

Software Specification and Development in Heterogeneous Environments*

Andrzej Tarlecki[†]

1 Introduction

Nowadays, software tends to be enormously complex. This truth has a number of consequences for the methodology and practice of software development, and should be adequately reflected by the theories that capture and support program development process. In a typical program a number of facets coexist, giving rise to a number of its possible and useful views. Specification of a complex program necessarily involves presentation, often separate presentation, of many aspects of its behaviour; practically useful methodologies must support this. A successful example here is UML, where a good dozen of various kinds of diagrams are used. Diagrams of each kind present a different program view, and in fact (leaving aside any doubts about the formal underpinnings) offer a formalism to deal with one particular kind of program properties. None of these individual views captures all the aspects of the program in question, and only considering them together may lead to an adequate overall view.

Abstracting away from many details: to adequately describe a program we need to use a number of logical formalisms, each targeted at a special kind of program properties. Indeed, the evident proliferation of logical systems used in computer science in general, and in software specification and development in particular, is not just theoreticians' fancy, but a necessary consequence of the practical needs to capture various aspects of software, as well as dealing with various programming languages and paradigms.

Consequently, what we are after is not a single “best” framework to cater for all needs and capture all possible properties and kinds of software, but a truly *heterogeneous environment*, where a number of formalisms may coexist and complement each other. In such an environment, it should be possible to vary formalisms in use to deal with various aspects of software, meaningfully interpret resulting specifications based on a number of logical formalisms, use various logical (and programming) means to deal with different components of

*This work has been partially supported by KBN grant 7T11C 002 21 and European AGILE project IST-2001-32747.

[†]Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

a single program, and switch between logical formalisms when progressing with program specification and development. Such an environment should also be *open*, meaning that a well-prescribed amount of work should be needed to add a new logical formalism to it and make it available for its users. Last but not least, it should come with a good support system, facilitating the use of each individual formalism incorporated in the environment, as well as the possibility to switch between these formalisms in the course of building specifications and then developing programs.

To achieve such an ambitious and far-fetching vision a good deal of foundational work is necessary, to precisely explicate the concepts and ideas involved, and to identify the potential problems and provide a solid basis to resolve them satisfactorily.

The first problem is a formalisation of the very concept of logical system as used here. A number of approaches are possible and provide potentially useful answers, including the rather popular in the theorem-proving community views of logics as captured in so-called *general logical frameworks* developed within appropriately rich type theories (with EDINBURGH LF being perhaps the prime example [15]). We follow here a more semantic, model-theoretic view, very much in the spirit of abstract model theory [1] and the tradition of algebraic specification [11], where the notion of a logical system has been usefully formalised as an *institution* [13, 28]. The framework of institutions provides a solid basis here for sketching some key issues related to the development of a heterogeneous environment.

Some standard concepts of the theory of institutions are recalled in Sect. 2, covering the definition of an institution as well as notions that capture various ways of relating one institution with another [27, 14]. These are used in Sect. 3 to present structured heterogeneous specifications, built over a number of institutions, following [29]. Section 4 then shows how an abstract view of software development [24, 30] may incorporate the possibility of (and need for) switching from one institution to another, which leads to the discussion of *heterogeneous design*, where various components of the same program might be specified and developed using different formalisms, thus resulting in a heterogeneous version of *architectural specification* [3]. Section 5 offers some concluding remarks. Necessarily, the presentation here is very sketchy, trying to concentrate more on the problems and ideas than on precise answers, with statements of any technical results (and explicit examples!) being sacrificed for the sake of brevity — a more detailed and complete presentation will be given in a forthcoming full version of this paper.

At least two projects have been successfully undertaken in a similar vein of exploiting the theory of institutions to explicate and support practical use of multiple logical systems in software specification. Perhaps the first such project leading to a practical system was CAFEOBJ based on a cube of logics formalised as institutions with institution morphisms between them [10]. The other one arose within the COFI working group, with activities centered around CASL, an up-to-date algebraic specification formalism [2, 8]. This work used the concepts of institution theory to structure and facilitate the design and formal descrip-

tion of CASL, to compare it with other specification formalisms, and to design its extensions [18, 8]. One of the recent CASL extensions is to provide a formalism to build heterogeneous specifications, with tool support offered by HETS, Heterogeneous Tool Set, and formal underpinnings developed in [17, 19, 20].

2 Institutional preliminaries

An *institution* [13] consists of a category **Sign** of *signatures*, functors **Sen**: **Sign** \rightarrow **Set** and **Mod**: **Sign**^{op} \rightarrow **Cat**, and for each $\Sigma \in |\mathbf{Sign}|$, a Σ -*satisfaction relation* $\models_{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$. For $\Sigma \in |\mathbf{Sign}|$, **Sen**(Σ) is the set of Σ -*sentences*, and **Mod**(Σ) is the category of Σ -*models* and their morphisms. For any *signature morphism* $\sigma: \Sigma \rightarrow \Sigma'$ in **Sign**, **Sen**(σ), written as $\sigma: \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma')$, is a translation of sentences, and **Mod**(σ), written as $_|\sigma: \mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ is a *reduct functor*. These are subject to the following *satisfaction condition*:

$$M'|\sigma \models_{\Sigma} \varphi \iff M' \models_{\Sigma'} \sigma(\varphi)$$

where $\sigma: \Sigma \rightarrow \Sigma'$ in **Sign**, $M' \in |\mathbf{Mod}(\Sigma')|$, $\varphi \in \mathbf{Sen}(\Sigma)$.

The requirements this definition imposes on a logical system are very mild, and examples of logical systems presented as institutions abound, including many typical logics such as various versions of equational and first-order logics (perhaps with partial operations, predicate symbols, error elements, subsorting, additional forms of higher-order constraints, etc). Just to hint that much more is covered, let us mention that modal logics, many-valued logics, and even programming language semantics may be formalized as institutions.

For any signature $\Sigma \in |\mathbf{Sign}|$, the satisfaction relation between Σ -models and Σ -sentences allows for reuse of standard logical concepts and results in an arbitrary institution. In particular, the definitions of the class of *models*¹ $Mod[\Phi] \subseteq |\mathbf{Mod}(\Sigma)|$ of a set of sentences $\Phi \subseteq \mathbf{Sen}(\Sigma)$, and of *semantic consequence* $\Phi \models_{\Sigma} \varphi$, for $\Phi \subseteq \mathbf{Sen}(\Sigma)$ and $\varphi \in \mathbf{Sen}(\Sigma)$, carry over without change.

In applications the institutions may be subject to further restrictions. For instance, in work on various forms of “putting together” signatures, theories and specifications [6], the category of signatures is assumed to be cocomplete, or at least to have pushouts, and things work smoothly if the amalgamation property holds over colimits of signatures (equivalently: the model functor is continuous, and so compatible model families over signature diagrams can be amalgamated to a model over the diagram colimit). Standard meta-logical properties may be abstractly formulated for an arbitrary institution, with various forms of interpolation over signature pushouts [26] being one important example. Moreover, one often assumes that the institutions of interest come with some additional structure; for instance, an additional “proof-theoretic” counterpart of the consequence relations may be required, captured by entailment relations $\vdash_{\Sigma} \subseteq \wp(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$, for $\Sigma \in |\mathbf{Sign}|$, which leads to the concept of a *general logic* [16] when each \vdash_{Σ} is sound for the semantic consequence \models_{Σ} .

¹We apologize for the traditional overloading of the term “model” here.

Given the formalisation of the concept of a logical system as an institution, much interesting work has been done to free the theory of algebraic specification and software development as originally developed for equational logic [11] from the limitations of using the equational logic only, leading to an abstract specification theory in an arbitrary but fixed institution, [24, 30].

We undertake now the next step: dealing with a number of institutions at the same time. Naturally, to make any sense of this, the institutions involved must be somehow linked with each other. A number of concepts of a formal mapping between institutions have been proposed, starting with the original *institution morphisms* in [13]. There is an obvious “taxonomy” for such mappings, based on the relative direction of translation of the three institution components (signatures, models, sentences), as perhaps first pointed out in [27] and then exploited to systematize the field and put forward key dualities in [14], see also [19]. Two such concepts seem of the crucial importance for our purposes.

Consider two institutions $\mathbf{I} = \langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_{\Sigma} \rangle_{\Sigma \in |\mathbf{Sign}|} \rangle$ and $\mathbf{I}' = \langle \mathbf{Sign}', \mathbf{Sen}', \mathbf{Mod}', \langle \models_{\Sigma'} \rangle_{\Sigma' \in |\mathbf{Sign}'|} \rangle$. An *institution morphism* $\mu: \mathbf{I}' \rightarrow \mathbf{I}$ consists of three components: a functor $\mu^{Sign}: \mathbf{Sign}' \rightarrow \mathbf{Sign}$ and two natural transformations, $\mu^{Mod}: \mathbf{Mod}' \rightarrow (\mu^{Sign})^{op}; \mathbf{Mod}$ and $\mu^{Sen}: \mu^{Sign}; \mathbf{Sen} \rightarrow \mathbf{Sen}'$, subject to the *satisfaction condition*:

$$M' \models'_{\Sigma'} \mu_{\Sigma'}^{Sen}(\varphi) \iff \mu_{\Sigma'}^{Mod}(M') \models_{\mu^{Sign}(\Sigma')} \varphi$$

where $\Sigma' \in |\mathbf{Sign}'|$, $M' \in |\mathbf{Mod}'(\Sigma')|$, $\varphi \in \mathbf{Sen}(\mu^{Sign}(\Sigma'))$. Disregarding the sentence component (and so the satisfaction condition as well) yields an institution *semi-morphism*.

An *institution comorphism* $\rho: \mathbf{I} \rightarrow \mathbf{I}'$ consists of three components: a functor $\mu^{Sign}: \mathbf{Sign} \rightarrow \mathbf{Sign}'$ and two natural transformations, $\mu^{Mod}: (\mu^{Sign})^{op}; \mathbf{Mod}' \rightarrow \mathbf{Mod}$ and $\rho^{Sen}: \mathbf{Sen} \rightarrow \mu^{Sign}; \mathbf{Sen}'$, subject to the *satisfaction condition*:

$$M' \models'_{\rho^{Sign}(\Sigma)} \rho_{\Sigma}^{Sen}(\varphi) \iff \mu_{\Sigma}^{Mod}(M') \models_{\Sigma} \varphi$$

where $\Sigma \in |\mathbf{Sign}|$, $M' \in |\mathbf{Mod}'(\rho^{Sign}(\Sigma))|$, $\varphi \in \mathbf{Sen}(\Sigma)$. Disregarding the sentence component (and so the satisfaction condition as well) yields an institution *semi-comorphism*.

The rough and highly informal idea is that an institution morphism as above captures a way in which “richer” institution \mathbf{I}' is built over more “poor” institution \mathbf{I} , by “extracting” simpler \mathbf{I} -signatures and models out of more complex \mathbf{I}' -signatures and models, and encoding \mathbf{I} -sentences as \mathbf{I}' -sentences. In turn, an institution comorphism as above captures a way in which more “poor” institution \mathbf{I} is represented within “richer” institution \mathbf{I}' , by mapping simpler \mathbf{I} -signatures to some \mathbf{I}' -signatures so that \mathbf{I} -models can be extracted out of \mathbf{I}' -models and \mathbf{I} -sentences may be represented as \mathbf{I}' -sentences.

In the following we will often omit super- and sub-scripts identifying the exact component of an institution (co)morphism in use. Thus, for instance, if $\Sigma' \in |\mathbf{Sign}'|$ then $\mu(\Sigma')$ stands for $\mu^{Sign}(\Sigma')$ and if $\mathcal{M}' \subseteq |\mathbf{Mod}'(\Sigma')|$ then $\mu(\mathcal{M}')$ stands for the image of \mathcal{M}' under $\mu_{\Sigma'}^{Mod}$.

3 Structured heterogeneous specifications

The use of an institution \mathbf{I} in software specification and development is based on the overall idea that its signatures determine the syntax of the programs to be specified, its models represent (the semantics of) the programs, and its sentences are used to specify their properties. Following this, we adopt a model-theoretic view of specifications, as put forward in [22]. The meaning of any specification SP built over \mathbf{I} is given by its *signature* $Sig[SP] \in |\mathbf{Sign}|$ and a class of its *models* $Mod[SP] \subseteq |\mathbf{Mod}(Sig[SP])|$.

One crucial insight going back to [6, 7] is that specifications must be built in some well-structured fashion; another one was that specific features of the logic at hand must not clutter the presentation and understanding of specification structure. Both points were further put forward in [22] by insisting that specifications are built from the simplest lists of axioms using some well-understood *specification-building operations* designed independently of the institution one works with. Following this, we consider a few very basic means to build specifications in an arbitrary institution:

Basic specification: indicates a signature and lists axioms to capture the desirable properties. For any $\Sigma \in |\mathbf{Sign}|$ and $\Phi \subseteq \mathbf{Sen}(\Sigma)$, $\langle \Sigma, \Phi \rangle$ is a specification with $Sig[\langle \Sigma, \Phi \rangle] = \Sigma$ and $Mod[\langle \Sigma, \Phi \rangle] = Mod[\Phi]$.

Union: combines constraints imposed by various specifications. For any SP_1 and SP_2 with $Sig[SP_1] = Sig[SP_2]$, $SP_1 \cup SP_2$ is a specification with $Sig[SP_1 \cup SP_2] = Sig[SP_1]$ and $Mod[SP_1 \cup SP_2] = Mod[SP_1] \cap Mod[SP_2]$.

Translation: renames and introduces new components, following a signature morphism. For any SP and $\sigma: Sig[SP] \rightarrow \Sigma'$, $\sigma(SP)$ is a specification with $Sig[\sigma(SP)] = \Sigma'$ and $Mod[\sigma(SP)] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid M'|_\sigma \in Mod[SP]\}$.

Hiding: hides auxiliary components, receding to the source of a signature morphism. For any SP' and $\sigma: \Sigma \rightarrow Sig[SP']$, $SP'|_\sigma$ is a specification with $Sig[SP'|_\sigma] = \Sigma$ and $Mod[SP'|_\sigma] = \{M'|_\sigma \mid M' \in Mod[SP']\}$.

The above definition exploits the structure of specifications to determine their semantics in a *compositional* manner. Compositionality is the key to effective use of large structured specifications; for instance, the structure of specifications may be used to guide a proof search for their semantic consequences defined as expected: $SP \models \varphi$, for a specification SP and $\varphi \in \mathbf{Sen}(Sig[SP])$, if φ holds in all models of SP . The following rules form a compositional proof system for structured specifications built as above in an institution with a proof-theoretic entailment $\langle \vdash_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|}$.

$$\frac{\varphi \in \Phi}{\langle \Sigma, \Phi \rangle \vdash \varphi} \quad \frac{SP_1 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi} \quad \frac{SP_2 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi}$$

$$\frac{SP \vdash \varphi}{\sigma(SP) \vdash \sigma(\varphi)} \quad \frac{SP' \vdash \sigma(\varphi)}{SP'|_\sigma \vdash \varphi} \quad \frac{\text{for } i \in \mathcal{I}, SP \vdash \varphi_i \quad \{\varphi_i\}_{i \in \mathcal{I}} \vdash \varphi}{SP \vdash \varphi}$$

Moreover, under reasonable assumptions about the underlying institution (we need pushouts of signatures, amalgamation and — most likely to raise problems — interpolation), the above rules extend any sound and complete entailments to a sound and complete proof system for consequences of structured specifications considered, see [5].

Heterogeneity of specifications may be achieved by introducing new specification building operations to move specifications from one institution to another. Of course, the institutions involved cannot be totally unrelated; at least their signatures and models must be linked (as these are the ingredients involved in the specification semantics). We will retain the property that each specification ultimately “resides” in a specific institution, with its semantics given in terms of signatures and models of this institution, even though some of its subspecifications may similarly reside in other institutions. We refer to specifications residing in an institution \mathbf{I} in this sense as \mathbf{I} -specifications.

Institution semi-morphisms and semi-comorphisms offer a number of possible ways of moving specifications between institutions. It seems, however, that the following two *inter-institutional* constructs are most natural and useful:

Translation: introduces new structure to specification models, following an institution semi-comorphism. For any institution semi-comorphism $\rho: \mathbf{I} \rightarrow \mathbf{I}'$ and \mathbf{I} -specification SP , $\rho(SP)$ is an \mathbf{I}' -specification with $Sig[\rho(SP)] = \rho(Sig[SP])$ and $Mod[\rho(SP)] = \{M' \in |\mathbf{Mod}'(\rho(Sig[SP]))| \mid \rho(M') \in Mod[SP]\}$.

Hiding: hides extra structure of specification models, following an institution semi-morphism. For any institution semi-morphism $\mu: \mathbf{I}' \rightarrow \mathbf{I}$ and \mathbf{I}' -specification SP' , $SP'|_{\mu}$ is an \mathbf{I} -specification with $Sig[SP'|_{\mu}] = \mu(Sig[SP'])$ and $Mod[SP'|_{\mu}] = \{\mu(M') \mid M' \in Mod[SP']\}$.

We deliberately reuse here the terminology for the corresponding specification-building operations within an institution, as indeed, the intuition behind inter- and intra-institutional versions of translation and hiding is quite similar.

Translation of an \mathbf{I} -specification receding to an indicated signature in the source of institution semi-morphism $\mu: \mathbf{I}' \rightarrow \mathbf{I}$, and hiding for an \mathbf{I}' -specification receding to an indicated signature in the source of institution semi-comorphism $\rho: \mathbf{I} \rightarrow \mathbf{I}'$ may be defined similarly — we omit these here though.

Given the new inter-institutional specification-building operations, one can construct structured specifications that span a diagram of institutions linked by institution semi-morphisms and semi-comorphisms. Various parts of such a specification may be build in other institutions involved, and thus capture properties that may not be even expressible in the “surface” institution. Moreover, as before, the operations indicate how to understand and work with such structured heterogeneous specifications in a compositional way.

For instance, consider the problem of extending the above compositional proof system for structured specifications to heterogeneous specifications. Evidently, there is little one can do if the sentences of the institutions the specifications involve are not related. So, for this purpose we assume that the

semi-morphisms and semi-comorphisms used to move specifications between institutions extend smoothly to sentences as well, that is, are in fact institution morphisms and comorphisms, respectively. Then the following two natural rules soundly extend the original system:

$$\frac{SP \vdash^{\mathbf{I}} \varphi}{\rho(SP) \vdash \rho(\varphi)} \quad \frac{SP' \vdash^{\mathbf{I}'} \mu(\varphi)}{SP'|_{\mu} \vdash \varphi}$$

There is another, less direct way to introduce heterogeneous specifications. Namely, given a diagram of institutions linked by (semi-)morphisms, one can construct a single institution which incorporates all the institution in the diagram and captures links between them by signature morphisms. This is given essentially by a (more elaborate version of) the Grothendieck construction to flattened indexed categories, see [9]. Moreover, similar construction may be applied to diagrams of institutions linked by institution (semi-)comorphisms [17], and further generalised to institution diagrams where links of both kinds may be used [19]. Now, any structured specification built over the resulting Grothendieck institution using the usual intra-institutional specification-building operations may be viewed as a possibly heterogeneous specification built over the considered diagram of institutions. Translation and hiding w.r.t. certain morphisms in this Grothendieck institution correspond to inter-institutional translation and hiding as explicitly introduced above. A warning is in order: the Grothendieck construction just puts the institutions involved next to each other, leaving them with their respective models and sentences, and only providing extra links between their signatures with institution morphisms used to define the induced model reducts and sentence translations. This has very little to do with the task of properly combining the institutions involved to form a new, more complex logical system, given in the most rudimentary way as the limit of the diagram in the (complete, see [26]) category of all institutions.

4 Heterogeneous software development

Given an institution \mathbf{I} with models capturing the essential semantic aspects of programs we aim at, an \mathbf{I} -specification SP , whether heterogeneous or not, determines the programming task to produce a program that correctly implements it, that is, a program with the semantics yielding a model of SP . Simplifying a lot, the specified task is to build a model $M \in |\mathbf{Mod}(Sig[SP])|$ such that $M \in Mod[SP]$. The key idea is that this should not be carried out in a single jump; instead, one should proceed step by step, adding gradually more and more detail and incorporating more and more design and implementation decisions.

We adopt here a view proposed in [23], where each individual *refinement step* leading from one specification to another involves an additional component, a *constructor*. Intuitively, constructors correspond to generic modules, like STANDARD ML *functors* [21]; semantically they are functions that map models to models. In the realm of institutions, such constructors may be given as reducts

w.r.t. signature morphisms, or via various forms of definitional extensions. Mixing the two leads to a powerful idea that a constructor may be given as a reduct w.r.t. a *derived* signature morphism [4], where symbols of the source signature may be mapped to terms, perhaps involving recursion, or even simply written in a programming language over the target signature.

With this concept in mind, we say that a specification SP' is a *constructor refinement* of SP via constructor $\kappa: |\mathbf{Mod}(Sig[SP'])| \rightarrow |\mathbf{Mod}(Sig[SP])|$, written $SP \rightsquigarrow_{\kappa} SP'$, whenever $\kappa(Mod[SP']) \subseteq Mod[SP]$. Development process takes the form of a chain of such constructor refinements, with requirements on individual refinement steps ensuring the overall correctness of the result [23, 24].

There is a natural way to allow such a development process to switch from one institution to another. All that is needed are inter-institutional constructors. Under the semantic view of a constructor as a function that maps models to models, any model component of an institution semi-morphism or semi-comorphism can be used. Consequently, for any institution semi-morphism $\mu: \mathbf{I}' \rightarrow \mathbf{I}$, \mathbf{I}' -specification SP' with $Sig[SP'] = \Sigma'$, and \mathbf{I} -specification SP with $Sig[SP] = \mu^{Sign}(\Sigma')$, SP' is a constructor refinement of SP via μ (or, to be more precise, via $\mu_{\Sigma'}^{Mod}$),

$$SP \rightsquigarrow_{\mu} SP'$$

provided that $\mu(Mod[SP']) \subseteq Mod[SP]$. Using such an inter-institutional refinement in a development process amounts to a decision to implement the requirements as captured by the \mathbf{I} -specification SP by first implementing the requirements captured by the \mathbf{I}' -specification SP' and then extracting an implementation of the original requirements from the result. A perhaps most typical case here has already been pointed out in [23]. The institution \mathbf{I} , where the original requirements specification is built, might be a standard institution of some usual algebraic logical system, like the institution of CASL [8] (in fact, SP then might be a CASL specification). Then the institution \mathbf{I}' , with a richer structure of models, might be an institution that captures some functional programming language, like STANDARD ML [21], with models corresponding to STANDARD ML structures, sentences to pieces of STANDARD ML code, and satisfaction relation capturing the semantics of STANDARD ML. A natural institution semi-morphism might then be given, which in essence abstracts away from some details of the semantics of STANDARD ML and views STANDARD ML structures as CASL models (many-sorted partial algebras). Then an inter-institutional refinement captures formally correct implementation of a CASL specification SP by a STANDARD ML specification (a program) SP' .

Note that there is no hope here to extend the institution semi-morphism to an institution morphism: this would require expressing STANDARD ML code using CASL sentences, which cannot be expected in general. The semantic presentation above does not require this: the institution semi-morphism is all that is needed. On the other hand, this does not provide any direct means to verify the correctness of inter-institutional refinement steps, as captured by the requirement $\mu(Mod[SP']) \subseteq Mod[SP]$. This is a separate task (here: of verifying STANDARD ML code w.r.t. CASL axioms and specifications) and separate calculi

and tools need to be provided to carry this out.

In the above formalisation of refinement steps we have entirely disregarded the possible internal structure of requirements specification. This was on purpose: in general there is no link between the structure used to capture requirements and the structure used to implement them. Quite simply, the two serve quite different purposes [12], which is perhaps even more visible in the context of heterogeneous specification and development. Using a number of institutions to build a requirements specification often reflects various views and aspects of the same program, and has nothing to do with identification of program components in any possible implementation.

Consequently, separate tools must be provided to design and capture in the development process the structure of program implementation. In CASL, these come in the form of *architectural specifications* [3], which might look as follows:

$$\begin{aligned} \text{arch spec } ASP = \text{units } & U_1: SP_1 \\ & \dots \\ & U_n: SP_n \\ \text{result } & \kappa(U_1, \dots, U_n) \end{aligned}$$

The architectural specification ASP above capture a branching point in the development process, namely a design decision to build the system by providing separate implementations for specifications SP_1, \dots, SP_n , (naming the resulting modules U_1, \dots, U_n , respectively) and putting them together using a multi-argument constructor κ . In CASL architectural specifications, the result constructor κ is composed of simpler constructors on models of the underlying institution built into the formalism. Perhaps the most important among them (disregarding instantiation of generic units, which we omit here) are reducts w.r.t. a signature morphism and amalgamation of models to the union of their signatures. While reducts are quite straightforward and always safe, amalgamation causes extra problems, since not every two models can be amalgamated [25].

Heterogeneity may be added to architectural specifications simply by extending the repertoire of basic constructors by some inter-institutional constructors, as discussed above. These can be model translations given by institution semi-morphisms, with typical examples sketched above. This opens the possibility for component specifications SP_1, \dots, SP_n to be given in various institutions, and their implementations to be developed using quite different programming paradigms. The resulting models, capturing implementations of individual system modules, can be translated to a common, typically more abstract framework (an institution) and combined there as required.

5 Final remarks

The aim of this note is to present some preliminary thoughts on possible ways of using multiple logical formalisms as well as programming languages in course of program specification and development. What emerges is a formal semantic view of a heterogeneous logical environment as a diagram of institutions linked

by institution (semi-)morphisms and (semi-)comorphisms, which captures both, logical systems used to build requirements specifications and programming languages and paradigms used for implementation as well as their mutual semantic relationships. Given this, we sketch how heterogeneous specifications can be built and how heterogeneous programs may emerge, with various logics and programming formalisms used at subsequent stages of program development and for various components of the program. This semantic view seems perfectly adequate to provide guidelines for the real work yet to come.

The first task is to choose the logical and programming formalisms to be offered, formalise them as institutions and develop (co)morphisms to link some of them. Providing such links is never an easy task: we often look for relationships between some models and concepts developed to capture radically different views and intuitions concerning software. In a way, relating various semantic views of programs is often the most difficult task here.

Once this is given, there are further problems to solve. For instance, when discussing structured heterogeneous specifications, we presented a proof system for showing their consequences, which works in a satisfactory way under some technical assumptions. When those are not satisfied though (for instance, because intra- or inter-institutional interpolation property fails) completeness of the compositional proof system must fail, and some non-compositional techniques may have to be used. Then, lifting this proof system to refinements between specifications requires an additional oracle for conservativity of extensions, perhaps even more troublesome in heterogeneous environments.

Our heterogeneous specifications ultimately reside in a specific institution, with other institutions in use playing in a sense an auxiliary role only. An interesting alternative to consider is to deal with diagrams of specifications distributed over the heterogeneous logical environment, understood as specifying a single program from a number of various perspectives. Issues of consistency and of “emerging features” seem to becoming even more important then.

Finally, the ease with which we treat heterogeneous development process is highly suspicious: we have in fact disregarded the true computational contents of implementations. While extracting an abstract model out of a program coded in a specific programming language is merely the task for the programming language semantics, combining two such models extracted from programs coded in possibly quite different programming languages, and especially resolving mutual dependencies between them in a computationally meaningful way cannot be restricted to semantic manipulation only, and must require some extra programming work to provide an interface between the languages involved.

Acknowledgments The ideas sketched here have roots in joint work with Don Sannella on numerous issues related to abstract specification theory, with Michel Bidoit on architectural specifications, and with Till Mossakowski on various aspects of institution theory, including heterogeneity. Thanks also to Andrzej Gąsienica-Samek for “pączki” and “kodki” which made me wonder about heterogeneous programming from a less semantic perspective.

References

- [1] J. Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic*, 7:221–265, 1974.
- [2] M. Bidoit and P. D. Mosses. *CASL User Manual*. LNCS 2900. Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.
- [3] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Comput.*, 13:252–273, 2002.
- [4] M. Bidoit, D. Sannella, and A. Tarlecki. Global development via local observational construction steps. In *Proc. 27th Intl. Symp. Mathematical Foundations of Computer Science*, LNCS 2420, pages 1–24. Springer, 2002.
- [5] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Comput. Sci.*, 286:197–245, 2002.
- [6] R. Burstall and J. Goguen. Putting theories together to make specifications. In *Proc. 5th Intl. Joint Conference on Artificial Intelligence, Cambridge, Mass. (USA)*, pages 1045–1058, 1977.
- [7] R. Burstall and J. Goguen. The semantics of CLEAR, a specification language. In *Proc. Copenhagen Winter School on Abstract Software Specification*, LNCS 86, pages 292–332. Springer, 1980.
- [8] CoFI (The Common Framework Initiative). *CASL Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.
- [9] R. Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10:383–402, 2002.
- [10] R. Diaconescu and K. Futatsugi. Logical foundations of CAFEOBJ. *Theoretical Comput. Sci.*, 285:289–318, 2002.
- [11] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.
- [12] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. In *Proc. 3rd Intl. Symp. VDM Europe: VDM and Z, Formal Methods in Software Development*, LNCS 428, pages 189–210. Springer, 1990.
- [13] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39:95–146, 1992.
- [14] J. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Comput.*, 13:274–307, 2002.
- [15] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40:143–184, 1993.

- [16] J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
- [17] T. Mossakowski. Comorphism-based Grothendieck logics. In *Proc. 27th Intl. Symp. Mathematical Foundations of Computer Science*, LNCS 2420, pages 593–604. Springer, 2002.
- [18] T. Mossakowski. Relating CASL with other specification languages: The institution level. *Theoretical Comput. Sci.*, 286:367–475, 2002.
- [19] T. Mossakowski. Foundations of heterogeneous specification. In *Recent Trends in Algebraic Development Techniques, 16th Intl. Workshop, WADT 2002, Revised Selected Papers*, LNCS 2755, pages 359–375. Springer, 2003.
- [20] T. Mossakowski. Heterogeneous specifications and the Heterogeneous Tool Set. This volume, 2004.
- [21] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [22] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.
- [23] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.
- [24] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Comput.*, 9:229–269, 1997.
- [25] L. Schröder, T. Mossakowski, A. Tarlecki, B. Klin, and P. Hoffman. Amalgamation in the semantics of CASL. *Theoretical Comput. Sci.* To appear.
- [26] A. Tarlecki. Bits and pieces of the theory of institutions. In *Proc. Intl. Workshop on Category Theory and Computer Programming*, LNCS 240, pages 334–363. Springer, 1986.
- [27] A. Tarlecki. Moving between logical systems. In *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, LNCS 1130, pages 478–502. Springer, 1996.
- [28] A. Tarlecki. Institutions: An abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, pages 105–130. Springer, 1999.
- [29] A. Tarlecki. Towards heterogeneous specifications. In *Frontiers of Combining Systems 2, 1998*, pages 337–360. Research Studies Press, 2000.
- [30] A. Tarlecki. Abstract specification theory: An overview. In *Models, Algebras and Logic of Engineering Software*, pages 43–79. IOS Press, 2003.