Mescal Requirements

Lambert Meertens*

Department of Algorithmics and Architecture, CWI, Amsterdam, and

Department of Computing Science, Utrecht University, The Netherlands

www.cwi.nl/~lambert

Version of June 4, 1998

0 What this document is about

Mescal is a (not yet existing) system for mechanical support in constructing and exploring formal theories, with an emphasis to calculational theories for software development. This document is an attempt to capture and clarify the primary and secondary functional requirements for Mescal. To avoid the repetitious use of modalities like "should" or "must", we use the present tense to signify the hopeful future.

The envisioned architecture for Mescal (described in a companion paper) is that of a *kernel system* "empowered" by libraries. We mention this right here, because it bears on the interpretation of questions of the form: *Can* Mescal *do X?* There are three possible answers: (1) *Yes, that is a* native *capability of* Mescal; (2) *Yes, provided appropriate libraries are developed and used*; (3) *No, unless someone is willing to do some kernel hacking.* In this document, the classes (1) and (2) will not be clearly distinguished.

Before going into the requirements, we list some *non-requirements*:

- The system finds the proofs for you. (Mescal is not a theorem prover. At best it is a proof verifier. There are some very good theorem-proving systems out there, and the intention is not to feebly duplicate that work.)

*Work performed while visiting Kestrel Institute, Palo Alto.

- A system-verified proof is correct. (*Mescal* "believes" whatever it is told to believe by the user. So a user can cheat, or just make a fundamental mistake in specifying some axioms or proof rules. It is not hard to feed *Mescal* a naive specification of set theory for which Russell's paradox applies. If you want strong verification, use one of many good proof verifiers.)

- The system compels its users to follow good mathematical standards. (On the contrary, *Mescal* allows its users to follow their own set ways of proving things. It is not meant for re-educating mathematicians. Read [OnTheShape].

That having been said, each "No" above is actually a "No, but ...", as will become clear in the course of the document.

Then what is *Mescal*? It combines the best of Math∫pad and Automath. Math∫pad is a WYSIWYG document editor that is optimized for documents containing mathematics. Users can easily define their own notations and enter formulas. However, there is no mathematical semantics of any kind associated with the formulas. Automath is not an interactive system, but a proof checker for documents (Automath "books") written in a hardly human-oriented language — although based more on the way working mathematicians prove theorems than on proof theory as developed by logicians. `[[NuPRL, Coq, Lego ...]]` Think of *Mescal* as a document editor like Math∫pad, except that the "formal parts" of the text do carry semantics. Some of the formulas being edited represent proofs, and *Mescal* can perform verifications on these proofs. On other formulas, *Mescal* can do other checks, such as type checking. Another viewpoint is that *Mescal* is a proof verifier that also has editing capabilities. However, this viewpoint is secondary. The editing has primacy.

Some important requirements are not specific to *Mescal*, but should apply to any advanced editor. These will not be dealt with at length in this document.

- There is a clear logical separation between the content and the presentation of a document.

- In particular, a document can consist of several parts that can be presented separately, and there can be several views on a document that may be presented simultaneously.

- The editor has "unlimited" UNDO.

- The editor keeps a persistent session state for each document. When a session is interrupted and restarted later, the effect is as if there had been no interruption.

- The editor can output either just the document or the complete session state in a portable document-interchange format, for example some form of HTML (and of course accepts this format as input).

For a discussion of such requirements we refer to [Ergonomics]. Three other potentially vital and interrelated issues that bring their own requirements will be ignored for the sake of simplicity, namely *multi-authoring* (using Mescal for cooperative work), *version control*, and using Mescal in a *distributed environment*.

Here are some primary requirements specific to Mescal. `[[Order by importance and grouping]]`

- Informal text and formal expressions may be mixed in a Mescal document.

- A user who never accesses the inbuilt "mathematical intelligence" of Mescal for verification purposes will never be bothered by it, but can use the system as a rather conventional WYSIWYG document editor.

- Mescal can be used to write research papers, but also to author web pages, or to produce programs. One output mode can for example be LaTeX source code, other modes being SGML (e.g. MathML), or Java. Users can define new output modes.

- In Mescal a user can perform "edit steps" on (incomplete) proofs corresponding to proof steps. There is no need to ask for verifying the step, since it was actually performed by the system itself — in response to the user's request — using the proof rules given to it.

- Users can also directly edit (parts of) proofs, bypassing Mescal's proof capabilities. Mescal can (to a certain extent) verify whether the proof steps are still instances of known proof rules. If not, Mescal is graceful and unobtrusive about it. The user can choose to ignore the problem and continue without hindrance, even if the offensive parts do not type check.

- If the user claims that some step is "obvious", *Mescal* will accept that. Compare this to interactive spelling checkers, that should believe a user's claim that "homset" is a well-spelled word, and not a misspelling of "hamster", although it is not in the dictionary and "hamster" is.

- *Mescal* can use libraries consisting of pre-defined theories. During a session a user can consult and modify these theories as well as create new ones, both global theories and "local" theories (theories that are private to the present document).

- *Mescal* facilitates exploration, in which the approach may be rather chaotic, theories may change on the fly, and the user may be less interested in seeing whether some claim follows from the known assumptions than in discovering what further conditions are needed to satisfy the claim. Revising something is at least as easy as it is with pencil and paper, and possibly much easier since systematically renaming a variable, or changing a notation, can be done at once for all uses.

- *Mescal* allows the use of multiple formalisms in one document (for example category-theoretical formulas next to Haskell programs),

- *Mescal* accepts overloading of symbols (for example $+$ both for addition of numbers and of matrices), symbols depending on implicit arguments ("Let $y = f(x)$ and assume that $y' = 0$."), and other potential sources of ambiguity.

- Ultimately, the user has control over the appearance of the text, being able to override locally the presentation rules of a notation.

Here are some secondary requirements.

- *Mescal* has some rudimentary theorem-proving capabilities: it can bridge some trivial gaps. An example is associativity. If operator $\oplus$ is known in some context to be associative, then a required matching of $a \oplus (b \oplus c)$ to $(a \oplus b) \oplus c$ will succeed.

- *Mescal* can make suggestions concerning a proof plan, as well as give information about what rules apply in some context.

- *Mescal* offers some possibilities for connecting to external engines.

- *Mescal* can display the undischarged proof obligations of a document or a part of it, and can also transform a proof with an undischarged proof obligation into one in which the undischarged part becomes an assumption.

- *Mescal* can signal that some assumption, definition or theorem is unused, as well as display which external definitions and theorems are appealed to in some part of the document.

1 Task analysis

We do the analysis by way of scenarios for some example problems. The problems are chosen to be of moderate complexity. They are not fully specified, for two reasons. First, in realistic settings, the original problem is typically not fully specified. Secondly, we want to discuss possible variations. In all cases, it is assumed that the user is aware of the literature in the field, but that no stock answer to the problem is available.

1.1 A distributed resource-allocation protocol

The assumption here is that the user wants to develop a distributed protocol for resource allocation. One possibility is that this is an exercise in fundamental research. For example, the user could be an academic scientist specializing in distributed protocols who wants to examine issues in resource allocation. Another possibility is that the user is an engineer who needs to solve an existing problem, possibly in order to improve upon an existing, working solution that is in some respects unsatisfactory. The existing protocol might entail too much overhead, or have certain limitations that are increasingly hard to put up with.

In either case, the user must examine a number of questions.

1. *What is a desirable degree of abstraction?* (The advantages of more abstraction include that the resulting protocol (if any) may be reusable in a larger variety of concrete settings, and that the treatment is less burdened by detail. On the other hand, too abstractly a formulation may mean that no efficient protocol is possible at all. The researcher is more likely to aim at an abstract formulation, possibly creating a

taxonomy of problems. The engineer will want to be sure that all relevant aspects of the existing problem are modelled.)

2. *What is the nature of the resources? How will they be modelled? What is to be considered an "allocation clash"? Can resources unexpectedly drop out?*

3. *What is the form of the requests? Are there priorities? Is pre-emption possible? Dynamic re-allocation? Is there a real-time aspect?*

4. *Can the problem be modelled as a (distributed) constraint-satisfaction problem?*

5. *What are the correctness criteria?* (Generally, such criteria can be seen as falling into two categories: *domain specific* and *general*. A domain-specific criterium would be: there must be no allocation clash (over-allocation). A generic criterium is progress: the protocol must not paint itself into a corner. A specific form of liveness could be dubbed "politeness": all requests must (eventually) be reacted to.)

6. *What is or may be the nature of the protocol? N-party? Can some participants have special roles (such as "broker")? Is communication through one-to-one channels, or can there be some form of shared memory, or broadcasting? And what about the network? Is it static, or can it be reconfigured dynamically? Or are we even into mobile environments here? What are the reliability assumptions on nodes and connections?*

7. *What is the objective? Optimal (and if so, in what sense)? Feasible? Some performance guarantee? Or all solutions (and if so, how are they represented)?*

8. *What formalism is to be used for describing the protocol. UNITY? Some process algebra? State charts?* (In the research setting, the choice of protocol formalism may be for the simplest one; in the engineering context, it may be preordained.)

9. *Should the protocol be reified in an existing implementation language?*

10. *What available formalism libraries can be used for this problem?*

11. *What degree of verification is desired? Everything? Some crucial aspects?*

12. *What is a good plan of approach?*

The answers to each of these questions may depend on many things, some of which are more a matter of pragmatics than others. In any case, whether the user is a research scientist or an engineer, it should be rather obvious that even on a moderate example like this it is unlikely that everything will be right on the first try. The number of choices is very large, too large to approach all with a systematic approach. A more concrete context, as when we are dealing with an engineering problem, may narrow this somewhat down, but even then there are many ways of modelling one same notion, and some combinations may be less felicitous then others. Also, a seemingly minor change in the assumptions can make the problem cross the border from very doable to completely hopeless.

A plausible general shape of a plan of approach is as follows. First find a non-distributed algorithm that by itself solves the allocation problem satisfactorily. If none is available straightaway, take a similar problem that has been solved (for example, Dijkstra's *Banker's Algorithm*) and try to adapt the algorithm to the case at hand. A way of doing that can be roughly indicated as follows. Create a derivation of the original algorithm. Modify the assumptions to fit the actual problem. Patch up the derivation, leading to the adapted algorithm. A fair amount of look-ahead, trying to "guess" what adaptations are needed, may help avoid blind alleys.

Next, try to use general techniques for converting a sequential algorithm into a distributed one.

During the first phase, it may turn out halfway, in trying to derive the "adapted" algorithm, that some assumptions are not necessary, while others need to be strengthened. It is further possible that the first phase is completed quite satisfactorily, but that the second phase requires some adaptations of the sequential algorithm for the conversion techniques to work. In fact, that is quite likely. In all of these cases, what a user definitely does not want to (have to) do, is go with a fine comb through all previously developed bits of theory, proofs, derivations and such, and make lots of scattered small changes. That is both boring and error-prone. And the last thing a user would want, is to have to redo all proofs. A system encouraging exploration can take much of that burden away.

For example, if an assumption is unused, none of the verified steps depend on it. So deleting the assumption should not invalidate any work. The same applies for adding an assumption. It may, of course, validate a previously invalid step.

A more complicated situation arises if an assumption that is in "active" use is changed. The *Mescal* system keeps track of all relationships between all forms, and will "know" which steps depend on the assumption. The proof rule applied there will then be re-applied. If valid, nothing happens. If (currently) invalid, it may be the case that this is a transitional situation, because the user is about to change some definition, say, accordingly, so that the step remains valid. It may also be the case that the user expects or hops the step would remain valid, but relies on the system to check that. The bottom line is that the system should detect all invalidated steps, and make them knowable to the user, but in an unobtrusive way that does not interfere with the work if the user has a plan and wants to press on. Requiring all inconsistencies to be repaired instantly will make revision unnecessarily awkward if not undoable, and thus effectively and thoroughly discourage exploration. (A similar problem exists with many structure editors that require the text to be fully syntactically valid at all times; even small changes that are painless with any other editor become hurdles in a revision.)

Obviously, the user will want to use existing libraries whenever applicable. How to find out what libraries apply? There must be a way of browsing through the libraries. Since it is foreseen that the formal expression of theories will not always be completely transparant at first sight, it should be possible to accompany these libraries with less formal counterparts, which we may think of as on-line manuals, that are equally browsable.

If, for the user, this is not a one-shot problem but one in a related set (which may apply both for the research scientist and the engineer), they may want to build their own sets of libraries, which should essentially have the same status as standard libraries packaged with the *Mescal* system, or libraries obtained from other repositories. Building one's own libraries should be within the reach of a typical user, and not require inordinate capabilities beyond those required to be able to use *Mescal* fruitfully in the first place.

1.2 Lecture Notes on Graph Theory

The user here is a lecturer revising existing lecture notes on Graph Theory, written in LaTeX. Although bright enough, this user has an unfortunate tendency to make many little, rather trivial, slips, in particular in a process of revision, which confuse the students if they slip through unnoticed into the lecture notes. The purpose of using Mescal here is to catch most of the silly little errors (such as mixing up variable names, or revising the example input to an algorithm but forgetting to adjust the output).

The major problem here is that of taking an existing body of work, created independently of Mescal, and turning it into a Mescal "object". Although that is never going to be a fully trivial exercise, it should be possible to develop, say, LaTeX-to-Mescal translators, similar to existing LaTeX-to-SGML translators, that bear the brunt of the pain. Such a tool will not be part of Mescal proper; the point is that the Mescal system should not make creating such a tool infeasible. In particular, there must be a well-defined target language for the translator that is acceptable Mescal input. Further, Mescal must not be (too) fussy about what is acceptable or not, in particular when it comes to semantic aspects. For example, a (representation of) a form that is identified as belonging to some formalism called GraphTheory should be accepted as a valid form, although it may contain undefined dummies, or operations of the wrong arity, and even if no formalism of that name is currently known to the system. What applies to micro-scale revision should equally apply to macro-scale revision.

Once the notes have been turned into a Mescal source, it should be possible, using the system, to focus on some segment of the notes containing forms with dummies, and declare that for that section certain typings apply, such as that x ranges over a set N of nodes, possibly overriding default typings introduced by the translator.

If any algorithms in the text are in an executable language, it should be possible to introduce a rule relating input+program to output, where the "execution" of the rule is delegated to an *external engine* (an implementation of that language) whose output is automatically inserted into the text. Modifying either the input or the program will invoke the rule, and ensure that the output as included is always up to date.

2 Interaction paradigms

The Edit-Run Loop Paradigm. This is the pre-interaction age paradigm: first the input is prepared, then run through a program, and if the outcome is not satisfactory the input is modified and the process is repeated. Systems using this paradigm tend to have a linearized-textual input format. Some examples among numerous systems are the Automath system [Automath] and LaTeX[Latex].

The Pocket-Calculator Paradigm. In the *Pocket-Calculator* interaction paradigm the various user operations replace the presentation being operated on *in situ* by the result, as in

$$
\begin{array}{lll}
\boxed{7} & \rightarrow & \boxed{\quad\quad\quad 7} \\
\boxed{5} & \rightarrow & \boxed{\quad\quad 75} \\
\boxed{*} & \rightarrow & \boxed{\quad\quad 75} \\
\boxed{7} & \rightarrow & \boxed{\quad\quad\quad 7} \\
\boxed{7} & \rightarrow & \boxed{\quad\quad 77} \\
\boxed{=} & \rightarrow & \boxed{\quad 5775}
\end{array}
$$

There is no way the user can tell afterwards how a result was arrived at. If a wrong button is hit, this may go unnoticed; if not, then in general there is no other recourse than to redo the whole calculation. With real pocket calculators, the domain is that of numbers. But basically the same paradigm is used by several of the early program-transformation systems: when the user performs a transformation step, the program *in statu nascendi* is replaced *in situ* by the transformed program. So the "state" is the program under construction. To revise a development you need a separate history mechanism. Examples are the CIP system [CIP-S] and KIDS [KIDS]

The Cash-Register Paradigm. Cash registers differ from pocket calculators in that they print a record of the steps. This is the basic paradigm of several computer-algebra systems, like Macsyma, Maple and Mathematica. The normal mode of interaction is a query-response loop, as in

`> v=x*(1-2*x)^2;`

$r1 = x(1-2x)^2$

`> diff(v,x);`

$r2 = (1-2x)^2 - 4x(1-2x)$

```
> solve(r2,x);
```
$r3 = [1/6, 1/2]$

Although this is (part of) a proof that the value of $x(1-2x)^2$ is maximal for $x = 1/6$, there is no notion of "proof object" here. The proof itself is not recorded; it exists only in the head of the user.

Shood go somewhere:

- Full verification is possible if you insist.
- Scratchpad scribbling
- Details may be hidden from view, to be recalled upon demand.
- hidable annotations
- definedness conditions

Related work:

- Program-transformation systems
- MathSpad
- Automath
- Deva
- Jape
- Mizar
- NuPRL, Coq, Lego, HOL, Isabelle, ...:
  - each based on one specific logic;

– adamant on keeping proof correct at all times;
– little user control over notation.

- Camino Real (?)

3 Where should this go?

- Allow definition by characterizing predicates:

$$[f(x) = \vartheta(y :: P(x, y))]$$

Definedness condition $\delta(\vartheta P) \equiv \exists P \wedge \mathsf{Uniq}.P$ (where $\mathsf{Uniq} := (P :: [Px \wedge Px' \Rightarrow x = x'])$.) Characterization: $[\vartheta P = x \equiv Px]$.

- Let **macro** P[**depends on** f, b] $:= (a :: f\ a = b)$. Define

$$\mathit{Injective}(f) := \forall(b : \exists P : \mathsf{Uniq}.P)$$

$$\mathit{inverse} := (f :: (b : \exists P : \vartheta P))$$

Then the definedness condition on $\mathit{inverse}(f)$ is:

$$\begin{array}{ll} & \forall(b : \exists P : \delta(\vartheta P)) \\ \equiv & \quad \{ \text{ above } \} \\ & \forall(b : \exists P : \exists P \wedge \mathsf{Uniq}.P) \\ \equiv & \quad \{ \text{ some rule } \} \\ & \forall(b : \exists P : \mathsf{Uniq}.P) \\ \equiv & \quad \{ \text{ above } \} \\ & \mathit{Injective}(f) \end{array}$$

So this suggests we could define

$$\mathit{Injective} := \delta \circ \mathit{inverse}$$

- The following happens quite often. Some property P has two equivalent formulations, W and S, where W is formally weaker than S, i.e., $W \Leftarrow S$ is immediate. If P is to be established, we prefer $P \Leftarrow W$. If P is to be exploited, we prefer $P \Rightarrow S$. This should be dealt with w/o fuss.

- What is the status of a chain like $x = a_0 < a_1 < a_2 = y$? One meaning is $x = a_0 \ \wedge \ a_0 < a_1 \ \wedge \ a_1 < a_2 \ \wedge \ a_2 = y$. In a Feijen-style proof this gives the proof obligation. The bottom-line demonstratum here would be $x < y$. In such a chain we require the connectives to be elements of a 'chainer'. A chainer is a monoid $(\mathcal{A}, \otimes)$ where $\mathcal{A}$ is a set of binary relations on the same domain $A \times A$ such that for any pair R, S in the set

  $$R \otimes S \supseteq R \circ S$$

  Equality/equivalence can be used for the neutral element. In the tables below we omit the corresponding rows and columns as being totally predictable. For example, for any transitive relation $\sqsubseteq$, defining $\sqsubset$ by $[x \sqsubset y := x \sqsubseteq y \wedge y \not\sqsubseteq x]$, we have the chainer

$\otimes$	$\sqsubset$	$\sqsubseteq$
$\sqsubset$	$\sqsubset$	$\sqsubset$
$\sqsubseteq$	$\sqsubset$	$\sqsubseteq$

  Any partial chainer can be made total by including $\top$ in $\mathcal{A}$. Again we leave out the corresponding rows and columns.

$\otimes$	$\neq$
$\neq$	$\top$

  The implicit proof rule in $aRbSc$ is

  $$a(R \otimes S)c \ \dashv \ aRb, bSc$$

There should be a stock set of chainers plus some "intelligence" in picking the appropriate one (and switching to more appropriate ones as the chain changes) so that the user need not specify the chain to be used when embarking upon a calculation.

- For a context $C[\_]$, define

  $$C[\_]\colon P \leftarrow\!\boxminus\!- Q$$

  to mean that the rule

  $$P(C[a]) \;\dashv\; Qa$$

  is universally valid. This forms a little category, so such contexts may be composed. We can extend this to relations thus:

  $$C[\_]\colon R \leftarrow\!\boxminus\!- S$$

  means universal validity of

  $$C[a]\; R\; C[b] \;\dashv\; a\, S\, b$$

  (In general we need a functor to describe the holes of source and target.) Instances:

  Leibniz: $C[\_]\colon (=) \leftarrow\!\boxminus\!- (=)$. Here $C[\_]$ may be anything.

  Monotonicity: $F[\_]\colon (\sqsubseteq) \leftarrow\!\boxminus\!- (\sqsubseteq)$. Here $F[\_]$ must be a monotonic context.

  Define further

  $$C[\_]\colon P \leftarrow\!\diamond\!- Q$$

  to imply universal validity of the rule

  $$Pa \;\dashv\; Q(C[a])$$

  Again this may be composed. Extendsing this likewise to relations, here are some instances:

Injectivity: $F[\_]\colon (=) \leftarrow\!\diamond\!- (=)$. Here $F[\_]$ must be an injective context.

Contraposition: $C[\_]\colon \overline{R} \leftarrow\!\diamond\!- S \;\dashv\vdash\; C[\_]\colon \overline{S} \leftarrow\!\boxminus\!- R$.

- If the rule to be applied is $X \dashv Y$, and t is given as justification, we match t to Y and if successful apply the matcher to X. If not directly successful, we can see if t is of the form Pt' and Y of the form QY' with $Qa \dashv Pa$, and try a "weakened" match of t' to Y'. For example, for $|a| = a \dashv a \geq 0$ we can match $f(x) = 0$ to $a \geq 0$ getting the "instance" $|f(x)| = f(x) \dashv f(x) = 0$.

- Moving theorems automatically up scopes, keeping theorem "name" (reference).

References

[1] Lambert Meertens and Steven Pemberton. *The Ergonomics of Computer Interfaces — Designing a System for Human Use.* Research Report CS–R9258, CWI, 1992.

[2] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments.* Lecture Notes in Computer Science 445. Springer Verlag, 1990.

TODO

`[[TODO]]`