Electronic Communications of the EASST
Volume 62 (2013)

Specification, Transformation, Navigation
Special Issue dedicated to Bernd Krieg-Brückner on the Occasion of his 60th Birthday

The VSE Refinement Method in HETS

Mihai Codescu and Bruno Langenstein and Christian Maeder and Till Mossakowski

34 pages

Guest Editors: Till Mossakowski, Markus Roggenbach, Lutz Schröder
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The VSE Refinement Method in HETS

Mihai Codescu[1] and Bruno Langenstein[2] and Christian Maeder[1] and Till Mossakowski[13]

[1]German Research Center for Artificial Intelligence (DFKI GmbH), Bremen, Germany
[2]DFKI GmbH, Saarbrücken, Germany
[3]SFB/TR 8 Spatial Cognition, University of Bremen, Germany

Abstract: We present the integration of the refinement method of the VSE verification tool, successfully used in industrial applications, in the Heterogeneous Tool Set HETS. The connection is done via introducing the dynamic logic underlying VSE and two logic translations in the logic graph of HETS. Thus the proof management formalism provided by HETS can be applied for VSE specifications without modification of the logic independent layers of HETS.

Keywords: heterogeneous specification, refinement, algebraic specifications, dynamic logic, institutions

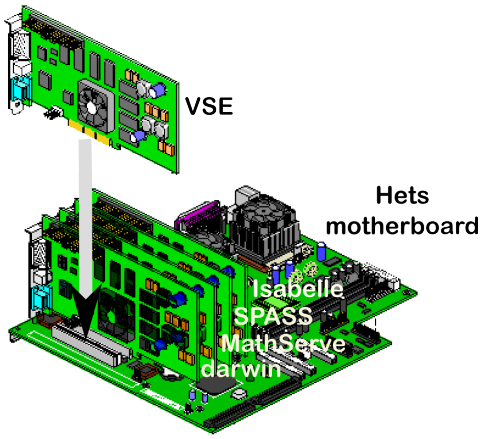
1 Introduction

Axiomatic specification of data and programs provide the means for developing formal models of software at a conceptual level, while dynamic logics and Hoare-style logics can express correctness criteria that stay closer to the actual programs. For a formal development or verification of software, typically both levels are needed, since using axiomatic modeling of the concepts alone misses the formal link to real programs, while using a Hoare-style or dynamic logic alone only allows for little formal conceptual modeling. In this work, we integrate two tools that both address the link between these two levels in different ways, and will provide added benefit through this integration.

The Heterogeneous Tool Set HETS [24, 22], developed at DFKI Bremen, is a tool for heterogeneous multi-logic specification, interfacing various theorem provers, model checkers and model finders. The specification environment Verification Support Environment (VSE) [4], developed at DFKI Saarbrücken, provides an industrial-strength methodology for specification and verification of imperative programs.

We want to combine the best of both worlds by establishing a connection between the VSE prover and the HETS proof management. For VSE, this brings additionally flexibility: VSE specifications can now be verified not only with the VSE prover, but also with provers like the first-order prover SPASS [30] and the higher-order prover Isabelle [27] which are interfaced with HETS. On the other hand, HETS benefits from VSE's industrial experience, including a practical relation between specification and programming languages together with the necessary proof support. Being interactive the VSE prover offers enough flexibility to tackle even challenging proof obligations, while a set of strong heuristics based on symbolic execution provide automation to keep the proof effort still small. VSE provides also a code generation mechanism to imperative programming languages like Ada or C.

In order to understand the specific way of integrating HETS and VSE, one needs to understand the philosophy behind HETS. The central idea of HETS is to provide a general integration and proof management framework. One can think of HETS acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations. The benefit of plugging VSE into HETS is that for both verification and refinement, we can use the general proof management mechanisms of the HETS motherboard, instead of the specialized refinement tools hard-wired into VSE. Moreover, the HETS motherboard already has plugged in a number of expansion cards (e.g., the theorem provers Isabelle, SPASS and more, as well as model finders) that can be used for VSE as well. The challenge is that typically, analysis and proof tools that shall be plugged into the HETS motherboard are not compatible with HETS expansion slots. Often, this is a matter of writing a suitable wrapper that encapsulates the tool in an expansion card that is compatible to the HETS motherboard. However, sometimes also the specification of the expansion slot has to be enhanced. Of course, such enhancements should only be done for very good reasons — otherwise, one will end up with slots containing hundreds of special pins. Since VSE provides a special notion of refinement, one is tempted to enhance the specification of the expansion slot in this case. However, we will see that we can do without such an enhancement.



Related work includes ad-hoc integration of (tools for) formal methods, see e.g. the integrated formal methods conference series [20], and integrations of decision procedures, model checkers and automated theorem provers into interactive theorem provers [11, 19]. However, these approaches are not as flexible as the HETS motherboard/expansion card mechanism. In many approaches, the interfaces for these integrations are ad-hoc and not re-used in many different contexts. Moreover, we will see in Sect. 6 below that the use of *logic translations as first class citizens* in the expansion card mechanism is crucial for integrating VSE and HETS in a modular way. This clearly is a novel feature of our approach.

The paper is an extension of [10] with full proofs of the meta theorems and a sample proof in VSE. It is organized as follows: Section 2 contains an informal description of HETS and its foundations. In particular, the notions of *institution* and *institution comorphism* can be imagined as the specification of two different types of expansion slot on the HETS motherboard. Section 3 presents the VSE methodology, and in Section 4, its underlying dynamic logic is (for the first time) organized as an institution, i.e. as an expansion card that can easily be plugged into the HETS motherboard. Section 5 recalls the algebraic specification notion of refinement and compares the way this concept is handled by HETS and VSE. In Section 6, we define two institution comorphisms, which can be thought of as further expansion cards that provide the VSE notion of refinement within HETS. In Section 7 we briefly present a standard example, illustrating the implementation of natural numbers as lists of binary digits, while Section 8 concludes the paper.

2 Presentation of HETS

HETS is a multi-logic proof management tool that heterogeneously integrates many languages, logics and tools on a strong semantic basis. The core of HETS is a heterogeneous extension of the specification language CASL, designed by the "Common Framework Initiative for Algebraic Specification and Development". However, HETS can also be used for languages that are completely different from CASL.

2.1 CASL

CASL has been designed from the outset as the central language in a family of languages. Soon sublanguages and extension of CASL, like the higher-order extension HASCASL, the coalgebraic extension COCASL, the modal logic extension MODALCASL, the reactive extensions CASL-LTL and CSP-CASL and others emerged. Luckily, CASL follows a separation of concerns — it has been designed in four different layers [3, 5, 26]:

basic specifications are unstructured collections of symbols, axioms and theorems, serving the specification of individual software modules. The specific logic chosen for CASL here is first-order logic with partial functions, subsorting and induction principles for datatypes;

structured specifications organize large specifications in a structured way, by allowing their translation, union, parameterization, restriction to an export interface and more. Still, structured specifications only cover the specification of individual software modules;

architectural specifications allow for prescribing the structure of implementations, thereby also determining the degree of parallelism that is possible in letting different programmers independently develop implementations of different subparts;

specification libraries allow the storage and retrieval of collections of specifications, distributed over the Internet.

2.2 Institutions

A crucial point in the design of these layers is that the syntax and semantics of each layer is orthogonal to that of the other layers. In particular, the layer of basic specifications can be changed to a different language and logic (e.g. an extension of CASL, or even a logic completely unrelated to CASL), while retaining the other layers. The central abstraction principle to achieve this separation of layers is the formalization of the notion of logical system as *institutions* [13], a notion that arose in the late 1970ies when Goguen and Burstall developed a semantics for the modular specification language Clear [8].

We recall informally this central notion here. An institution provides

- a notion of signature, carrying the context of user-defined (i.e. non-logical) symbols, and a notion of signature morphisms (translations between signatures);
- for each signature, notions of sentence and model, and a satisfaction relation between these (parameterized by signature);

- for each signature morphism, a sentence translation and a model reduction (the direction of the latter being opposite to the signature morphism), such that satisfaction is invariant under translation resp. reduction along signature morphisms. This has been summarized as *Truth is invariant under change of notation and enlargement of context.*

This leads to the following formal definition. Let $\mathscr{CAT}$ be the category of categories and functors.[1]

Definition 1 An *institution* $I = (\mathbf{Sign}^I, \mathbf{Sen}^I, \mathbf{Mod}^I, \models^I)$ consists of

- a category $\mathbf{Sign}^I$ of *signatures*,
- a functor $\mathbf{Sen}^I : \mathbf{Sign}^I \longrightarrow \mathbf{Set}$ giving, for each signature Σ, the set of *sentences* $\mathbf{Sen}^I(\Sigma)$, and for each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$, the *sentence translation map* $\mathbf{Sen}^I(\sigma) : \mathbf{Sen}^I(\Sigma) \longrightarrow \mathbf{Sen}^I(\Sigma')$, where often $\mathbf{Sen}^I(\sigma)(e)$ is written as $\sigma(e)$,
- a functor $\mathbf{Mod}^I : (\mathbf{Sign}^I)^{op} \longrightarrow \mathscr{CAT}$ giving, for each signature Σ, the category of *models* $\mathbf{Mod}^I(\Sigma)$, and for each signature morphism $\sigma : \Sigma \longrightarrow \Sigma'$, the *reduct functor* $\mathbf{Mod}^I(\sigma) : \mathbf{Mod}^I(\Sigma') \longrightarrow \mathbf{Mod}^I(\Sigma)$, where often $\mathbf{Mod}^I(\sigma)(M')$ is written as $M'|_\sigma$,
- a satisfaction relation $\models^I_\Sigma \subseteq |\mathbf{Mod}^I(\Sigma)| \times \mathbf{Sen}^I(\Sigma)$ for each $\Sigma \in \mathbf{Sign}^I$,

such that for each $\sigma : \Sigma \longrightarrow \Sigma'$ in $\mathbf{Sign}^I$ the following *satisfaction condition* holds:

$$M' \models^I_{\Sigma'} \sigma(e) \Leftrightarrow M'|_\sigma \models^I_\Sigma e$$

for each $M' \in \mathbf{Mod}^I(\Sigma')$ and $e \in \mathbf{Sen}^I(\Sigma)$. □

We will omit the index I when it is clear from the context.

A very prominent example is the institution $FOL^=$ of many-sorted first-order logic with equality [13]. Signatures are many-sorted first-order signatures, i.e. many-sorted algebraic signatures enriched with predicate symbols. Models are many-sorted first-order structures, and model reduction is done by translating a symbol that needs to be interpreted along the signature morphism before looking up its interpretation in the model that is being reduced. Sentences are first-order formulas, and sentence translation means replacement of the translated symbols. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure.

The institution $CFOL^=$ [26] adds sort generation constraints to $FOL^=$. These express that some of the carriers sets are generated by some of the operations (and possibly the other carrier sets); this amounts to an induction principle (in Sect. 4 below, formal details for similar formulas will be provided). $SubPCFOL^=$, the CASL institution [26], further equips $CFOL^=$ with subsorting and partial functions (which, however, will not play a role in this paper).

With the notion of institution providing the abstraction barrier between the layer of basic specifications on the one hand and the other layers on the other hand, it was quite natural (though also a great challenge) to realize this abstraction barrier *also at the level of tools*. HETS provides

[1] Strictly speaking, $\mathscr{CAT}$ is not a category but only a so-called quasi-category, which is a category that lives in a higher set-theoretic universe [2].

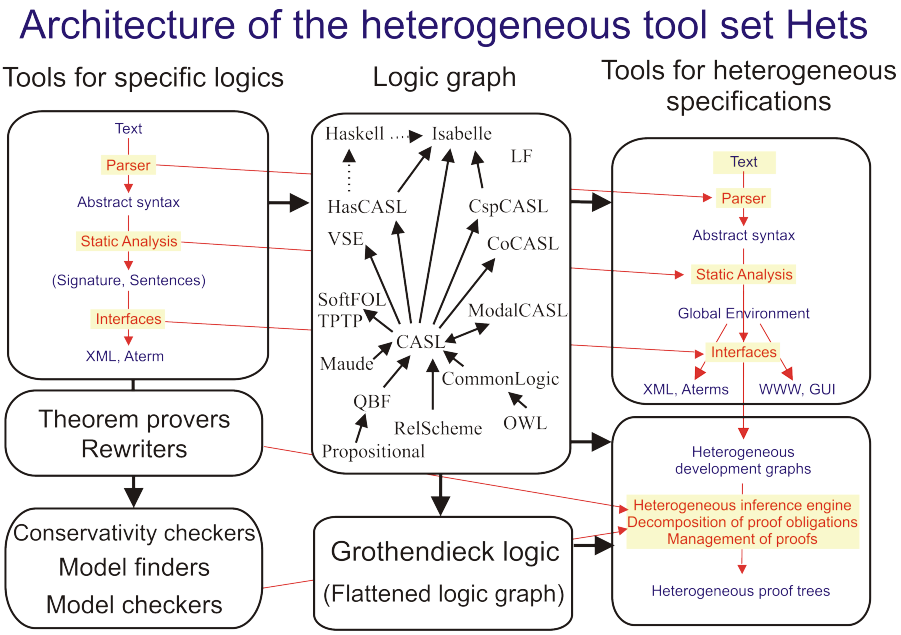


Figure 1: Architecture of HETS

an interface for logics and their proof tools, realized through a Haskell type class and which is similar to classes in object-oriented paradigm. This is exactly the specification of expansion slots mentioned in the introduction. This specification is heavily based on institutions, that is, the individual components of an institution are reflected in the interface. Of course, to be practically useful, the expansion slot specification contains additional components like concrete syntax, parsers, static analysis tools, and, last but not least, proof tools. The interface captures both interactive and automatic proof tools.

HETS allows for relating specifications written in different logics, e.g. CASL specifications can be imported for CASL extensions, or refinements can occur across different logics. In order to support this, HETS treats logic translations, formalized as *institution comorphisms (and morphisms)* [14], as first-class citizens (i.e., they are a different type of expansion card). An institution comorphism captures the idea of encoding or embedding between two institutions. It provides

- a translation of signatures (and signature morphisms),
- a translation of sentences,
- a translation of models (going backwards to the direction of the comorphism),

such that satisfaction is invariant under translation of sentences resp. models.

This leads to the following formal definition:

Definition 2 Given institutions I and J, an *institution comorphism* $\rho = (\Phi, \alpha, \beta) : I \longrightarrow J$ consists of

- a functor Φ: $\mathbf{Sign}^I \longrightarrow \mathbf{Sign}^J$,
- a natural transformation α: $\mathbf{Sen}^I \longrightarrow \mathbf{Sen}^J \circ \Phi$,
- a natural transformation β: $\mathbf{Mod}^J \circ \Phi^{op} \longrightarrow \mathbf{Mod}^I$

such that the following *satisfaction condition* is satisfied for all $\Sigma \in \mathbf{Sign}^I$, $M' \in \mathbf{Mod}^J(\Phi(\Sigma))$ and $e \in \mathbf{Sen}^I(\Sigma)$:

$$M' \models^J_{\Phi(\Sigma)} \alpha_\Sigma(e) \Leftrightarrow \beta_\Sigma(M') \models^I_\Sigma e.$$

Comorphisms can also be defined in a *simple theoroidal* variant [14], when signatures of the source institution I are mapped by Φ to *theories* of the target institution J, i.e. pairs (Σ', E') where Σ' is a signature of J and E' is a set of Σ'-sentences.

HETS is based on a *logic graph* of institutions and comorphisms, which is a parameter to the tools acting at the structured, architectural and library layers. The logic graph can be changed and extended without the need to change those logic independent analysis tools. The architecture of HETS is shown in Fig. 1. HETS' development graph component [23], inspired by the tool MAYA [16] (a cousin of VSE, also developed in Saarbrücken) provides a proof management for heterogeneous specifications, relying on proof tools for the individual logics involved.

We give several institutional notions that will be used in the following sections.

Definition 3 An institution I has the *weak amalgamation property* if its category of signatures has pushouts and moreover, given any span $\Sigma_1 \leftarrow \Sigma \rightarrow \Sigma_2$ and any two models M_i of Σ_i with the same Σ-reduct, there is a model M' of Σ' that reduces to M_i, where Σ' is obtained from the pushout $\Sigma_1 \rightarrow \Sigma' \leftarrow \Sigma_2$ of the span.

Moreover, if such model exists uniquely, we say that I has the amalgamation property.

Structured specifications are then introduced independently of the underlying institution I. At the basic level we have I-theories (Σ, E) and CASL provides a number of *structure-building operations*, taking a number of argument specifications and having a result a specification again. For details see [26]. The semantics of specifications is model-theoretic, in the sense that each specification SP is assigned the signature of the specification, $Sig(SP)$, and the model class of the specification, $Mod(SP)$. Moreover, if $\rho = (\Phi, \alpha, \beta) : I \rightarrow J$ is an institution comorphism and SP is a I-specification with $Sig(SP) = \Sigma$, we denote $\rho(SP)$ the J-specification with $Sig(\rho(SP)) = \Phi(\Sigma)$ and $Mod(\rho(SP)) = \{M' \in \mathbf{Mod}(\Phi(\Sigma)) \mid \beta_\Sigma(M') \in Mod(SP)\}$.

Definition 4 Let SP and SP' be two Σ-specifications and let $e \in Sen(\Sigma)$. Then:

- $SP \models_\Sigma e$ (SP *logically entails* e) if $M \models e$ for any $M \in Mod(SP)$;
- $SP \rightsquigarrow SP'$ (SP *refines to* SP') if $Mod(SP') \subseteq Mod(SP)$.

Definition 5 A comorphism $\rho = (\Phi, \alpha, \beta) : I \rightarrow J$

- has the *model expansion property* if each β_Σ is surjective on objects;
- admits *borrowing of entailment* if for any Σ-specification SP and any Σ-sentence e, $SP \models^I_\Sigma e \iff \rho(SP) \models^J_{Sig(\Phi(SP))} \alpha_\Sigma(e)$;

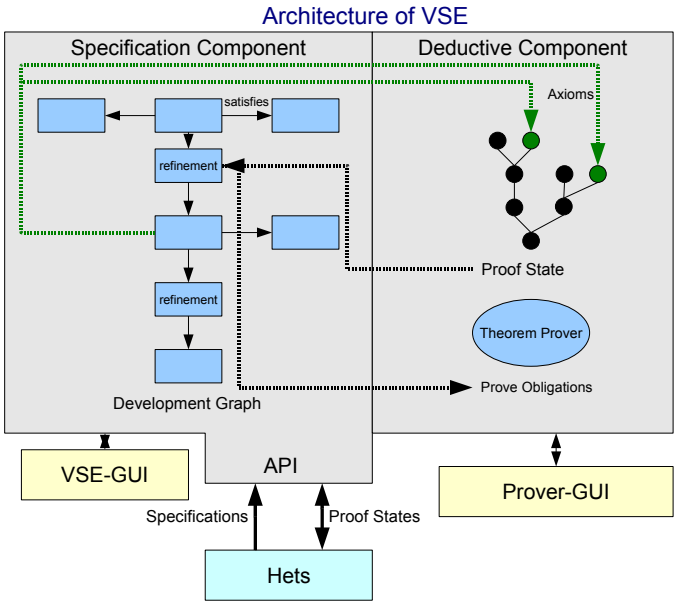


Figure 2: Architecture of VSE

- admits *borrowing of refinement* if $SP_1 \rightsquigarrow SP_2$ iff $\rho(SP_1) \rightsquigarrow \rho(SP_2)$.

3 Presentation of VSE

The *V*erification *S*upport *E*nvironment (VSE) is a tool that supports the formal development of complex large scale software systems from abstract high level specifications down to the code level. It provides both an administration system to manage structured formal specifications and a deductive component to maintain correctness on the various abstraction levels (see Fig. 2). Taken together, these components guarantee the overall correctness of the complete development. The structured approach allows the developer to combine specifications in an algebraic functional style with state-based formal descriptions of concurrent systems.

VSE has been developed in two phases on behalf the German Bundesamt für Sicherheit in der Informationstechnik (BSI) to satisfy the needs in software developments according to the upcoming standards ITSEC and Common Criteria. Since then, VSE has been successfully applied in several industrial and research projects, many of them being related to software evaluation [17, 4, 18, 9]. The models[2] developed with VSE comprise among others the control system of a heavy robot facility, the control system of a storm surge barrier, a formal security policy model conforming to the German signature law and protocols for chip card-based biometric identification.

[2] This use of the term "model" is in the sense of modeling, while the institutional use is in the sense of logic and model theory, see Sect. 4.3.

3.1 The VSE Methodology

VSE supports a development process that starts with a modular formal description of the *system model*, possibly together with separate *requirements* or *security objectives*. Logically the requirements have to be derivable from the system model. Therefore, the requirements lead to proof obligations that must be discharged by using the integrated deductive component of VSE.

In a *refinement process* the abstract system model can be related to more concrete models. This is in correspondence with a software development that starts from a high-level design and then descends to the lower software layers such that in a sense higher layers are implemented based on lower layers. Each such step can be reflected by a *refinement step* in VSE. These steps involve programming notions in the form of abstract implementations, that can later be exploited to generate executable code. Each refinement step gives rise to proof obligations showing the correctness of the implementations. Refinements also can be used to prove consistency of specifications, because they describe a way how to construct a model. This plays a major role for the formal specifications required for Common Criteria, which only need to cover higher abstraction levels.

In addition to the *vertical* structure given by refinement steps, VSE also allows the specification to be structured *horizontally* to organize the specifications on one abstraction level. Each single (sub)specification can be refined vertically or further decomposed horizontally, such that the complete development is represented by a *development graph*. The deductive component is aware of this structure. This is an important aspect for the interactive proof approach, as the structure helps the user to prove lemmas or proof obligations that require properties from various parts of the specification.

4 Institution of Dynamic Logic

VSE provides an interactive prover, which supports a Gentzen-style natural deduction calculus for dynamic logic. This logic is an extension of first-order logic with two additional kinds of formulas that allow for reasoning about programs. One of them is the box formula $[\alpha]e$, where α is a program written in an imperative language, and e is a dynamic logic formula. The meaning of $[\alpha]e$ can be roughly put as "After every terminating execution of α, e holds.". The other new kind of formulas is the diamond formula $\langle\alpha\rangle e$, which is the dual counter part of a box formula. The meaning of $\langle\alpha\rangle e$ can be described as "After some terminating execution of α, e holds".

We will now describe the formalization of this dynamic logic as an institution, denoted $CDyn^{=}$, in detail, because this has not been done in the literature so far. Moreover, as stated in the introduction, this step is crucial for turning VSE into an expansion card that can be plugged into the HETS motherboard.

4.1 Signatures

The starting point for dynamic logic signatures are the signatures of first-order logic with equality ($\mathrm{FOL}^{=}$) that have the form $\Sigma_{\mathrm{FOL}^{=}} = (S, F, P)$ consisting of a set S of sorts, a family F of function symbols and a family P of predicate symbols. Because we need to name procedures, we add an $S^* \times S^*$-sorted family $PR = (PR_{v,w})_{v,w \in S^*}$ of procedure symbols, leading to signatures of the

form $\Sigma = (S, F, P, PR)$. We have two separate lists v and w of the argument sorts of the procedure symbols in $PR_{v,w}$, in order to distinguish the sorts of the input parameters (v) from those of the output parameters (w). In VSE syntax, the input parameters of the procedures are preceded by **IN** and the output parameters, by **OUT**. In the case of functional procedures, since all parameters except the last are input parameters, these annotations are not present. When the string of output parameters consists of just one sort s, we can mark some of the procedures of $PR_{v,s}$ as *functional procedures* and we denote this subset as $FP_{v,s}$.

A signature morphism between two signatures maps sorts, operation symbols, predicate symbols and procedure symbols in a way such that argument and result sorts are preserved. Also, signature morphisms are required to map functional procedures to functional procedures.

Moreover, it is assumed that all signatures have a sort *Boolean* together with two constants *true* and *false* on it and this subsignature is preserved by signature morphisms.

4.2 Sentences

Let $\Sigma = (S, F, P, PR)$ be a dynamic logic signature with $PR = (PR_{v,w})_{v,w \in S^*}$. The variables will be taken from an arbitrary but fixed countably infinite set $\mathbb{X}$ which is required to be closed under disjoint unions.

First we define the syntax of the programs that may appear in dynamic logic formulas. The programs contain Σ-terms, which are predicate logical terms of $(S, (F_{v,s} \cup FP_{v,s})_{v \in S^*, s \in S}, P)$, i.e. in addition to variables and function symbols we allow symbols of *functional* procedures to occur in these terms. The set $\mathbb{P}_\Sigma$ of Σ-programs is the smallest set containing:

- **abort**
- **skip**
- $x := \tau$
- **declare** $x : s = \tau$
- **declare** $x : s = ?$
- $\alpha; \beta$
- **if** ε **then** α **else** β **fi**
- **while** ε **do** α **od**
- $p(x_1, x_2, \ldots, x_n; y_1, y_2, \ldots, y_m)$,

where $x, x_1, x_2, \ldots, x_n \in \mathbb{X}$ are variables, $y_1, y_2, \ldots, y_m \in \mathbb{X}$ are pairwise different variables, τ a Σ-term of sort s, ε a boolean Σ-formula (i.e. a Σ-formula without quantifiers, boxes and diamonds)[3], $\alpha, \beta \in \mathbb{P}_\Sigma$, p a procedure symbol, such that the sorts of $x_1, \ldots, x_n, y_1, \ldots, y_m$ match the argument and result sorts of p. Moreover, in the case of functional procedures, programs also contain **return** τ, where τ is a term of the result sort of the functional procedure.

[3] This restriction is motivated by the straightforward translation of such formulas into program expressions.

These kinds of program statements can be explained informally as follows: **abort** is a program that never terminates. **skip** is a program that does nothing. $x := \tau$ is the assignment. **declare** $x : s = \tau$ is the deterministic form of a variable declaration which sets x to the value of τ. Its nondeterministic form **var** $x : s =?$ sets x to an arbitrary value.[4] Notice that the nondeterministic declaration can not be used for functional procedures. $\alpha;\beta$ is the composition of the programs α and β, such that α is executed before β. The *conditional* **if** ε **then** α **else** β **fi** means that α is executed if ε holds, otherwise β is computed. The *loop* **while** ε **do** α **od** checks the condition ε, in case of validity executes α and repeats the loop. Finally, $p(x_1,x_2,\ldots,x_n;y_1,y_2,\ldots,y_m)$ calls the procedure p with input parameters $x_1,x_2,\ldots,x_n$ and output parameters $y_1,y_2,\ldots,y_m$.

There are three kinds of sentences that may occur in a Σ-dynamic logic specification.

1. The set of dynamic logic Σ-formulas is the smallest set containing
   - *True* and *False*
   - the (S,F,P)-first-order formulas e;
   - for any dynamic logic Σ-formulas e, e_1, e_2, any variable $x \in \mathbb{X}$, any sort $s \in S$, and any Σ-program α the formulas $[\alpha]e$, $\langle\alpha\rangle e$ and $\neg e$, $e_1 \wedge e_2$ and $\forall x : s.e$;

2. Procedure definitions are expressions of the form:

   defprocs
   procedure $pr_1(x_1^1,\ldots,x_{n_1}^1,y_1^1,\ldots,y_{m_1}^1)\alpha_1$
   $\ldots$
   procedure $pr_k(x_1^k,\ldots,x_{n_k}^k,y_1^k,\ldots,y_{m_k}^k)\alpha_k$
   defprocsend

   where $pr_i \in PR_{v_i,w_i}$ for some $v_i, w_i \in S^*$, $x_1^i,\ldots,x_{n_i}^i,y_1^i,\ldots,y_{m_i}^i$ are variables of the corresponding sorts in v_i, w_i, and $\alpha_i \in \mathbb{P}_\Sigma$ is a Σ-program with free variables from $\{x_1^i,\ldots,x_{n_i}^i,y_1^i,\ldots,y_{m_i}^i\}$. Notice that in VSE the functional procedures are written using **function** rather than **procedure** and without a formal output parameter; to simplify notation, we will ignore this in this section.

3. *Restricted sort generation constraints* express that a set of values defined by restriction procedure can be generated by the given set of procedures, the *constructors*. Syntactically a restricted sort generation constraints takes the form

   generated types
   $s_1 ::= p_1^1(\ldots)|p_2^1(\ldots)|\ldots|p_n^1(\ldots)$ **restricted by** r^1 ,
   $\ldots$
   $s_k ::= p_1^k(\ldots)|p_2^k(\ldots)|\ldots|p_n^k(\ldots)$ **restricted by** r^k ,

   where s_i are sort symbols, $p_1^i,\ldots,p_n^i$ are functional procedure symbols, the dots in $p_j^i(\ldots)$ etc. have to be replaced by a list of the argument sorts, and r^i is a procedure symbol taking one argument of sort s_i.

[4] In VSE one can also declare more than one variable in a **declare** list; for simplicity we restrict to the case of a single variable.

In order to make the satisfaction condition hold, we formally define a sort generation constraint over a signature Σ as a tuple (S', F', PR', θ), where $\theta : \Sigma_0 \to \Sigma$, $\Sigma_0 = (S_0, F_0, P_0, PR_0)$, $S' \subseteq S_0$, $F' \subseteq F_0$ and $PR' \subseteq PR_0$ such that in PR' there is a restriction procedure r_i for any sort s_i in S'.

For any signature morphism $\sigma : \Sigma \to \Sigma'$, the translation of Σ-sentences along σ is done by translating each symbol according to the sort, operation symbol, predicate symbol and procedure symbol mappings respectively. In the case of quantified sentences, $\forall X.e$ gets mapped to $\forall X'.\sigma(e)$, where for any sort s' of Σ', $X'_{s'}$ is defined as the disjoint union of X_s for all s such that $\sigma(s) = s'$. The translation of a sort generation constraint (S', F', PR', θ) over Σ along σ is defined as $(S', F', PR', \theta; \sigma)$.

4.3 Models

Let $\Sigma = (S, F, P, PR)$ be a dynamic logic signature with $F = (F_{w,s})_{w \in S^*, s \in S}, P = (P_w)_{w \in S^*}, PR = (PR_{v,w})_{v,w \in S^*}$. A (dynamic logic) Σ-model M maps each sort symbol $s \in S$ to a carrier set M_s, each function symbol $f \in F_{w,s}$ to a total function $M_f : M_w \to M_s$, each predicate symbol $p \in P_w$ to a relation $M_p \subseteq M_w$ and each procedure symbol $pr \in PR_{v,w}$ to a relation $M_{pr} \subseteq M_v \times M_w$, where $M_{(s_1,\ldots,s_n)}$ denotes $M_{s_1} \times M_{s_2} \times \ldots \times M_{s_n}$ for $(s_1, s_2, \ldots, s_n) \in S^*$. Functional procedures are required to be interpreted as total functions over their domain. Thus, such a model can be viewed as a $CFOL^=$ structure extended with the interpretation of procedure symbols.

For any signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct $M'|_\sigma$ of a Σ'-model M' interprets x as the interpretation of $\sigma(x)$ in the original model, where x can be either a sort, a function symbol, a predicate symbol or a procedure symbol.

4.4 Satisfaction of Dynamic Logic Formulas

Semantics is defined in a Kripke-like manner. For a given signature Σ and a Σ-model M the (program) states are variable valuations, i.e. partial functions taking sorted variables $x : s$ to values of M_s where s is a sort of Σ and $x \in \mathbb{X}$. We assume that for each sort there is a designated output variable for storing the return values of functional procedures; let us denote that variable o.

First, we need to define the interpretation of a term τ in a model M and a state q. Notice that because the states are partial, the interpretation can be undefined. The interpretation of terms is then done as usual inductively on the structure of terms:

- if τ is a variable $x : s$, then $\tau^{M,q} := q(x : s)$ (i.e. the value of the variable $x : s$ in state q when defined and undefined otherwise);
- if $\tau = f(\tau_1, \ldots, \tau_n)$ and $f \in F_{w,s}$ or $f \in FP_{w,s}$, then $\tau^{M,q} := M_f(\tau_1^{M,q}, \ldots, \tau_n^{M,q})$, and undefinedness of any of the $\tau_i^{M,q}$ is propagated;

The semantics of a program α with respect to a model M is a predicate $\_[\![\alpha]\!]^M\_$ on two program states. $q[\![\alpha]\!]^M r$ can be read as: If α is started in state q it may terminate after having changed the state to r.

- $q[\![\mathbf{skip}]\!]^M q$
- not $q[\![\mathbf{abort}]\!]^M r$
- $q[\![x := \tau]\!]^M r \Leftrightarrow r = q[x : s \leftarrow \tau^{M,q}]$ and $\tau^{M,q}$ is defined, where $s = sort(\tau)$
- $q[\![x := \tau]\!]^M r$ does not hold for any r if $\tau^{M,q}$ is not defined
- $q[\![\alpha;\beta]\!]^M r \Leftrightarrow$ for some state s : $q[\![\alpha]\!]^M s$ and $s[\![\beta]\!]^M r$
- $q[\![\mathbf{declare}\, x : s = \tau]\!]^M r \Leftrightarrow q[\![x := \tau]\!]^M r$
- $q[\![\mathbf{declare}\, x : s = ?]\!]^M r \Leftrightarrow$ for some $a \in s^M : r = q[x \leftarrow a]$
- $q[\![\mathbf{if}\, \varepsilon\, \mathbf{then}\, \alpha\, \mathbf{else}\, \beta\, \mathbf{fi}]\!]^M r \Leftrightarrow (q \models \varepsilon$ and $q[\![\alpha]\!]^M r)$ or $(q \models \neg\varepsilon$ and $q[\![\beta]\!]^M r)$
- $q[\![\mathbf{while}\, \varepsilon\, \mathbf{do}\, \alpha\, \mathbf{od}]\!]^M r \Leftrightarrow q([\![\mathbf{if}\, \varepsilon\, \mathbf{then}\, \alpha\, \mathbf{else}\, \mathbf{skip}\, \mathbf{fi}]\!]^M)^* r$ and $r \models \neg\varepsilon$
- $q[\![pr(x_1,\ldots,x_n;y_1,\ldots,y_m)]\!]^M r \Leftrightarrow pr_M(q(x_1),\ldots,q(x_n);r(y_1),\ldots,r(y_m))$
- $q[\![\mathbf{return}\, \tau]\!] r \Leftrightarrow r = q[o : s \leftarrow \tau^{M,q}]$, where o is the output variable of sort $sort(\tau)$

where for any program α, $([\![\alpha]\!]^M)^*$ is the reflexive transitive closure of the relation $[\![\alpha]\!]^M$, and the state $q[x \leftarrow a]$ is defined as $q[x \leftarrow a](y) = \begin{cases} q(y) & \text{if } y \neq x \\ a & \text{if } y = x \end{cases}$, τ is a Σ-term without predicate symbols and $\tau^{M,q}$ is the evaluation of the term τ with respect to the model M and state q.

We define satisfaction on a model M and a program state r as follows:

- $M, r \models True$ and $M, r \not\models False$
- $M, r \models p(\tau_1,\ldots\tau_n) \Leftrightarrow$ for all $i = 1,\ldots,n$, $\tau_i^{M,r}$ is defined and $M_p(\tau_1^{M,r},\ldots,\tau_n^{M,r})$
- $M, r \models \tau_1 = \tau_2 \Leftrightarrow \tau_1^{M,r} = \tau_2^{M,r}$ and interpretation of both terms in the state r is defined
- $M, r \models \neg e \Leftrightarrow M, r \not\models e$
- $M, r \models e \wedge e' \Leftrightarrow M, r \models e$ and $M, r \models e'$
- $M, r \models e \vee e' \Leftrightarrow M, r \models e$ or $M, r \models e'$
- $M, r \models \forall x : s.e \Leftrightarrow$ for all $a \in M_s$: $\quad M, r[x : s \leftarrow a] \models e$
- $M, r \models [\alpha]e \Leftrightarrow$ for all program states q with $r[\![\alpha]\!]^M q$: $M, q \models e$

The formula $\langle\alpha\rangle e$ is to be read as an abbreviation for $\neg[\alpha]\neg e$. Finally a formula e holds on a model M ($M \models e$), if for all program states r it holds on M and r ($M, r \models e$).

4.5 Satisfaction of Procedure Definitions

The procedures in our model will not have any side effects (except for modifying the output parameters).

Unwinding a procedure call by replacing it by the body of the procedure and substituting the formal parameter variables by the actual parameters should not change the result of a program. Therefore, for a signature Σ, a Σ-model M is a model of a procedure declaration without recursion

defprocs
procedure $pr_1(x_1^1,\ldots,x_{n_1}^1,y_1^1,\ldots,y_{m_1}^1)\alpha_1$
$\ldots$
procedure $pr_k(x_1^k,\ldots,x_{n_k}^k,y_1^k,\ldots,y_{m_k}^k)\alpha_k$
defprocsend

if

$$M \models \forall x_1^i,\ldots,x_{n_i}^i,r_1^i,\ldots,r_{m_i}^i :$$
$$(\langle pr(x_1^i,\ldots,x_n^i;y_1^i,\ldots,y_m^i)\rangle y_1^i = r_1^i \wedge \cdots \wedge y_{m_i}^i = r_{m_i}^i) \Leftrightarrow \langle\alpha\rangle y_1^i = r_1^i \wedge \cdots \wedge y_{m_i}^i = r_{m_i}^i$$

holds for any $i = 1\ldots k$. Abbreviating the procedure declaration as Π, we then write $M \mid\!\models \Pi$.

Unfortunately, in the presence of recursion this does neither make the procedure definitions non-ambiguous, nor compliant with conventional semantics of programming languages. Therefore, from several models complying with the definitions, the minimal model (with respect to some order) will be chosen. The order compares the interpretations of the procedures symbols, such that the order relation $M_1 \leq_\Pi M_2$ holds for two models M_1 and M_2 for the same signature $\Sigma = (S,F,P,PR)$ iff $pr_i^{M_1} \subseteq pr_i^{M_2}$ for all procedure symbols pr, and the interpretations of sort, function, predicate symbols and procedure symbols which are not part of Π are identical. Moreover, we say that a model M_2 is a Π-*variant* of M_1, written $M_1 \equiv^\Pi M_2$, if M_1 and M_2 agree on the interpretations of all symbols except possibly the procedure symbols in Π. Then we define that the satisfaction of a procedure declaration Π by M as follows:

$$M \models \Pi \quad \text{iff} \quad M \mid\!\models \Pi \text{ and for all } \Pi\text{-variants } M' \text{ of } M, M' \mid\!\models \Pi \text{ implies } M \leq_\Pi M'.$$

4.6 Satisfaction of restricted sort generation constraints

A restricted sort generation constraint (S',F',PR',θ) written as

generated types $s_i ::= p_1^i(\ldots)|p_2^i(\ldots)|\ldots|p_n^i(\ldots)$ **restricted by** r^i ,

is said to hold in a model M, if the subset of the carrier $(M|_\theta)_{s_i}$ on which the restriction procedure r^i terminates is generated by the functional procedures p_1^i, p_2^i, $\ldots$ p_n^i (called *constructor procedures*). In more detail: for each element a of $(M|_\theta)_{s^i}$ such that $M|_\theta, q \models \langle r^i(x)\rangle true$ when q is a state such that $q(x) = a$, there must by a term t built with constructor procedures only and having no free variables of sorts s_i and a state v such that v is defined only for variables of sorts distinct from the s_i and that $t^{M|_\theta,v} = a$ holds.

4.7 Satisfaction condition

Proposition 1 *Let $\Sigma = (S,F,P,PR)$ and $\Sigma' = (S',F',P',PR')$ be two dynamic logic signatures, $\sigma : \Sigma \to \Sigma'$ a signature morphism, M' a Σ'-model and e a Σ-sentence. Then:*

$$M' \models \sigma(e) \iff M'|_\sigma \models e$$

Proof.
We prove the satisfaction condition by case distinction on e.

1. e is a dynamic logic formula.

   In this case, we prove by induction over e that $M',t' \models \sigma(e) \iff M'|_\sigma, t'|_\sigma \models e$, where for any state t' for Σ' and M', we define the state $t'|_\sigma$ for Σ and $M'|_\sigma$ by taking $t'|_\sigma(x:s) = t(x:\sigma(s))$ for each sort s of Σ. The proof is pretty much routine, the interesting case being when e is of form $[\alpha]e'$. Let us denote $M = M'|_\sigma$.

   We begin with a lemma:

   Lemma 1 *For any state t' for Σ' and M', any state q for Σ and M and any Σ-program α, $t'|_\sigma [\![\alpha]\!]^M q$ iff there is a state q' such that $t'[\![\sigma(\alpha)]\!]^{M'} q'$ and $q'|_\sigma = q$.*

   which can be proven by induction on α, by making use of the fact that the variables used in α are bijectively renamed in the states q and q'.

   Let t' be a state for Σ' and M' such that $M',t' \models \sigma([\alpha]e')$. By definition this means that for any state p' such that $t'[\![\sigma(\alpha)]\!]^{M'} p'$, $M',p' \models \sigma(e')$. Let q be a state such that $t'|_\sigma [\![\alpha]\!]^M q$. We need to prove that $M,q \models e'$. Using Lemma 1, there is a state q' such that $t'[\![\sigma(\alpha)]\!]^{M'} q'$ and $q'|_\sigma = q$. By the hypothesis we get that $M',q' \models \sigma(e')$. By the inductive hypothesis for e', we obtain $M,q'|_\sigma \models e'$. Since $q'|_\sigma = q$, this means $M,q \models e'$. Since q was arbitrary such that $t'|_\sigma [\![\alpha]\!]^M q$, we obtain $M,t'|_\sigma \models [\alpha]e'$.

   For the reverse implication, let t' be a state for Σ' and M' such that $M,t'|_\sigma \models [\alpha]e'$. By definition this means that for any state p such that $t'|_\sigma [\![\alpha]\!]^M p$, $M,p \models e'$. Let q' be a state such that $t'[\![\sigma(\alpha)]\!]^{M'} q'$. By Lemma 1 we get $t'|_\sigma [\![\alpha]\!]^M q'|_\sigma$. By the hypothesis we get that $M,q'|_\sigma \models e'$. By the inductive hypothesis for e' we get that $M',q' \models \sigma(e')$ and since q' was arbitrary, by definition $M',t' \models [\sigma(\alpha)]\sigma(e')$.

2. e is a procedure definition.

   We first prove the following lemma:

   Lemma 2 *Let $\sigma : \Sigma \to \Sigma'$, let Π be a procedure definition in Σ and let N be a Σ'-model, and let $M = N|_\sigma$. Then*

   $N \leq_{\sigma(\Pi)} N'$ for any $\sigma(\Pi)$-variant N' of N such that $N' \mid\!\models \sigma(\Pi)$
   iff $M \leq_\Pi M'$ for any Π-variant M' of M such that $M' \mid\!\models \Pi$.

Proof: For the left to right implication, assume for a contradiction that M is not minimal among its Π-variants satisfying Π. Then there exists a Π-variant of M, M^0, satisfying Π such that $M \not\leq_\Pi M^0$. We define a σ-expansion N^0 of M^0 by interpreting all symbols outside Π as in the model N and taking $N^0_{\sigma(\pi)} = M^0_\pi$ for any procedure π defined in Π. The well-definedness is ensured by $M^0 \mid\!\models \Pi$ and moreover, by the satisfaction condition we get that $N^0 \mid\!\models \sigma(\Pi)$. Since $N \not\leq_{\sigma(\Pi)} N^0$, we get a contradiction with the minimality of N.

For the right to left implication, assume for a contradiction that N is not minimal. Then there exists a $\sigma(\Pi)$-variant of N, denoted N^0 such that $N \not\leq_{\Sigma(\Pi)} N^0$. Let $M^0 := N^0|_\sigma$. By the satisfaction condition we get $M^0 \mid\!\models \Pi$. By definition of reduct it follows that $M^0_\pi = N^0_{\sigma(\pi)}$ and since $M_\pi = N_{\sigma(\pi)}$ we get $M \not\leq_\Pi M^0$ which contradicts the minimality of M.

□

The satisfaction condition follows from the definition of the model reduct for procedure symbols (which ensures minimality) and from Lemma 2.

3. *e is a restricted sort generation constraint.*

   The satisfaction condition is obvious.

5 Refinement

The methodology of formal software development by stepwise refinement describes the ideal process (which in practice is more a loop with continuous feedback) as follows: starting from initial informal requirements, these are translated to a formal requirement specification, which is then further refined to a formal design specification and then to an executable program.

Simple refinements between specifications can be expressed as so-called *views* in CASL, which are just specification morphisms: given two structured specifications SP_1 and SP_2 such that $Sig(SP_i) = \Sigma_i$ and $Mod(SP_i) = \mathscr{M}_i$, for $i = 1,2$, a specification morphism $\sigma : SP_1 \to SP_2$ is a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ such that $M|_\sigma \in \mathscr{M}_1$ for any model $M \in \mathscr{M}_2$. The degree of looseness diminishes along a refinement (technically, the model class shrinks). For more complex refinements involving architectural decompositions (i.e. branching points in the emerging refinement tree), a refinement language has been designed [25]. Sometimes (e.g. when refining arrays to stacks with pointers), an observational interpretation of specifications is needed. This means that values exhibiting the same observable behavior are identified (that is, observational congruence is generated implicitly). This has been developed in theory to some degree [6, 29], but not implemented in HETS yet. By contrast, the VSE specification language supports a refinement approach based on explicit submodels and congruences [28], an idea that dates back to Hoare [15]. This more specific and simpler approach has been successfully applied in practice, and moreover, it is linked with a mechanism for generating code in imperative programming languages like Ada or C. Hence, integrating this approach into HETS brings considerable advantages.

VSE's refinements associate an abstract data type specification, called the *export specification* of the refinement, with an implementation. The implementation is based on another theory,

called the *import specification* and contains several functional procedures written in an imperative language. These procedures use the functions and predicates of the import specifications. A so called *mapping* relates each sort of the export specification to a sort of the import specification, while the functions and procedures are mapped to procedures in the import specification.

A refinement describes the construction of a model for the signature of the export specification (export model) from a model of the import specification (import model). The functions and predicates are interpreted by the computations of the procedures. The elements of the carrier sets of the export model are constructed from the carrier sets of the import model. The implementations are allowed to represent a single value in the export specification by several values of the import specifications. For example, when implementing sets by lists, a set might be represented by any list containing all elements of the set in any order. Furthermore, VSE does not require that all values of a sort in the import specification really represent a value of the export specification. In the example below where we will implement natural numbers by binary words, we will exclude words with leading zeroes. In order to describe the construction of the carrier sets, the refinement contains two additional procedures for each sort: a procedure defining a *congruence* relation and a procedure defining a *restriction*. The restriction terminates on all elements that represent export specification values. The congruence relation determines the equivalence classes that represent the elements of the export model.

We can express this formally as in the following definition:

Definition 6 Let SP be a $CFOL^{=}$-specification and SP' a $CDyn^{=}$-specification. Then SP' is a *refinement* of SP, denoted $SP \rightsquigarrow_{VSE} SP'$, if for all $M \in Mod^{CDyn^{=}}(SP')$, $\langle M \rangle/_{\equiv} \in Mod^{CFOL^{=}}(SP)$, where $\langle M \rangle/_{\equiv}$ denotes the model obtained from M by restricting the elements of each sort according to the restriction procedures and taking the quotient to the congruence relation. [5]

Note that the definition of refinement is given in terms of semantics. The VSE system generates proof obligations that are sufficient for guaranteeing that a $CDyn^{=}$-specification is indeed a refinement of a $CFOL^{=}$-specification using only syntactical deduction.

6 VSE Refinement as an Institution Comorphism

When integrating VSE and its notion of refinement into HETS, a naive approach would extend HETS with a new notion of *restriction-quotient refinement link* in HETS, and would extend both the HETS motherboard and the expansion slot specification in a way that makes it possible to deal with such refinement links. VSE easily could be turned into an expansion card that is able to prove these refinement links.

However, this approach has a severe disadvantage: the specification of expansion slots needs to be extended! If we did this for every tool that is newly integrated into HETS (and every tool comes with its own special features), we would quickly arrive at a very large and unmanageable expansion slot specification.

Fortunately, the heterogeneity of HETS offers a better solution: we can encode VSE refinement as ordinary refinement in HETS, with the help of an institution comorphism that does the actual

[5] For a definition of quotients of first-order models, see [29].

restriction-quotient construction. With this approach, only the HETS logic graph needs to be extended by a logic and a comorphism; actually, we will see that two comorphisms are necessary. That is, we add two further expansion cards doing the work, while the logic-independent part of HETS, i.e. the motherboard and the expansion slot specification, can be left untouched!

6.1 The Refinement Comorphism

We model the refinement notion of VSE by a comorphism from the CASL institution $CFOL^{=}$ to the VSE institution $CDyn^{=}$. The intuition behind it can be summarized as follows. At the level of signatures, for each sort we need to introduce procedure symbols for the equality relation and for the restriction formula together with axioms specifying their expected behaviour, while for function and predicate symbols, we need to introduce procedure symbols for their implementations. For all these symbols, we assign no procedure definition but rather leave them loosely specified; in this way, the choice of a possible implementation is not restricted. The sentence translation is based on translation of terms into programs implementing the representation of the term. The model reduct performs the submodel/quotient construction, leaving out the values that do not satisfy the restriction formula and quotienting by the congruence generated by the equality procedure.

We now define the simple theoroidal comorphism $CASL2VSERefine : CFOL^{=} \to CDyn^{=}$. Each CASL signature $\Sigma = (S,F,P)$ is mapped to the $CDyn^{=}$ theory $((S,\emptyset,\emptyset,PR),E)$, denoted $\Phi(\Sigma)$. PR contains (1) for each sort s, a symbol $restr\_s \in PR_{[s],[]}$ for the *restriction* on the sort and a symbol $eq_s \in PR_{[s,s],[Boolean]}$ for the *equality* on the sort and (2) for each function symbol $f : w \to s \in F_{w,s}$, a symbol $gn\_f : w \to s \in PR_{w,[s]}$ and for each predicate symbol $p : w \in P_w$, a symbol $gn\_p : w \to [Boolean] \in PR_{w,[Boolean]}$.

The set of axioms E contains sentences saying that for each sort s, (1) eq_s is a congruence and it terminates for inputs satisfying the restriction and (2) the procedures that implement functions/predicates terminate for inputs satisfying the restriction and their results also satisfy the restriction. These properties are to be proven when providing an actual implementation. The general pattern of the translation is presented in Fig. 3, which gives the symbols and the sentences introduced in the resulting VSE theory for each symbol of the CASL theory that is translated. Notice that to improve readability, we only considered the case of unary function/predicate symbols; the generalization to symbols of arbitrary arity is obvious.

A CASL signature morphism $\sigma : \Sigma \to \Sigma'$ is mapped to the $CDyn^{=}$ morphism $\Phi(\sigma) : \Phi(\Sigma) \to \Phi(\Sigma')$ which works like σ on sorts and procedure symbols corresponding to function/predicate symbols in Σ and for each sort s of Σ maps eq_s to $eq_{\sigma(s)}$ and $restr_s$ to $restr_{\sigma(s)}$.

Given a CASL signature $\Sigma = (S,F,P)$ and a model M' of its translation $\Phi(\Sigma) = ((S,\emptyset,\emptyset,PR), E)$, we define the translation of M' to an (S,F,P)-model, denoted $M = \beta_\Sigma(M')$. The interpretation of a sort s in M is constructed in two steps. First we take the subset $M_{restr\_s} \subseteq M'_s$ of elements, for which the restriction predicate holds. Then we take the quotient $M_{restr\_s}/_{\equiv}$ according to the congruence relation $\equiv$ defined by eq_s, such that for all $a,b \in M'_s$, $a \equiv b$ is equivalent to $M', t \models \langle eq_s(x_1,x_2;y)\rangle y = true$ whenever t is a state such that $t(x_1) = a$ and $t(x_2) = b$. For each function symbol f, we define the value of M_f in the arguments $a_1,\ldots,a_n$ to be the value returned by the call of procedure M'_{gn_f} on inputs $a_1,\ldots,a_n$, that is $M_f(a_1,\ldots,a_n) = b$ if and

CASL	VSE	VSE sentences
sort s	sort s $eq_s \in PR_{[s,s],[Boolean]}$ $restr_s \in PR_{[s],[]}$	$\langle restr_s(x)\rangle true \wedge \langle restr_s(y)\rangle true \Rightarrow \langle eq_s(x,y;e)\rangle true$
		$\langle restr_s(x) true\rangle \Rightarrow \langle eq_s(x,x;e)\rangle e = true$
		$\langle restr_s(x)\rangle true \wedge \langle restr_s(y)\rangle true \wedge \langle eq_s(x,y;e)\rangle e = true \Rightarrow \langle eq_s(y,x;e)\rangle e = true$
		$\langle restr_s(x)\rangle true \wedge \langle restr_s(y)\rangle true \wedge \langle restr_s(z)\rangle true \wedge \langle eq_s(x,y;e)\rangle e = true \wedge \langle eq_s(y,z;e)\rangle e = true \Rightarrow \langle eq_s(x,z;e)\rangle e = true$
$f \in F_{s\to t}$	$gn\_f \in PR_{[s],[t]}$	$\langle restr_s(x)\rangle true \wedge \langle restr_s(y)\rangle true \wedge \langle eq_s(x,y;e)\rangle e = true \Rightarrow \langle y1 := gn\_f(x)\rangle\langle y2 := gn\_f(y)\rangle\langle eq_t(y1,y2;e)\rangle e = true$
		$\langle restr_s(x)\rangle true \implies \langle gn\_f(x;y)\rangle\langle restr_t(y)\rangle true$
$p \in P_s$	$p \in PR_{[s],[Boolean]}$	$\langle restr_s(x)\rangle true \wedge \langle restr_s(y)\rangle true \wedge \langle eq_s(x,y;e)\rangle e = true \Rightarrow \langle gn\_p(x;r1)\rangle\langle gn\_p(y;r2)\rangle r1 = r2$
		$\langle restr_s(x)\rangle true \Rightarrow \langle gn\_p(x;e)\rangle true$

Figure 3: Summary of the signature translation part of the comorphism *CASL2VSERefine* (for simplicity, only unary symbols are shown).

only if $M',t \models \langle gn\_f(x_1,\ldots,x_n;y)\rangle y = z$ when t is a state such that $t(x_i) = a_i$ for any $i = 1,\ldots,n$ and $t(z) = b$. Axioms (1) and (2) in E ensure that M_f is total and well-defined. Similarly and using the same notations, for each predicate symbol p, $M_p(a_1,\ldots,a_n)$ holds iff $M',t \models \langle gn\_p(x_1,\ldots,x_n;y)\rangle y = true$.

Proposition 2 *The model translation is natural.*

Proof. Follows easily from definition of the model translation and the definition of model reducts. □

Sentence translation is based on translation of terms into programs that compute the representation of the term. Basically, each function application is translated to a procedure call of the implementing procedure, and new output variables are introduced:

- a variable x is mapped to $x := x$, where the left-hand side x is the output variable and the right-hand side x is the logical variable;
- a constant c is mapped to $gn\_c(;y)$, where gn_c is the procedure implementing the constant and y is a new output variable;
- a term $f(t_1,\ldots,t_n)$ is mapped to $\alpha_1;\ldots\alpha_n;a := gn\_f(y_1,\ldots,y_n)$, where α_i is the translation of t_i with the output variable y_i and a is a new output variable.

Then the sentence translation is defined inductively:

- an equation $t_1 = t_2$ is translated to

$$\langle\alpha_1\rangle;\langle\alpha_2\rangle;\langle eq_s(y_1,y_2;y)\rangle y = true$$

where α_i is the translation of the term t_i, with the output variable y_i

- a predicate application $p(t_1,\ldots,t_n)$ is translated to

$$\langle\alpha_1\rangle\ldots\langle\alpha_n\rangle\langle gn\_p(y_1,\ldots,y_n;y)\rangle y = true$$

where α_i is the translation of the term t_i, with the output variable y_i

- Boolean connectives of formulas are translated into the same connections of their translated formulas;
- for universally and existentially qualified formulas one also has to make sure that the bound variables are assigned a value that satisfies the restriction: e.g $\forall x : s.e$ gets translated to $\forall x : s.\langle restr_s(x)\rangle true \Rightarrow \alpha(e)$, where we denoted with $\alpha(e)$ the translation of e.

An example of how a CASL sentence is translated along the *CASL2VSERefine* comorphism will be introduced in the next section in Fig. 6.

Sort generation constraints are translated to restricted sort generation constraints over implementing procedures. For example, assume we have in the abstract specification of natural numbers a sort generation constraint:

$$generated\ type\ nat\ ::=\ 0 \mid suc\ (nat)$$

Then in the VSE theory resulting from translation along comorphism, the restricted sort generation constraint

$$\textbf{generated type}\ nat\ ::=\ gn\_0 \mid gn\_suc(nat)\ \textbf{restricted by}\ restr\_nat\ .$$

is introduced, where gn_0 and gn_suc are the procedures implementing the constructors and $restr\_nat$ is the restriction procedure symbol on sort nat.

Proposition 3 *The sentence translation is natural.*

Proof. Follows easily by induction on sentences and by noticing that for any CASL signature morphism $\sigma : \Sigma \to \Sigma'$ and any Σ-term t we have that the program computing the representation of $\sigma(t)$ is the σ-image of the program computing the representation of t. □

Lemma 3 *Let Σ be a CFOL$^=$-signature and let M' be a model of the theory $\Phi(\Sigma)$. Denoting $M = \beta_\Sigma(M')$ we have that for any Σ-term t and any state q for $\Phi(\Sigma)$ and M', $M_t = a$ iff $M', q \models \langle\alpha_\Sigma(t)\rangle y = z$, where y the output variable of $\alpha_\Sigma(t)$ and z is a variable such that $q(z) = a$.*

Proof: Follows by induction on the structure of the term t. □

Theorem 1 *The satisfaction condition for the comorphism CASL2VSERefine holds.*

Proof: The proof follows by induction on the structure of sentences and making use of Lemma 3. □

Notice that this construction follows very faithfully the steps of the refinement method of VSE, as described in section 5. The export specification of VSE is a first-order specification that we can translate along the comorphism *CASL2VSERefine* to generate the same kind of proof obligations that VSE would generate to prove correctness of a refinement. The difference is that now they are built using abstract (i.e. loose) procedure names and actual implementations are to be later plugged in by means of a view which corresponds to the VSE mapping, with the exception that instead of pairing export specification symbols with implementations, the view rather pairs abstract procedures with implementations. Moreover, the correctness of the view ensures us that a model of the implementation reduces along the signature morphism induced by the view to a model of the translation of the original export specification, that we can further translate along the comorphism to obtain a model of the export specification. Thus we achieve that the model semantics of the refinement in VSE [1] and of the refinement expressed using the comorphism *CASL2VSERefine* coincide.

Definition 7 Let I an J be two institutions, $\rho = (\phi, \alpha, \beta) : I \to J$ be an institution comorphism. Let SP be a I specification and SP' be a J specification. We say that SP' is a *heterogeneous refinement of SP along ρ* if for each $M \in Mod^J(SP')$, $\beta(M) \in Mod^I(SP)$.

Our result can be formulated as follows:

Theorem 2 *Let SP be a* CASL *specification and SP' a VSE specification. Then $SP \leadsto_{VSE} SP'$ iff SP' is a heterogeneous refinement of SP along CASL2VSERefine.*

We can now show that the proof calculus for heterogeneous development graphs, combined with the VSE prover, can be used for discharging refinement proof obligations in a sound way.

The following two lemmas follow easily:

Lemma 4 *The institution $CDyn^{=}$ has the amalgamation property.*

Proof idea: Similar to the proof for first-order logic, see for example [12].

Lemma 5 *The comorphism CASL2VSERefine admits model expansion.*

Proof: For any $CFOL^{=}$ signature Σ and each Σ-model M, we build a $\Phi(\Sigma)$-model by interpreting sorts s as M_s, functions gn_f like M_f, predicates gn_p as M_p, the equality as the set-theoretical equality and the restriction as always returning $true$. It is easy to see that the model such built satisfies the axioms of $\Phi(\Sigma)$ and it reduces via β_Σ to M. □

Notice that in VSE we do not have hiding as a structuring operation, and therefore all specifications are flattenable (see e.g. [29]). The following corollary follows then directly from Lemma 5 and a result from [21].

Corollary 1 *The comorphism CASL2VSERefine admits borrowing of entailment and of refinement.*

Unfortunately, we cannot expect completeness here, because first-order dynamic logic is not

finitely axiomatisable [7].

6.2 Structuring in Context of Refinement

Consider a refinement from an abstract to a refined specification where a theory of a library (e.g. the natural numbers) or a parameter theory that will be instantiated later occurs both in the abstract and the refined specification. Such common import specifications should not be refined, but rather kept identically — and this is indeed the case in VSE.[6]

To handle this situation in the present context, the import of a first-order specification into a dynamic logic specification is not done along the trivial inclusion comorphism from $CFOL^{=}$ to $CDyn^{=}$ — this would mean that the operations of the import need to be implemented as procedures. Instead, we define a comorphism $CASL2VSEImport : CFOL^{=} \to CDyn^{=}$, which, besides keeping the first-order part, will introduce for the symbols of the import specification new procedure symbols, similarly to *CASL2VSERefine*. Namely each $CFOL^{=}$ signature (S,F,P) is translated to the $CDyn^{=}$ theory $((S,F,P,PR),E)$ where PR is the same as in the definition of the translation of (S,F,P) along *CASL2VSERefine* and E contains the following types of sentences:

- for each sort $s \in S$, sentences giving implementations for the restriction and the equality of s, as follows:

  PROCEDURE $restr_s(x)$
  BEGIN SKIP END;

  and respectively

  FUNCTION $eq_s(x,y)$
  BEGIN IF $x = y$ **THEN RETURN True**
  ELSE RETURN False FI END;

  with the intuitive meaning that no element of the sort is restricted and the equality on the sort is defined as the (meta-)equality on the interpretation of the sort;

- for each operation symbol $f \in F_{s\to t}$, a sentence giving the implementation of the corresponding procedure symbol $gn\_f \in PR_{[s],[t]}$:

  FUNCTION $gn\_f(x)$
  BEGIN DECLARE $y : t := f(x)$; **RETURN** y **END**;

  which means that the implementation of the functional procedure for f returns as result exactly the value $f(a)$ for each input a;

- for each predicate symbol $p \in P_s$, a sentence giving the implementation of the corresponding procedure symbol $gn\_p \in PR_{[w],[Boolean]}$:

  FUNCTION $gn\_p(x)$
  BEGIN DECLARE $y : Boolean := p(x)$; **RETURN** y **END**;

[6] This resembles a bit the notion of imports of parameterized specifications in CASL [26], where the import is shared between formal and actual parameter and is kept identically.

again with the meaning that the functional procedure gn_p is implemented as returning true only on those inputs that make the predicate p hold.

The result of choosing these implementations is that the sorts are not restricted, the congruence on each sort is simply the equality and the functional procedures introduced for operation/predicate symbols have the same behavior as the original symbols, i.e. give the same results on same inputs. Also notice that the signature morphisms translation of the comorphism *CASL2VSEImport* is the straightforward one, the translation of CASL sentences along the comorphism is simply the identity, and the models can be reduced in an obvious way by simply forgetting the interpretations of procedure symbols. The satisfaction condition of the comorphism follows immediately.

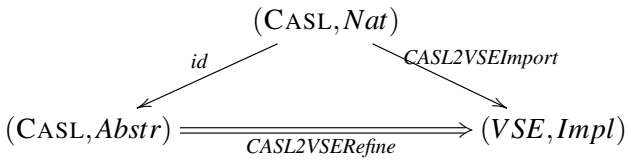


Figure 4: Common import.

For example, let us consider the situation in Fig. 4, where the natural numbers are imported both in the abstract and the concrete specification and the (heterogeneous) refinement link is represented by the double arrow. The label *CASL2VSEImport* on the right arrow indicates that *Nat* is translated via the import comorphism before being imported in *Impl*. We can assume for simplicity that *Nat* has only a sort *nat* and then in *Impl* we have procedure symbols for identification and restriction on *nat*, together with procedure definitions saying that no element is restricted and the identification procedure is simply equality.

On the other side, when *Abstr* is translated along the refinement comorphism to $CDyn^{=}$, no distinction between the sorts defined in *Abstr* and the imported ones is made, so in the resulting translated theory we will have symbols for the restriction on sort *nat* and for identification. These symbols are then mapped identically by the $CDyn^{=}$-signature morphism that labels the refinement and since in *Impl* no restriction and no identification on *nat* is made, the quotient on *nat* is trivial. For any function/predicate symbols from *Nat* we would get the same behavior: the default implementations provided by *CASL2VSEImport* act only as wrappers for functions/procedures, without changing their values, and therefore the imported specification symbols are kept identically.

7 Example: Implementing natural numbers by binary words

As an example, we present the implementation of natural numbers as lists of binary digits, slightly abridged from [28].[7] The abstract CASL specification, NATS (introduced in Fig. 5),

[7] The complete example can be found at https://svn-agbkb.informatik.uni-bremen.de/Hets-lib/trunk/Refinement/natbin_refine.het.

is the usual specification of natural numbers with 0, successor and addition with the Peano axioms. Notice that the predecessor function is defined to take *zero_n* to *zero_n*; this is because in VSE there are no partial functions. In Fig. 6 [8], we present a fragment of the theory obtained by translating NATS along the comorphism *CASL2VSERefine*: the resulting signature and the translation of the first axiom - the other three translated axioms and the sentences introduced by the comorphism are similar.

spec NATS =
free type
$nats ::= zero\_n \mid succ\_n(nats)$
op $zero\_n : nats$
op $succ\_n : nats \to nats$
op $prdc\_n : nats \to nats$
op $add\_n : nats \times nats \to nats$
vars $m, n : nats$
• $prdc\_n(zero\_n) = zero\_n$
• $prdc\_n(succ\_n(m)) = m$
• $add\_n(m, zero\_n) = m$
• $add\_n(m, succ\_n(n))$
$= succ\_n(add\_n(m, n))$
end

Figure 5: CASL specification of natural numbers.

sort *nats*
PROCEDURES
$gn\_add\_n : IN\ nats, IN\ nats \to nats;$
$gn\_eq\_nats : IN\ nats, IN\ nats \to Boolean;$
$gn\_prdc\_n : IN\ nats \to nats;$
$gn\_restr\_nats : IN\ nats;$
$gn\_succ\_n : IN\ nats \to nats;$
$gn\_zero\_n : \to nats$
$\forall\ gn\_x0 : nats; gn\_x1 : nats; gn\_x2 : nats;$
$gn\_x3 : Boolean$
• $<: gn\_x1 := gn\_zero\_n;$
$gn\_x0 := gn\_prdc\_n(gn\_x1);$
$gn\_x2 := gn\_zero\_n;$
$gn\_x3 := gn\_eq\_nats(gn\_x0, gn\_x2) :>$
$gn\_x3 = (op\ True : Boolean)$
...

Figure 6: Natural numbers translated along the comorphism *CASL2VSERefine*.

The VSE implementation, NATS-IMPL (Fig. 7), provides procedures for the implementation of natural numbers as binary words, which are imported as data part along *CASL2VSEImport* [9] from the CASL specification BIN (here omitted). We illustrate the way the procedures are written with the example of the restriction procedure, *nlz*, which terminates whenever the given argument has no leading zeros. The implementation of the other procedures is similar and therefore omitted. Notice that the equality is in this case simply the equality on binary words.

Fig. 8 presents the view BINARY_ARITH expressing the fact that binary words, restricted to those with non-leading zeros, represent a refinement of natural numbers, where each symbol of NATS is implemented by the corresponding procedure in the symbol mapping of the view.

In Fig. 9, we present some of the proof obligations introduced by the view. The resulting development graph is displayed in Fig. 10, where the double arrows correspond to translations along comorphisms and the red arrow is introduced by the view. Notice that these proof obligations are translations of the sentences of the theory presented in Fig. 6 along the signature morphism induced by the view. The first two sentences that we included here are introduced

[8] HETS uses $<: \alpha :> \phi$ as input syntax for $\langle\alpha\rangle\phi$.
[9] The HETS construction *SP with logic C* translates a specification *SP* along the comorphism *C*.

spec NATS_IMPL =
BIN **with logic** *CASL2VSEImport*
then *PROCEDURES*
hnlz : *IN bin*; *nlz* : *IN bin*; *i_badd* : *IN bin*, *IN bin*, *OUT bin*, *OUT bin*;
i_add : *IN bin*, *IN bin* → *bin*; *i_prdc* : *IN bin* → *bin*;
i_succ : *IN bin* → *bin*; *i_zero* : → *bin*; *eq* : *IN bin*, *IN bin* → *Boolean*
• *DEFPROCS*
PROCEDURE hnlz(*x*)
BEGIN
IF x = *b_zero THEN ABORT*
ELSE IF x = *b_one THEN SKIP ELSE hnlz*(*pop*(*x*)) *FI*
FI
END;
PROCEDURE nlz(*x*)
BEGIN IF x = *b_zero THEN SKIP ELSE hnlz*(*x*) *FI END*
DEFPROCSEND
%% ...

Figure 7: Implementation using lists of binary digits.

view BINARY_ARITH : { NATS **with logic** *CASL2VSERefine* } **to** NATS_IMPL =
nats ↦ *bin*, *gn_restr_nats* ↦ *nlz*, *gn_eq_nats* ↦ *eq*,
gn_zero_n ↦ *i_zero*, *gn_succ_n* ↦ *i_succ*, *gn_add_n* ↦ *i_add*

Figure 8: Natural numbers as binary words.

%% Proof obligations introduced by the view
%% equality procedure terminates on valid inputs
$\forall\ gn\_x, gn\_y : bin \bullet <:nlz(gn\_x):> true \wedge <:nlz(gn\_y):> true$
$\Rightarrow <:gn\_b := eq(gn\_x, gn\_y):> true$
%% procedure implementing addition terminates and gives valid results on valid inputs
$\forall\ gn\_x1, gn\_x2 : bin \bullet <:nlz(gn\_x1):> true \wedge <:nlz(gn\_x2):> true$
$\Rightarrow <:gn\_x := i\_add(gn\_x1, gn\_x2):> <:nlz(gn\_x):> true$
%% translation of : forall m : nats . add_n(m, zero_n) = m
$\forall\ gn\_x0, gn\_x1, gn\_x2, gn\_x3 : bin; gn\_x4 : Boolean;$
$m : bin$
$\bullet <:nlz(m):> true$
$\Rightarrow <:gn\_x1 := m\ ;$
$gn\_x2 := i\_zero;$
$gn\_x0 := i\_add(gn\_x1, gn\_x2);$
$gn\_x3 := m;$
$gn\_x4 := eq(gn\_x0, gn\_x3):>$
$gn\_x4 = (op\ True : Boolean)$

Figure 9: Generated proof obligations

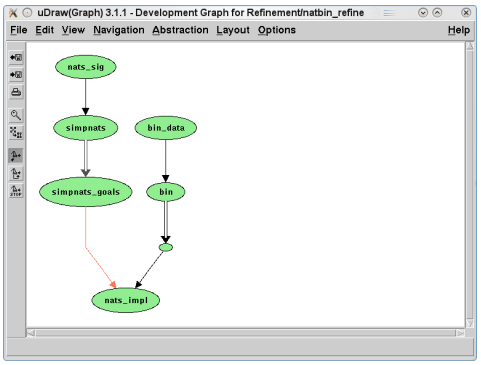


Figure 10: The development graph of natural numbers example

by the signature translation of the comorphism and state that (1) equality terminates on inputs for which the restriction formula *nlz* holds and (2) the procedure implementing addition, *i_add*, terminates for valid inputs and the result is again valid. Also the translation of an axiom of NATS along the comorphism *CASL2VSERefine* is presented.

The proof obligations together with all axioms can be handed over to the VSE prover; the user is presented with the interactive prover interface, where the current proof state can be inspected and proofs of obligations or lemmas can be started. Fig. 11 shows the situation after starting the proof for the obligation resulting from the axiom describing zero as the neutral element with respect to addition. There is a window containing the current goal and another window with a list of applicable rules to choose from.

The prover uses a sequent calculus. Therefore the goals have the form of sequents $e_1, e_2, \ldots, e_n \vdash e'_1, e'_2, \ldots, e'_m$ meaning that under the assumption of $e_1, e_2, \ldots, e_n$ one of the formulas e'_1, e'_2, ..., or e'_m holds.

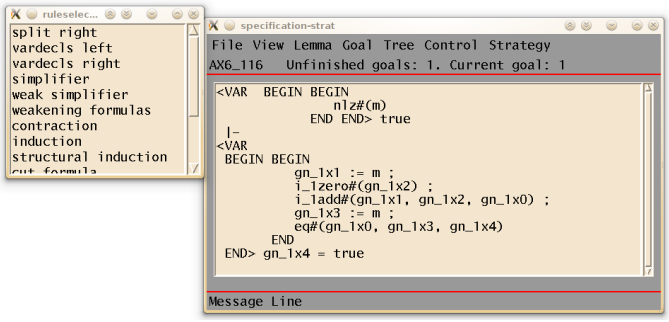


Figure 11: A sample proof goal in VSE

The user could now complete the proof by selecting rules by hand. For this kind of dynamic logic goals rules for each program construct are available. For example for a conditional **if** ε **then** α **else** β **fi** we have the rule

$$\frac{\Gamma, \varepsilon \vdash \langle\alpha\rangle e, \Delta \quad \Gamma, \neg\varepsilon \vdash \langle\beta\rangle e, \Delta}{\Gamma \vdash \langle \textbf{if}\ \varepsilon\ \textbf{then}\ \alpha\ \textbf{else}\ \beta\ \textbf{fi}\rangle e, \Delta}$$

where Γ and Δ are sequences of formulas. Applying this rule would generate two new goals, one assuming the condition ε holds which allows us to replace the conditional with α, and the other one assuming $\neg\varepsilon$.

The rule for an assignment statement $x := \tau$ will change the sequent in a way that it reflects the state after the assignment. It will remove all formulas where x occurs freely and add the equation $x = \tau$. In the following rule Γ' resp. Δ' are obtained from Γ resp. Δ by removing all formulas with free occurrences of the variable x:

$$\frac{\Gamma', x = \tau \vdash e, \Delta'}{\Gamma \vdash \langle x := \tau\rangle e, \Delta}$$

A simplifier is run after each rule application. When appropriate, it will apply a substitution $x = \tau$ on e and remove the equation $x = \tau$. For example, starting from the goal in Fig. 11, the user would soon want to get rid of the assignment $gn\_1x1 := m$, which results in the following new goal:

```
<nlz#(m)> true
 |-
<i_1zero#(gn_1x2)>
    <i_1add#(m, gn_1x2, gn_1x0)>
       <gn_1x3 := m> <eq#(gn_1x0, gn_1x3, gn_1x4)> gn_1x4 = true
```

Next the call of procedure *i_1zero* has to be dealt with. There is a rule which allows to unwind procedure calls. In this case it will yield the following new goal:

```
<nlz#(m)> true
 |-
<VAR  BEGIN BEGIN
               res-i_1zero := b_1zero
            END END>
    <i_1add#(m, res-i_1zero, gn_1x0)>
       <gn_1x3 := m> <eq#(gn_1x0, gn_1x3, gn_1x4)> gn_1x4 = true
```

The proof then would continue always choosing rules for the first top level program construct. As this is boring, the user will rather activate heuristics for that. Then most of the remaining proof is done automatically. As these heuristics are mainly driven by a program appearing in one of the formulas and the result looks like executing the program with symbolic terms instead of values, it is called *symbolic execution*.

In general VSE performs a heuristic loop, which means that after each rule application it tries to apply heuristics from a given list of heuristics the user has chosen. In case all heuristics should fail, there is also a last resort heuristic which allows the user to select a rule from the set of all applicable rules. Finally, a proof tree as shown in Fig. 12 results, where each goal is shown as a node and each rule application lets the tree grow upwards.

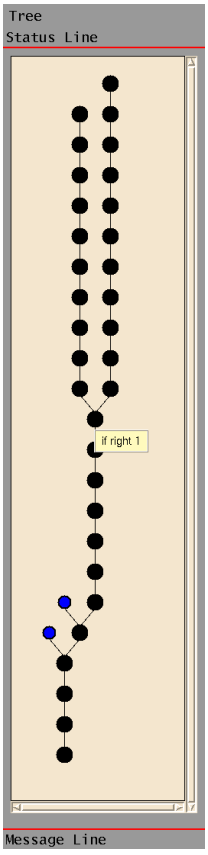


Figure 12: Proof tree in VSE

A more involved example is the proof obligation that will show that the procedure *i_add*, if applied to well-formed input arguments, terminates and produces a well-formed result (in the sense of the restriction procedure):

```
<nlz#(gn_1x1)> true, <nlz#(gn_1x2)> true
 |-
< i_1add#(gn_1x1, gn_1x2, gn_1x)>
  < nlz#(gn_1x)> true
```

Many proof steps still can be done by symbolic execution. However, as *i_add* is recursive this could fail to complete the proof and lead to an infinite loop instead. To prevent this, an induction proof is required, in this case structural induction on the first input argument *i_add*. The induction hypothesis can then be used for recursive calls of *i_add*. The proof should also be made more concise by avoiding to unwind the *i_succ* calls occurring in *i_add*. Instead these calls should be handled by using the similar proof obligation

```
<nlz#(gn_1x1)> true
 |-
< i_1succ#(gn_1x1, gn_1x)> < nlz#(gn_1x)> true
```

as a lemma.

Most of the proof obligations for this example can be treated similar to the two obligations we have discussed. After finishing the proof with the VSE prover, HETS is informed about the obligations that have been completed.

The two comorphisms have been implemented and are part of the latest HETS release; the VSE tool is also going to become available under public license. Provided VSE is installed, the example can be fully checked in HETS.

8 Conclusions and future work

We have integrated VSE's mechanism of refining abstract specifications into procedural implementations into HETS. Via a new logic and two logic translations, one of them doing the usual restriction-quotient construction, we could avoid entirely the introduction of new types of "refinement links" into HETS, but rather could re-use the present machinery of heterogeneous development graphs and thus demonstrate its flexibility. Visually spoken, we could avoid extending the HETS motherboard and expansion slot specification, but rather just construct several expansion cards related to VSE and plug them into the HETS motherboard.

However, there is a point when it actually makes sense to enhance the expansion slot specification. Currently, it is based on the assumption that expansion cards (i.e. theorem provers) can only handle flat unstructured theories. However, VSE can also handle structured theories, and takes advantage of the structuring during proof construction. Hence, we plan to extend the expansion slot specification in a way that allows the transmission (between HETS and VSE) of whole acyclic directed development graphs of theories with connecting definition links, reflecting the import hierarchy. We expect to use this enhancement of the expansion slot specification also for other theorem provers supporting structured theories, like Isabelle.

Another direction of future work will try to exploit synergy effects between VSE and HETS e.g. by using automatic provers like SPASS (which are now available through the integration) during some sample VSE refinement proofs. The refinement method could also be extended from first-order logic to the richer language CASL, which also features subsorting and partial functions.

Acknowledgments

Work on this paper has been supported by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe) and the German Research Council (DFG) under grant MO-971/2-1. We thank Werner Stephan for conceptual discussions and Erwin R. Catesbeiana for pointing out a class of specifications particularly easily usable as targets of refinements.

Bibliography

[1] Spezifikationssprache VSE-SL, 1997. Part of the VSE documentation.

[2] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Wiley, New York, 1990.

[3] E. Astesiano, M. Bidoit, B. Krieg-Brückner, H. Kirchner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL - the common algebraic specification language. *Theoretical Computer Science*, 286:153–196, 2002.

[4] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: Formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer, Special issue on Mechanized Theorem Proving for Technology*, 3(1), september 2000.

[5] Michel Bidoit and Peter D. Mosses. CASL *User Manual*, volume 2900 of *LNCS (IFIP Series)*. Springer, 2004.

[6] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Observational interpretation of CASL specifications. *Math. Struct. in Comp. Sci.*, 18(2):325–371, 2008.

[7] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter (Eds.). *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA, 2006.

[8] R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, LNCS 86, pages 292–332. Springer, 1980.

[9] Lassaad Cheikhrouhou, Georg Rock, Werner Stephan, Matthias Schwan, and Gunter Lassmann. Verifying a chipcard-based biometric identification protocol in VSE. In Janusz Górski, editor, *SAFECOMP 2006*, volume 4166 of *LNCS*, pages 42–56. Springer, 2006.

[10] Mihai Codescu, Bruno Langenstein, Christian Maeder, and Till Mossakowski. The VSE refinement method in HETS. In K. Breitman and A. Cavalcanti, editors, *ICFEM 2009*, volume 5885 of *Lecture Notes in Computer Science*, pages 660–678. Springer, 2009.

[11] Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, and Thomas F. Melham. The prosper toolkit. *STTT*, 4(2):189–210, 2003.

[12] R. Diaconescu. *Institution-independent Model Theory*. Studies in Universal Logic. Birkhäuser, 2008.

[13] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[14] Joseph Goguen and Grigore Roşu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[15] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[16] Dieter Hutter. Management of change in structured verification. In *ASE*, pages 23–, 2000.

[17] Dieter Hutter, Bruno Langenstein, Georg Rock, Jörg Siekmann, Werner Stephan, and Roland Vogt. Formal software development in the verification support environment. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(4):383–406, December 2000.

[18] Bruno Langenstein, Roland Vogt, and Markus Ullmann. The use of formal methods for trusted digital signature devices. In James N. Etheredge and Bill Z. Manaris, editors, *FLAIRS Conference*, pages 336–340. AAAI Press, 2000.

[19] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.

[20] Dominique Méry and Stephan Merz, editors. *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, volume 6396 of *Lecture Notes in Computer Science*. Springer, 2010.

[21] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.

[22] T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation thesis, Universität Bremen, 2005.

[23] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.

[24] T. Mossakowski, Ch. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *TACAS 2007*, LNCS 4424, pages 519–522. Springer, 2007.

[25] T. Mossakowski, D. Sannella, and A.Tarlecki. A simple refinement language for CASL. In Jose Luiz Fiadeiro, editor, *WADT 2004*, volume 3423 of *Lecture Notes in Computer Science*, pages 162–185. Springer Verlag, 2005.

[26] Peter D. Mosses (Editor). CASL *Reference Manual*, volume 2960 of *LNCS (IFIP Series)*. Springer, 2004.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[28] Wolfgang Reif. Verification of large software systems. In R. K. Shyamasundar, editor, *FSTTCS*, volume 652 of *LNCS*, pages 241–252. Springer, 1992.

[29] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2012.

[30] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, LNCS 2392, pages 275–279. Springer-Verlag, 2002.

A Complete specification of natural numbers example

spec NATS_SIG =
  sort *nats*
  op *zero_n* : *nats*
  op *succ_n* : *nats* → *nats*
  op *prdc_n* : *nats* → *nats*
  op *add_n* : *nats* × *nats* → *nats*
end

spec SIMPNATS =
  NATS_SIG
then free type *nats* ::= *zero_n* | *succ_n*(*nats*)
  vars *m*, *n* : *nats*
  • *prdc_n*(*zero_n*) = *zero_n*
  • *prdc_n*(*succ_n*(*m*)) = *m*
  • *add_n*(*m*, *zero_n*) = *m*
  • *add_n*(*m*, *succ_n*(*n*)) = *succ_n*(*add_n*(*m*, *n*))
end

logic VSE

spec SIMPNATS_GOALS =
  SIMPNATS **with logic** → *CASL2VSERefine*
end
logic CASL

spec BIN_DATA =
  free type *bin* ::= *b_zero* | *b_one* | *s0*(*bin*) | *s1*(*bin*)
  op *pop* : *bin* → *bin*
  var *x* : *bin*
  • *pop*(*s0*(*x*)) = *x*
  • *pop*(*s1*(*x*)) = *x*
end

spec BIN =
  BIN_DATA
then op *top* : *bin* → *bin*
  vars *x*, *y*, *z* : *bin*
  • *top*(*b_zero*) = *b_zero*
  • *top*(*b_one*) = *b_one*
  • *top*(*s0*(*x*)) = *b_zero*
  • *top*(*s1*(*x*)) = *b_one*
end

logic VSE

spec NATS_IMPL =
BIN **with logic**→ *CASL2VSEImport*
then *PROCEDURES*
hnlz : *IN bin*;
nlz : *IN bin*;
i_badd : *IN bin*, *IN bin*, *OUT bin*, *OUT bin*;
i_add : *IN bin*, *IN bin* → *bin*;
i_prdc : *IN bin* → *bin*;
i_succ : *IN bin* → *bin*;
i_zero : → *bin*;
eq : *IN bin*, *IN bin* → *Boolean*
• *DEFPROCS*
PROCEDURE hnlz(*x*)
BEGIN
IF x = *b_zero*
THEN ABORT
ELSE IF x = *b_one THEN SKIP ELSE hnlz*(*pop*(*x*)) *FI*
FI
END;

PROCEDURE nlz(*x*)
BEGIN IF x = *b_zero THEN SKIP ELSE hnlz*(*x*) *FI END*
DEFPROCSEND

%(restr)%

• *DEFPROCS*
PROCEDURE i_badd(*a*, *b*, *z*, *c*)
BEGIN
IF a = *b_zero*
THEN c := *b_zero*;
z := *b*
ELSE c := *b*;
IF b = *b_one*
THEN z := *b_zero*
ELSE z := *b_one*
FI
FI
END;

FUNCTION i_add(*x*, *y*)
BEGIN
DECLARE

```
z : bin := b_zero, c : bin := b_zero, s : bin := b_zero;
IF x = b_zero
THEN s := y
ELSE IF y = b_zero
        THEN s := x
        ELSE IF x = b_one
                THEN s := i_succ(y)
                ELSE IF y = b_one
                        THEN s := i_succ(x)
                        ELSE i_badd(top(x), top(y), z, c);
                                IF c = b_one
                                THEN s := i_add(pop(x), pop(y))
                                ELSE s := i_succ(pop(x));
                                        s := i_add(s, pop(y))
                                FI;
                                IF z = b_zero
                                THEN s := s0(s)
                                ELSE s := s1(s)
                                FI
                        FI
                FI
        FI
FI;
RETURN s
END;

FUNCTION i_prdc(x)
BEGIN
DECLARE
y : bin := b_zero;
IF x = b_zero ∨ x = b_one
THEN y := b_zero
ELSE IF x = s0(b_one)
        THEN y := b_one
        ELSE IF top(x) = b_one
                THEN y := s0(pop(x))
                ELSE y := i_prdc(pop(x));
                        y := s1(y)
                FI
        FI
FI;
RETURN y
END;
```

```
FUNCTION i_succ(x)
BEGIN
DECLARE
y : bin := b_one;
IF x = b_zero
THEN y := b_one
ELSE IF x = b_one
        THEN y := s0(b_one)
        ELSE IF top(x) = b_zero
                THEN y := s1(pop(x))
                ELSE y := i_succ(pop(x));
                     y := s0(y)
                FI
        FI
FI;
RETURN y
END;

FUNCTION i_zero()
BEGIN RETURN b_zero END
DEFPROCSEND
```

%(impl)%

- ```
  DEFPROCS
  FUNCTION eq(x, y)
  BEGIN
  DECLARE
  res : Boolean := False;
  IF x = y THEN res := True FI;
  RETURN res
  END
  DEFPROCSEND
  ```

%(congruence)%

end

logic VSE

view REFINE :
SIMPNATS_GOALS **to** NATS_IMPL =
$nats \mapsto bin$, $gn\_restr\_nats \mapsto nlz$, $gn\_eq\_nats \mapsto eq$,
$gn\_zero\_n \mapsto i\_zero$, $gn\_succ\_n \mapsto i\_succ$,
$gn\_prdc\_n \mapsto i\_prdc$, $gn\_add\_n \mapsto i\_add$
end
  ```