Toward Component-Oriented Formal Software Development: An Algebraic Approach*

Michel Bidoit[1], Donald Sannella[2], and Andrzej Tarlecki[3]

[1] Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France
[2] Laboratory for Foundations of Computer Science, University of Edinburgh, UK
[3] Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

Abstract. Component based design and development of software is one of the most challenging issues in software engineering. In this paper, we adopt a somewhat simplified view of software components and discuss how they can be conveniently modelled in a framework that provides a modular approach to formal software development by means of stepwise refinement. In particular we take into account an observational interpretation of requirements specifications and study its impact on the definition of the semantics of specifications of (parametrized) components. Our study is carried out in the context of CASL architectural specifications.

1 Introduction

Nowadays component based design is perceived as a key technology for developing systems in a cost- and time effective manner. However, there is still no clear understanding of what is a component, and in particular of how to provide a formal semantics of components. Similarly, formal software development has received relatively little attention in the context of component based approaches.

We focus here on a rather restrictive view of components, namely *software components* (understood as pieces of code) in contrast with *system components* (understood as self-contained computers with their own hardware and software interacting with each other and the environment by exchanging messages across linking interfaces). However, our view of (software) components is consistent with the best accepted definition in the software industry, see [Szy98]: a (software) component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently.

There has been a great deal of work in the algebraic specification tradition on formalizing the intuitive idea of software development by stepwise refinement, including [EKMP82,GM82,Gan83,Sch87,ST88b,ST89,Sch90]; the general ideas go back at least to [Hoa72]. For a recent survey, see [EK99]. There are many issues that make this a difficult problem, and some of them are rather subtle,

* This work has been partially supported by KBN grant 7T11C 002 21 and European AGILE project IST-2001-32747 (AT), CNRS–PAS Research Cooperation Programme (MB, AT), and British–Polish Research Partnership Programme (DS, AT).

one example being the relationship between specification structure and software structure. An overview that covers most of our own contributions is [ST97], with some more recent work addressing the problem of how to prove correctness of refinement steps [BH98], and the design of a convenient formalism for writing specifications [ABK$^+$02,BST02a].

In this paper we discuss how software components can be modelled in an algebraic framework providing a modular approach to formal software development by means of stepwise refinement. In particular we take into account an observational interpretation of requirements specifications and study its impact on the definition of the semantics of specifications of (parametrized) components. Our study is carried out in the context of CASL architectural specifications. Architectural specifications, for describing the modular structure of software systems, are probably the most novel feature of CASL. We view them here as a means of making complex refinement steps, by defining well-structured constructions to be used to build a software system from implementations of individual components (these also include parametrized components, acting as constructions providing local construction steps to be used in a more global context).

We begin with a brief introduction to CASL basic and structured specifications in Sect. 2. Then we present our basic view of formal software development by means of stepwise refinement in Sect. 3, motivating CASL architectural specifications introduced in Sect. 4. In Sect. 5 we motivate and recall the observational interpretation of specifications, and we study in Sect. 6 the impact of this observational interpretation on the semantics of specifications of parametrized components. An example is given in Sect. 7 to illustrate a few of the main points. Further work is discussed in Sect. 8. The present paper is a high-level overview that concentrates on presenting and justifying the concepts without dwelling on the technicalities, which are presented in [BST02b] and elsewhere.

2 CASL Essentials

A basic assumption underpinning algebraic specification and derived approaches to software specification and development is that programs are modelled as algebras (of some kind) with their "types" captured by algebraic signatures (again, adapted as appropriate). Then specifications include axioms describing their required properties. This leads to quite a flexible framework, which can be tuned as desired to cope with various programming features of interest by selecting the appropriate variation of algebra, signature and axiom. This flexibility has been formalized via the notion of *institution* [GB92] and related work on the theory of specifications and formal program development [ST88a,ST97,BH93].

CASL is an algebraic specification language to describe CASL *models*: many-sorted algebras with subsorts, partial and total operations, and predicates. CASL models are classified by CASL *signatures*, which give *sort* names (with their subsorting relation), partial and total *operation* names, and *predicate* names, together with *profiles* of operations and predicates. In CASL models, subsorts and supersorts are linked by implicit *subsort embeddings* required to compose with each other and to be compatible with operations and predicates with the same names. For each signature Σ, the class of all Σ-models is denoted $Mod(\Sigma)$.

The basic level of CASL includes *declarations* to introduce components of signatures and *axioms* to give properties that characterize *models* of a specification. The logic used to write the axioms is essentially first-order logic (so, with quantification and the usual logical connectives) built over *atomic formulae* which include strong and existential equalities, definedness formulae and predicate applications, with generation constraints added as special, non-first-order sentences. A basic CASL specification SP amounts to a definition of a signature Σ and a set of axioms Φ. It denotes the class $[\![SP]\!] \subseteq Mod(\Sigma)$ of SP-models, which are those Σ-models that *satisfy* all the axioms in Φ: $[\![SP]\!] = \{A \in Mod(\Sigma) \mid A \models \Phi\}$.

Apart from basic specifications as above, CASL provides ways of building complex specifications out of simpler ones by means of various *structuring constructs*. These include translation, hiding, union, and both free and loose forms of extension. *Generic specifications* and their *instantiations* with pushout-style semantics [BG80,EM85] are also provided. Structured specifications built using these constructs can be given a compositional semantics where each specification SP determines a signature $Sig[SP]$ and a class $[\![SP]\!] \subseteq Mod(Sig[SP])$ of models.

3 Program Development and Refinement

The intended use of CASL, as of any such specification formalism, is to specify programs. Each CASL specification should determine a class of programs that correctly realize the specified requirements. To fit this into the formal view of CASL specifications, programs must be written in a programming language having a semantics which assigns[4] to each program its *denotation* as a CASL model. Then each program P determines a signature $Sig[P]$ and a model $[\![P]\!] \in Mod(Sig[P])$. The denotation $[\![SP]\!]$ of a specification SP is a description of its admissible realizations: a program P is a *(correct) realization* of SP if $Sig[P] = Sig[SP]$ and $[\![P]\!] \in [\![SP]\!]$.

In an idealized view of program development, we start with an initial loose requirements specification SP_0 and refine it step by step until we have a specification SP_{last} that records all the intended design decisions:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{last}$$

Stepwise refinement only makes sense if the above chain of refinements guarantees that any correct realization of SP_{last} is also a correct realization of SP_0: for any P, if $[\![P]\!] \in [\![SP_{last}]\!]$ then $[\![P]\!] \in [\![SP_0]\!]$. This is ensured by the definition of refinement. For any SP and SP' with the same signature, we define:

$$SP \rightsquigarrow SP' \iff [\![SP']\!] \subseteq [\![SP]\!]$$

[4] This may be rather indirect, and in general involves a non-trivial abstraction step. It has not yet been attempted for any real programming language, but see [SM02] for an outline of how this could be done for Haskell. See also the pre-CASL work on Extended ML [KST97].

The construction of a program to realize SP_{last} is a separate task which strongly depends on the target programming language. If SP_{last} is relatively small and sufficiently detailed, then achieving this task tends to be straightforward for clean functional programming languages and problems that are appropriately coded in such languages, see for instance the example in Sect. 7. If the target programming language is C or a similar low-level language then of course there is a larger gap between specification and program, largely because a lot of work must be devoted to mundane and complex matters like management of heap space that are handled automatically by more modern languages. This step is outside the scope of CASL, which provides means for building specifications only; in this sense it is not a "wide-spectrum" language [BW82]. See [AS02] for a more detailed discussion. Furthermore, there is no construct in CASL to explicitly express refinement between specifications. All this is a part of the meta-level, though firmly based on the formal semantics of CASL specifications.

A more satisfactory model of refinement allows for modular decomposition of a given development task into several tasks by refining a specification to a sequence of specifications, each to be further refined independently. (Of course, a development may branch more than once, giving a tree structure.)

$$SP \rightsquigarrow BR \begin{cases} SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_{1,last} \\ \vdots \\ SP_n \rightsquigarrow \cdots \rightsquigarrow SP_{n,last} \end{cases}$$

Given realizations $P_1, \ldots, P_n$ of the specifications $SP_{1,last}, \ldots, SP_{n,last}$, we should be able to put them together with no extra effort to obtain a realization of SP. So for each such branching point we need an operation to combine arbitrary realizations of $SP_1, \ldots, SP_n$ into a realization of SP. This may be thought of as a linking procedure $LINK_{BR}$ attached to the branching point BR, where for any $P_1, \ldots, P_n$ realizing $SP_1, \ldots, SP_n$, $LINK_{BR}(P_1, \ldots, P_n)$ realizes SP:

$$\text{if } [\![P_1]\!] \in [\![SP_1]\!], \ldots, [\![P_n]\!] \in [\![SP_n]\!] \text{ then } [\![LINK_{BR}(P_1, \ldots, P_n)]\!] \in [\![SP]\!]$$

Crucially, this means that whenever we want to replace a realization P_i of a component specification SP_i with a new realization P'_i (still of SP_i), all we need to do is to "re-link" it with realizations of the other component specifications, with no need to modify them in any way. $LINK_{BR}(P_1, \ldots, P'_i, \ldots, P_n)$ is guaranteed to be a correct realization of SP, just as $LINK_{BR}(P_1, \ldots, P_i, \ldots, P_n)$ was. In other words, the only interaction between the components happens via $LINK_{BR}$, so the components may be developed entirely independently from each other.

The nature of $LINK_{BR}$ depends on the nature of the programs considered. They could be for instance just "program texts" in some programming language like Pascal (in which case $LINK_{BR}$ may be a simple textual operation, say, regrouping the declarations and definitions provided by the component programs) or actual pieces of compiled code (in which case $LINK_{BR}$ would really be linking in the usual sense of the word). Our preferred view is that the programming language in use has reasonably powerful and flexible modularization facilities, such as those in Standard ML [Pau96] or Ada [Ada94]. Then $P_1, \ldots, P_n$ are program modules (structures in Standard ML, packages in Ada) and LINK_{BR} is a module expression or a *generic module* with formal parameters for which the actual modules $P_1, \ldots, P_n$ may be substituted. Note that if we later replace a module P_i by P'_i as above, "recompilation" of $\mathit{LINK}_{BR}(P_1, \ldots, P'_i, \ldots, P_n)$ might be required but in no case will it be necessary to modify the other modules.

One might expect that BR above is just a *specification-building operation* OP (or a specification construct expressible in CASL), and branching could be viewed as "ordinary" refinement $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$. Further refinement of $OP(SP_1, \ldots, SP_n)$ might then consist of separate refinements for $SP_1, \ldots, SP_n$ as above. This requires at least that OP is "monotonic" with respect to inclusion of model classes.[5] Then the following "refinement rule" is sound:

$$\frac{SP_1 \rightsquigarrow SP'_1 \qquad \cdots \qquad SP_n \rightsquigarrow SP'_n}{OP(SP_1, \ldots, SP_n) \rightsquigarrow OP(SP'_1, \ldots, SP'_n)}$$

This view is indeed possible provided that the specification-building operation OP is *constructive* in the following sense: for any realizations $P_1, \ldots, P_n$ of $SP_1, \ldots, SP_n$, we must be able to construct a realization $\mathit{LINK}_{OP}(P_1, \ldots, P_n)$ of $OP(SP_1, \ldots, SP_n)$. In that case, $OP(SP_1, \ldots, SP_n)$ will be consistent whenever $SP_1, \ldots, SP_n$ are. However, simple examples show that some standard specification-building operations (like the union of specifications) do not have this property. It follows that refining SP to $OP(SP_1, \ldots, SP_n)$, where OP is an arbitrary specification-building operation, does not ensure that we can provide a realization of SP even when given realizations of $SP_1, \ldots, SP_n$. (See [HN94] for a different approach to this problem.)

Another problem with the refinement step $SP \rightsquigarrow OP(SP_1, \ldots, SP_n)$ is that it does not explicitly indicate that subsequent refinement is to proceed by independently refining each of $SP_1, \ldots, SP_n$, so preserving the structure imposed by the operation OP. The structure of the specification $OP(SP_1, \ldots, SP_n)$ in no way prescribes the structure of the final program. And this is necessarily so: *while preserving the structure of a specification throughout the ensuing development is convenient when it is natural to do so, refinements that break this structure must also be allowed.* Otherwise, at very early stages of the development process we would have to fix the final structure of the resulting program: any decision about structuring a specification would amount to a decision about the structure of the final program. This is hardly practical, as the aims of structuring specifications in the early development phases (and at the requirements engineering phase) are quite distinct from those of structuring final programs.

On the other hand, at certain stages of program development we need to fix the structure of the system under development: the design of the modular

[5] Most of the specification constructs of CASL and other specification languages are indeed monotonic. The few exceptions — like imposing the requirement of freeness — can be viewed as operations which add "constraints" to specifications rather than as fully-fledged specification-building operations, cf. data constraints in Clear [BG80].

structure of the system is often among the most important design decisions in the development process. In CASL, this is the role of *architectural specifications*, introduced in the next section.

4 Architectural Specifications

In CASL, an architectural specification *prescribes* a decomposition of the task of implementing a requirements specification into a number of subtasks to implement specifications of "modular components" (called *units*) of the system under development. The units may be parametrized or not. Another essential part of an architectural specification is a prescription of how the units, once developed, are to be put together using a few simple operators. Thus, an architectural specification may be thought of as a definition of a construction that implements a requirements specification in terms of a number of specified units to be developed subsequently.

For the sake of readability, we present here only a very simple version of CASL architectural specifications, with a limited (but representative) number of constructs, shaped after a somewhat less simplified fragment used in [SMT+01].

Architectural specifications: $ASP ::= \textbf{arch spec } Dcl^* \textbf{ result } T$
An architectural specification consists of a list of unit declarations followed by a unit result term.

Unit declarations: $Dcl ::= U\colon SP \mid U\colon SP_1 \xrightarrow{\iota} SP_2$
A unit declaration introduces a unit name with its type, which is either a specification or a specification of a parametrized unit, determined by a specification of its parameter and its result, which extends the parameter via a signature morphism ι.

Unit terms: $T ::= U \mid U[T \textbf{ fit } \sigma] \mid T_1 \textbf{ and } T_2$
A unit term is either a (non-parametrized) unit name, or a (parametrized) unit application with an argument that fits via a signature morphism σ, or an amalgamation of (non-parametrized) units.

The semantics of this CASL fragment can be defined following the same lines as for full CASL, see [CoFI03,SMT+01,BST02b]. Let us just discuss here the semantics of specifications of parametrized units. Consider for instance the following simple architectural specification:

arch spec AS
  units $U_1 : SP_1$;
     $F\colon SP_1 \xrightarrow{\iota} SP_2$;
  result $F[U_1]$

To be well-formed, the specification $SP_1 \xrightarrow{\iota} SP_2$ of the parametrized unit F should be such that $[\![SP_2]\!]\big|_\iota \subseteq [\![SP_1]\!]$.[6] Let Σ_1 and Σ_2 be the respective sig-

[6] Given a signature morphism $\iota\colon \Sigma_1 \to \Sigma_2$ and a Σ_2-model $M_2 \in Mod(\Sigma_2)$, $M_2\big|_\iota \in Mod(\Sigma_1)$ is the *reduct* of M_2 w.r.t. ι to a Σ_1-model defined in the usual way; this obviously extends further to classes of Σ_2-models.

natures of SP_1 and SP_2. To realize the specification $SP_1 \xrightarrow{\iota} SP_2$, we should provide a "program fragment" ΔP (i.e., a parametrized program, see [Gog84]) that extends any realization P_1 of SP_1 to a realization P_2 of SP_2, which we will write as $\Delta P(P_1)$. The basic semantic property required is that for all programs P_1 such that $[\![P_1]\!] \in [\![SP_1]\!]$, $\Delta P(P_1)$ is a program that extends P_1 and realizes SP_2 (semantically: $[\![\Delta P(P_1)]\!]|_\iota = [\![P_1]\!]$ and $[\![\Delta P(P_1)]\!] \in [\![SP_2]\!]$). This amounts to requiring ΔP to determine a partial function[7] $[\![\Delta P]\!]: Mod(\Sigma_1) \rightarrow? Mod(\Sigma_2)$ that "preserves" its argument whenever it is defined, is defined on (at least) all models in $[\![SP_1]\!]$,[8] and yields a result in $[\![SP_2]\!]$ when applied to a model in $[\![SP_1]\!]$. This leads to the following definitions.

Definition 1. *Given a signature morphism $\iota: \Sigma_1 \rightarrow \Sigma_2$, a* local construction *along ι is a persistent partial function $F: Mod(\Sigma_1) \rightarrow? Mod(\Sigma_2)$ (for each $A_1 \in dom(F)$, $F(A_1)|_\iota = A_1$). We write $Mod(\Sigma_1 \xrightarrow{\iota} \Sigma_2)$ for the class of all local constructions along ι.*

Definition 2. *A local construction F along $\iota: Sig(SP_1) \rightarrow Sig(SP_2)$ is* strictly correct w.r.t. SP_1 and SP_2 *if for all models $A_1 \in [\![SP_1]\!]$, $A_1 \in dom(F)$ and $F(A_1) \in [\![SP_2]\!]$. We write $[\![SP_1 \xrightarrow{\iota} SP_2]\!]$ for the class of all local constructions along ι that are strictly correct w.r.t. SP_1 and SP_2.*

The class $[\![SP_1 \xrightarrow{\iota} SP_2]\!]$ is empty if there is some model of SP_1 that cannot be extended to a model of SP_2; then we say that $SP_1 \xrightarrow{\iota} SP_2$ is *inconsistent*.

The crucial idea here is that while programs (closed software components) are represented as CASL models, parametrized programs (generic software components) are represented as persistent partial functions mapping models to models.

Instantiation of such generic modules is then simply function application. This may be further complicated in CASL by cutting the argument out of a larger "global" context via some fitting morphism, and putting the result back into that context. (Hence the terminology "*local* construction".) The latter involves the amalgamation construct, which puts together models (non-parametrized units) sharing common parts, as discussed in detail in [SMT+01].

Note that in the architectural specification AS above, although F is generic it is only instantiated once. Genericity is used here merely to separate the task of implementing SP_1 from the task of implementing the rest of SP_2. This may be stressed by making the generic unit anonymous, and treating U_1 as an *import* for the unit that implements SP_2. In CASL, this is written as follows:

arch spec AS′
  units $U_1 : SP_1$;
    $U_2 : SP_2$ **given** U_1;
  result U_2

Semantically, this is essentially equivalent to AS except that the generic unit remains anonymous and there is an explicit name for the result.

[7] As in CASL, $X \rightarrow? Y$ denotes the set of partial functions from X to Y.

[8] Intuitively, $\Delta P(P_1)$ is "statically" well-formed as soon as P_1 has the right signature, but needs to be defined only for arguments that realize SP_1.

5 Observational Equivalence

So far, we have followed the usual interpretation for basic specifications given as sets of axioms over some signature, which is to require models of such a basic specification to satisfy all its axioms. However, in many practical examples this turns out to be overly restrictive. The point is that only a subset of the sorts in the signature of a specification are typically intended to be directly observable — the others are treated as internal, with properties of their elements made visible only via *observations*: terms producing a result of an observable sort, and predicates. Often there are models that do not satisfy the axioms "literally" but in which all observations nevertheless deliver the required results. This calls for a relaxation of the interpretation of specifications, as advocated in numerous "observational" or "behavioural" approaches, going back at least to [GGM76,Rei81]. Two approaches are possible:

- introduce an "internal" *observational indistinguishability* relation between elements in the carrier of each model, and re-interpret equality in the axioms as indistinguishability; or
- introduce an "external" *observational equivalence* relation on models over each signature, and re-interpret specifications by closing their class of models under such equivalence.

It turns out that under some acceptable technical conditions, the two approaches are closely related and coincide for most basic specifications [BHW95,BT96]. We follow the second approach here.

From now on, for the sake of simplicity, we will assume that the set of observable sorts is empty and so predicates are the only observations. Note that this is not really a restriction, since one can always treat a sort as observable by introducing an "equality predicate" on it.

Definition 3. *Consider a signature Σ. A* correspondence *between two Σ-models A, B, written $\rho\colon A \bowtie B$, is a relation $\rho \subseteq |A| \times |B|$ that is closed under the operations*[9] *and preserves and reflects the predicates.*[10] *Two models $A, B \in Mod(\Sigma)$ are* observationally equivalent*, written $A \equiv B$, if there exists a correspondence between them.*

This formulation is due to [Sch87] (cf. "simulations" in [Mil71] and "weak homomorphisms" in [Gin68]) and is equivalent to other standard ways of defining observational equivalence between algebras, where a special role is played by *observable equalities*, i.e., equalities between terms of observable sorts.

[9] That is, for $f\colon s_1 \times \ldots \times s_n \to s$ in Σ, $a_1 \in |A|_{s_1}, \ldots, a_n \in |A|_{s_n}$ and $b_1 \in |B|_{s_1}, \ldots, b_n \in |B|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $f_A(a_1, \ldots, a_n)$ is defined iff $f_B(b_1, \ldots, b_n) \in \rho_s$ is defined, and then $(f_A(a_1, \ldots, a_n), f_B(b_1, \ldots, b_n)) \in \rho_s$.

[10] That is, for $p\colon s_1 \times \ldots \times s_n$ in Σ, $a_1 \in |A|_{s_1}, \ldots, a_n \in |A|_{s_n}$ and $b_1 \in |B|_{s_1}, \ldots, b_n \in |B|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $p_A(a_1, \ldots, a_n) \iff p_B(b_1, \ldots, b_n)$.

For any specification SP with $Sig(SP) = \Sigma$, we define its *observational interpretation* by abstracting from the standard interpretation as follows:

$$Abs_{\equiv}(SP) = \{A \in Mod(\Sigma) \mid A \equiv B \text{ for some } B \in [\![SP]\!]\}.$$

6 Observational Interpretation of Architectural Specifications

The observational interpretation of specifications sketched in the previous section leads to a more liberal notion of refinement of specifications. Given two specifications SP and SP' with the same signature, we define:

$$SP \overset{\equiv}{\leadsto} SP' \iff [\![SP']\!] \subseteq Abs_{\equiv}(SP)$$

This observational refinement concept means that now we consider that a program P is a correct realization of a specification SP if it determines a $Sig(SP)$-model which is observationally equivalent to an SP-model, thus relaxing the requirements spelled out in Sect. 3.

The crucial issue is now to understand how to re-interpret the semantics of architectural specifications to take account of the observational interpretation of specifications. Surprisingly enough, there is not much to change in the semantics of architectural specifications, the essential modifications to be made concerning only the semantics of specifications of parametrized units.

The key insight is that we should require local constructions to satisfy a "stability" property, see [Sch87].

Definition 4. *A local construction F along $\iota\colon \Sigma_1 \to \Sigma_2$ is* stable *if it preserves observational equivalence of models, i.e., for any Σ_1-models A, B such that $A \equiv B$, if $A \in dom(F)$ then $B \in dom(F)$ and $F(A) \equiv F(B)$.*

While stability seems to be an obvious requirement to be imposed on local constructions, it turns out that it is not quite strong enough for our purposes. The reason is that when applying a local construction, the argument may be "cut out" of a larger global context where more observations are available. To restrict attention to conditions that will be both strong enough and essentially local to the local constructions involved, we define *local stability* as follows.

Definition 5. *A local construction F along $\iota\colon \Sigma_1 \to \Sigma_2$ is* locally stable *if for any Σ_1-models A, B and correspondence $\rho_1\colon A \bowtie B$, $A \in dom(F)$ if and only if $B \in dom(F)$ and moreover, if this is the case then there exists a correspondence $\rho_2\colon F(A) \bowtie F(B)$ that extends ρ_1 (i.e., $\rho_2\big|_{\iota} = \rho_1$).*

Obviously, local stability implies stability. However, it also implies that using the local construction in a global context as suggested above yields a stable construction at the global level as well.

We now have the necessary ingredients to provide a re-interpretation of specifications of parametrized units.

Definition 6. *A local construction F along $\iota\colon Sig(SP_1) \to Sig(SP_2)$ is* observationally correct w.r.t. SP_1 and SP_2 *if for every model $A_1 \in [\![SP_1]\!]$, $A_1 \in dom(F)$ and there exists a model $A_2 \in [\![SP_2]\!]$ and correspondence $\rho_2\colon A_2 \bowtie F(A_1)$ such that $\rho_2\big|_\iota$ is the identity. We write $Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$ for the class of all locally stable constructions along ι that are observationally correct w.r.t. SP_1 and SP_2.*

This implies that $A_2 \equiv F(A_1)$ and $A_2\big|_\iota = A_1$, which is in general stronger than $F(A_1) \in Abs_{\equiv}(SP_2)$. It follows that if $F \in Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$ then there is some $F' \in [\![SP_1 \xrightarrow{\iota} SP_2]\!]$ such that $dom(F') = dom(F)$ and for each $A_1 \in [\![SP_1]\!]$, $F'(A_1) \equiv F(A_1)$. But in general $[\![SP_1 \xrightarrow{\iota} SP_2]\!] \not\subseteq Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2)$, as strictly correct local constructions need not be stable. Moreover, it may happen that there are no stable observationally correct constructions, even if there are strictly correct ones: that is, we may have $Mod_{lc}(SP_1 \xrightarrow{\iota} SP_2) = \emptyset$ even if $[\![SP_1 \xrightarrow{\iota} SP_2]\!] \neq \emptyset$. For a more detailed treatment of stability and of observational interpretation of architectural specifications, see [BST02b].

The conclusion here is that now parametrized program modules, i.e., generic software components, are modelled as *persistent locally stable partial functions*. It is important to understand that stable constructions are those that respect modularity in the software construction process. That is, such constructions can use the ingredients provided by their parameters, but they cannot take advantage of their particular internal properties. Thus, (local) stability is a directive for language design, rather than a condition to be checked on a case-by-case basis: in a language with good modularization facilities, all constructions that one can code should be locally stable.

7 Example

The following example illustrates some of the points in the previous sections. The notation of CASL is hopefully understandable without further explanation; otherwise see [ABK+02].

We start with a simple specification of sets of strings.

spec STRING = **sort** *String* ...
spec STRINGSET = STRING
  then sort *Set*
    ops *empty* : *Set*;
      add : *String* × *Set* → *Set*
    pred *present* : *String* × *Set*
    $\forall\ s, s' : String, t : Set$
      • $\neg present(s, empty)$
      • $present(s, add(s, t))$
      • $s \neq s' \Longrightarrow (present(s, put(s', t)) \iff present(s, t))$

We now refine this specification to introduce the idea of using a hash table implementation of sets.

$\textbf{spec}\ \textsc{Int} = \textbf{sort}\ \mathit{Int} \ldots$
$\textbf{spec}\ \textsc{Elem} = \textbf{sort}\ \mathit{Elem}$
$\textbf{spec}\ \textsc{Array}[\textsc{Elem}] = \textsc{Elem}\ \textbf{and}\ \textsc{Int}$
$\quad \textbf{then sort}\ \mathit{Array}[\mathit{Elem}]$
$\qquad \textbf{ops}\ \mathit{empty} : \mathit{Array}[\mathit{Elem}];$
$\qquad\qquad \mathit{put} : \mathit{Int} \times \mathit{Elem} \times \mathit{Array}[\mathit{Elem}] \to \mathit{Array}[\mathit{Elem}];$
$\qquad\qquad \mathit{get} : \mathit{Int} \times \mathit{Array}[\mathit{Elem}] \ \to ?\ \mathit{Elem}$
$\qquad \textbf{pred}\ \mathit{used} : \mathit{Int} \times \mathit{Array}[\mathit{Elem}]$
$\qquad \forall\ i, j : \mathit{Int}; e, e' : \mathit{Elem}; a : \mathit{Array}[\mathit{Elem}]$
- $i \neq j \Longrightarrow \mathit{put}(i, e', \mathit{put}(j, e, a)) = \mathit{put}(j, e, \mathit{put}(i, e', a))$
- $\mathit{put}(i, e', \mathit{put}(i, e, a)) = \mathit{put}(i, e', a)$
- $\neg \mathit{used}(i, \mathit{empty})$
- $\mathit{used}(i, \mathit{put}(i, e, a))$
- $i \neq j \Longrightarrow (\ \mathit{used}(i, \mathit{put}(j, e, a)) \iff \mathit{used}(i, a)\ )$
- $\mathit{get}(i, \mathit{put}(i, e, a)) = e$

$\textbf{spec}\ \textsc{Elem\_Key} = \textsc{Elem}\ \textbf{and}\ \textsc{Int}$
$\quad \textbf{then op}\ \mathit{hash} : \mathit{Elem} \to \mathit{Int}$
$\textbf{spec}\ \textsc{HashTable}[\textsc{Elem\_Key}] = \textsc{Elem\_Key}\ \textbf{and}\ \textsc{Array}[\textsc{Elem}]$
$\quad \textbf{then ops}\ \mathit{add} : \mathit{Elem} \times \mathit{Array}[\mathit{Elem}] \to \mathit{Array}[\mathit{Elem}];$
$\qquad\qquad \mathit{putnear} : \mathit{Int} \times \mathit{Elem} \times \mathit{Array}[\mathit{Elem}] \to \mathit{Array}[\mathit{Elem}]$
$\qquad \textbf{preds}\ \mathit{present} : \mathit{Elem} \times \mathit{Array}[\mathit{Elem}]$
$\qquad\qquad \mathit{isnear} : \mathit{Int} \times \mathit{Elem} \times \mathit{Array}[\mathit{Elem}]$
$\qquad \forall\ i : \mathit{Int}; e : \mathit{Elem}; a : \mathit{Array}[\mathit{Elem}]$
- $\mathit{add}(e, a) = \mathit{putnear}(\mathit{hash}(e), e, a)$
- $\neg \mathit{used}(i, a) \Longrightarrow \mathit{putnear}(i, e, a) = \mathit{put}(i, e, a)$
- $\mathit{used}(i, a) \wedge \mathit{get}(i, a) = e \Longrightarrow \mathit{putnear}(i, e, a) = a$
- $\mathit{used}(i, a) \wedge \mathit{get}(i, a) \neq e \Longrightarrow$
  $\mathit{putnear}(i, e, a) = \mathit{putnear}(\mathit{succ}(i), e, a)$
- $\mathit{present}(e, a) \iff \mathit{isnear}(\mathit{hash}(e), e, a)$
- $\neg \mathit{used}(i, a) \Longrightarrow \neg \mathit{isnear}(i, e, a)$
- $\mathit{used}(i, a) \wedge \mathit{get}(i, a) = e \Longrightarrow \mathit{isnear}(i, e, a)$
- $\mathit{used}(i, a) \wedge \mathit{get}(i, a) \neq e \Longrightarrow$
  $(\mathit{isnear}(i, e, a) \iff \mathit{isnear}(\mathit{succ}(i), e, a))$

$\textbf{spec}\ \textsc{StringKey} = \textsc{String}\ \textbf{and}\ \textsc{Int}$
$\quad \textbf{then op}\ \mathit{hash} : \mathit{String} \to \mathit{Int}$
$\textbf{spec}\ \textsc{StringHashTable} =$
$\quad \textsc{HashTable}[\textsc{StringKey}]\ \textbf{with}\ \mathit{Array}[\mathit{String}] \mapsto \mathit{Set}$
$\qquad \textbf{reveal}\ \mathit{String}, \mathit{Set}, \mathit{empty}, \mathit{add}, \mathit{present}$

It is easy to check that StringHashTable is a refinement of StringSet. This is a fairly natural structure, building on a specification of arrays that is presumably already available, and including a generic specification of hash tables that may be reused in future.

However, the structure of this specification does not prescribe the structure of the final implementation! For example, we may decide to adopt the structure given by the following architectural specification:

$$
\begin{array}{l}
\textbf{arch spec } \textsc{StringHashTableDesign} = \\
\quad \textbf{units } N : \textsc{Int}; \\
\qquad\qquad S : \textsc{String}; \\
\qquad\qquad SK : \textsc{StringKey} \textbf{ given } S, N; \\
\qquad\qquad A : \textsc{Elem} \to \textsc{Array}[\textsc{Elem}] \textbf{ given } N; \\
\qquad\qquad HT : \textsc{StringHashTable} \\
\qquad\qquad\qquad\qquad \textbf{given } \{A[SK] \textbf{ with } \mathit{Array}[\mathit{String}] \mapsto \mathit{Set}\} \\
\quad \textbf{result } HT \textbf{ reveal } \mathit{String}, \mathit{Set}, \mathit{empty}, \mathit{add}, \mathit{present}
\end{array}
$$

The structure here differs in an essential way from the earlier one since we have chosen to forego genericity of hash tables, implementing them for the special case of strings.

Further development might lead to a final implementation in Standard ML, including the following modules. The task of extracting Standard ML signatures (`ARRAY_SIG` etc.) from the corresponding CASL signatures of the specifications given above is left for the reader.

```
functor A(E : ELEM_SIG) : ARRAY_SIG =
  struct
    open E
    type array = int -> elem
    exception unused
    fun empty(i) = raise unused
    fun put(i,e,a)(j) = if i=j then e else a(j)
    fun get(i,a) = a(i)
    fun used(i,a) = (a(i); true) handle unused => false
  end

structure HT : STRING_HASH_TABLE_SIG =
  struct
    open SK
    structure ASK = A(struct type elem=string end); open ASK
    type set = array
    fun putnear(i,s,t) =
       if used(i,t)
       then if get(i,t)=s then t else putnear(i+1,s,t)
       else put(i,s,t)
    fun add(s,t) = putnear(hash(s),s,t)
    fun isnear(i,s,t) =
       used(i,t) andalso (get(i,t)=s orelse isnear(i+1,s,t))
    fun present(s,t) = isnear(hash(s),s,t)
  end
```

The functor `A` is strictly correct with respect to ELEM and ARRAY[ELEM], and the structure `HT` satisfies the axioms of HASHTABLE[STRINGKEY] literally (at least on the reachable part, and assuming the use of extensional equality on functions). The former would not hold if, for instance, we implemented arrays so as to store the history of updates:

```
functor A'(E : ELEM_SIG) : ARRAY_SIG =
  struct
    open E
    type array = int -> elem list
    fun empty(i) = nil
    fun put(i,e,a)(j) = if i=j then e::a(j) else a(j)
    fun get(i,a) = let val e::_=a(i) in e end
    fun used(i,a) = not(null a(i))
  end
```

Then `A'` is not strictly correct with respect to ELEM and ARRAY[ELEM] (it violates the axiom $put(i, e', put(i, e, a)) = put(i, e', a)$) but it is observationally correct. Similarly we might change the code for `HT` to count the number of insertions of each string. This would violate the axiom $used(i, a) \wedge get(i, a) = s \Longrightarrow putnear(i, s, a) = a$, but again would be correct under an observational interpretation.

The Standard ML functors above are locally stable: they respect encapsulation since they do not use any properties of their arguments other than what is spelled out in their parameter signatures. Indeed, it is impossible to express non-stable functors in Standard ML.

Now let us try to instead directly implement the structure expressed by the specification STRINGHASHTABLE. That structure may be captured by the following architectural specification:

> **arch spec** STRINGHASHTABLEDESIGN$'$ =
> **units** N : INT;
> A : ELEM $\to$ ARRAY[ELEM] **given** N;
> HT' : ELEM_KEY $\times$ ARRAY[ELEM] $\to$ HASHTABLE[ELEM_KEY];
> S : STRING;
> SK : STRINGKEY **given** S, N;
> **result** $HT'[SK][A[S]]$ **reveal** $String, Set, empty, add, present$

Then we might try

```
functor HT'
    (structure EK : ELEM_KEY_SIG and A : ARRAY_ELEM_KEY_SIG
         sharing type EK.elem=A.elem) : HASH_TABLE_ELEM_KEY_SIG =
  struct
    open EK A
    fun putnear(i,e,a) =
       if used(i,a)
       then if get(i,a)=e then a else putnear(i+1,e,a)
       else put(i,e,a)
    fun add(e,a) = putnear(hash(e),e,a)
```

```
  fun isnear(i,e,a) =
     used(i,a) andalso (get(i,a)=e orelse isnear(i+1,e,a))
  fun present(e,a) = isnear(hash(e),e,a)
end
```

However, this does not define a locally stable functor, and is in fact not correct code in Standard ML, since it requires equality on `elem` (in `get(i,a)=e`) which is not required by `ELEM_KEY_SIG`. There is no locally stable functor satisfying the required specification. So, what is a reasonable structure for the requirements specification, as expressed in StringHashTable, turned out to be inappropriate as a modular design.

There is of course more than one way of changing the structure to make it appropriate; one was provided above in StringHashTableDesign. Another would be to require equality on *Elem* in Elem_Key, by introducing an equality predicate — this corresponds to making `elem` an "eqtype" in `ELEM_KEY_SIG`. One point of architectural specifications is that such change of structure is an important design decision that deserves to be recorded explicitly.

8 Conclusions and Further Work

In this paper, we have recalled how the standard and quite general view of formal software development by stepwise refinement can be refined to take into account some notion of components, leading to what is called architectural specifications in Casl. An important outcome of this view is the interpretation of software components by means of local constructions.

In a second step, we have taken into account the fact that a program need not be a strictly correct realization of the given requirements specification: it need only be observationally correct. Then we have pointed out how observational interpretation of specifications leads to the key — and quite natural — stability requirement on local constructions, and how this leads to a re-interpretation of software components by means of persistent locally stable partial functions.

A challenging issue is now to understand how far the concepts developed for our somewhat simplified view of software components can be inspiring for a more general view of components, in particular for the case of system components implemented as communicating processes. While this is clearly beyond the scope of this paper, we nevertheless imagine that a promising direction of future research would be to look for an adequate counterpart of (local) stability in this more general setting.

References

[Ada94] *Ada Reference Manual: Language and Standard Libraries*, version 6.0. International standard ISO/IEC 8652:1995(E). http://www.adahome.com/rm95/ (1994).

[AS02] D. Aspinall and D. Sannella. From specifications to code in Casl. *Proc. 9th Intl. Conf. on Algebraic Methodology and Software Technology, AMAST'02*. Springer LNCS 2422, 1–14 (2002).

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella and A. Tarlecki. Casl: The Common Algebraic Specification Language. *Theoretical Computer Science* 286:153–196 (2002).

[AKBK99] E. Astesiano, B. Krieg-Brückner and H.-J. Kreowski, eds. *Algebraic Foundations of Systems Specification*. Springer (1999).

[BW82] F. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer (1982).

[BH93] M. Bidoit and R. Hennicker. A general framework for modular implementations of modular systems. *Proc. 4th Int. Conf. on Theory and Practice of Software Development TAPSOFT'93*, Springer LNCS 668, 199–214 (1993).

[BH98] M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. *Acta Informatica* 35(11):951–1005 (1998).

[BHW95] M. Bidoit, R. Hennicker and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming* 25:149–186 (1995).

[BST02a] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in Casl. *Formal Aspects of Computing*, 13:252–273 (2002).

[BST02b] M. Bidoit, D. Sannella and A. Tarlecki. Global development via local observational construction steps. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science, MFCS'02*. Springer LNCS 2420, 1–24 (2002).

[BT96] M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. *Proc. 20th Coll. on Trees in Algebra and Computing CAAP'96*, Linköping, Springer LNCS 1059, 241–256 (1996).

[BG80] R. Burstall and J. Goguen. The semantics of Clear, a specification language. *Proc. Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, 292–332 (1980).

[CoFI03] The CoFI Task Group on Semantics. Semantics of the Common Algebraic Specification Language Casl. Available from `http://www.cofi.info/` (2003).

[EK99] H. Ehrig and H.-J. Kreowski. Refinement and implementation. In: [AKBK99], 201–242.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science* 20:209–263 (1982).

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer (1985).

[Gan83] H. Ganzinger. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems* 5:318–354 (1983).

[GGM76] V. Giarratana, F. Gimona and U. Montanari. Observability concepts in abstract data type specifications. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Springer LNCS 45, 576–587 (1976).

[Gin68] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press (1968).

[Gog84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM* 39:95–146 (1992).

[GM82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. 9th Intl. Coll. on Automata, Languages and Programming*. Springer LNCS 140, 265–281 (1982).

[HN94] R. Hennicker and F. Nickl. A behavioural algebraic framework for modular system design and reuse. *Selected Papers from the 9th Workshop on Specification of Abstract Data Types*, Caldes de Malavella. Springer LNCS 785, 220–234 (1994).

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica* 1:271–281 (1972).

[KST97] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Comp. Sci.* 173:445–484 (1997).

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence*, London, 481–489 (1971).

[Pau96] L. Paulson. *ML for the Working Programmer*, 2nd edition. Cambridge Univ. Press (1996).

[Rei81] H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Comp. Sci. Conference*, 27–39 (1981).

[ST88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76:165–210 (1988).

[ST88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Colloq. on Current Issues in Programming Languages, Intl. Joint Conf. on Theory and Practice of Software Development TAPSOFT'89*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269 (1997).

[Sch87] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sch90] O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming* 14:43–57 (1990).

[SM02] L. Schröder and T. Mossakowski. HasCasl: Towards integrated specification and development of functional programs. *Proc. 9th Intl. Conf. on Algebraic Methodology and Software Technology, AMAST'02*. Springer LNCS 2422, 99–116 (2002).

[SMT+01] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman and B. Klin. Semantics of architectural specifications in Casl. *Proc. 4th Intl. Conf. Fundamental Approaches to Software Engineering FASE'01*, Genova. Springer LNCS 2029, 253–268 (2001).

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New-York, N.Y. (1998).