose of demonstrating the concept, this kind of tool is adequate: it is only ever the author who will use these tools.

However, after the initial proof-of-concept implementation, other scientific questions may arise that can only be answered by human-factors studies, such as observing how users actually use a given tool, and scrutinizing their activities. Or a colleague might want also use these tools and replicate results achieved using them. Or it might be interesting to give students access to the tool, using it in the classroom or in a project. Clearly, the initial basic tool is not up to such challenges. Lacking the above quality attributes is a massive impediment to evolving a mere exploratory prototype into a tool that can be used by people other than the author.

Turning a research prototype into a "proper" tool implies substantial effort that the original author may not want to spend, since this is usually a pure

Email address: `hsto@dtu.dk` (Harald Störrle) engineering effort with little contribution to the research as such. In some cases, money may not be a limiting factor, so that the development can be outsourced to a commercial software developer, but that is a rare case indeed. More frequently, researchers would turn such a task into a thesis project for a Bachelor- or Master's-student. But that is not always possible: maybe the task is too large or too small, no appropriate student is available/willing to do it on short notice, or a candidate does a poor job of the assignment.

In the end, all too often, the researcher may abandon a strand of research for lack of resources. Or, he finds himself implementing a tool, which is a poor allocation of resources if the resulting tool is intended to match the degree of polishing seen in commercial or large-scale open-source projects. If this is the case, how can the researcher create a decent tool quickly, and cheaply? How can the effort of such developments be reduced while still allowing him to deploy it to poorly qualified users, such as students? In this paper, we will show our approach to solving this dilemma.

2. From Rationale to Architecture

When implementing a tool working on models there are three important design decisions. First, whether the tool shall be integrated in a modeling environment or stand-alone, and if the former, in which one. Second, what programming language to use, in particular in the present case, when the existing code base is in Prolog. Third, whether to create a mere command-line UI or a fully-blown GUI.

2.1. Goals, Constraints, and Requirements

The author's research work focuses on advanced operations on UML models that are beyond the scope of existing modeling tools. Over the years, many small exploratory prototypes have been created. They do not provide huge value each on their own, but together their contribution is substantial. In creating the MACH tool, we focused on students and scholars, not industrial users: some understanding of models and underlying concepts is required, and we can accept some limitations and shortcomings that would not be tolerable in commercial tools. Observe that most of these prototypes have been implemented using the Prolog programming language. The main rationale for choosing that particular language is, of course, to allow rapid prototyping. Long-term usage of the resulting code, usage by third parties, or long-term-evolution of the code base, on the other hand, were not considered at the time of creation.

This results in three overall goals:

- Combine as many of the existing prototypes into a single tool;
- make the major functions available to students and colleagues; but
- strictly limit the effort in creating the tool to the bare minimum.

Table 1: Requirements for MACH: **RQA** stands for required quality attribute, **C** stands for constraint, and **F** stands for feature.

ID	Type	Description
R1	RQA	sufficient usability for the intended audiences
R2	RQA	high levels of portability and stability
R3	RQA	easy maintenance and extension with little effort
R4	C	minimum effort in creation and maintenance
R5	C	existing code base is in Prolog
R6	F	unifying interface for all or most of the prototypes
R7	F	auxiliary functions to work with models

We translate these goals into the requirements shown in Table 1. In this table, the requirements are classified by their type: required quality attributes (also known as non-functional requirements), constraints, and features. In the next section we will discuss architectural alternatives to satisfy all of these requirements simultaneously, in a balanced way.

2.2. Implementation platform

Clearly, integrating a tool into a modeling environment provides much easier access to other model-related capabilities, and thus promises a better blending with the modeling process, i.e., higher usability. Also, when reusing an existing framework such as Eclipse[1] or a commercial modeling tool with an API, a substantial benefit is to be expected from reusing the existing code base. The down side is, of course, that the integration as such requires substantial coding effort, and massive learning overhead is to be expected when learning to use an existing framework. Also, using an existing framework is a substantial risk in that things may not work as smoothly in reality as the documentation promises. Of course, this is a constraint only when this knowledge is not present. If a research or development group is already working on a given platform (such as Eclipse) and already has experience in creating tools with this framework, the additional investment is smaller or non-existent. Clearly, this is a defining characteristic for *any* framework.

Obviously, creating a new modeling environment from scratch would completely negate **R4**. This left us with two realistic candidates to be used as an integration framework. On the one hand, the Eclipse Rich Client Platform could be used, a straightforward option because of its openness and widespread use in academia, notably the model-based software development community. However, we have witnessed many a student struggle with Eclipse. The effort of learning

[1]Note that refer to Eclipse here in its role as a framework (i.e., Eclipse RCP), or in its role as a modeling tool (Eclipse plus EMF-related plug-ins), not in its role as an IDE; this discussion is entirely unrelated to IDEs of any kind.

to use Eclipse as a programming framework is so high, that such an investment cannot possibly pay back for the project we consider here.

The other option was to use MagicDraw UML (MD), one of the best commercial UML modeling tools on the market. Our group had used MD for a long time with very good experiences. MD is available on all popular architectures and comes with a great choice of features. Crucially, MD also offers an open API which we had found stable and reasonably well-documented when exploring it. Connecting the prototypes could be achieved using the Java-to-Prolog library (JPL, see `swi-prolog.org`). An exploration project by a student showed, however, that the JPL version current at the time suffered from instability, incurring an incalculable project risk. Furthermore, the effort required to create the user interface was found to be rather higher than originally expected. Thus, we have ruled both Eclipse and MD as frameworks, and decided to create a stand-alone tool without integration.

2.3. Implementation Language

A decision for Eclipse or MD would have implied using Java, since this is the implementation language of both of them. Since we decided for a stand-alone solution, however, this was an open decision again. The portability requirement (**R2**) ruled out using the Microsoft technology stack. Java is an obvious choice of implementation language because of its rich ecosystem and the ability to interface to Prolog code through the Java-to-Prolog library (JPL), though this would have been a gamble on a future version fixing the bug related to the underlying processor architecture of JPL. Also, Java seems to be almost canonical for academic software development.

The other straightforward option was to use Prolog because it had been used for creating the existing code base so there is no integration effort, no problems can arise out of multi-language interaction problems and so on. There was no reason to consider *another* obscure language like Python, Ruby, or Tcl/Tk, so the decision was between Java and Prolog.

We anticipated additional effort from a more complex setting with two programming languages, in particular in the long run, conflicting with requirement **R3**. Thus we decided to go with Prolog as our implementation language. Since SWI-Prolog (see `swi-prolog.org`) had been used all along, we kept using this implementation.

2.4. User Interface type

Finally, we had to decide on what kind of user interface we wanted to implement. Since there are mature freely available GUI-frameworks for Prolog (e.g., the XPCE library for SWI-Prolog) , we still had both options at this point.

Given the usability requirement **R1**, many people might only ever consider a graphical user interface. However, it is much more difficult to create a good GUI than one would believe, and the process involves a large amount of very dull ("boiler plate") code. A command-line UI, on the other hand, is much easier to realize. We also expected it to be easier to maintain since the number of files to touch for changes could be very much reduced (one, in fact). So, with a view to the expected effort (requirement **R4**), we settled for a textual UI as the first step, with the plan of upgrading to a UI later if this should become necessary. So we first implemented option (E) (MACH-1). Fig. 1 visualizes the decision making process.

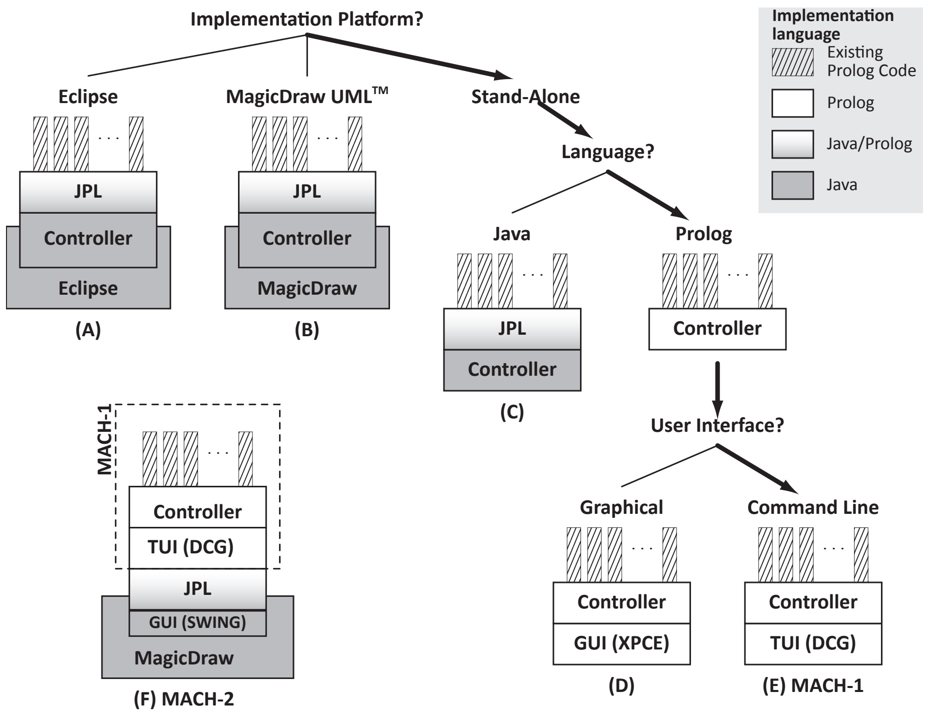


Figure 1: Architectural alternatives for addressing the requirements of MACH.

In an independent project, a student project provided as a by-product a solution that allowed us to plug MACH-1 as-is into MagicDraw (option F in Fig. 1), thus yielding a fully integrated version of MACH at little extra cost, though this is currently a prototype itself, due to the shortcomings described above. We hope a future version of JPL will enable us to achieve a higher level of stability, and thus allow us to deploy MACH-2, too.

3. Components of MACH

In this section we describe the different tools that have been unified in MACH. For each of them, we describe the benefit we realize by integrating it in MACH, and what difficulty we have faced in the process of doing so.

3.1. Model querying

Today, there are few facilities for ad-hoc querying of UML models. Basically, modelers have to use full-text search or exhaustive browsing of models, both of which have serious drawbacks: full text search has very low precision, exhaustive browsing is tedious and not applicable for all but the smallest models. Some tools also offer specific predefined queries or views, but obviously these suffer from limited expressiveness. So, defining model query languages has attracted some interest over the past decade. But how do they compare to each other? Clearly, we cannot compare just one tool with another, since then we would be (mostly) testing the tools rather than the concepts behind them. So we need a common platform to compare different model query facilities side by side.

Given the textual UI of MACH, our approach is limited to textual query languages, but even so there are at least three candidates worth studying: full-text search as the gold-standard, OCL as the only query language as such with a user base worth mentioning, and MOCQL (Model Constraint and Query Language, [5]), our own experimental model query language that shows promise to deliver a much higher degree of usability than OCL. Integrating MOCQL into MACH is trivial, as is implementing a full-text search facility for MACH. Integrating existing OCL-implementations, on the other hand, is difficult if not impossible, but creating a (simple) OCL interpreter in SWI-Prolog is a task that may be achieved by a student in a thesis project.

A major challenge for a textual UI query facility is the integration into (visual) modeling tools. Clearly, users want to have a convenient display of search results, allowing to investigate the results, and navigate to their context in the base model. MACH offers a tabular view of the query results with varying degrees of detailedness, but navigating from MACH to a modeling tool is not straightforward. We have solved this problem by creating a plug in to the modeling tool of our choice that runs MACH inside it. This way, modelers can simply click on an element displayed in MACH's UI to navigate to the respective model element in the modeling tool. As an added benefit, the plug in also allows to execute queries expressed in the Visual Model Query Language (VMQL, see [3]), another query facility we have defined before.

3.2. Size metrics

Measuring the size of the relevant artifacts clearly is of crucial importance in turning software development from craft into engineering. Basic size metrics are essential components in measures of productivity, actual and estimated effort, artifact complexity, and so on. Obviously, the first challenge is to come up with the "right" size metrics of models (see [2]), justify that it is suitable and indeed better than alternatives, and implement it to demonstrate its capabilities. Clearly, this research agenda requires a tool to compute different model sizes, and easily add/modify implemented metrics. By integrating this component into MACH, we can now very easily explore new size metrics and test their effectiveness in class.

This application actually exhibits a drawback in our choice of architecture since we would like to experiment with different interactive visualizations of size metrics, and lacking a GUI, this is difficult to do in MACH. In order to mitigate this problem, we have provided some basic output using ASCII-graphics for tables, histograms, and scatter-plots.

3.3. Clone detection

Model clones are unwanted duplicate model fragments. It has been argued, that model clones have the same adversary effect on models as code clones have on source code, possibly even more so. Thus, detecting clones is an important task in the quality assurance of models. In [6], we have proposed an algorithm for this problem and provided an implementation. The resulting tool had a multitude of parameters and options, that were very difficult to use effectively. We have isolated a configuration that strikes a good compromise for practical purposes, hardwired it in a wrapper around the tool, and plugged it into MACH. Using this function is trivial, and while the results may not be quite as good as when fine-tuning all options, this is the first time such an option has been made easily accessible to non-expert users.

3.4. Comparing models and presenting differences

Another tool we have created compares subsequent versions of a model and allows different presentations of the result [4]. However, this cannot be satisfactorily addressed in a stand-alone-tool since part of the quality of difference presentation is determined by the integration with a modeling environment. By integrating these functions into MACH, we can now study the usage effectiveness of our approach, and explore new usage scenarios.

3.5. Infrastructure components of MACH

Apart from the features proper, MACH also provides a number of supportive functions that are not in itself interesting, but greatly improve the practical work with MACH.

- The first of these components is a package of functions to load models in XMI format and covert them to an internal data structure, and provide aliases to them such that to reduce typing effort when accessing a model. In order to support working with aliases, there are commands for showing and deleting aliases, individually or collectively. This is particularly important, since we frequently use long and very descriptive filenames that are easy to mistype or forget.

- The second auxiliary component is one that provides facilities to navigate and manage the file system, offering commands much like those well known from UNIX command line interpreters, e.g., `pwd`, `cd`, `mv`, and so on.

- The fourth component is the textual UI as such. It benefits from Prolog's built-in capacity of interpreting definite clause grammars (DCGs). This way, the command line interpreter is expressed as a single, relatively compact file defining the complete syntax and linking commands to calls in the code.

Together, these components amount to a few hundred lines of code, the interpreter as such fitting into a single file and using less than 200 lines of code. This is largely due to the built-in capacity of interpreting definite-clause grammars directly as Prolog programs.

Additionally, there is the MagicDraw plug-in that allows to execute Prolog code in a console window in MagicDraw. It is this component that allows us to turn MACH-1 into MACH-2. This component is a by-product of the Model Query tool (`MQ`, see [1]). In contrast to all other components, `MQ` is implemented in Java, and enables to run any Prolog code embedded in the MagicDraw UML modeling tool, including MACH. `MQ` provides functions to run visual queries and also allows to relate model element identifiers defined in MagicDraw to those identifiers used in MACH.

4. Usage experiences

We have field-tested MACH in an undergraduate course on model based software development in the spring of 2013, leading to some initial insights. The students were working in groups of 3 to 5 over a period of 13 weeks on one case study. There were four different case studies, each of which was used by 2-3 concurrent groups. These case studies have been used in similar courses over the past 4 years so we can assert that the resulting models are usually comparable in terms of size and complexity.

4.1. Usability

We found no problems whatsoever related to usability of MACH. This was quite surprising to us since the students using MACH had a very small level of technical prowess: many of them were completely unfamiliar with command line tools, and some had never even used a terminal application or any of the popular UNIX shells before. For some students, issuing a command such as `cd ..` was a challenge. Interestingly, this problem arose mostly for MAC-Users, despite the fact that MAC OS is in fact a Unix-variant with a readily available command shell.

Once such problems were out of the way, however, using MACH was straightforward. Observe that MACH does not provide any built-in help system or man-pages, and the error messages are not exactly human-readable. The only documentation students had was a cheat-sheet like two-page summary of the available commands. Clearly, this level of help was sufficient for most students. So, we can safely conclude that usability is not an issue, and text-based user interfaces can be used by non-proficient users such as undergraduate students without great difficulty.

4.2. Stability

Another concern that we had before deploying MACH was stability. In our experience, running a tool by a few dozens of students will expose every conceivable error very quickly, and incomprehensible error messages, unexpected aborts, and even lost work will quickly lead to frustration. Given that many students have a relatively low frustration threshold, this would be a potential problem, and we were expecting this to happen. However, finding and eliminating each and every problem, and providing high-quality error messages and on-line help incurs substantial effort.

Instead, we decided to simply wrap all problems with a global exception handler issuing a polite warning to the students. The real error message was hidden and accessible only by specifically asking for it, so that the instructors could actually find out about the underlying problem, while the students would not be confused. This approach worked surprisingly well and allowed the instructors to effectively assist students.

4.3. Installability/Portability

Deploying the tool turned out to be a major challenge. We had originally planned to have MACH installed on the computers available to students in the lab rooms. However, students strongly prefer to use their own machinery instead, mostly, because it allows them to work on their homework assignments whenever and wherever they like, independent of the opening hours of the computing labs. Since the students own a wide variety of machines, hardware architectures, and operating systems, and an even wider variety of local settings, we had to provide a universal solution.

In particular, MACH requires an executable SWI-Prolog Interpreter. When trying to shrink wrap the application for uniform deployment, we realized that different machines would have different file names for this executable, and people would install it to different locations in their local directory trees. Finding and resolving these problems was difficult, as a pre-deployment test revealed. Thus, we chose to deploy MACH as source code and provided a build script that would compile the code and create an executable locally. Students would then have to install MACH in two steps. First, SWI-Prolog needed to be installed, but students could choose just the right version and any location they preferred, following the instructions and using the resources provided on the SWI-Prolog web-site. In a second step, they would simply call the install script, which creates an executable that may be started by double clicking. If this should fail, for whatever reason, MACH can still be run from the Prolog command line by issuing a single command.

5. Related tools

Many commercial modeling tools offer capabilities similar to those implemented in MACH. For instance, MagicDraw does provide a built-in suite of consistency checks, model differencing, and size metrics. Similar functions exist for Eclipse based tools. However, the functions actually offered by MACH are beyond those implemented in any tool today, e.g., it generates class model difference descriptions in English prose, it creates graphs of meta class distributions, and it detects clones in UML models, for which there seems to be no other tool at all, not even research prototypes.

Moreover, the objective of MACH is to provide a unified tool out of existing prototypes, not as a green-field development. The goal of MACH is to explore new algorithms and approaches in a dynamic way, with short feedback cycles, without spending more than the bare minimum of effort on the tool's infrastructure.

6. Conclusions and future work

In this paper we present the MACH tool and outline the forces that influenced its design process. MACH is implemented in Prolog and provides a command-line user interface, both of which are relatively exotic choices, today. MACH can be obtained online at `www2.imm.dtu.dk/~rvac/workgroup/mach.html`. However, initial usage experience suggests that our approach was successful in satisfying the goals associated to it:

- MACH is usable by students in a classroom setting, as our experience demonstrates. Thus, requirement **R1** has been satisfied.
- MACH can be deployed to all popular platforms, but a simpler procedure is desirable. So, requirement **R2** has been met largely, though not completely.
- The implementation effort was minimal, and during a period of six months, it proved to be straightforward to gradually add features to MACH, which is largely due to the extremely compact and readable code. We conclude that **R3** and **R4** have been addressed.

We have succeeded in making advanced functionality available to students. Thus, the experience of creating MACH has clearly shown that there are architectural alternatives to EclipseRCP and that a command-line GUI can serve its purpose just as well as a GUI – at lower cost, since creating a light-weight tool with only a textual interface can be achieved with very little effort.

Probably the biggest disadvantage of our approach is the difficulty to interface with tools created by different research groups: given the obscure technology we use, most other tool implementations will be difficult to connect, even if they are created in a way facilitating reuse by others.

Our ongoing work focuses on incremental improvements of the UI, including better documentation, on-line help, and better error messages, and adding more and more advanced functions for UML model analysis, model transformations such as weaving, merging, refactoring; model comparison and similarity measurements; and checking models for compliance with style-guides.

References

[1] Vlad Acretoaie and Harald Störrle. MQ-2: A Tool for Prolog-based Model Querying. In *Proc. Eur. Conf. Modelling Foundations and Applications (ECMFA)*, volume 7349 of *LNCS*, pages 328–331. Springer Verlag, 2012.

[2] Harald Störrle. Large Scale Modeling Efforts: A Survey on Challenges and Best Practices. In Wilhelm Hasselbring, editor, *Proc. IASTED Intl. Conf. Software Engineering*, pages 382–389. Acta Press, 2007.

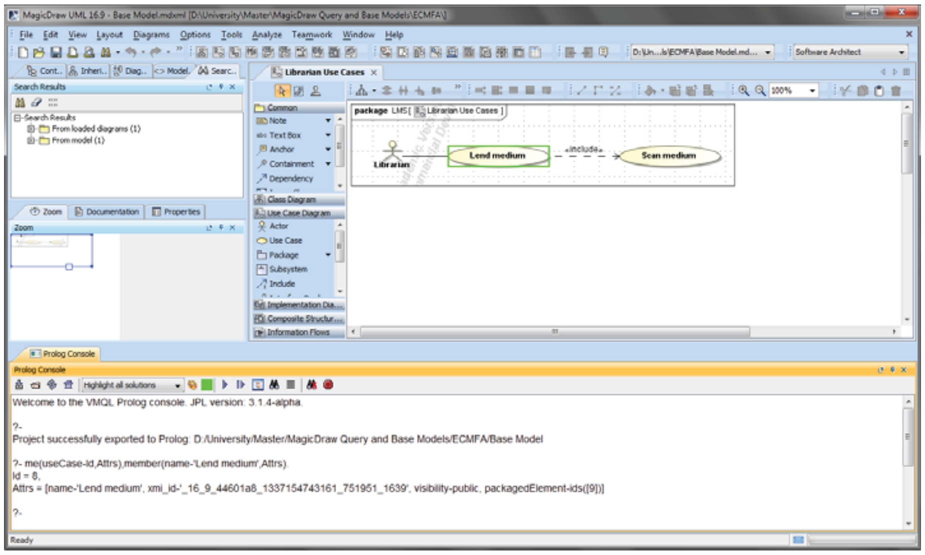
[3] Harald Störrle. VMQL: A Visual Language for Ad-Hoc Model Querying. *J. Visual Languages and Computing*, 22(1), February 2011.

[4] Harald Störrle. Making Sense to Modelers - Presenting UML Class Model Differences in Prose. In Joaquim Filipe, Rui Csar das Neves, Slimane Hammoudi, and Lus Ferreira Pires, editors, *Proc. 1st Intl. Conf. Model-Driven Engineering and Software Development*, pages 39–48. SCITEPRESS, 2013.

[5] Harald Störrle. MOCQL: A Declarative Language for Ad-Hoc Model Querying. In Pieter Van Gorp, Tom Ritter, and Louis M. Rose, editors, *Eur. Conf. Proc. Modelling Foundations and Applications (ECMFA)*, number 7949 in LNCS, pages 3–19. Springer Verlag, 2013.

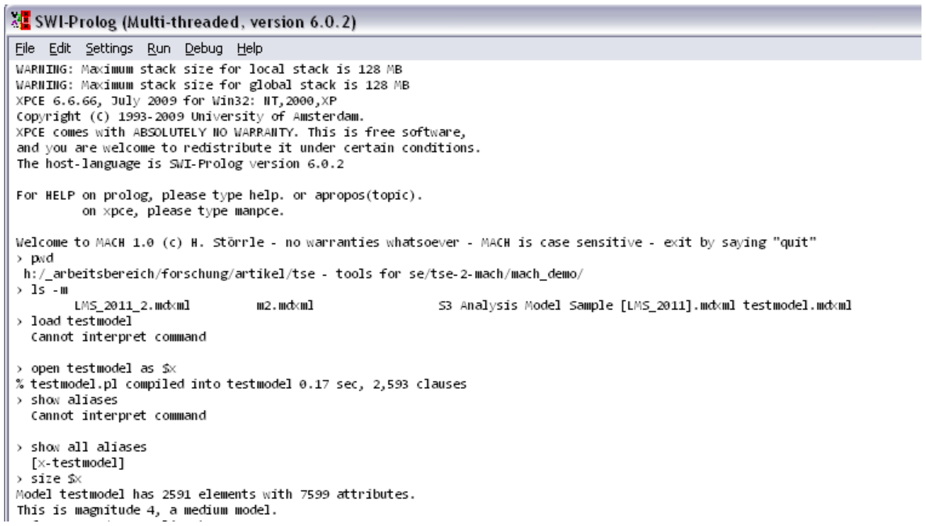
[6] Harald Störrle. Towards Clone Detection in UML Domain Models. *J. Software and Systems Modeling*, 12(2), 2013. (accepted in 2011).

Appendix A. Usage Scenarios for MACH



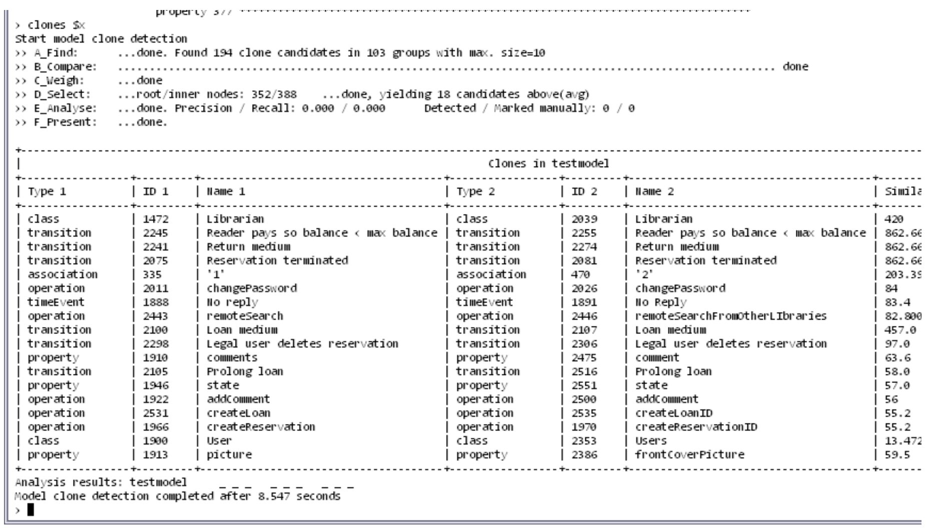
```
Welcome to the VMQL Prolog console. JPL version: 3.1.4-alpha.

?-
Project successfully exported to Prolog: D:/University/Master/MagicDraw Query and Base Models/ECMFA/Base Model

?- me(useCase-Id,Attrs),member(name-'Lend medium',Attrs).
Id = 8,
Attrs = [name-'Lend medium', xmi_id-'_16_9_44601a8_1337154743161_751951_1639', visibility-public, packagedElement-ids([9])]

?-
```

Figure A.2: Using MACH-2 to query models: Running MACH-1 in the MQ-console inside of MagicDraw allows smooth integration. A simple click to a model element allows to navigate to the model element's definition in the host tool.


SWI-Prolog (Multi-threaded, version 6.0.2)

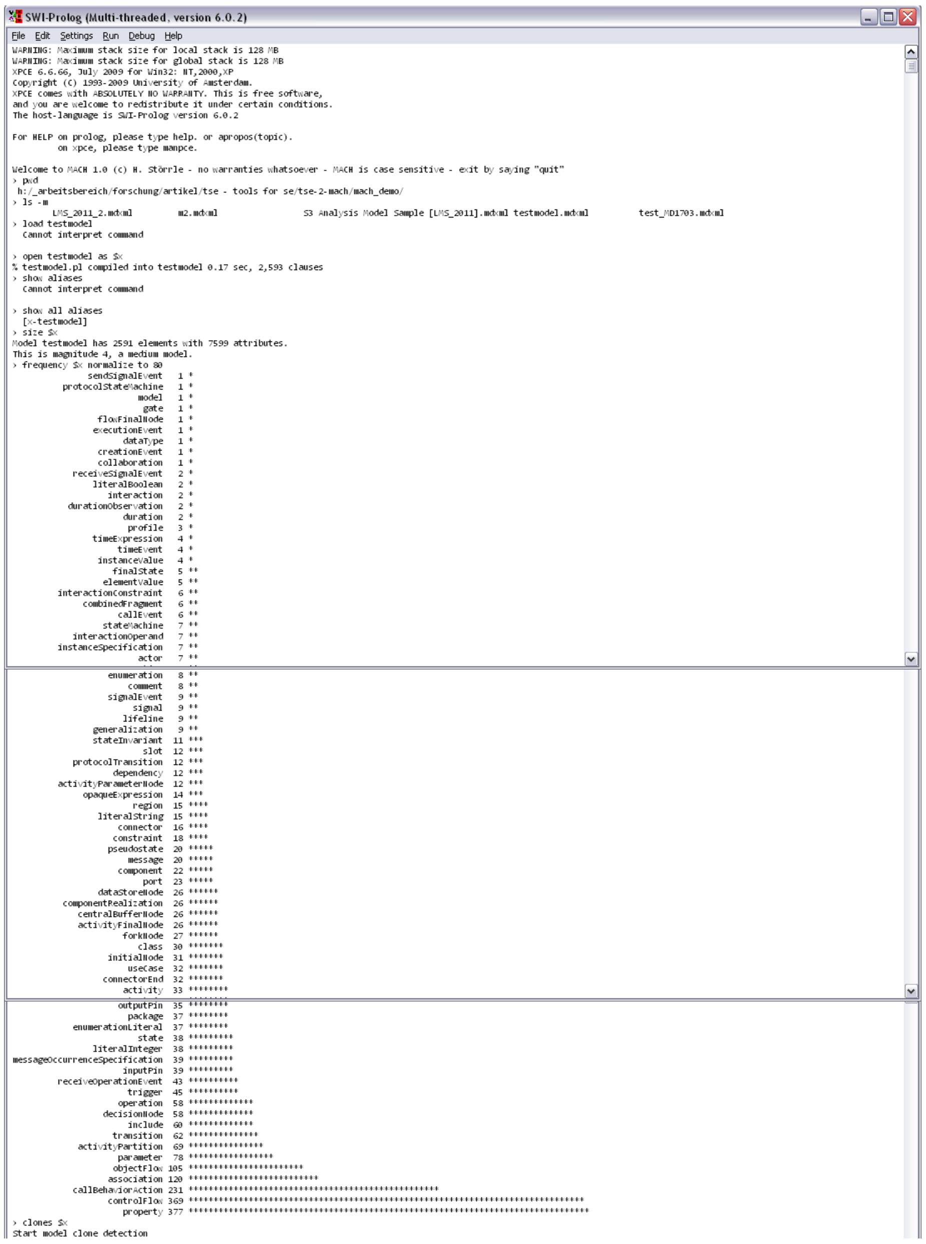
```
WARNING: Maximum stack size for local stack is 128 MB
WARNING: Maximum stack size for global stack is 128 MB
XPCE 6.6.66, July 2009 for Win32: NT,2000,XP
Copyright (C) 1993-2009 University of Amsterdam.
XPCE comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
The host-language is SWI-Prolog version 6.0.2

For HELP on prolog, please type help. or apropos(topic).
         on xpce, please type manpce.

Welcome to MACH 1.0 (c) H. Störrle - no warranties whatsoever - MACH is case sensitive - exit by saying "quit"
> pwd
 h:/_arbeitsbereich/forschung/artikel/tse - tools for se/tse-2-mach/mach_demo/
> ls -m
        LMS_2011_2.mdxml         m2.mdxml                 S3 Analysis Model Sample [LMS_2011].mdxml testmodel.mdxml
> load testmodel
  Cannot interpret command

> open testmodel as $x
% testmodel.pl compiled into testmodel 0.17 sec, 2,593 clauses
> show aliases
  Cannot interpret command

> show all aliases
  [x-testmodel]
> size $x
Model testmodel has 2591 elements with 7599 attributes.
This is magnitude 4, a medium model.
```

Figure A.3: Opening models in MACH, assigning an alias, and determining the size of a model as numbers of model elements and attributes.



```
                   property 377 ..........................................................................
> clones $x
Start model clone detection
>> A_Find:      ...done. Found 194 clone candidates in 103 groups with max. size=10
>> B_Compare:   ................................................................................... done
>> C_Weigh:     ...done
>> D_Select:    ...root/inner nodes: 352/388    ...done, yielding 18 candidates above(avg)
>> E_Analyse:   ...done. Precision / Recall: 0.000 / 0.000      Detected / Marked manually: 0 / 0
>> F_Present:   ...done.

+----------------------------------------------------------------------------------------------------------
|                                                Clones in testmodel
+---------------+--------+--------------------------------+---------------+--------+--------------------------------+------
| Type 1        | ID 1   | Name 1                         | Type 2        | ID 2   | Name 2                         | Simila
+---------------+--------+--------------------------------+---------------+--------+--------------------------------+------
| class         | 1472   | Librarian                      | class         | 2039   | Librarian                      | 420
| transition    | 2245   | Reader pays so balance < max balance | transition | 2255 | Reader pays so balance < max balance | 862.66
| transition    | 2241   | Return medium                  | transition    | 2274   | Return medium                  | 862.66
| transition    | 2075   | Reservation terminated         | transition    | 2081   | Reservation terminated         | 862.66
| association   | 335    | '1'                            | association   | 470    | '2'                            | 203.39
| operation     | 2011   | changePassword                 | operation     | 2026   | changePassword                 | 84
| timeEvent     | 1888   | No reply                       | timeEvent     | 1891   | No Reply                       | 83.4
| operation     | 2443   | remoteSearch                   | operation     | 2446   | remoteSearchFromOtherLIbraries | 82.800
| transition    | 2100   | Loan medium                    | transition    | 2107   | Loan medium                    | 457.0
| transition    | 2298   | Legal user deletes reservation | transition    | 2306   | Legal user deletes reservation | 97.0
| property      | 1910   | comments                       | property      | 2475   | comment                        | 63.6
| transition    | 2105   | Prolong loan                   | transition    | 2516   | Prolong loan                   | 58.0
| property      | 1946   | state                          | property      | 2551   | state                          | 57.0
| operation     | 1922   | addComment                     | operation     | 2500   | addComment                     | 56
| operation     | 2531   | createLoan                     | operation     | 2535   | createLoanID                   | 55.2
| operation     | 1966   | createReservation              | operation     | 1970   | createReservationID            | 55.2
| class         | 1900   | User                           | class         | 2353   | Users                          | 13.472
| property      | 1913   | picture                        | property      | 2386   | frontCoverPicture              | 59.5
+---------------+--------+--------------------------------+---------------+--------+--------------------------------+------
Analysis results: testmodel     _ _ _  _ _ _  _ _ _
Model clone detection completed after 8.547 seconds
> █
```

Figure A.4: Using MACH to detect clones in a model.


SWI-Prolog (Multi-threaded, version 6.0.2)
File Edit Settings Run Debug Help

```
WARNING: Maximum stack size for local stack is 128 MB
WARNING: Maximum stack size for global stack is 128 MB
XPCE 6.6.66, July 2009 for Win32: NT,2000,XP
Copyright (C) 1993-2009 University of Amsterdam.
XPCE comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
The host-language is SWI-Prolog version 6.0.2

For HELP on prolog, please type help. or apropos(topic).
         on xpce, please type manpce.

Welcome to MACH 1.0 (c) H. Störrle - no warranties whatsoever - MACH is case sensitive - exit by saying "quit"
> pwd
 h:/_arbeitsbereich/forschung/artikel/tse - tools for se/tse-2-mach/mach_demo/
> ls -m
        LMS_2011_2.mdxml        m2.mdxml                S3 Analysis Model Sample [LMS_2011].mdxml testmodel.mdxml         test_MD1703.mdxml
> load testmodel
  Cannot interpret command

> open testmodel as $x
% testmodel.pl compiled into testmodel 0.17 sec, 2,593 clauses
> show aliases
  Cannot interpret command

> show all aliases
  [x-testmodel]
> size $x
Model testmodel has 2591 elements with 7599 attributes.
This is magnitude 4, a medium model.
> frequency $x normalize to 80
               sendSignalEvent   1 *
          protocolStateMachine   1 *
                         model   1 *
                          gate   1 *
                 flowFinalNode   1 *
                executionEvent   1 *
                      dataType   1 *
                 creationEvent   1 *
                 collaboration   1 *
            receiveSignalEvent   2 *
                literalBoolean   2 *
                   interaction   2 *
           durationObservation   2 *
                      duration   2 *
                       profile   3 *
                timeExpression   4 *
                     timeEvent   4 *
                 instanceValue   4 *
                    finalState   5 **
                  elementValue   5 **
         interactionConstraint   6 **
              combinedFragment   6 **
                     callEvent   6 **
                  stateMachine   7 **
            interactionOperand   7 **
         instanceSpecification   7 **
                         actor   7 **
                   enumeration   8 **
                       comment   8 **
                   signalEvent   9 **
                        signal   9 **
                      lifeline   9 **
                generalization   9 **
                stateInvariant  11 ***
                          slot  12 ***
            protocolTransition  12 ***
                    dependency  12 ***
         activityParameterNode  12 ***
              opaqueExpression  14 ***
                        region  15 ****
                 literalString  15 ****
                     connector  16 ****
                    constraint  18 ****
                   pseudostate  20 *****
                       message  20 *****
                     component  22 *****
                          port  23 *****
                 dataStoreNode  26 ******
          componentRealization  26 ******
             centralBufferNode  26 ******
             activityFinalNode  26 ******
                      forkNode  27 ******
                         class  30 *******
                   initialNode  31 *******
                       useCase  32 *******
                  connectorEnd  32 *******
                      activity  33 ********
                     outputPin  35 ********
                       package  37 ********
            enumerationLiteral  37 ********
                         state  38 *********
                literalInteger  38 *********
messageOccurrenceSpecification  39 *********
                      inputPin  39 *********
         receiveOperationEvent  43 **********
                       trigger  45 **********
                     operation  58 *************
                  decisionNode  58 *************
                       include  60 *************
                    transition  62 **************
             activityPartition  69 ***************
                     parameter  78 *****************
                    objectFlow 105 ***********************
                   association 120 **************************
            callBehaviorAction 231 **************************************************
                   controlFlow 369 *******************************************************************************
                      property 377 ********************************************************************************
> clones $x
Start model clone detection
```

Figure A.5: Emulating the graphics capabilities missing in a textual UI to visualize meta class frequency distribution.