Management of Change in Declarative Languages

Mihnea Iancu and Florian Rabe

Jacobs University, Bremen, Germany

Abstract. Due to the high degree of interconnectedness of formal mathematical statements and theories, human authors often have difficulties anticipating and tracking the effects of a change in large bodies of symbolic mathematical knowledge. Therefore, the automation of change management is often desirable. But while computers can in principle detect and propagate changes automatically, this process must take the semantics of the underlying mathematical formalism into account. Therefore, concrete management of change solutions are difficult to realize.
The MMT language was designed as a generic declarative language that captures universal structural features while avoiding a commitment to a particular formalism. Therefore, it provides a promising framework for the systematic study of changes in declarative languages. We leverage this framework by providing a generic change management solution at the MMT level, which can be instantiated for arbitrary specific languages.

1 Introduction

Mathematical knowledge is growing at an enormous rate. Even if we restrict attention to formalized mathematics, libraries are reaching sizes that users have difficulties overseeing. Since this knowledge is also highly interconnected, it is getting increasingly difficult for humans to anticipate and follow the effects of changes. Therefore, management of change (MoC) for mathematics has received attention recently.

In this paper, we focus on change management for formalized mathematics, which — contrary to traditional, semi-formal mathematics — permits mechanically computing and verifying declarations. In principle, this should permit change management tools to automatically identify and recheck those declarations that are affected by a change. However, current computer algebra and deduction systems have not been designed systematically with change management in mind. In fact, the question of how to do that is still open.

A major motivation of our work was to provide change management for the LATIN library [CHK+11], a collection of formalizations of logics and related languages in a logical framework. Using the Little Theories approach [FGT92], the LATIN library takes the form of a highly modular and inter-connected network of theories, which creates an urgent need for change management.

We contribute to the solution of this problem by studying change management for the MMT language [RK11]. Because it was introduced as a foundation-

independent, modular, and scalable representation language for formal mathematical knowledge, it is a very promising framework for change management. Firstly, **foundation-independence** means that MMT avoids a syntactic or semantic commitment to any particular formalism. Thus, an MMT-based change management system could be applied to virtually any formal system. Secondly, **modularity** is a well-known strategy to rein in the impacts of changes and has been the basis of successful change management solutions such as [AHMS02]. Thirdly, MMT deemphasizes sequential in-memory processing of declarations in favor of maintaining a **large scale** network of declarations that are retrieved on demand, a crucial prerequisite for revisiting exactly the affected declarations.

We introduce a formal notion of differences between MMT documents, an abstract notion of semantic dependency relation, and a change propagation algorithm that guarantees that validity is preserved. We state our results for a small fragment of MMT, but our treatment extends to the full language. Our solution is implemented within the MMT system [Rab08], thus providing a generic change management system for formal mathematical languages.

In Sect. 2, we briefly introduce the MMT language in order to be self-contained. In Sect. 3, we refine our problem statement and compare it to related work. Then we develop the theory of change management in MMT in Sect. 4 and give an overview of our implementation in Sect. 5.

2 The MMT Language

Theory Graph	$\mathcal{G}$	::=	$\cdot \mid \mathcal{G},\ Mod$
Module Declaration	Mod	::=	$T = \{Sym^*\} \mid v : T \to T = \{Ass^*\}$
Symbol Declaration	Sym	::=	$c : \omega = \omega$
Assignment Declaration	Ass	::=	$c := \omega$
Term	ω	::=	$\bot \mid T?c \mid x \mid \omega\,\omega^+ \mid \omega X.\omega \mid \omega^v$
Variable Context	X	::=	$\cdot \mid X,\ x : \omega = \omega$
Module Identifier	M	::=	$T \mid v$
Theory Identifier	T	::=	`MMT URI`
Morphism Identifier	v	::=	`MMT URI`
Local Declaration Name	c	::=	`MMT Name`

Fig. 1. Simplified MMT Grammar

We will only give a brief overview of MMT and refer to [RK11] for details. The fragment of the MMT grammar that we discuss in this paper is given in Fig. 1. In particular, we have to omit the MMT module system for simplicity. The central notion is that of a **theory graph**, a list of modules, which are theories T or theory morphisms v.

A **theory** declaration $T = \{Sym^*\}$ introduces a theory with name T containing a list of symbol declarations. A **symbol** declaration $c : \omega = \omega'$ introduces

a symbol named c with **type** ω and **definiens** ω'. Both type and definiens are optional. However, in order to reduce the number of case distinctions, we use the special term $\bot$: If the type or definiens is omitted, we assume they are $\bot$.

Terms ω over a theory T are formed from constants $T?c$ declared in T, bound variables x, application $\omega\,\omega_1\,\ldots\,\omega_n$ of a function ω to a sequence of arguments, bindings $\omega X.\omega'$ using a binder ω, a bound variable context X, and a scope ω', and morphism application ω^v. Except for morphism application, this is a fragment of the OpenMath language [BCC+04], which can express virtually every object.

Theory morphism declarations $v : T \to T' = \{Ass^*\}$ introduce a morphism with name v from T to T' containing a list of assignment declarations. Such a morphism must contain exactly one assignment $c := \omega'$ for each undefined symbol $c : \omega = \bot$ in T; here ω' is some term over T'. Theory morphisms extend homomorphically to a mapping of T-terms to T' terms.

Intuitively, a theory morphism formalizes a translation between two formal languages. For example, the inclusion from the theory of semigroups to the theory of monoids (which extends the former with two declarations for the unit element and the neutrality axiom) can be formalized as a theory morphism. More complex examples are the Gödel-Gentzen negative translation from classical to intuitionistic logic or the interpretation of higher-order logic in set theory.

Every Mmt declaration is identified by a canonical, globally unique URI. In particular, the URIs of symbol and assignment declarations are of the form $T?c$ and $v?c$.

Mmt symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules. Their syntax and semantics is determined by the foundation, in which Mmt is parametric. In particular, the validity of a theory graph is defined relative to a type system provided by the **foundation**:

Definition 1. *A **foundation** provides for every theory graph $\mathcal{G}$ a binary relation on terms that is preserved under morphism application. This relation is denoted by $\mathcal{G} \vdash \omega : \omega'$, i.e., we have $\mathcal{G} \vdash \omega : \omega'$ implies $\mathcal{G} \vdash \omega^v : \omega'^v$.*

Constant declarations $c : \omega = \omega'$ in a theory graph $\mathcal{G}$ are valid if $\mathcal{G} \vdash \omega' : \omega$. Thus, a foundation also has to define typing for the special term $\bot$: The judgment $\mathcal{G} \vdash \bot : \omega$ is interpreted as "ω is a well-typed universe, i.e., it is legal to declare constants with type ω". Similarly, $\mathcal{G} \vdash \omega : \bot$ means that ω may occur as the definiens of an untyped constant. This way the foundation can precisely control what symbol declarations are well-formed. Similarly, an assignment $c := \omega$ in a morphism v is valid if $\mathcal{G} \vdash \omega : \omega'^v$ where ω' is the type of c in the domain of v.

Running Example 1 *Below we present a simple* Mmt *theory for propositional logic over two revisions Rev_1 and Rev_2. For simplicity, we will assume that the* Mmt *module system is used and that the symbols* `type`*, $\to$, and λ have been imported from a theory representing the logical framework LF, and that all theory graphs are validated relative to a fixed foundation for LF. PL of Rev_1 introduces*

a type bool of formulas and three binary connectives, the last of which is defined in terms of the other two. This theory is valid. In Rev_2, bool is renamed to form, $\vee$ is deleted, and $\neg$ is added. The other declarations remain unchanged, thus making the theory invalid.

Rev_1

$$
\begin{array}{l}
PL = \{ \\
\quad bool : \texttt{type} = \bot \\
\quad \vee : bool \to bool \to bool = \bot \\
\quad \wedge : bool \to bool \to bool = \bot \\
\quad \Rightarrow : bool \to bool \to bool \\
\qquad = \lambda x.\lambda y.y \vee (x \wedge y) \\
\}
\end{array}
$$

Rev_2

$$
\begin{array}{l}
PL = \{ \\
\quad form : \texttt{type} = \bot \\
\quad \neg : form \to form = \bot \\
\quad \wedge : bool \to bool \to bool = \bot \\
\quad \Rightarrow : bool \to bool \to bool \\
\qquad = \lambda x.\lambda y.y \vee (x \wedge y) \\
\}
\end{array}
$$

3 Related Work

MoC has been applied successfully in a number of **domains** such as software engineering (e.g., [EG89]) or file systems ([Apa00,CVS,Git]). A typical MoC work flow in this setting uses *compilation units*, e.g., the classes of a Java program: These are compiled independently, and a compilation manager can record the dependency relation between the units. In particular, if compilation units correspond to source files, changes in a file can be managed by recompiling all depending source files.

Intuitively, this work flow can be applied to declarative languages for mathematics as well if we replace "compilation" with "validation" where the latter includes, e.g., type reconstruction, rewriting, and theorem proving. However, there are a few key differences. Firstly, the validation units are individual types and definitions (which includes assertions and proofs in Mmt), of which there are many per source file (around 50 on average in the Mizar library [TB85]). Their validation can be expensive, and there may be many dependencies within the same theory and many little theories in the same source file. Therefore, validation units cannot be mapped to files so that the notions of change and dependency must consider fragments of source files. Moreover, since foundations may employ search with backtracking, the validation of a unit U may access more units than the validity of U depends on. Therefore, the dependency relation should not be recorded by a generic Mmt validation manager but produced by the foundation. Recently several systems have become able to produce such dependency relations, in particular Coq and Mizar [AMU11].

MoC systems for mathematical languages can be classified according to the **nature** of changes. In principle, any change in a declarative language can be expressed as a sequence of *add* and *delete* operations on declarations. But using additional change natures is important for scalability. We use *update*s to change the type or definiens of a declaration without changing its Mmt URI, and *rename*s to change only the Mmt URI. We do not use *reordering* operations because Mmt already guarantees that the order of declarations has no effect on

the semantics. More complex natures have been studied in [BC08], which uses *split*s in ontologies to replace one concept with two new ones. Dually, we could consider *merge* changes, which identify two declarations.

Moreover, MoC systems can be classified by the **abstraction level** of their document model. The most concrete physical and bit level are relatively boring, and standard MoC tools operate at the character level treating documents as arrays of lines [Apa00,CVS,Git]. More abstract document models such as XML are better suited for mathematical content [AM10,Wag10] and have been applied to document formats for mathematics [Wag10,ADD+11]. Our work continues this development to more abstract document models by using MMT, which specifically models mathematical data structures. For example, the order of declarations, the flattening of imports, and the resolution of relative identifiers are opaque in XML but transparent in MMT representations. Moreover, MMT URIs are more suitable to identify the validation units than the XPath-based URLs usually employed in generic XML-based change models.

The development graph model [AHMS99], which has been applied to change management in [Hut00,AHMS02], is very similar to MMT: Both are parametric in the underlying formal language, and both make the modular structure of mathematical theories explicit. The main difference is that MMT uses a concrete (but still generic) declarative language for mathematical theories; modular structure is represented using special declarations. Somewhat dually, development graphs use an abstract category of theories using diagrams to represent modular structure; the declarations within a theory can be represented by refining the abstract model as done in [AHM10].

At an even more abstract level, document models can be specific to one foundational language. While foundation-independent approaches like ours can only identify potentially affected validation units, those could determine and possibly repair the impact of a change. That would permit treating even subobjects as validation units. However, presently no systems exists that can provide such foundation-specific information so that such MoC systems remain future work.

Finally we can classify systems based on how they **propagate** changes. Our approach focuses on the theoretical aspect of identifying the (potentially) affected parts. The most natural post-processing steps are to revalidate them, as, e.g., in [AHMS02], or to present them for user interaction as in [ADD+11]. The MMT system can be easily adapted for either one. A very different treatment is advocated in [KK11] based on using only references that include revision numbers so that changes never affect other declarations (because each change generates a new revision number).

4 A Theory of Changes

4.1 A Data Structure for Changes

Just like we can consider only an exemplary fragment of MMT here, we can only consider some of the possible changes. We will only treat changes of declarations

within modules. This is justified because these occur most frequently. However, our treatment can be generalized to changes of any declaration in the full MMT language. The grammar for our formal **language of changes** is given in Fig. 2.

Diff	Δ	::=	$\cdot \mid \Delta \bullet \delta$
Change	δ	::=	$\mathcal{A}(M, c : \omega = \omega) \mid \mathcal{D}(M, c : \omega = \omega) \mid \mathcal{U}(M, c, o, \omega, \omega) \mid \mathcal{R}(T, c, c)$
Component	o	::=	`def` | `tp`
Box Terms	ω	::=	$\boxed{\omega} \mid \boxed{\cdot}$ in addition to existing productions for ω

Fig. 2. The Grammar for MMT Changes

We use terms as validation units because they are the smallest units that can be validated separately by foundations. Therefore, besides adding and deleting whole declarations, we use updates that change a term. In updates, we use **components** o to distinguish between changes to the type (o = `tp`) or the definiens (o = `def`). More precisely:

- $\mathcal{A}(M, c : \omega = \omega')$ **adds** a declaration to the module M
- $\mathcal{D}(M, c : \omega = \omega')$ **deletes** a declaration from the module M.
- $\mathcal{U}(M, c, o, \omega, \omega')$ **updates** component o of declaration $M?c$ from ω to ω'.
- $\mathcal{R}(T, c, c')$ **renames** the declaration c in theory T to c'.

Finally, **Diffs** Δ are sequences of changes. In our implementation, we locate changes even more precisely by referring to subobjects of type and definiens. This is important for user interaction: If an impact has been detected, this permits showing the user exactly what change caused the impact.

Notation 1. In order to unify the cases of changing symbols in a theory and assignments in a morphism, we use the following convention: A declaration $c := \omega'$ in a morphism v abbreviates a declaration $c : \omega = \omega'$ and the components `tp` and `def` are defined accordingly. The type ω is uniquely determined by MMT to ensure the type preservation of morphisms: Its value is τ^v where τ is the type of c in the domain of v. Updates to assignments work in the same way as updates to symbols except that the component `tp` cannot be changed.

We need one additional detail in our grammar: We add two special productions for terms ω, which we call **box terms**. These represent invalid terms that are introduced during change propagation.

$\boxed{\cdot}$ represents a missing term. $\boxed{\omega}$ represents a possibly invalid term ω. More sophisticated box terms can also record the required type, which gives users a hint what change is needed and permits applications to type-check a declaration relative to the box terms in it. We omit this here for simplicity.

Algebraically, the set of diffs Δ is the free monoid generated from changes δ. As we will see below, the operation of applying a diff to a theory graph can be regarded as this monoid acting on the set of theory graphs.

As seen on the right, our diffs are **invertible**. This permits transactions where partially applied diffs are rolled back if they cause an error. This is also useful to offer undo-redo functionality in a user interface.

$\cdot^{-1}$	$= \cdot$
$(\Delta \bullet \delta)^{-1}$	$= \delta^{-1} \bullet \Delta^{-1}$
$\mathcal{A}(M, c : \omega = \omega')^{-1}$	$= \mathcal{D}(M, c : \omega = \omega')$
$\mathcal{D}(M, c : \omega = \omega')^{-1}$	$= \mathcal{A}(M, c : \omega = \omega')$
$\mathcal{R}(T, c, c')^{-1}$	$= \mathcal{R}(T, c', c)$
$\mathcal{U}(M, c, o, \omega, \omega')^{-1}$	$= \mathcal{U}(M, c, o, \omega', \omega)$

In order to talk efficiently about MMT theory graphs, we introduce a few definitions that permit looking up information in the theory graph:

Definition 2 (Lookup in Theory Graphs). *For a theory graph $\mathcal{G}$, we write*

- $\vdash \mathcal{G}(M) = Mod$ *if a module declaration Mod with URI M is present in $\mathcal{G}$.*
- $\mathcal{G} \vdash T?c : \omega = \omega'$ *if T is a theory URI in $\mathcal{G}$ and the symbol declaration $c : \omega = \omega'$ exists in the body of T. We also define the corresponding notation for morphisms.*
- $\vdash \mathcal{G}(M?c) = Sym$ *if* $\vdash \mathcal{G}(M) = Mod$ *and Sym is the declaration with name c in the body of Mod.*
- $\vdash \mathcal{G}(M?c/o) = \omega$ *if* $\vdash \mathcal{G}(M?c) = Sym$ *and ω is the component o of Sym.*
- $\mathcal{G} \vdash \pi$ *if* $\vdash \mathcal{G}(\pi) = Dec$ *for some module or symbol declaration Dec.*

We will now define the **application** of diffs Δ on theory graphs $\mathcal{G}$, which we denote by $\mathcal{G} \ll \Delta$. In MoC tools, this is sometimes called *patching*.

Definition 3. *A diff Δ is called **applicable** to the theory graph $\mathcal{G}$ if $\mathcal{G} \vdash \Delta$ according to the rules in Fig. 3.*

$$\frac{\mathcal{G} \vdash M \quad \mathcal{G} \nvdash M?c}{\mathcal{G} \vdash \mathcal{A}(M, c : \omega = \omega')}\,\mathcal{A}_{dec} \qquad \frac{\mathcal{G} \vdash M?c : \omega = \omega'}{\mathcal{G} \vdash \mathcal{D}(M, c : \omega = \omega')}\,\mathcal{D}_{dec}$$

$$\frac{\vdash \mathcal{G}(T?c/o) = \omega}{\mathcal{G} \vdash \mathcal{U}(T, c, o, \omega, \omega')}\,\mathcal{U}_{sym} \qquad \frac{\vdash \mathcal{G}(v?c/\texttt{def}) = \omega}{\mathcal{G} \vdash \mathcal{U}(v, c, \texttt{def}, \omega, \omega')}\,\mathcal{U}_{ass}$$

$$\frac{\mathcal{G} \vdash T?c \quad \mathcal{G} \nvdash T?c'}{\mathcal{G} \vdash \mathcal{R}(T, c, c')}\,\mathcal{R}_{dec}$$

$$\frac{}{\mathcal{G} \vdash \cdot}\,\Delta_{base} \qquad \frac{\mathcal{G} \vdash \Delta \quad \mathcal{G} \ll \Delta \vdash \delta}{\mathcal{G} \vdash \Delta \bullet \delta}\,\Delta_{dec}$$

Fig. 3. Applicability of Changes

Definition 4 (Change Application). *Given a theory graph $\mathcal{G}$ and a $\mathcal{G}$-applicable change δ, we define $\mathcal{G} \ll \delta$ as follows:*

- *If $\delta = \mathcal{A}(M, c : \omega = \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from $\mathcal{G}$ by adding the declaration $c : \omega = \omega'$ to module M.*
- *If $\delta = \mathcal{D}(M, c : \omega = \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from $\mathcal{G}$ by deleting the declaration $c : \omega = \omega'$ from module M.*
- *If $\delta = \mathcal{U}(M, c, o, \omega, \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from $\mathcal{G}$ by updating the component at $M?c/o$ from ω to ω'.*
- *If $\delta = \mathcal{R}(T, c, c')$ then $\mathcal{G} \ll \delta$ is the graph constructed from $\mathcal{G}$ by renaming the declaration at $T?c$ to $T?c'$.*

Moreover, we define $\mathcal{G} \ll \Delta$ by $\mathcal{G} \ll \cdot = \mathcal{G}$ and $\mathcal{G} \ll (\Delta \bullet \delta) = (\mathcal{G} \ll \Delta) \ll \delta$.

Running Example 2 (Continuing Ex. 1) *We have $Rev_1 \ll \Delta = Rev_2$ where Δ is the diff: $\mathcal{D}(PL, bool : \texttt{type} = \bot) \bullet \mathcal{A}(PL, form : \texttt{type} = \bot) \bullet \mathcal{D}(PL, \vee : bool \to bool \to bool = \bot) \bullet \mathcal{A}(PL, \neg : form \to form = \bot)$. Alternatively, we could use a rename $\mathcal{R}(PL, bool, form)$ instead of the add-delete pair.*

The following simple theorem permits lookups in a hypothetical patched theory graph. This is important for scalability in the typical case where a large $\mathcal{G}$ should be neither changed nor copied:

Theorem 1. *Assume a theory graph $\mathcal{G}$ and a $\mathcal{G}$-applicable diff Δ. Then*

$$
\begin{array}{ll}
\vdash\ (\mathcal{G} \ll \cdot)(M?c/o) & = \mathcal{G}(M?c/o) \\
\vdash\ (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M, c : \omega = \omega')))\,(M?c/o) & = \begin{cases} \omega & \text{if } o = \texttt{tp} \\ \omega' & \text{if } o = \texttt{def} \end{cases} \\
\vdash\ (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M, c : \omega = \omega')))\,(M?c/o) & = \text{undefined} \\
\vdash\ (\mathcal{G} \ll (\Delta \bullet \mathcal{U}(M, c, o, \omega, \omega')))\,(M?c/o) & = \omega' \\
\vdash\ (\mathcal{G} \ll (\Delta \bullet \mathcal{R}(M, c', c)))\,(M?c/o) & = (\mathcal{G} \ll \Delta)\,(M?c'/o) \\
\vdash\ (\mathcal{G} \ll (\Delta \bullet \_))\,(M?c/o) & = (\mathcal{G} \ll \Delta)\,(M?c/o)
\end{array}
$$

where $\_$ is any change not covered by the previous cases.

Proof. This is straightforward to prove using the definitions.

We will now introduce and study an equivalence relation between diffs. Intuitively, two diffs are equivalent if their application has the same effect:

Definition 5. *Given a theory graph $\mathcal{G}$, two $\mathcal{G}$-applicable diffs Δ and Δ' are called $\mathcal{G}$-**equivalent** iff $\mathcal{G} \ll \Delta = \mathcal{G} \ll \Delta'$. We write this as $\Delta \equiv^{\mathcal{G}} \Delta'$.*

Our main theorem about change application is that diffs can be normalized. We need some auxiliary definitions first:

Definition 6. *The **referenced** URIs of a change are defined as follows: For both $\mathcal{A}(M, c : \omega = \omega')$ and $\mathcal{D}(M, c : \omega = \omega')$ they are $M?c/\texttt{tp}$ and $M?c/\texttt{def}$, for $\mathcal{U}(M, c, o, \omega, \omega')$ it is only $M?c/o$, and for $\mathcal{R}(T, c, c')$ they are $T?c/\texttt{tp}$, $T?c/\texttt{def}$,*

*$T?c'/\texttt{tp}$ and $T?c'/\texttt{def}$. Two changes δ and δ' have a **clash** if they reference the same URI.*

*A diff Δ is called **minimal** if there are no clashes between any two changes in Δ. A minimal diff is called **normal** if it is of the form $\Delta_1 \bullet \Delta_2$ where Δ_1 contains no renames and Δ_2 contains only renames.*

Theorem 2. *Reordering the changes in a minimal diff yields an equivalent diff.*

Proof. In a minimal diff, each change affects a different declaration so the order of application is irrelevant.

Definition 7. *$\mathcal{G}' - \mathcal{G}$ is obtained as follows:*

1. *The diff Δ contains the following changes (in any order):*

$$\begin{array}{lll} \mathcal{U}(M,c,o,\omega,\omega') & \text{for} & \mathcal{G}(M?c/o)=\omega,\quad \mathcal{G}'(M?c/o)=\omega',\quad \omega\neq\omega' \\ \mathcal{D}(M?c:\omega=\omega') & \text{for} & \mathcal{G}\vdash M?c:\omega=\omega',\quad \mathcal{G}'\not\vdash M?c \\ \mathcal{A}(M?c:\omega=\omega') & \text{for} & \mathcal{G}'\vdash M?c:\omega=\omega',\quad \mathcal{G}\not\vdash M?c \end{array}$$

2. *We say that a pair (A,D) of changes in Δ* matches *if $A=\mathcal{A}(T,c:\omega=\omega')$ and $D=\mathcal{D}(T,c':\omega=\omega')$. They match* uniquely *if there is no other A' that matches D and no other D' that matches A.*
3. *$\mathcal{G}'-\mathcal{G}$ arises from Δ by removing every uniquely matching pair (A,D) and appending the respective rename $\mathcal{R}(T,c,c')$.*

This definition first generates an add or delete for every URI that exists only in $\mathcal{G}'$ or $\mathcal{G}$, respectively, and $0-2$ updates for every URI that exists in both. Then uniquely matching add-delete pairs are replaced with renames. The uniqueness constraint is necessary to make the last step deterministic.

Running Example 3 (Continuing Ex. 2) *The first step of the computation of the difference $Rev_2 - Rev_1$ yields the diff from Ex. 2. The next steps simplify this diff to $\mathcal{D}(PL, \vee : bool \to bool \to bool = \bot) \bullet \mathcal{A}(PL, \neg : form \to form = \bot) \bullet \mathcal{R}(PL, bool, form)$.*

Theorem 3. *$\mathcal{G}'-\mathcal{G}$ is normal, $\mathcal{G}$-applicable, and $\mathcal{G} \ll (\mathcal{G}'-\mathcal{G}) = \mathcal{G}'$.*

Proof. The proof is straightforward from the definition.

Theorem 4. *If $\mathcal{G}' = \mathcal{G} \ll \Delta$, then there is a normal diff Δ' such that $\Delta \equiv^{\mathcal{G}} \Delta'$.*

Proof. We put $\Delta' = (\mathcal{G} \ll \Delta) - \mathcal{G}$. Then the result follows from Thm. 3.

4.2 A Data Structure for Dependencies

As our validation units are the components of MMT declarations, we need to formulate the validity of MMT theory graphs in a way that permits separate validation of each component:

Definition 8. *A theory graph $\mathcal{G}$ is called* ***foundationally valid*** *if for all symbol or assignment declarations $\mathcal{G} \vdash M?c : \omega = \omega'$ (recall Not. 1), we have $\mathcal{G} \vdash \omega' : \omega$.*

Now we can make formal statements how the validity of a theory graph is affected by changes. First, a typical property of typing relations is that they satisfy a weakening property: Additional information can not invalidate a type inference:

Definition 9. *A foundation is called* ***monotonous*** *if the following rules are admissible for any $A = \mathcal{A}(M, c : \_ = \_)$ and for any $U = \mathcal{U}(M, c, o, \bot, \_)$:*

$$\frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash A}{\mathcal{G} \ll A \vdash \omega : \omega'} \qquad \frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash U}{\mathcal{G} \ll U \vdash \omega : \omega'}$$

Almost all practical foundations for MMT are monotonous. This includes even substructural type theories like linear LF [CP02] because we only require weakening for the set of global declarations, not for local contexts. A simple counter-example is a type theory with induction in which constructors can be added as individual declarations: Then adding a constructor will break an existing induction. But most type theories introduce all constructors in the same declaration.

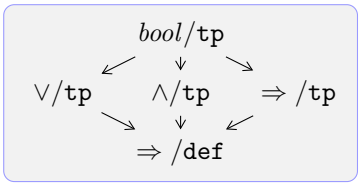
While monotony permits handling additions to a theory graphs in general, we must introduce dependency relations between components to handle updates and deletes. Intuitively, if a validation unit U does not depend on U', then deleting U' is guaranteed not to affect the validity of U:

Definition 10. *A* ***dependency relation*** *for a theory graph $\mathcal{G}$ is a binary relation $\rightsquigarrow$ between declaration components $M?c/o$ and $M'?c'/o'$ such that the following rules are admissible:*

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\rightsquigarrow M'?c'/\mathtt{tp}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \bot) \vdash \bot : \omega}$$

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\rightsquigarrow M'?c'/\mathtt{def}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \bot) \vdash \omega' : \omega}$$

Note that dependency relations are not necessarily transitive. That way changes can be propagated one dependency step at a time, and intermediate revalidation can show that no further propagation is necessary. Of course, the transitive closure (in fact: any larger relation) is again a dependency relation. Our definition of a dependency relation was inspired by the one in [RKS11].



Running Example 4 (Continuing Ex. 3) *For the theory graph Rev_1, we obtain a dependency relation by assuming a dependency whenever a constant occurs in a component. We also assume a dependency from each definiens to its type. The graph in the figure above illustrates this relation.*

4.3 Change Propagation

It is tempting to study the propagation of only a change δ. But this does not cover the important case of transactions, where multiple changes are propagated together. This is typical in practice when an author makes multiple related changes. But it is very complicated to propagate an arbitrary diff. Our key insight is to focus on the **propagation of minimal diffs**. These are very easy to work with, and due to Thm. 3, this is not a loss of generality.

The central idea of our propagation algorithm is to introduce box terms that mark expressions as impacted. This has the advantage that propagation can be formalized as a closure operator on sets of changes so that no additional data structures for impacts are needed.

Definition 11. *For a term ω and a rename $R = \mathcal{R}(T, c, c')$, we define ω^R as the term obtained from ω by replacing all occurrences of $T?c$ with $T?c'$. Similarly, if Δ contains only renames, we define ω^Δ by $\omega^{\Delta \bullet R} = (\omega^\Delta)^R$ and $\omega^{\cdot} = \omega$.*

Definition 12. *For the purposes of Def. 13, we say that a component $M?c/o$ is **modified** by Δ if Δ contains a change of the form $\mathcal{D}(M, c : \_ = \_)$ or $\mathcal{U}(M, c, o, \omega, \_)$.*

The following definition and theorem express our main result. We state them for the special case for a diff that does not add or delete assignments. The general case holds as well but is more complicated.

Definition 13 (Propagation). *Assume a fixed theory graph $\mathcal{G}$ and a fixed dependency relation $\rightarrowtail$ (which we omit from the notation). Assume a $\mathcal{G}$-applicable diff in normal form $\Delta = \Delta_1 \bullet \Delta_2$ that does not contain any adds or deletes of assignments. We define the propagation $\overline{\Delta}$ of Δ in multiple steps as follows:*

1. *Δ_1' contains the following changes (in any order): whenever $M?c/o \rightarrowtail M'?c'/o'$ and $M?c/o$ is modified by Δ_1, the change*

$$\mathcal{U}(M', c', o', \omega, \boxed{\omega}) \qquad \text{for} \qquad \mathcal{G} \ll \Delta \vdash M'?c'/o' = \omega$$

2. *Δ_2' contains the following changes (in any order): whenever $\mathcal{R}(T, c, \_) \in \Delta_2$ and $T?c/o \rightarrowtail M'?c'/o'$, the change*

$$\mathcal{U}(M', c', o', \omega, \omega^{\Delta_2}) \qquad \text{for} \qquad \mathcal{G} \ll \Delta \bullet \Delta_1' \vdash M'?c'/o' = \omega$$

3. *Δ_3' contains the following changes for every morphism $\mathcal{G} \vdash v : T \to T'$ (in any order):*
   - *whenever $\mathcal{A}(T, c : \_ = \bot) \in \Delta$ or $\mathcal{U}(T, c, \texttt{def}, \_, \bot) \in \Delta$, the change*

$$\mathcal{A}(v, c := \boxed{\cdot})$$

   - *whenever $\mathcal{D}(T, c : \_ = \bot) \in \Delta$ or $\mathcal{U}(T, c, \texttt{def}, \bot, \_) \in \Delta$, the change*

$$\mathcal{D}(v, Ass) \qquad \text{for} \qquad \mathcal{G} \ll \Delta \bullet \Delta_1' \bullet \Delta_2' \vdash v?c = Ass$$

– *whenever* $\mathcal{R}(T, c, c') \in \Delta$ *and* $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash T?c/\texttt{def} = \bot$ *and* $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash v?c := \omega$*, the changes*

$$\mathcal{D}(v, c := \omega), \mathcal{A}(v, c' := \omega)$$

4. $\overline{\Delta}$ *is obtained as* $\Delta'_1 \bullet \Delta'_2 \bullet \Delta'_3$*.*

Intuitively, Δ'_1 updates all impacted terms to box terms. If $\rightsquigarrow$ is transitive, this includes all terms that depended on the now boxed terms. Δ'_2 updates all references to renamed declarations to the new name.

Δ'_3 ensures that all morphisms have exactly one assignment for every undefined constant in the domain. The first two subcases add empty assignments or delete existing ones if necessary. The third subcase renames those assignments where the corresponding constant in the domain has been renamed.

Theorem 5. *Consider the situation of Def. 13. Assume that the foundation is monotonous, that* $\mathcal{G}$ *is foundationally valid, and that* $\rightsquigarrow$ *is transitive. Let* $\mathcal{G}' = \mathcal{G} \ll \Delta \bullet \overline{\Delta}$*, and let* $\mathcal{G}^*$ *be a theory graph that arises from* $\mathcal{G}'$ *by replacing every box term with a term that type checks in the sense of Def. 8. Then* $\mathcal{G}^*$ *is foundationally valid.*

Proof. Let us first consider the special case without renames or morphisms. We apply Def. 8 to $\mathcal{G}^*$. Due to Δ'_1 and the transitivity of $\rightsquigarrow$, all possibly ill-typed terms have been replaced with box terms in $\mathcal{G}'$; and according to the assumptions, these are replaced with well-typed terms in $\mathcal{G}^*$. Thus, the claim follows.

If there are renames, care must be taken to update all references to the renamed declarations. If there are adds, care must be taken to guarantee the totality of morphisms. Both conditions are already fulfilled in $\mathcal{G}'$. We omit the details.

A typical situation where we would apply Thm. 5 is after a user made the changes Δ. Then propagation marks all terms that have to be revalidated and — if not well-typed — replaced interactively with well-typed terms. The theorem guarantees the resulting graph is valid again.

$$\begin{aligned} PL = \{ & \\ & form : \texttt{type} = \bot \\ & \neg : form \to form = \bot \\ & \wedge : form \to form \to form = \bot \\ & \Rightarrow : form \to form \to form \\ & \quad = \boxed{\lambda x.\lambda y.y \vee (x \wedge y)} \\ \} & \end{aligned}$$

Running Example 5 (Continuing Ex. 3 and 4)
Using $\Delta = Rev_2 - Rev_1$ *and the dependency relation from Ex. 4, we compute* $\overline{\Delta}$*. First* $\Delta = \Delta_1 \bullet \Delta_2$ *with* $\Delta_1 = \mathcal{D}(PL, \vee : bool \to bool \to bool = \bot) \bullet \mathcal{A}(PL, \neg : form \to form = \bot)$ *and* $\Delta_2 = \mathcal{R}(PL, bool, form)$*. Then*

$$\overline{\Delta'_1} \quad = \quad \mathcal{U}(PL, \Rightarrow, \texttt{def}, \lambda x.\lambda y.y \vee (x \wedge y), \boxed{\lambda x.\lambda y.y \vee (x \wedge y)})$$

as well as $\overline{\Delta'_2} = \mathcal{U}(PL, \wedge, \texttt{tp}, bool \to bool \to bool, form \to form \to form) \bullet \mathcal{U}(PL, \Rightarrow, \texttt{tp}, bool \to bool \to bool, form \to form \to form)$ *and* $\overline{\Delta'_3} = \cdot$*. Finally, the theory graph* $Rev_1 \ll \Delta \bullet \overline{\Delta}$ *is shown above. As stated in Thm. 5, it becomes foundationally valid after replacing the box term with a term of the right type.*

5 A Generic Change Management API

Implementation We have implemented the data structures and algorithms from Sect. 4 as a part of the MMT API [Rab08]. In fact, our implementation covers a much larger fragment of MMT than discussed in this paper.

In particular, the API now contains functions that compute the **difference** $\mathcal{G}' - \mathcal{G}$ of two theory graphs. The difference of two modules can be computed as well. The two arguments can either be provided directly or the previous revision can be pulled automatically from an SVN repository.

We also added functions for **change propagation** that enrich a normal diff with its direct impacts according to Def. 13. The generated box terms are represented as OPENMATH error objects. During the propagation algorithm, we make crucial use of Thm. 1 to increase the efficiency.

Both of these algorithms are implemented foundation-independently. The foundation is only needed to obtain the dependency relation and to revalidate the impacted declarations. Both are special cases of type checking.

The MMT API relegates a **type checking** obligation $\omega' : \omega$ to a plugin for the respective foundation. In particular, there is a plugin for a monotonous foundation for the logical framework LF [HHP93], which induces implementations of type checking for all formal systems represented in LF (i.e., for a lot of formal systems [CHK+11]).

The plugin interface is such that the plugin calls back to the main system whenever it needs to look up any component $M?c/o$. In the simplest case, we can trace these callbacks to obtain the set of components $Used(\omega', \omega)$ that were used to validate $\omega' : \omega$. When the system validates a theory graph $\mathcal{G}$ according to Def. 8, we obtain a **dependency relation** by putting for every symbol or assignment $\mathcal{G} \vdash M'?c' : \omega = \omega'$

$$M?c/o \not\rightarrow M'?c'/\texttt{tp} \quad \text{if} \quad M?c/o \in Used(\bot, \omega)$$

$$M?c/o \not\rightarrow M'?c'/\texttt{def} \quad \text{if} \quad M?c/o \in Used(\omega', \omega)$$

$$M'?c'/\texttt{tp} \not\rightarrow M'?c'/\texttt{def}$$

Note that we first check the type of the declaration and then separately check the definiens against that type even though the latter implies the former. This is important because the type will usually have much less dependencies than the definiens.

This dependency relation is stored in the MMT ontology, which MMT maintains together with the content [HIJ+11]. Alternatively, the foundation can explicitly provide a dependency relation, or we can import dependency relations externally, e.g., the ones from [AMU11].

Application We have applied the resulting system to obtain a change management API for the LATIN library. Using the MMT plugins for LF — the language underlying the LATIN library — we obtain a foundation that validates the library and computes a dependency relation for it. Fig. 4 gives a summary of the

dependency relation, where we include only the about 1700 components falling into the fragment of MMT treated in this paper. The tables group the components by the number of components that they depend on (left) or that depend on them (right). This includes only direct dependencies — taking the transitive closure increases the numbers by about 20 %.

dependencies	**components (%)**
$0-5$	1373 (79)
$6-10$	271 (15.6)
$11-15$	81 (4.7)
$16-26$	13 (0.7)

impacts	**components (%)**
$0-5$	1504 (86.5)
$6-10$	101 (5.8)
$11-25$	76 (4.4)
$26-50$	31 (1.8)
$50-449$	26 (1.5)

Fig. 4. Components grouped by dependencies and impacts

The number of dependencies and impacts is generally low. This is a major benefit of our choice of using type and definiens as separate validation units, which avoids the exponential blowup one would otherwise expect. Indeed, on average a type has 3 times as many impacts as a definiens.

Our differencing algorithm can detect and propagate changes easily, and it is straightforward to revalidate the impacted components. The numbers show that even manual inspection (as opposed to automatic revalidation) is feasible in most cases: For example, changes to 86 % of the components impact only 5 or less components. Even if the number of impacted components is so small, it is usually very difficult for humans to identify exactly which components are impacted. Our MoC infrastructure, on the other hand, does not only identify them automatically but also guarantees that all other components stay valid.

6 Conclusion

We have presented a theory of change management based on the MMT language including difference, dependency, and impact analysis. As MMT is foundation-independent, our work yields a theory of change management for an arbitrary declarative language. Our work is implemented as a part of the MMT API and thus immediately applicable to any language that is represented in MMT. The latter includes in particular the logical framework LF and thus every language represented in it.

Because we use fine-grained dependencies, change propagation can identify individual type checking obligations (which subsume proof obligations) that have to be revalidated. The MMT API already provides a scalable framework for validating individual such obligations efficiently. Therefore, our work provides the foundation for a large scale change management system for declarative languages.

While our presentation has focused on a small fragment of MMT, the results can be generalized to the whole MMT language, in particular the module system.

Presently the most important missing feature is a connection between the MMT abstract syntax and the concrete syntax of individual languages. Therefore, change management currently requires an export into MMT's abstract syntax (which exists for, e.g., Mizar [TB85], TPTP [SS98], and OWL [W3C09]). Consequently, **future work** will focus on developing fast bidirectional translations between human-friendly source languages and their MMT content representation. If these include fine-grained cross-references between source and content, MMT can propagate changes into the source language; this could happen even while the user is typing.

More generally, this approach extends to pure mathematics where the source language is, e.g., LaTeX. If the source is formalized manually, it is sufficient to include cross-references in the above sense. Then changes in the LaTeX source can be treated and propagated like changes in the formalization. Alternatively, we can avoid a manual formalization if certain annotations are present in the source: firstly, annotations that map a line number to the identifier of the statement (definition, theorem, etc.) made at that line; secondly, annotations that explicate the dependency relation between statements. For example, the sTeX package for LaTeX permits such annotations in a way that supports automated extraction.

References

ADD+11. S. Autexier, C. David, D. Dietrich, M. Kohlhase, and V. Zholudev. Workflows for the Management of Change in Science, Technologies, Engineering and Mathematics. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 164–179. Springer, 2011.

AHM10. S. Autexier, D. Hutter, and T. Mossakowski. Change Management for Heterogeneous Development Graphs. In S. Siegler and N. Wasser, editors, *Verification, Induction, Termination Analysis, Festschrift in honor of Christoph Walther*. Springer, 2010.

AHMS99. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

AHMS02. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002.

AM10. S. Autexier and N. Müller. Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In M. Gormish and R. Ingold, editors, *Proceedings of 10th ACM Symposium on Document Engineering (DocEng2010)*, 2010.

AMU11. Jesse Alama, Lionel Mamane, and Josef Urban. Dependencies in Formal Mathematics. *CoRR*, abs/1109.3687, 2011.

Apa00. Apache Software Foundation. Apache Subversion, 2000. see http://subversion.apache.org/.

BC08. A. Bundy and M. Chan. Towards Ontology Evolution in Physics. In W. Hodges and R. de Queiroz, editors, *Logic, Language, Information and Computation*, pages 98–110. Springer, 2008.

BCC+04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

CHK+11. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.

CP02. I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002.

CVS. Concurrent Versions System: The open standard for Version Control. Web site at http://cvs.nongnu.org/. seen February 2012.

EG89. C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM, 1989.

FGT92. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

Git. Git. Web Site at: http://git-scm.com/.

HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

HIJ+11. F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.

Hut00. D. Hutter. Management of change in structured verification. In *Proceedings Automated Software Engineering, ASE-2000*, pages 23–34, 2000.

KK11. Andrea Kohlhase and Michael Kohlhase. Versioned links. In *Proceedings of the 29th annual ACM international conference on Design of communication (SIGDOC)*, 2011.

Rab08. F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.

RK11. F. Rabe and M. Kohlhase. A Scalable Module System. see http://arxiv.org/abs/1105.0548, 2011.

RKS11. F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2011.

SS98. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

W3C09. W3C. OWL 2 Web Ontology Language, 2009. http://www.w3.org/TR/owl-overview/.

Wag10. M. Wagner. *A change-oriented architecture for mathematical authoring assistance*. PhD thesis, Universität des Saarlands, 2010.