

Formal Software Development in MAYA

Serge Autexier and Dieter Hutter*

German Research Center for Artificial Intelligence, Stuhlsatzenhausweg 3,
D-66123 Saarbruecken, Germany, Email: {autexier | hutter}@dfki.de

Abstract. The formal development of industrial-size software is an error-prone and therefore an evolutionary process. Verifying formal specifications usually reveals hidden errors causing the change of parts of the specification. Also adding new functionality will result in changes of the specification which always endangers the verification work already done. In this paper we describe the system MAYA which maintains formal developments. The MAYA-system supports an evolutionary formal development since it allows users to specify and verify developments in a structured manner, incorporates a uniform mechanism for verification in-the-large to exploit the structure of the specification, and maintains the verification work already done when changing the specification. MAYA relies on development graphs as a uniform representation of structured specifications, which enables the use of various (structured) specification languages to formalize the software development. Moreover, MAYA allows the integration of different theorem provers to deal with the actual proof obligations arising from the specification, i.e. to perform verification in-the-small.

1 Introduction

Formal methods are used in the software development process to increase the security and safety of software. The software systems as well as their requirement specifications are formalized in a textual manner in some specification language like CASL [4] or VSE-SL [7]. The specification languages provide constructs to structure the textual specifications to ease the reuse of components. Exploiting this structure, e.g. by identifying shared components in the system specification and the requirement specification, can result in a drastic reduction of the proof obligations, and hence of the development time which again reduces the overall project costs.

Creating the arising proof obligations in a naive way by postulating all parts of the security requirements as theorems of the system design would result in umpteen redundant proof obligations relating to common datastructures. Exploiting the given (graph-) structure of specifications allows one to reveal this redundancy. In [2] we proposed the use of development graphs to represent defined and postulated properties of formal specifications in a logical way. We will introduce a calculus \mathcal{DG} to verify postulated properties. The calculus rules decompose conjectures between specifications into conjectures between parts of

* This work was supported by the German Ministry for Education and Technology (BMBF).

the specification and check whether some of those are already subsumed by the specification structure. We denote this activity by *verification in-the-large*. Those conjectures that can neither be further decomposed nor subsumed give rise to the proof obligations that must actually be tackled by some theorem prover, which is denoted by *verification in-the-small*.

However, the logical formalization of software systems is error-prone. Since even the verification of small-sized industrial developments requires several person months, specification errors revealed in late verification phases pose an incalculable risk for the overall project costs. An *evolutionary formal development* approach is absolutely indispensable. In all applications so far development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a *formal reflection* of partial developments rather than just a way to assure and prove more or less evident facts. Revealed flaws give rise to changes of the specification and to the need for an update of all the proof work done before. Loosing this work would be an incalculable risk of the overall project costs for large verification tasks that arise in practice. Hence, we introduce a management of change based on the notion of development graphs to incrementally adjust existing proofs to a changed specification, while preserving as much information about proven conjectures as possible.

We start with a general overview of how MAYA supports the formal development of software in Sect. 2. In Sect. 3 we introduce the formal notion of development graphs that underly the MAYA system. Sect. 4 describes the general methodology to define translation of an input specification given in some specification language \mathcal{L} into development graph. The methodology is illustrated by providing the definition of the translation of CASL-specifications into development graphs. Sect. 5 is concerned with the computation of differences between specifications and how to update their logical representation inside the development graph. Sect. 6 presents the management of change for both, verification in-the-large and verification in-the-small, while we discuss its implementation in MAYA and related work in Sects. 7 and 8.

2 General Overview

A user interacts with the MAYA-system via a formal specification language. Such a formal specification gives rise to a logical modeling of the specification and the proof obligations arising from commitments made inside the specification. Examples of such commitments are that theories satisfy specific properties specified inside so-called security models or that basic specifications imply specific properties, so-called theorems. Translating the textual specification into a structured logic representation, which we call a development graph, proof obligations are denoted either as so-called theorem links between theories, indication that both theories are related to each other wrt. a specific property or as so-called theorems representing lemmata inside a particular theory. The problem of establishing properties between theories is dealt with inside the MAYA-system utilizing the

overall structure of the graph until we end up with elementary proof obligations which are tackled by external theorem provers.

The user performs changes of her specification always on the textual representation, which gives rise to the problem of tracking the changes in the textual specification to arising changes in the corresponding logical representation and last not least also in changes or adaptation of the proofs. Changed textual specifications are translated into their logical counterpart. The analysis which parts of the specification have changed is done on the logical level. The result of the analysis is an operational description of how to adjust the existing development graph such that it fits the changed specification. The adjustment of the proof work is based on the operation description incorporating precompiled knowledge how individual operations will affect the validity of proofs.

3 Development Graphs

In order to define development graphs we start with a short recapitulation of the basics of logics as they are given, for instance, in [11]. Thereby the notion of a logic is based on the notions of an *institution* and an *entailment system*.

An *institution* $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ consists of a category of signatures \mathbf{Sign} , two functors \mathbf{Sen} and \mathbf{Mod} giving respectively the set of valid sentences $\mathbf{Sen}(\Sigma)$ and the models $\mathbf{Mod}(\Sigma)$ for some signature, and a satisfaction relation $\models_{\Sigma} \subseteq \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ for each signature Σ . An *entailment system* $\mathcal{E} = (\mathbf{Sign}, \mathbf{Sen}, \vdash)$ consists of a category \mathbf{Sign} of *signatures*, a functor $\mathbf{Sen}: \mathbf{Sign} \rightarrow \mathbf{Set}$ giving the set of *sentences* over a given signature, and entailment relations $\vdash_{\Sigma} \subseteq |\mathbf{Sen}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ with the following properties:

1. *reflexivity*: for any $\varphi \in \mathbf{Sen}(\Sigma)$, $\{\varphi\} \vdash_{\Sigma} \varphi$,
2. *monotonicity*: if $\Gamma \vdash_{\Sigma} \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_{\Sigma} \varphi$,
3. *transitivity*: if $\Gamma \vdash_{\Sigma} \varphi_i$, for $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_{\Sigma} \psi$, then $\Gamma \vdash_{\Sigma} \psi$,
4. *\vdash -translation*: if $\Gamma \vdash_{\Sigma} \varphi$, then for any $\sigma: \Sigma \rightarrow \Sigma'$ in \mathbf{Sign} , $\sigma[\Gamma] \vdash_{\Sigma'} \sigma(\varphi)$.

A logic is then defined as a 5-tuple $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \vdash, \models)$ such that: (1) $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ is an institution (denoted by $inst(\mathcal{LOG})$), (2) $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ is an entailment system (denoted by $ent(\mathcal{LOG})$), and (3) the following *soundness condition* is satisfied: for any $\Sigma \in |\mathbf{Sign}|$, $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ and $\varphi \in \mathbf{Sen}(\Sigma)$, $\Gamma \vdash_{\Sigma} \varphi$ implies $\Gamma \models_{\Sigma} \varphi$. Throughout the rest of the paper, we will work with an arbitrary but fixed logic $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \vdash, \models)$.

Structured specifications are represented on a logical basis as development graphs. The nodes of such a graph represent individual theories. Definition links are used to specify theory inclusions (with respect to some morphism) between different theories. The axiomatic specification of a single theory is distributed to the subgraph of the corresponding node since the definition of the theory of a node depends on the local axioms attached to the node combined with the axioms or theories of the nodes imported by definition links.

In order to formulate proof obligations denoting properties between different theories (*verification in-the-large*) we introduce so-called theorem links. These

links are similar in appearance to definition links but do not influence the theories denoted by the nodes. Formally we define

Definition 1. A development graph \mathcal{S} is a directed graph $\langle \mathcal{N}, \Psi \rangle$, where

- \mathcal{N} is a finite set of nodes. Each node $N \in \mathcal{N}$ is a pair (Σ_i^N, Φ_i^N) consisting of a **local signature** Σ_i^N and a set of **local axioms** $\Phi_i^N \subset \mathbf{Sen}(\Sigma^N)$ of N .
- $\Psi = \Psi_D \uplus \Psi_T$ is a finite set of directed links between elements of \mathcal{N} consisting of an acyclic¹ set Ψ_D of **definition links** and a set Ψ_T of **theorem links**. Each link from a node M to a node N in Ψ is either **global** (denoted $M \xrightarrow{\sigma} N$) or **local** (denoted $M \xrightarrow{\sigma} N$) and is annotated with a signature morphism $\sigma : \Sigma^M \rightarrow \Sigma^N$.

For all $N \in \mathcal{N}$ the **signature** Σ^N of N is given by:

$$\Sigma^N = \Sigma_i^N \cup \{ \sigma(f) \mid f \in \Sigma^M, M \xrightarrow{\sigma} N \in \Psi_D \} \cup \{ \sigma(f) \mid f \in \Sigma_i^M, M \xrightarrow{\sigma} N \in \Psi_D \}$$

For the implementation, we represent a signature morphism σ by a set of finite pairs (f_{in}, f_{out}) with $\sigma(f) = g$ if there is a pair $(f, g) \in \sigma$ and $\sigma(f) = f$ otherwise.

The proof theoretical semantics of a development graph is given by the following definition:

Definition 2. Let $\mathcal{S} = \langle \mathcal{N}, \Psi \rangle$ be a development graph and $\Delta \subseteq \Psi$, Δ acyclic. Let $N \in \mathcal{N}$, then the **theory** $Th_\Delta(N)$ of N relative to Δ is defined by

$$Th_\Delta(N) = \left[\Phi_i^N \cup \bigcup_{K \xrightarrow{\sigma} N \in \Delta} \sigma(Th_\Delta(K)) \cup \bigcup_{K \xrightarrow{\sigma} N \in \Delta} \sigma(\Phi_i^K) \right]^{\vdash_{\Sigma^N}}$$

where $[\Gamma]^{\vdash_{\Sigma^N}}$ denotes the closure of Γ under the entailment relation \vdash_{Σ^N} . The **theory** $Th(N)$ of N is defined as $Th_{\Psi_D}(N)$.

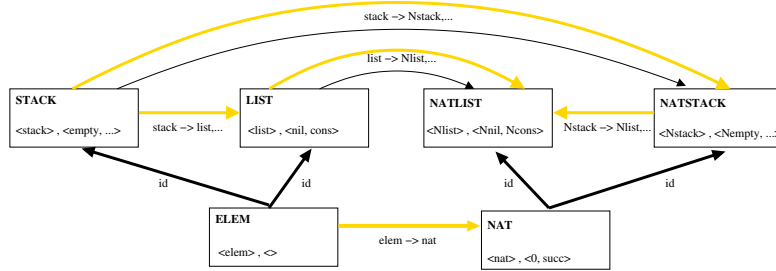


Fig. 1. Structured Specifications of NATLIST

Fig. 1 presents a development graph for lists LIST and stacks STACK over arbitrary elements and their respective instantiations to lists NATLIST and stacks

¹ A set of links is acyclic iff the graph denoted by these links is acyclic.

NATSTACK over natural numbers. While we included the local signatures of the nodes in the figure we have omitted the local axioms because of shortage of space. The theories of generic lists LIST and generic stacks STACK are defined with the help of a theory ELEM, indicated by the global definition links from ELEM to LIST and STACK, and local axioms in LIST and STACK specifying that LIST and STACK are freely generated. The global theorem link between STACK and LIST represents the proof obligation, that STACK can be implemented by LIST.

The theories of lists and stacks of natural numbers, NATLIST and NATSTACK, are instantiations of generic lists and stacks with natural numbers NAT. Thus, both NATLIST and NATSTACK import NAT via a global definition link and the local axioms of LIST and STACK respectively via local definition links. The global theorem links between LIST and NATLIST, and STACK and NATSTACK denote the proof obligations that NATLIST and NATSTACK are respective instances of LIST and NAT. The global theorem link between ELEM and NAT denotes the proof obligation that the actual parameter NAT satisfies the requirements of the formal parameter ELEM. The proof obligation that NATSTACK can be implemented by NATLIST is represented by a global theorem link from NATSTACK to NATLIST.

This toy example illustrates how the important concepts from structured specifications are represented with development graphs. In practice the system and requirement specifications and hence the resulting development graph are much larger. A development graph of a typical size is sketched in Fig. 2.

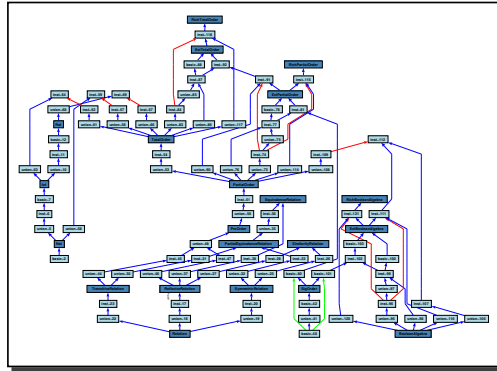


Fig. 2. Example of development graphs for real software engineering problems

4 Translating Specifications into Development Graphs

In formal software development, specification languages like CASL [4] or VSE-SL [7] are used to describe a system and its requirements as well as proof obligations arising from the requirement that the system must satisfy these requirements. The notion of development graph from Sect. 3 provides a uniform representation of general structured formal developments. The representation is independent of any specification language, which eases the use of different specification languages as input format. To use a specification language \mathcal{L} as input for MAYA we must define a mapping from \mathcal{L} -specifications into the representation language of development graphs. Roughly speaking, such a translation of some \mathcal{L} -specification S works as follows:

- it maps basic (unstructured) parts of the specification, like the specification of simple abstract datatypes, into a collection of (local) axioms within some theory node,
- it translates the structuring operations of the specification language \mathcal{L} into the notion of definition links, and
- it reformulates proof obligations given in the specification either into theorem links connecting corresponding theories or into conjectures considered as lemmata of a specific theory in the development graph.

Obviously, the definition of such a translation entails the requirement to prove the adequacy of the translation of \mathcal{L} -specifications into the notion of development graphs.

In the following we illustrate the translation of specifications into development graphs by giving the rigorous definition of the mapping of CASL-specifications into development graphs.

4.1 Translation of CASL into Development Graphs

For the translation of CASL into development graphs we restrict ourselves to a subset of the CASL-language, namely CASL-specifications without the structuring operations hiding and freeness (cf. [4]) and without architectural specifications. We use the CATS-parser [12] to parse the specifications, which also provides encoding of the logical parts of specifications from the CASL-logic into different target logics. Since the logic underlying the actual implementation of development graphs in MAYA is many-sorted higher-order logic, we use second-order logic without subsorting as the target logic. Furthermore, the CATS-parser performs a static analysis of the specifications.

Basic specifications constitute the nucleus of CASL-specifications. Consisting of a (local) signature and a set of axioms, basic specifications are translated into the higher-order logic with the help of the CATS-parser which provides second-order logic encodings. *Structured specifications* are used to combine *basic specifications* with the help of structuring operations, like for example extension (**then**), union (**and**), or to actualize parameterized specifications.

A CASL-specification itself consists of a list of specification parts which are either *named specifications*, *named views*, or *fitting views*, which are constructed with the help of structured specifications. We will describe these constructs in more details lateron.

To translate a CASL-specification, we define a top-level translation function τ_{CASL} that iterates over the specification parts and that iteratively constructs the corresponding development graph. The major difficulty of this translation is the encoding of the so-called *linear visibility constraint*, which is implicitly given by the CASL-semantics: the semantics of a specification part depends on its global environment which depends on previously parsed specification parts. Thus besides a list of CASL-specifications, τ_{CASL} requires the global environment as an additional argument that provides information about translated specifications and views before parsing the actual list of CASL-specifications.

τ_{CASL} returns the actualized development graph enlarged by nodes and links corresponding to the parsed specification list and provides the new global environment. Inside this translation information we accumulate for instance the information how named specifications or named views have been translated. Lateron we make use of this information to perform actualizations. Formally, we define τ_{CASL} by recursion as follows:

$$\begin{aligned} \tau_{CASL}(\langle \rangle, \mathcal{S}, \mathcal{P}) &:= (\mathcal{S}, \mathcal{P}) \\ \tau_{CASL}(\langle spec-part, restlist \rangle, \mathcal{S}, \mathcal{P}) &:= \tau_{CASL}(restlist, \mathcal{S}', \mathcal{P}') \\ &\text{with } (\mathcal{S}', \mathcal{P}') := \tau_{part}(spec-part, \mathcal{S}, \mathcal{P}) \end{aligned}$$

τ_{part} is the corresponding translation function for the individual specification parts. It takes as arguments a structured specification *spec-part*, a development graph \mathcal{S} , and a translation information \mathcal{P} . It provides a pair $(\mathcal{S}', \mathcal{P}')$ where \mathcal{S}' is a new development graph that includes the subgraph resulting of the translation of the structured specifications and \mathcal{P}' is the updated translation information.

Since CASL specification parts are constructed with the help of structured specifications, we will introduce a third translation function τ to translate structured specifications into development graphs. τ takes as argument the specification *Spec*, the development graph \mathcal{S} and translation information \mathcal{P} . It returns a triple $(\mathcal{I}', \mathcal{S}', \mathcal{O}')$ with \mathcal{S}' being the development graph updated with the translated *Spec*. The linear visibility constraints for structured CASL-specifications defines *how* specification parts are visible when parsing the next specification. Translating this requirement in terms of development graphs, the development graphs of previous specifications have to be imported to the graph of the specification part under consideration. Therefore τ returns also a set \mathcal{I}' of nodes denoting the import interface to the global environment and a node \mathcal{O}' which corresponds to the exported global environment.

4.2 Named Specifications.

A named specification in CASL is of the form

$$\mathbf{spec} \text{ } SN[SP_1] \dots [SP_n] \mathbf{given} \text{ } SP'_1, \dots, SP'_m = SP \mathbf{end}$$

where *SN* is the name of the specification, SP_1, \dots, SP_n are the parameter specifications, SP'_1, \dots, SP'_m specifications that are visible inside the parameter specifications, and *SP* is the body of the named specification. For the definition of the translation, we use the translation function τ for specification bodies. This function takes three arguments: (1) the CASL-specification body to translate, (2) the actual development graph, (3) the actual translation information. This function returns a triple $(\mathcal{I}, \mathcal{S}, \mathcal{O})$, containing the new development graph \mathcal{S} , the theory nodes \mathcal{I} which import the visible environment of the argument specification body, and the theory node \mathcal{O} which exports the new visible environment.

In order to satisfy the visibility rules from CASL the translation is done according to the following steps

1. The union of the “given” specifications SP'_1, \dots, SP'_m is translated:

$$\langle \mathcal{I}_0, \mathcal{S}_0, O_0 \rangle := \tau(SP'_1 \text{ and } \dots \text{ and } SP'_m, \mathcal{S}, \mathcal{P})$$

2. Parameter specifications SP_i are translated for all i , $1 \leq i \leq n$ by:

$$\langle \mathcal{I}_i, \mathcal{S}_i, O_i \rangle := \tau(SP_i, \mathcal{S}_{i-1}, \mathcal{P})$$

and new definition links are inserted to import the output theory node O_0 into all elements of all \mathcal{I}_i with $1 \leq i \leq n$: Let \mathcal{S}_n be $\langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ then

$$\mathcal{S}_{n+1} := \langle \mathcal{N}, (\Psi_D \cup \{O_0 \xrightarrow{\lambda} N \mid N \in \mathcal{I}_i \wedge 1 \leq i \leq n\}) \uplus \Psi_T \rangle$$

3. Next, the body SP of the named specification is translated by:

$$\langle \mathcal{I}_{n+2}, \mathcal{S}_{n+2}, O_{n+2} \rangle := \tau(SP, \mathcal{S}_{n+1}, \mathcal{P})$$

and new definition links are added to import the parameters O_1, \dots, O_n into each element N of \mathcal{I}_{n+2} obtain by translating the body. Let \mathcal{S}_{n+2} be $\langle \mathcal{N}', \Psi'_D \uplus \Psi'_T \rangle$ then

$$\begin{aligned} \langle \mathcal{I}_{name}, \mathcal{S}_{name}, O_{name} \rangle := \\ \langle \mathcal{I}_0, \langle \mathcal{N}', (\Psi'_D \cup \{O_i \xrightarrow{\lambda} N \mid N \in \mathcal{I}_{n+2} \wedge 1 \leq i \leq n\}) \uplus \Psi'_T \rangle, O_{n+2} \rangle \end{aligned}$$

As translation information we store that the CASL-specification of name SN has the top-level output node O_{name} , and add the information about the translation of the parameter specifications $(\langle \mathcal{I}_i, O_i \rangle)$ with $1 \leq i \leq n$ as well the output node O_0 of the given part. This is used for the translation of instantiations of SN .

The final result of the translation of the named specification definition is then

$$\begin{aligned} \tau_{part}(\text{spec } SN[SP_1] \dots [SP_n] \text{ given } SP'_1, \dots, SP'_m = SP \text{ end}, \mathcal{S}, \mathcal{P}) \\ := \langle \mathcal{S}_{name}, \mathcal{P} \cup [SN, O_{name}, (\langle \mathcal{I}_1, O_1 \rangle, \dots, \langle \mathcal{I}_n, O_n \rangle), O_0] \rangle \end{aligned}$$

4.3 Views.

A named view in CASL is of the general form

$$\text{view } VN [SP_1] \dots [SP_n] \text{ given } SP_1, \dots, SP_m : SP \text{ to } SP' = SM \text{ end. (1)}$$

$[SP_1] \dots [SP_n] \text{ given } SP_1, \dots, SP_m : SP$ represents a specification similar to the definition of named specifications. The view constitutes the proof obligation that the models of this specification can be mapped to models of SP' using the signature morphism given by SM .

To translate a named view we translate a dummy named specification

$$\text{spec } SN [SP_1] \dots [SP_n] \text{ given } SP_1, \dots, SP_m = SP$$

which results as described in the previous paragraph in

$$\langle \mathcal{S}_{name}, \mathcal{P} \cup [SN, O_{name}, (\langle \mathcal{I}_1, O_1 \rangle, \dots, \langle \mathcal{I}_n, O_n \rangle), O_0] \rangle.$$

Next we translate the structured specification SP' by

$$\langle \mathcal{I}', \mathcal{S}', O' \rangle := \tau(SP', \mathcal{S}_{name}, \mathcal{P})$$

and add a global theorem link from O_{name} to O' with the morphism SM . The final result of τ_{part} on (1) consists of this new development graph and the parameter information for the named view. Let $\mathcal{S}' = \langle \mathcal{N}', \Psi'_D \uplus \Psi'_T \rangle$ then

$$\begin{aligned} \tau_{part}(\mathbf{view} \ VN \ \dots \ \mathbf{end}, \mathcal{S}, \mathcal{P}) := \\ \langle \langle \mathcal{N}', \Psi'_D \uplus (\Psi'_T \cup \{O_{name} \xrightarrow{SM} O'\}) \rangle, \\ \mathcal{P} \cup \{[VN, (O_{name}, O'), (\mathcal{I}_1, O_1), \dots, (\mathcal{I}_n, O_n)], O_0] \} \rangle \end{aligned}$$

4.4 Structured specifications.

We now define τ for basic specifications and each of the structuring operations in CASL.

Basic Specifications. A basic specification is a pair (Σ, Φ) of a signature Σ and a set of second-order logic axioms Φ . We create a new node in the development N with local signature $\Sigma_l^N := \Sigma$ and local axioms $\Phi_l^N := \Phi$ and add it to the development graph. The node N is both the node where the actual visible environment shall be imported into as well as the node that contains the visible environment “after” parsing the basic specification.

$$\tau((\Sigma, \Phi), \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) := \langle \{N\}, \langle \mathcal{N} \cup \{N\}, \Psi_D \uplus \Psi_T \rangle, N \rangle$$

Translations. A translation is of the form SP **with** SM , where SP is a structured specification and SM a symbol morphism. Let

$$\langle \mathcal{I}', \mathcal{S}', O' \rangle := \tau(SP, \mathcal{S}, \mathcal{P})$$

with $\mathcal{S}' = \langle \mathcal{N}', \Psi'_D \uplus \Psi'_T \rangle$ and let N be a new node in \mathcal{N}' with empty local signature and axioms. We add this new node to the new development graph and import the top-level node O' from SP into N via a global definition link with morphism SM .

$$\tau(SP \ \mathbf{with} \ SM, \mathcal{S}, \mathcal{P}) := \langle \mathcal{I}', \langle \mathcal{N}' \cup \{N\}, (\Psi'_D \cup \{O' \xrightarrow{SM} N\}) \uplus \Psi'_T \rangle, N \rangle$$

Extensions. There are two kinds of extensions in CASL, namely SP **then** SP' and SP **then** $\% \mathbf{implies} \ SP'$. The first is a *regular* extension of SP by SP' , while the second denotes a *conservative* extension, i.e. it is in fact a conjecture that all axioms in SP' are theorems in the theory of SP .

– *Regular Extensions* are translated as follows. Let

$$\begin{aligned} \langle \mathcal{I}', \mathcal{S}', O' \rangle &:= \tau(SP, \mathcal{S}, \mathcal{P}) \\ \langle \mathcal{I}'', \langle \mathcal{N}'', \Psi''_D \uplus \Psi''_T \rangle, O'' \rangle &:= \tau(SP', \mathcal{S}', \mathcal{P}) \end{aligned}$$

then

$$\tau(SP \text{ then } SP', \mathcal{S}, \mathcal{P}) := \langle \mathcal{I}', \langle \mathcal{N}'' \rangle, (\Psi_D'' \cup \{O' \xrightarrow{\lambda} I'' \mid I'' \in \mathcal{I}''\}) \uplus \Psi_T'', O'' \rangle$$

- *Conservative Extensions:* The CASL-semantics requires from a conservative extension $SP \text{ then } \% \text{implies } SP'$ that SP' is a *basic specification* without local signature. Thus, let

$$\begin{aligned} \langle \mathcal{I}, \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, O \rangle &:= \tau(SP, \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) \text{ and} \\ \langle \{N\}, \langle \mathcal{N}' \cup \{N\}, \Psi_D' \uplus \Psi_T' \rangle, N \rangle &:= \tau(SP', \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, \mathcal{P}) \end{aligned}$$

Then the translation of this conservative extension consists of adding the local axioms of N as local lemmata to O and returning $\langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle$.

$$\tau(SP \text{ then } \% \text{implies } SP', \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) := \langle \mathcal{I}, \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, O \rangle$$

where O is the updated O .

Union. A union of specifications in CASL is of the form $SP \text{ and } SP'$. Let

$$\begin{aligned} \langle \mathcal{I}, \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, O \rangle &:= \tau(SP, \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) \text{ and} \\ \langle \mathcal{I}', \langle \mathcal{N}'', \Psi_D'' \uplus \Psi_T'' \rangle, O' \rangle &:= \tau(SP', \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, \mathcal{P}) \end{aligned}$$

In order to represent the union of the specifications, we add a new empty theory node N to \mathcal{N}'' and import both O and O' into N via global definition links.

$$\begin{aligned} \tau(SP \text{ and } SP', \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) &:= \\ \langle \mathcal{I} \cup \mathcal{I}', \langle \mathcal{N}'' \cup \{N\}, (\Psi_D'' \cup \{O \xrightarrow{\lambda} N, O' \xrightarrow{\lambda} N\}) \uplus \Psi_T'' \rangle, N \rangle & \end{aligned}$$

The theory nodes to import the global environment is the union of both \mathcal{I} and \mathcal{I}' , while the visible environment “after” the union is the global signature of the new node N .

Closed specifications. They are of the form $\text{closed}\{SP\}$. The semantics is that the global environment is not visible inside SP , but shall still be visible “after” $\text{closed}\{SP\}$ together with the environment generated from SP . Thus, the translation of the closed specification consists in creating a new empty node N , import the environment from SP into N via a global definition link, and returning N has both the import and output node for the global environment. Thus, if $\langle \mathcal{I}, \langle \mathcal{N}', \Psi_D' \uplus \Psi_T' \rangle, O \rangle := \tau(SP, \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P})$, then

$$\tau(\text{closed}\{SP\}, \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle, \mathcal{P}) := \langle N, \langle \mathcal{N}' \cup \{N\}, (\Psi_D' \cup \{O \xrightarrow{\lambda} N\}) \uplus \Psi_T' \rangle, N \rangle$$

Actualization. An actualization in CASL is of the general form

$$SN [SP_1 \text{ fit } SM_1] \dots [SP_n \text{ fit } SM_n].$$

Its semantics is that, the formal parameter of the formerly declared named specification SN are instantiated with the SP_i and the “**given**”-specifications of SN is imported into the actual parameters SP_i . This is only sound if the actual parameter fit the formal parameter theories modulo the morphisms SM_i . A parameter information for SN is $[SN, O, (\langle \mathcal{I}'_1, O'_1 \rangle, \dots, \langle \mathcal{I}'_n, O'_n \rangle), O_I]$, where O is the top-level theory for SN , $\langle \mathcal{I}'_i, O'_i \rangle$ the information about input and output theories of the parameter theories, and O_I the top-level theory node that is imported into the parameters. Given this parameter information for the named specification SN , let

$$\begin{aligned} \langle \mathcal{N}_0, \Psi_D^0 \uplus \Psi_T^0 \rangle &:= \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle \\ \tau(SP_i, \langle \mathcal{N}_{i-1}, \Psi_D^{i-1} \uplus \Psi_T^{i-1} \rangle, \mathcal{P}) &:= \langle \mathcal{I}_i, \langle \mathcal{N}_i, \Psi_D^i \uplus \Psi_T^i \rangle, O_i \rangle \\ &\text{for all } 1 \leq i \leq n \end{aligned}$$

Then we import the “**given**” environment theory O_I into each theory in \mathcal{I}_i , for all $1 \leq i \leq n$:

$$\langle \mathcal{N}_n, (\Psi_D^n \cup \{O_I \xrightarrow{\lambda} I \mid I \in \bigcup_{i=1}^n \mathcal{I}_i\}) \uplus \Psi_T^n \rangle$$

We further encode the soundness condition required by **fit** by introducing global theorem links from each O'_i to O_i with morphism SM_i :

$$\langle \mathcal{N}_n, (\Psi_D^n \cup \{O_I \xrightarrow{\lambda} I \mid I \in \bigcup_{i=1}^n \mathcal{I}_i\}) \uplus (\Psi_T^n \cup \{O'_i \xrightarrow{SM_i} O_i \mid 1 \leq i \leq n\}) \rangle$$

Finally, we create the node N_I to encode the instantiated theory: This node imports globally the top-level node N for SN , as well as the top-level nodes O_i of the actual parameter theories.

$$\langle \mathcal{N}_n \cup \{N_I\}, (\Psi_D^n \cup \{O_I \xrightarrow{\lambda} I \mid I \in \bigcup_{i=1}^n \mathcal{I}_i\}) \cup \{N \xrightarrow{SM} N_I, O_1 \xrightarrow{\lambda} N_I, \dots, O_n \xrightarrow{\lambda} N_I\} \rangle \\ \uplus (\Psi_T^n \cup \{O'_i \xrightarrow{SM_i} O_i \mid 1 \leq i \leq n\})$$

where $SM := \bigcup_{i=1}^n SM_i$.

This completes the definition of the translation of structured specification.

4.5 Fitting Views

A fitting view is of the form $VN [SP_1] \dots [SP_n]$, where VN is the name of a view. The translation of this fitting view is analogously to the translation of an actualization of a named specification, except that an additional global theorem link from the actualized theory to the top-level theory node obtained for the target SP of the view SN is inserted.

5 Difference Analysis & Basic Operations

Due to its evolutionary nature, (formal) software development can be seen as a chain of specifications $Spec_1, Spec_2, \dots$ which corresponds to a chain of development graphs DG_1, DG_2, \dots such that DG_i is the logical representation of the specification $Spec_i$. Working on the verification side we try to verify the various proof obligations within a particular development graph, say DG_i . Changing the specification to $Spec_{i+1}$ and compiling it into its logical representation DG_{i+1} , we loose all information about previous proof work, which is stored in DG_i , at first. Hence, the idea is to incrementally adjust DG_i and its annotated proofs until the resulting development graph DG_{i+1} denotes a logical representation of $Spec_{i+1}$. Two problems have to be solved to implement this approach:

First, we need a set of operations which allow us to modify development graphs in such a way that as much proof work as possible can be reused from the previous development graph. We call these operations, that manipulate individual links, theories or axioms, *basic operations*.

Second, we have to compute the differences between two specifications $Spec_i$ and $Spec_{i+1}$ and translate these differences into a sequence of basic operations to be performed on the development graph DG_i in order to obtain DG_{i+1} .

5.1 Basic Operations

To allow for a reuse of proof work, basic operations have to be as granular as possible. Since development graphs consists of nodes and links, basic operations allow one to modify single nodes or links. In principle each of these individual objects can be inserted, deleted or modified. As nodes are composed of a local signature and local axioms, the modification of nodes is done by insertion, deletion or modification of signature entries or local axioms. Formally the set of basic operations consists of the following functions that take, between others, a development graph $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ as argument and return a new development graph \mathcal{S}' :

Nodes: $ins_{node}(N, \mathcal{S})$ inserts a new (isolated) node N to \mathcal{N} , and $del_{node}(N, \mathcal{S})$ removes a node N from \mathcal{N} and deletes also all links in Ψ_D and Ψ_T connected to N .

Links: $ins(N, M, \sigma, Type, \mathcal{S})$ inserts a link to Ψ as a global/local definition/theorem link depending on the value of $Type$. $del(L, \mathcal{S})$ removes the link L from $\Psi_D \uplus \Psi_T$, and $ch(L, \sigma, \mathcal{S})$ replaces the morphism of the link L by σ .²

Local Signature: $ins_{sig}(f, N, \mathcal{S})$ inserts the symbol f into the local signature of N , where f can be either a sort, a constant, or a function. $del_{sig}(f, N, \mathcal{S})$ removes the symbol f from the local signature of N .

Local Axioms: $ins_{ax}(N, Ax, \mathcal{S})$ inserts the local axiom Ax into the node N , $del_{ax}(f, N, \mathcal{S})$ deletes the local axiom Ax from the node N . $ch_{ax}(N, Ax, Ax', \mathcal{S})$ replaces the local axiom Ax by the new local axiom Ax' in the node N .

² There are no operations to change the source or target node of a link. In this case the old link must be deleted and a new link is inserted.

For each basic operation the manner how it affects the development graph is known. This knowledge is exploited by the proof transformation techniques, that adapt the proofs of old global proof obligations to the new global proof obligations. We will describe these techniques in the Sect. 6.

Starting with a legal development graph, the application of basic operations may result in inconsistent intermediate states. A typical example is the insertion of a new function symbol into the source node of a link. Then in general, the morphism attached to the link has to be adjusted to cope with the new symbol. Therefore we allow for intermediate inconsistent states of the development graph and delay the update of the proof work until we reach a consistent state which is indicated by calling a special *update*-function initiating a consistency check and a propagation of the proof work done so far.

5.2 Computing Differences

When computing differences between specifications, the question arises how to define the granularity up to which differences are determined between the old and the new development graph. Note that along a scale of granularity levels for difference analysis the worst granularity level is the one only stating that the whole global proof obligation changed, in which case the proof transformation consists of redoing the whole proof, whereby any information about established conjectures are lost.

The overall aim is to enable the preservation of as many validated conjectures during the transformation of the old proof to the new development graph. The recorded information establishing the validity of a conjecture consists of proofs for those conjectures. However, not every theorem prover returns a proof object. In that case, we must assume that any axiom available at prove time might have been used during the proof. Thus, the information about a proof contains at least a set of axioms. If any of those is deleted or changed, the proof gets invalid. The implication is that we have to determine the difference between the old and new development graph at least on the level of axioms.

The axioms are build from the available signature symbols, like sorts, constants and functions. In order to maintain a sound development graph, we must also be able to determine the differences between signatures. As presented in Sect. 3, the signature of some node is defined from the local signature defined on that node and the signatures of the nodes imported via definition links, after application of the morphism attached to those links.

To determine the differences of signatures and axioms between two development graphs requires first to define an equivalence relation between graphs that identifies nodes and links. This problem has no optimal solution and hence we rely on some heuristics checking their equivalence. In principle two nodes are equivalent if their local signature and axioms are equal as well as their respective incoming definition links. However, this equivalence relation is too strict for our purpose, since if we added or deleted an axiom to some node, its old and new version are not identified. Thus, instead of performing an equality check, we perform a similarity check on nodes, that is based on the number of shared local

signature symbols as well as the similarity of the incoming definition links. Applying that similarity check results in an equivalence relation associating nodes and links of the old to nodes and links in the new development graph.

The equivalence relation is the basis to determine the differences between both graphs. From it we determine (1) which nodes have been deleted or added, (2) which local signature symbols and axioms have been deleted or added to some node, and (3) how the morphisms of links have changed.

5.3 Heuristic Determination of Similarities

In this Section we describe the heuristic implemented in MAYA which is used to compute the similarities between two versions of development graphs. The conducted experiments showed that it is sufficiently reliable for our needs. The similarity is expressed by a *mapping* among theory nodes and links. Formally, given two development graphs $\langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ and $\langle \mathcal{N}', \Psi'_D \uplus \Psi'_T \rangle$, the mapping is a triple $(\mapsto_N, \mapsto_D, \mapsto_T)$, such that $\mapsto_N: \mathcal{N} \leftrightarrow \mathcal{N}'$, $\mapsto_D: \Psi_D \leftrightarrow \Psi'_D$, and $\mapsto_T: \Psi_T \leftrightarrow \Psi'_T$ are partial functions.

The heuristic to construct such a mapping works as follows. We start with partial functions \mapsto_N, \mapsto_D , and \mapsto_T that have an empty domain. Then, for each node N in \mathcal{N} we determine the “most similar node” in \mathcal{N}' that is not in the image of \mapsto_N . If there is such a node N' , we extend \mapsto_N by setting $\mapsto_N(N) := N'$; otherwise N has no similar theory in \mathcal{N}' . Finally, for each link in Ψ_D (resp. Ψ_T) we determine the “most similar link” in Ψ'_D (resp. Ψ'_T), and extend \mapsto_D (resp. \mapsto_T) accordingly, if such a link exists.

The whole heuristic is based on the notion of similarities of two nodes and links. These two notions are defined by mutually recursion and make use of already established mappings between old and new theory nodes and links. The similarity of nodes and links is some value from $[0..1]$ and is denoted by $Similarity(N, N')$ for nodes and $Similarity(l, l')$ for links. The values are estimated as follows:

- *Similarity of nodes.* Let $N \in \mathcal{N}$ and $N' \in \mathcal{N}'$ be two nodes and $(\mapsto_N, \mapsto_D, \mapsto_T)$ the actual mapping.
 - If $\mapsto_N(N) = N'$, then $Similarity(N, N') := 1$.
 - If both N and N' have a *defined name*, like for example if they are both the top-level theory nodes obtained for some named CASL-specification, then if those names are equal, then $Similarity(N, N') := 1$, otherwise $Similarity(N, N') := 0$.
 - If none of them has a defined name, then we take the average of on the one hand the similarity between the local sorts in Σ_l^N and $\Sigma_l^{N'}$, and on the other hand the similarity between the sets of incoming definition links into N and those for N' .
 - Otherwise, $Similarity(N, N') := 0$.
- *Similarity of links.* We illustrate the computation for definition links, i.e. links from Ψ_D and Ψ'_D . The computation is analogously for theorem links. Let $l \in \Psi_D$ and $l' \in \Psi'_D$ be two definition links and $(\mapsto_N, \mapsto_D, \mapsto_T)$ the actual mapping.

- If one is a global link while the other is not, then $\text{Similarity}(l, l') := 0$.
- Otherwise, if they have the same morphism, then we set the similarities of the two links to be the average of the similarity between the source nodes and the similarity of the target nodes.
- If they don't have the same morphism, then the similarity is 0.

6 Maintaining Proof Work

The development graph represents a justification-based truth maintenance system for structured specifications. Based on underlying theorem provers it provides justifications for proof obligations (encoded as theorem links) and is able to remember and adjust derivations which were computed previously. There are two different types of justifications corresponding to the verification in-the-large and to the verification in-the-small which both have to be updated each time the graph is changed. In the following we describe this propagation of proof work for the verification in-the-large and the verification in-the-small separately.

6.1 The DG-calculus

The theory of a node N depends on theories of all nodes connected to N via (global) definition links. Local definition links hide the theories of underlying subnodes. The next definition specifies possible paths to include the theory or the local axioms of the source node to the theory of the target node.

Definition 3. *Let Ψ be a set of links.*

- Ψ contains a **global path** $N_1 \xrightarrow{\sigma} \Psi N_k$ from N_1 to N_k via a morphism σ if there is either a sequence of links $N_1 \xrightarrow{\sigma_1} N_2, N_2 \xrightarrow{\sigma_2} N_3 \dots N_{k-1}, \xrightarrow{\sigma_{k-1}} N_k$ in Ψ with $\sigma = \sigma_1 \circ \dots \circ \sigma_{k-1}$ or $N_1 = N_k$ and σ is the identity function.
- Ψ contains a **local path** $N_1 \xrightarrow{\sigma} \Psi N_k$ from N_1 to N_k via a morphism σ if there is a sequence of links $N_1 \xrightarrow{\sigma_1} N_2, N_2 \xrightarrow{\sigma_2} N_3 \dots N_{k-1}, \xrightarrow{\sigma_{k-1}} N_k$ in Ψ with $\sigma = \sigma_1 \circ \dots \circ \sigma_{k-1}$.

Given a development graph $\langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$, the definition links Ψ_D are used to specify the semantics, i.e. theory, of the individual nodes. Ψ_T constitutes the proof obligations inside the graph. In the following we define when a development graph satisfies these proof obligations:

Definition 4. *Let $\mathcal{S} = \langle \mathcal{N}, \Psi \rangle$ be a development graph and $\Delta \subseteq \Psi$ be acyclic. Δ satisfies a link $M \xrightarrow{\sigma} N \in \Psi$ (or $M \xrightarrow{\sigma} N \in \Psi$ resp.) iff $\sigma(\text{Th}_\Delta(M)) \subseteq \text{Th}_\Delta(N)$ (or $\sigma(\Phi_l^M) \subseteq \text{Th}_\Delta(N)$ resp.). Δ satisfies a set Γ of links if it satisfies all elements in Γ .*

*A development graph $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ is **verified** iff Ψ_D satisfies Ψ_T .*

A global definition link includes the theory of the source node into the theory of the target node while a local definition link includes only the local axioms of the source node. Due to the \vdash -translation property of the underlying entailment

relation, any global definition link starting at the target node of such a link will export this imported theory or axioms in turn to other theories. Theorem links which are satisfied by the definition links can be treated in the same manner as definition links:

Lemma 1. *Let $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ be a development graph and let Ψ_D satisfy a set of links Δ . Then the following holds:*

1. $N \xrightarrow{\sigma} \Psi_D \uplus \Delta M$ implies $\sigma(\text{Th}(N)) \subset \text{Th}(M)$ and
2. $N \xrightarrow{\sigma} \Psi_D \uplus \Delta M$ implies $\sigma(\Phi_i^N) \subset \text{Th}(M)$

Proof. We sketch the proof for global paths which is done by induction on the length of the path:

Base case: If the path is empty, then σ is the identity and $N = M$. Thus $\sigma(\text{Th}(N)) \subset \text{Th}(M)$ holds trivially.

Step case: As an induction hypothesis we assume that if $N \xrightarrow{\sigma'} \Psi_D \uplus \Delta K$ then $\sigma'(\text{Th}(N)) \subset \text{Th}(K)$. Let $K \xrightarrow{\sigma''} M \in \Psi_D \uplus \Delta$. Thus, $\sigma''(\text{Th}(K)) \subset \text{Th}(M)$ (Def. 2 or Def. 4 resp.) and therefore $\sigma''(\sigma'(\text{Th}(N))) \subset \text{Th}(M)$. \square

In order to verify a development graph we introduce a calculus \mathcal{DG} operating on links to perform a so-called *verification in-the-large* and providing a *local decomposition rule* to establish elementary relations between theories by usual theorem proving, which we call *verification in-the-small*.

Definition 5 (Calculus \mathcal{DG}). *The calculus \mathcal{DG} is a sequent-style calculus. Sequents are of the form $\Gamma \vdash \Delta$, where Γ, Δ are sets of links. A sequent $\Gamma \vdash \Delta$ holds iff Γ satisfies Δ . The sequent calculus rules of \mathcal{DG} are:*

Axiom (AX): $\frac{}{\Gamma \vdash \emptyset}$

Global decomposition (GD):

$$\frac{\Gamma \vdash N \xrightarrow{\sigma} M, \bigcup_{K \xrightarrow{\rho} N \in \Gamma} \{K \xrightarrow{\sigma \circ \rho} M\}, \bigcup_{K \xrightarrow{\rho} N \in \Gamma} \{K \xrightarrow{\sigma \circ \rho} M\}, \Delta}{\Gamma \vdash N \xrightarrow{\sigma} M, \Delta}$$

Local decomposition (LD):

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash N \xrightarrow{\sigma} M, \Delta} \quad \text{if for all } \phi \in \Phi_i^N : \sigma(\phi) \in \text{Th}_\Gamma(M)$$

Global subsumption (GS):

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash N \xrightarrow{\sigma} M, \Delta}$$

if $N \xrightarrow{\sigma} \Gamma \cup \Delta M$

Local subsumption (LS):

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash N \xrightarrow{\sigma} M, \Delta}$$

if $N \xrightarrow{\sigma} \Gamma \cup \Delta M$

Theorem 1. *Let $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ be a development graph and $\Delta \subseteq \Psi_T$. Then, $\Psi_D \vdash \Delta$ is derivable in the deduction system \mathcal{DG} iff Ψ_D satisfies Δ .*

Proof (of Theorem 1).

Soundness: We induce on the length n of the deduction:

Base case: let $n = 1$. Thus, $\Delta = \emptyset$ and \mathcal{S} satisfies Δ trivially.

Step case: let $n > 1$ and $\Psi_D \vdash \Delta'$ be the immediate predecessor of $\Psi_D \vdash \Delta$. As induction hypothesis we assume that \mathcal{S} satisfies Δ' . We do a case split according to the applicable rules:

GD: Hence $N \xrightarrow{\sigma} M \in \Delta'$, $K \xrightarrow{\sigma \circ \rho} M \in \Delta'$ for all $K \xrightarrow{\rho} N \in \Psi_D$ and $K \xrightarrow{\sigma \circ \rho} M \in \Delta'$ for all $K \xrightarrow{\rho} N \in \Psi_D$. Since \mathcal{S} satisfies Δ' , we know that $\sigma(\Phi_i^N) \subseteq Th(M)$, $\sigma(\rho(Th(K))) \subseteq Th(M)$ for all $K \xrightarrow{\rho} N \in \Psi_D$ and $\sigma(\rho(\Phi_i^K)) \subseteq Th(M)$ for all $K \xrightarrow{\rho} N \in \Psi_D$. Thus, from the \vdash -translation property of the underlying entailment relation we get

$$[\sigma(\Phi_i^N) \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \sigma(\rho(Th(K))) \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \sigma(\rho(\Phi_i^K))] \vdash^{\Sigma^N} \subseteq Th(M).$$

Hence, $\sigma([\Phi_i^N \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \rho(Th(K)) \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \rho(\Phi_i^K)]) \vdash^{\Sigma^M} \subseteq Th(M)$ holds due to the \vdash -translation property, i.e. $\sigma(Th(N)) \subseteq Th(M)$ and thus Ψ_D satisfies $N \xrightarrow{\sigma} M$.

LD: $\sigma(\phi) \in Th(M)$ for all $\phi \in \Phi_i^N$ implies that Ψ_D satisfies $N \xrightarrow{\sigma} M$ and thus \mathcal{S} satisfies Δ .

GS: Since $N \xrightarrow{\sigma} \Psi_D \cup \Delta' M$ holds and \mathcal{S} satisfies Δ' we know that $\sigma(Th(N)) \subseteq Th(M)$ holds, i.e. \mathcal{S} satisfies $N \xrightarrow{\sigma} M$.

LS: Since $N \xrightarrow{\sigma} \Psi_D \cup \Delta' M$ and \mathcal{S} satisfies Δ' we know that $\sigma(\Phi_i^N) \subseteq Th(M)$ holds, i.e. \mathcal{S} satisfies $N \xrightarrow{\sigma} M$.

Completeness: Suppose, Ψ_D satisfies Δ . Since the development graph $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ is acyclic with respect to Ψ_D we define the *depth* of a node $N \in \mathcal{N}$ as the length of the longest path of links in Ψ_D from some leaf node in \mathcal{N} to N . We induce on the multiset $depths(\Delta)$ of depths of the source nodes in Δ .

Base case: If $depths(\Delta) = \emptyset$ then $\Delta = \emptyset$ and $\Psi_D \vdash \emptyset$ holds by rule AX.

Induction step: Let $depths(\Delta) \neq \emptyset$. As an induction hypothesis suppose the conjecture holds for all Δ' which are smaller than Δ with respect to the multiset-ordering on $depths$.

- Let $N \xrightarrow{\sigma} M \in \Delta$ with $depth(N) = \max(depths(\Delta))$. Since Ψ_D satisfies $\Delta - \{N \xrightarrow{\sigma} M\}$, applying the induction hypothesis yields $\Psi_D \vdash \Delta - \{N \xrightarrow{\sigma} M\}$. As Ψ_D satisfies $N \xrightarrow{\sigma} M$, we know that $\sigma(\Phi_i^N) \subseteq Th(M)$ holds and apply rule LD to deduce finally $\Psi_D \vdash \Delta$.
- Let $N \xrightarrow{\sigma} M \in \Delta$ with $depth(N) = \max(depths(\Delta))$. For all links $K \xrightarrow{\rho} N \in \Psi_D$ $\sigma(\rho(Th(K))) \subseteq Th(M)$ holds. Analogously for all links $K \xrightarrow{\rho} N \in \Psi_D$ holds $\sigma(\rho(\Phi_i^K)) \subseteq Th(M)$. Thus, \mathcal{S} satisfies both $\bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\}$ and $\bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\}$. Applying the induction hypothesis yields $\Psi_D \vdash (\Delta - \{N \xrightarrow{\sigma} M\}) \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\} \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\}$. Since \mathcal{S} satisfies also $N \xrightarrow{\sigma} M$ we use the argumentation of the first case to deduce $\Psi_D \vdash (\Delta - \{N \xrightarrow{\sigma} M\}) \cup \{N \xrightarrow{\sigma} M\} \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\} \cup \bigcup_{K \xrightarrow{\rho} N \in \Psi_D} \{K \xrightarrow{\sigma \circ \rho} M\}$ and apply rule GD to derive $\Psi_D \vdash \Delta$. \square

The \mathcal{DG} -calculus is based on an oracle to check if $\sigma(\phi) \in Th_{\Psi_D}(M)$ holds. In general $Th_{\Psi_D}(M)$ is an infinite set of formulas and we need a finite axiomatization for it. It is well-known that structured specifications excluding hiding³ are flatable. The following lemma describes the finite axiomatization of the theory of a node:

Lemma 2. *Let $\mathcal{S} = \langle \mathcal{N}, \Psi \rangle$ be a development graph and let the **axiomatization** of some node $N \in \mathcal{N}$ **relative** to some $\Delta \subseteq \Psi$, Δ acyclic, be defined by*

$$\Phi_{\Delta}^N = \Phi_l^N \cup \bigcup_{K \xrightarrow{\sigma} N \in \Delta} \sigma(\Phi_{\Delta}^K) \cup \bigcup_{K \xrightarrow{\sigma} N \in \Delta} \sigma(\Phi_l^K)$$

Then, $Th_{\Delta}(N) = [\Phi_{\Delta}^N]^{\vdash_{\mathcal{S}^N}}$ holds for all $N \in \mathcal{N}$.

Proof. Directly from Def. 2 and the \vdash -translation property of the underlying entailment relation. \square

To verify the proof obligations on local theorem links, which we call *verification in-the-small*, we make use of standard theorem provers like ISABELLE [17] or INKA 5.0 [1]. The reader is referred to [3] for a description how development graph and theorem provers are technically connected.

6.2 Verification In-the-Large

Verification in-the-large is concerned with the reduction of the overall problem of verifying an development graph \mathcal{S} to the problem of proving as few as possible proof obligations denoted by local theorem links. Verification-in-the-large is done with the help of the \mathcal{DG} -calculus. Obviously, applying global and local subsumption rules as often as possible will reduce the number of arising proof obligations in-the-small. To support the maintenance of the proofs in-the-large, MAYA provides explicit proof objects for the \mathcal{DG} -calculus. Theorem links are annotated with explicit proof objects, which are instances of the \mathcal{DG} -calculus rules. Each \mathcal{DG} -calculus rule reduces the proof of a theorem link to the problem of proving a set of other theorem links. Thus, the proof object of a theorem link is distributed through the development graph and only the first inference step, the so-called *local proof object*, is stored at the theorem link while the remaining part always coincides with proof objects of other theorem links.

Definition 6. *Let $\psi = N \xrightarrow{\sigma} M$ then*

- $pr_{\psi} := GD(\psi_0, \langle \psi'_1, \dots, \psi'_n \rangle, \langle \psi''_1, \dots, \psi''_m \rangle)$ is a local proof object.
- pr_{ψ} is locally valid iff $\psi_0 = N \xrightarrow{\sigma} M$, $\{\psi'_1, \dots, \psi'_n\} = \bigcup_{K \xrightarrow{\rho} N \in \Gamma} \{K \xrightarrow{\sigma \circ \rho} M\}$ and $\{\psi''_1, \dots, \psi''_m\} = \bigcup_{K \xrightarrow{\rho} N \in \Gamma} \{\psi \xrightarrow{\sigma \circ \rho} M\}$

³ See [14] for an extension of development graphs by hiding which translates proof obligations in theories based on hiding to proof obligations in theories without hiding.

- $pr_\psi := GS(\psi_1, \dots, \psi_n)$ is a local proof object. pr_ψ is locally valid iff ψ_1, \dots, ψ_n constitutes a relation $N \xrightarrow{\sigma} M$.

Let $\psi = N \xrightarrow{\sigma} M$ then

- $pr_\psi := LS(\psi_1, \dots, \psi_n)$ is a local proof object. pr_ψ is locally valid iff ψ_1, \dots, ψ_n constitutes a relation $N \xrightarrow{\sigma} M$.
- $pr_\psi := LD(\sigma, (Ax_1, \Phi_1), \dots, (Ax_k, \Phi_k))$ is a proof object where each Φ_i is either an atom *NoProof*, *ProofExists* or a set of triples (τ, K, Ω) with $\Omega \subset \Phi_i^K$. pr_ψ is locally valid iff for all (Ax_i, Φ_i) with $1 \leq i \leq n$, $(\bigcup_{(\tau, K, \Omega) \in \Phi_i} \tau(\Omega)) \vdash \sigma(Ax_i)$ and for all triple $(\tau, K, \Omega) \in \Phi_i$ $K \xrightarrow{\tau} M$ holds.

$\Psi(pr_\psi)$ is defined as the set of all links occurring in the proof object pr_ψ of ψ . $\Psi^*(pr_\psi)$ denotes the transitive closure of $\Psi(pr_\psi)$ and is defined by $\Psi^*(pr_\psi) = \Psi(pr_\psi) \cup \bigcup_{\psi' \in \Psi(pr_\psi)} \Psi^*(pr_{\psi'})$.

Lemma 3. Let $\mathcal{S} = \langle \mathcal{N}, \Psi_D \uplus \Psi_T \rangle$ be a development graph. If there are locally valid proof objects pr_ψ with $\psi \notin \Psi^*(pr_\psi)$ for all $\psi \in \Psi_T$ then Ψ_D satisfies Ψ_T .

Proof. Since $\psi \notin \Psi^*(pr_\psi)$ holds for all $\psi \in \Psi_T$ there is a partial ordering $<$ on Ψ_T with $\psi' < \psi$ iff $\psi' \in \Psi^*(pr_\psi)$. We can extend such a partial ordering to a total ordering \ll on Φ_T . It is an easy inductive argument that we can construct a \mathcal{DG} -calculus proof in the following way: we start with the problem of proving $\Psi_D \vdash \Psi_T$ and apply the proof rule attached to the maximal element of Ψ_T wrt. \ll . Since the proof object is locally valid the rule is applicable and we have reduced the problem to a problem of proving $\Psi_D \vdash \Psi_T \setminus \{\psi\}$. Iterating this approach by choosing always the maximal element of the set of remaining theorem links we end up in the trivial case of proving $\Psi_D \vdash \emptyset$.

Verification in-the-large is concerned with the problem of creating and maintaining local proof objects of the types GD, GS and LS such that each of these local proof objects is locally valid and such that the proof object of a link ψ does not depend on itself, i.e. $\psi \notin \Psi^*(pr_\psi)$. The problem of maintaining LD-proof objects is discussed in section 6.3. We call a development graph verified in-the-large if and only if all GD, GS, LS-proof objects are locally valid and do not contain cycles (i.e. $\psi \notin \Psi^*(pr_\psi)$).

Starting with an empty development, which is trivially verified, the graph is manipulated by using basic operations like for instance the insertion, deletion, or change of links or axioms. After a sequence of basic operations (updating the development graph according to the change of specification made by the user) the proof objects are adapted to the needs of the actual graph. Hence, each subsequent development graph is verified reusing the old proof objects annotated in the former development graph.

To describe the update-process, assume now that we manipulated a verified development graph with the help of a sequence of basic operations. To establish the validity of the resulting development graph we perform the following steps:

Checking GD-proof objects: In the first phase, existing GD-proof objects are updated to be locally valid proof objects. Starting at the top-level theories (like LIST in our example), we traverse the graph according to the depth of the theories. Reasons for an invalidated GD-proof object $pr_\psi = GD(\psi_0, \langle \psi'_1, \dots, \psi'_n \rangle, \langle \psi''_1, \dots, \psi''_m \rangle)$ are the change of the morphism of some link or the insertion or deletion of definition links targeting at the source of the theorem link. In the first case we replace an inappropriate link by a link with an appropriate morphism. Either such a link already exists (e.g. as a definition link) or it is created and added to Ψ_T while it inherits the (invalid) proof object of the replaced link (this proof object will be fixed in the ongoing procedure). In case of insertion or removal of definition links, both link lists $\langle \psi'_1, \dots, \psi'_n \rangle$ and $\langle \psi''_1, \dots, \psi''_m \rangle$ in pr_ψ are updated accordingly. This, again, might result in the creation of new theorem links, which are again added to Ψ_T , or the deletion of theorem links from Ψ_T if they have been once created using the GD-rule and are of no use anymore (i.e. they do not occur in any proof object anymore).

Checking GS- and LS-proof objects: In the second phase, proof objects concerned with subsumption rules are checked for validity. For each of these proof objects $pr_\psi = GS(\psi_1, \dots, \psi_n)$ we prove whether all links ψ_i do still exist and whether the morphism of the denoted path still coincides with the morphism of the theorem link ψ . If any of these conditions fails then the proof object is removed; otherwise the proof object pr_ψ is still locally valid.

Establish new proof object: In the third phase local proof objects are generated for theorem links which do not possess any proof object. Either these links have been newly created or their proof objects have been removed in an earlier stage of the procedure. Given a theorem link ψ , firstly we search for an application of the GS- or LS-rule. Thus, we search for a path starting at the source of ψ and ending at the target of ψ which coincides with ψ also in its morphism. In order to obtain an acyclic proof object, each link ψ' in the path has to satisfy the property $\psi \notin \Psi^*(pr_{\psi'})$. In practice we restrict this search for a path inside a graph in the following way: First, we do not search for paths in which a node is visited twice (although in general, running through a circle may result in a different overall morphism of the path). Second, proving a theorem link $K \xrightarrow{\sigma \circ \rho} M$ which was created while verifying a theorem link $N \xrightarrow{\sigma} M$ in presence of a definition link $K \xrightarrow{\rho} N$, we do not consider paths starting with this definition link. If we would find such a path then we could strip off the definition link to obtain a path for $N \xrightarrow{\sigma} M$ (but this was already checked during the verification of this link!). If we cannot find a suitable path to establish a GS- or LS-proof object, a GD-proof object for ψ is generated. This may cause the generation of new theorem links to be added to Ψ_T if no suitable links are already available in the graph.

To illustrate our approach, consider our example in Fig. 3. As we have started with the empty development graph there are no GD, GS or LS-proof objects to be updated and we continue with phase three:

Descending the graph according to the depth of the theories, we first establish a new proof object for the global theorem link from LIST to NATLIST. The GS-rule is not applicable since there is no corresponding global path from LIST to NATLIST. Hence, the GD-rule is applied which results in a proof object $GD(\psi_0, \langle \psi_{elem} \rangle, \langle \rangle)$. ψ_0 is the local definition link from LIST to NATLIST while ψ_{elem} is a newly generated theorem link from ELEM to NATLIST (corresponding to the import of ELEM in LIST). Similarly, we obtain a local proof object $GD(\psi'_0, \langle \psi'_{nat} \rangle, \langle \psi'_{stack} \rangle)$ for the global theorem link ψ' from NATSTACK to NATLIST. ψ'_0 is a newly generated local theorem link parallel to ψ' , ψ'_{nat} is the global definition link from NAT to NATLIST. ψ'_{stack} is a newly generated local theorem link from STACK to NATLIST. Using the LS-rule ψ'_{stack} is proven by the path of (global) theorem links from STACK over LIST to NATLIST. Since NATSTACK has no local axioms, ψ'_{nat} is trivially proven using the LD-rule. Applying the GD-rule to the global theorem link from STACK to NATSTACK introduces a global theorem link from ELEM to NATSTACK which is proven using the GS-rule by the path of global links from ELEM over NAT to NATSTACK. At the end we are left with open proofs for the local theorem links from STACK to LIST and from ELEM to NAT which are tackled by the verification in-the-small.

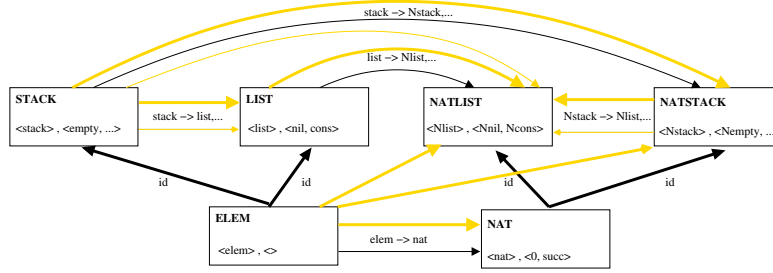


Fig. 3. Management of change for NATLIST

Suppose now, we change the graph structure by the insertion of a new theory REL introducing a new symbol R and imported by ELEM. Therefore all morphisms of the links from ELEM, LIST and STACK to NAT, NATLIST and NATSTACK will be changed in order to incorporate an appropriate mapping of R . In the first phase of the revision process the GD-proof objects of the corresponding global theorem links are adjusted to incorporate the mapping of R . Additionally, the proof object of the global theorem link from ELEM to NAT is changed to $GD(\psi'_0, \langle \psi_{REL} \rangle, \langle \rangle)$ where ψ_{REL} denotes a global theorem link from REL to NAT (corresponding to the new definition link from REL to ELEM). In the second phase nothing has to be done since all GS- and LS-proof objects are still valid although the mapping have changed. In the third phase the new theorem link ψ_{REL} is proven with the help of the GD-rule which introduces a local theorem link from REL to NAT denoting the proof obligations arising from the local axioms in REL to be proven in NAT.

6.3 Verification In-the-Small

Applying the local decomposition (LD-)rule gives rise to proof obligations that each local axiom of the source node mapped by the attached morphism of the link is a theorem of the target theory. To tackle these proof obligations, the system has to compute the axiomatization of the theory (ref. lemma 2) and to apply the morphism of the theorem link to the axioms of the source theory. Since the computation of the axiomatization is expensive the system caches the computed axiomatization of the target node. The axiomatization is annotated by the information about the origin, applied morphisms and used paths of mapped axioms. Once the axiomatization of a different node is needed to tackle another proof obligation, the path information attached to the cached axiomatization is used to incrementally compute the axiomatization of the new node by comparing the needs with the annotated information. Thus, we obtain a set of axioms to be removed from the cached axiomatization and a set of axioms to be inserted to the cached axiomatization to obtain the axiomatization of the new node.

Suppose $\psi = N \xrightarrow{\sigma} M$ is a local theorem link with an attached local proof object $LD(\sigma, (Ax_1, \Phi_1), \dots, (Ax_k, \Phi_k))$. Each axiom Ax_i of N is related to the proof description Φ_i . Φ_i is either an atom `NOPROOF` indicating that this proof obligation has not been proven yet, or an atom `PROOFEXISTS` indicating that some theorem prover has proven the problem but did not return an explicit proof object or at least the set of used axiom, or the set of axioms used to prove $\sigma(Ax_i)$ inside the theory of M . In this case Φ_i breaks down the used axioms according to their origins and the morphism with the help of which they are imported to the target theory.

Changing either the axioms of N , the morphism σ or the subgraph of M may render the proof object pr_ψ invalid. In the following we discuss the repair of the proof object pr_ψ for these three cases:

Change in N : The change of source axioms results in corresponding changes of the proof obligations. Insertion of a new axiom Ax_{k+1} will result in a new entry $(Ax_{k+1}, \text{NOPROOF})$, where `NOPROOF` indicates that $\sigma(Ax_{k+1})$ is still to be proven by some theorem prover. Deletion of some Ax_i will result in the removal of the corresponding pair (Ax_i, Φ_i) . Change of a source axiom Ax_i to Ax'_i causes an invalidation of Φ_i . If the system provides explicit proof objects (instead of the set of used axioms) the system supports the theorem prover by additionally providing Ax_i and the old proof for $\sigma(Ax_i)$ when proving $\sigma(Ax'_i)$ to allow for a reuse of the old proof.

Change in morphism σ : Changing the morphism σ attached to the theorem link to σ' may result in a change of some proof obligations depending how the change of the morphism affects the mapping of local axioms of the source theory. If $\sigma(Ax_i) = \sigma'(Ax_i)$ we can reuse the old proof otherwise the proof information is invalidated but stored for a later reuse when tackling the proof obligation $\sigma'(Ax_i)$ by some theorem prover.

Change in M : Since the theory of M depends on its subgraph, every change in this subgraph may affect the theory of M . We distinguish two different approaches depending whether Φ_i is PROOF EXISTS or description of used axioms.

1. In the latter case we know about all used axioms (and their origins). The proof is still valid if all used axioms are still part of the theory of M . Instead of computing the changes in the axiomatization of M we check for all triples (τ, K, Ω) whether $\tau(\Omega)$ is still imported to M from K via a morphism τ' with $\tau'(\Omega) = \tau(\Omega)$.
2. If there is no explicit proof object, we assume that all axioms accessible at the time of the proof have been used for the proof. Thus a proof is invalid if some axiom of a node inside the subgraph of M has been changed or deleted, or some definition link has been changed or deleted and there is no alternative path with the same morphism. This check is restricted to objects which have existed at the time when the proof was done. Hence each object (links, nodes, axioms, etc.) contains timestamps of its creation, its deletion, or its change. For example, changing a morphism does not affect the validity of a proof if all signature entries which are affected by these changes were introduced after the computation of the proof.

Consider our running example and suppose we had already proven some axioms of STACK mapped as theorems to LIST when we inserted the theory REL. As REL only adds new axioms to the theory of LIST, all proofs of the axioms are still valid. This holds although the morphism τ of the local theorem link from STACK to LIST has changed to τ' in order to incorporate the mapping of the new relation R . In case the local proof object provides the list of used axioms we can easily check that $\tau(Ax_i) = \tau'(Ax_i)$ holds for all $1 \leq i \leq n$. Otherwise, the morphisms τ and τ' are compared which results in the fact that the only differences between both morphisms concern the mapping of the relation R which has been introduced after doing the proofs of any Ax_i . Thus, changing τ to τ' will not affect the proofs of any Ax_i done before the insertion of the theory REL.

7 Implementation

The development graph as well as the techniques for their maintenance are implemented in the MAYA system (cf. [9]). Currently the fixed logic underlying the development graph is higher-order logic. The uniform representation of structured theories in the development graph supports evolutionary formal software development with respect to arbitrary specification languages, provided there exists an adequate mapping from the specification language into development graphs. Currently MAYA integrates parsers for the specification languages CASL (cf. [1]) and VSE-SL (cf. [7]). With respect to the verification in-the-small, MAYA supports the use of arbitrary theorem provers for higher-order logic. To this end a generic interface to propagate the changes of theories to the theorem provers has been implemented. Currently, the HOL-CASL instance of Isabelle/HOL (cf. [3])

and the INKA 5.0 theorem prover (cf. [1]) are integrated into MAYA via this interface. The Lisp sources of MAYA can be obtained from the MAYA-webpage [9].

8 Related Work

The KIV system [18] incorporates a development graph similar to the one presented in this paper. However, instead of having basic structuring mechanism like our global and local links, the KIV structure mechanisms are heavy tailored to the structuring constructs of their specification languages. Although this allows for a more adequate representation of global proof obligations, it lacks the ability to easily integrate support for further specification languages. With respect to the verification in-the-large, it also supports the maintenance of established proof obligations when changing the specification, but lacks a mechanism for redundancy checking and elimination. This is due to the absence of decomposition of proof obligations between graphs into proof obligations between the respective subgraphs. With respect to the verification in-the-small, when the specification is changed, the effects on the axioms usable by the theorem prover cannot be determined in an as granular manner as in the MAYA system. Finally, the tight integration of the KIV development graph with the built-in theorem prover hampers the use of further theorem provers.

The SPECWARE system [10] is a formal software design environment. It follows the paradigm of top-down formal software development using refinement, modularization, and parameterization. The whole design and refinement process is explicitly represented in some kind of development graph and the arising proof obligations are proven using theorem provers. However, like for the KIV system, the basic structuring mechanisms are tailored to the specification language, which hampers the use of other specification languages. Finally, it lacks the support for redundancy checking and elimination, as well as the maintenance of established proof obligations.

The *Little Theories* approach [8] provides a subset of the theory structuring mechanism of development graphs, i.e. global definition links and proven global theorem links. It is more general than development graph, because each theory (node) can have its own logic, whereas for the current implementation of development graphs presented in this paper, the whole graph is with respect to a single logic. The extension of development graphs to deal with different logics has been achieved in theory in [14]. However, little theories lack on the one hand the ability to represent intermediate states of the development, i.e. a state where there still exist yet unproven postulated global theorem links. On the other hand, there are no mechanisms that exploit the graph structure to reduce the amount of proof obligations and to deal with non-monotonic changes of the theories.

9 Conclusion

For the development of industrial-size software systems, the preservation of the structure of specifications is essential not only for the specification of the systems, but also for their verification. Indeed, the structure can be exploited in order to reduce the amount of proof obligations and to support efficiently the revision of specifications, which usually arises in practice.

We presented the implementation of a system for verification in-the-large about structured specifications. It enables to formally find and eliminate redundant proof obligations. Furthermore, it incorporates strategies to transform a proof for some former specification to some new specification, while preserving as many established conjectures as possible.

The theorem proving mechanisms for verification in-the-large are the kernel of the MAYA system [9]. Around that kernel are build on the one hand a uniform interface for parsers of arbitrary specification languages⁴, and on the other hand a uniform interface to use theorem provers for verification in-the-small. These functionalities enable MAYA to bridge the gap between parsers for specification languages and state of the art automated or interactive theorem provers, and deals with all aspects of evolutionary formal software development based on structured specifications.

Future work will consist of extending the verification in-the-large mechanisms to support development graphs with hiding [14] as well as heterogenous development graphs [13]. Further work will also be concerned with the generation of proof-objects for completed developments from MAYA's internal "in-the-large" proof representation and the annotated "in-the-small" proofs. This proof object shall be used to proof check a completed development, which formally certifies a completed formal software development.

References

1. S. Autexier, D. Hutter, H. Mantel, A. Schairer. System description: INKA 5.0 - a logic voyager. In *H. Ganzinger (Ed.): 16th International Conference on Automated Deduction*, Springer, LNAI 1632, 1999.
2. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy and D. Bert, editors, *Proceedings Workshop on Algebraic Development Techniques, WADT-99*. Springer, LNCS 1827, 2000.
3. S. Autexier, T. Mossakowski. Integrating HOL-CASL into the Development Graph Manager MAYA. In A. Armando (Ed.) *Frontiers of Combining Systems (FroCoS'02)*, Santa Margherita Ligure, Italy, Springer LNAI, April, 2002.
4. CoFI Language Design Task Group. *The common algebraic specification language (CASL) - summary*, 1998. Version 1.0 and additional Note S-9 on Semantics, available from <http://www.brics.dk/Projects/CoFI>.

⁴ Provided there is an adequate translation of the logic and the structuring constructs of the specification language into the development graph structure.

5. M. Cerioli, J. Meseguer. May I borrow your logic? *Theoretical Computer Science*, 173(2):311-347, 1997.
6. D. Hutter. Management of change in verification systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, pages 23–34. IEEE Computer Society, 2000.
7. D. Hutter et al.: Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, 1996.
8. W. M. Farmer. An infrastructure for intertheory reasoning, In: D. McAllester, ed., *Automated Deduction – CADE-17*, LNCS, 1831:115-131, 2000.
9. MAYA-webpage: <http://www.dfki.de/~inka/maya.html>.
10. J. McDonald, J. Anton. SPECWARE - Producing Software Correct by Construction. Kestrel Institute Technical Report KES.U.01.3., March 2001.
11. J. Meseguer. General logics, In *Logic Colloquium 87*, pages 275–329, North Holland, 1989.
12. T. Mossakowski: CASL: From Semantics to Tools. In S. Graf (Ed.) *TACAS 2000*, LNCS volume 1785, pages 93-108. Springer, 2000.
13. T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In M. Nielsen (Ed.) *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS02)*, Grenoble, France, Springer LNCS, 2002.
14. T. Mossakowski, S. Autexier, and D. Hutter: Extending Development Graphs With Hiding. In H. Hußmann (Ed.), *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001)*, Italy. LNCS 2029, 269–283. Springer, 2001.
15. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In *Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002*. Springer-Verlag, 2002.
16. D. Hutter and A. Schairer. Proof transformations for evolutionary formal software development. In *Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002*. Springer-Verlag, 2002.
17. L. C. Paulson. *Isabelle - A Generic Theorem Prover*. LNCS 828. Springer, 1994.
18. W. Reif: The KIV-approach to Software Verification, In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software - Final Report*, LNCS 1009, 339-368. Springer, 1995.