On the Specification of a Component Repository

Nuno Rodrigues[1] and Luis Barbosa[2]

[1] Sidereus, S. A., Porto, Portugal
nfr@sidereus.pt

[2] Dep. Informática, Univ. Minho, Braga, Portugal
lsb@di.uminho.pt

Abstract. The lack of a commonly accepted definition of a software component, the proliferation of competing 'standards' and component frameworks, is here to stay, raising the fundamental question in component-based development of how to cope in practice with heterogeneity. This paper reports on the design of a Component Repository aimed to give at least a partial answer to the above question. The repository was fully specified in VDM and a working prototype is currently being used in an industrial environment.

1 Introduction

"I'm sure this problem has already been solved" and *"probably, others would like to (re)use my solution"*, are common concerns in the everyday life of a programmer. Concerns that the emerging of *component-based* programming paradigm aims to transform into effective programming practises. Actually, all engineering disciplines rely on standard *components* to design and build their artifacts and development methodologies based on *third-party assembly* of components — software engineering should not be the exception.

Although the term *software component* has been around for a long time, *component-based programming* has become a buzzword since mid 1990's (see, *e.g.*, [10, 15, 14, 9, 17]). The basic motivation is to replace conventional programming by system's construction by composition and configuration of reusable off-the-shelf units, often regarded as *'abstractions with plugs'*. Or, quoting from [15], *'independently deployable unit of composition with contractually defined interfaces'*. The paradigm is often illustrated by the visual metaphor of a *palette* of computational units, treated as black boxes, and a *canvas* into which they can be dropped and plugged together through *wires*, corresponding to some sort of gluing code.

However, as it happened before with *object* orientation, component programming has grown up to popular technologies before consensual definitions and principles, let alone formal foundations, have been put forward. In particular, it has given rise to a number of increasingly popular technologies designed around specific *interaction paradigms* — *e.g.*, CORBA and JAVABEANS based on object-like composition, JAVASPACES and TSPACES on top of *shared data space* models, typical of classical coordination languages, and *channel* based techniques

inherited from process languages, as in IWIM/Manifold or Drawin. Such technologies also differ at the application level — *e.g.* acting as source-level languages extensions (such as Corba or JavaSpaces), or as connector frameworks for dynamic (binary-level) units (as in *e.g.*, Com and .NET).

From an engineering point of view a key issue to the success of the component paradigm is *integration*. In fact, unless one restricts himself to a particular framework or, even worse, to a specific tool within a framework, it is no easy task to identify, select, re-use and compose heterogeneous software components from a virtual 'global market' (as represented, for example, by the Internet). The lack of a commonly accepted definition of 'component', the proliferation of competing 'standards' and component frameworks, is here to stay, raising the question of how to cope in practice with heterogeneity.

This paper reports on a concrete approach to a component integration problem found in the context of an european-wide industrial software development project (the Ikf Eureka E!2235 consortium). The challenge was to build a component repository (of both source code and interface information) able to register and marketing all the components produced by different teams in the project. The repository was also supposed to act as an *exchange market*: each user being able not only to register its own components, but also to announce its plans to issue specific components and even to publicise its needs and associated requirements in the form of collections of APIs. Although such *component marketplaces* are emerging in the Web, the underlying description techniques remain rather informal, mainly textual, plus an additional, limited, classification in terms the intended business areas.

In a sense API, the acronym for *Application Program Interface*, emerged as a keyword. But what is in an API, when the component technology used ranges from Java and Corba code to Haskell or Wsdl? Coming back to Szyperski's definition above, what seems crucial to handle is the definition of what an *interface* is.

Initial prospects of implementing the repository as an API database and resorting to text-based retrieve methods for querying, soon reveal impractical given the heterogeneity of the APIs supplied by the different project teams. As the complexity of the individual components and the size of the repository increases, higher levels of abstraction are required, and, for all practical purposes, going abstract means going formal. The Vdm meta-language [5, 4] was chosen as the common language to which different sorts of APIs were mapped. A complete, working prototype of the repository, named CompSrc to express the idea of a 'component source indexing space', has been developed in the Vdm ToolBox complemented with a dedicated web interface for easier access.

The paper is organised as follows: the CompSrc architecture and the generic component model underlying the whole system are described in the following two sections. Section 4 discusses some of the composition patterns automatically detected by CompSrc; an example is briefly discussed in section 5. We conclude with some prospect of future work in section 6.

2 The CompSrc Architecture

The CompSrc repository is built around a broad characterisation of what an interface to component source code is, specified in the Vdm meta-language in the early project stages. This is called, in the context of this system, the *abstract API*. The component source code database is indexed by such abstract APIs which are 'extracted' from the actual component APIs. This is done by extractor functions specified for the typical target languages used in the project. Such extractors correspond to the incoming arrows in Figure 1.

In the centre of the structure diagram is the interface repository indexing the actual component source data base. The repository supports the usual management operations — *e.g.*, registering, removing, updating, etc. The most relevant feature, however, is the support of a *component calculator* which makes it possible to compute new services based on the automatic composition of old functionality, according to more or less strict interaction patterns, and, then, to build new APIs and new components which would act as if the actual source code has been merged. The interaction patterns available are different kinds of *pipelining* (linear, flattened or monadic, as described below) as well as *multiplicative* and *additive* aggregation, corresponding to component parallel composition and choice, respectively. The underlying component calculus is described in detail in [2, 1]. Once a new component is built, by the calculator, the new abstract API is registered in the system and the component 'combined' code is supplied, in the form of a conditional compiling script in the .Net notation. This corresponds to the outgoing arrows in the diagram.

3 Abstract Interfaces

The 'quest' for a common *abstract API* format started from a reverse specification of the (static information part of) different sorts of component APIs used in the project. These ranged from the 'object-oriented flavour', emphasising hierarchical class structures, to 'functional modules' described by service signatures and, eventually, datatype constraints. The formal analysis of typical API examples in Java, Corba, Microsoft Visual Basic, Haskell and Wsdl, which is documented in [11], lead to the description of a component as an indexed collection of services (referred to as *modules* in the sequel) and mirroring the overall component architecture described in Figure 2.

Each module is, then, defined as an ordered tuple of information including the following aspects:

1. **Desc -** a text description of the current module
2. **INH -** a set of imported modules
3. **Ex -** a map enumerating examples concerning the module usage
4. **Func -** the set of functions (fine-grain services) defined in the module
5. **State -** the implementation state of the module which is one of :
   - **Implemented -** the module is implemented and ready to use

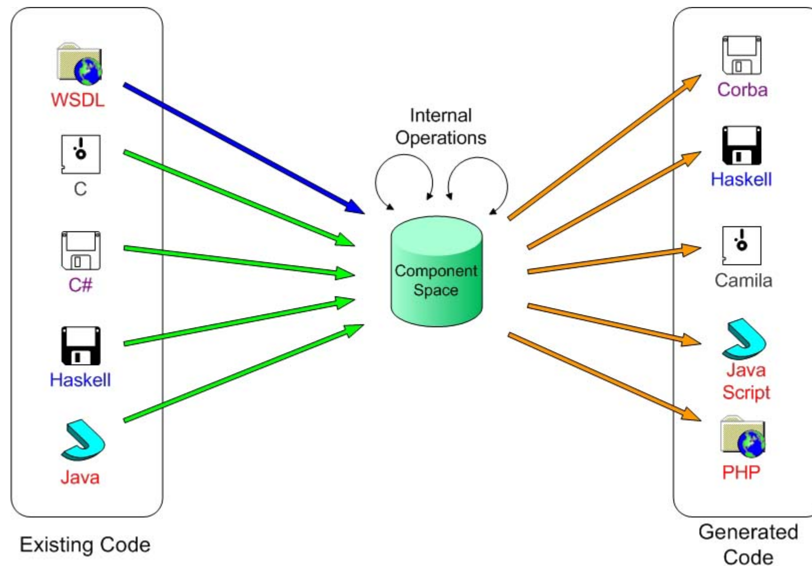

Fig. 1. Component Space Architecture

- **In Implementation** - the module is currently in implementation process
- **Requested** - someone is interested in this module and is willing to pay for it
- **To Convert** - the module is specified in a language but needs to be implemented in a different platform
- **Generated** - generated by the 'component calculator' in CompSrc

6. **Platform** - the target implementation platform
7. **StatDep** - the static dependencies set, i.e., environment variables, runtime environments, ...
8. **Protocols** - a brief description of the communication protocols that the module implements
9. **SpecAPI** - specific (platform-dependent) module data for each kind of API, basically distinguishing purely functional interfaces from state-based ones (as found in object-orientation)

4 Component Assembly

For each API submitted to CompSrc a corresponding *abstract API* (*i.e.*, a value of the respective Vdm datatype) is built by the (language-dependent) *interface extractors* mentioned above. Such abstract APIs provide the basis for classifying, locating and retrieving components from the underlying repository. Furthermore

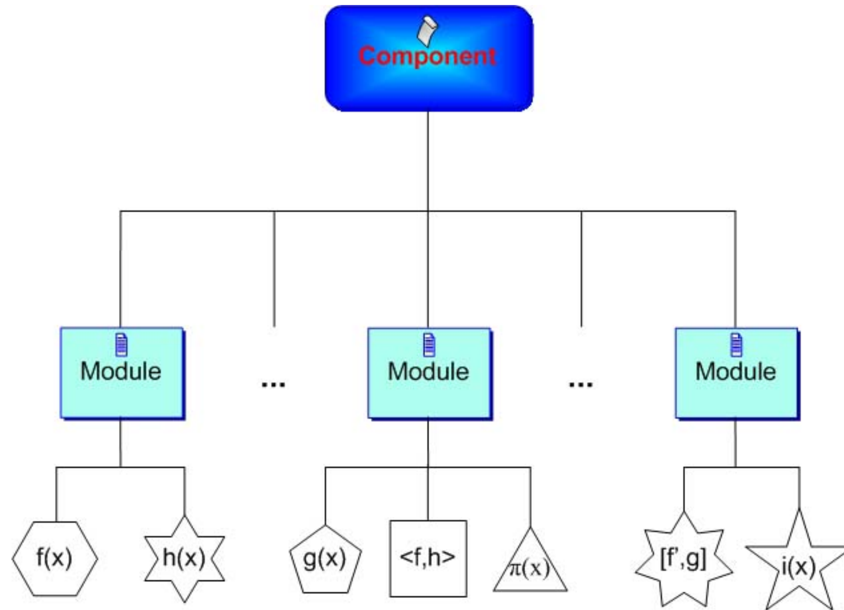

Fig. 2. Structure of a Component

$$\begin{aligned} &CompData :: Modules : ModuleId \stackrel{m}{\rightarrow} ModuleData \\ &\qquad\qquad\quad Name : String \\ &ModuleId = \mathbb{Z}; \end{aligned}$$

Fig. 3. Component DataType

they become the 'raw material' used by the *component calculator* to generate new components.

Such generation is done in two different, but complementary, ways: *aggregation* and *wiring*.

Component *aggregation* is achieved by the application of a small set of operators acting on the abstract interfaces as a whole [1]. In particular, they model:

- Interface *restriction* to a set of *modules* or even, within a particular module, to a set of services.
- *Additive aggregation*, in which different modules coming from two different interfaces are selected and packaged into a new one. If both interfaces are

[1] see [2] for an overview of the corresponding calculus partially implemented in CompSrc.

$$
\begin{aligned}
ModuleData :: \; & Desc : [\mathsf{token}] \\
& INH : Inheritance\text{-}\mathsf{set} \\
& EX : Examples \\
& Funcs : Func\text{-}\mathsf{set} \\
& State : \textsc{Implemented} \mid \textsc{InImplemetation} \mid \\
& \qquad\quad \textsc{PayToImplement} \mid \textsc{Moduled} \mid \\
& \qquad\quad \textsc{Generated} \\
& Platform : [String] \\
& StatDep : [\mathsf{token}] \\
& Protocls : \mathsf{token}\text{-}\mathsf{set} \\
& SpecAPI : SpecOOAPI \mid SpecFAPI \\
Examples = \; & ExId \xrightarrow{m} \mathsf{token}; \\
ExId = \; & \mathbb{Z};
\end{aligned}
$$

Fig. 4. Module DataType

indexing actual code written in compatible implementation languages (*i.e.*, related by an *embedding*) the code generation process is activated.
- *Multiplicative* aggregation, corresponding to the synchronous execution of modules in two different components.

The *wiring* process, on the other hand, is based on the search for *composition possibilities* among the collections of functions of a specified set of components. Such search can be systematic, exposing all possible connections arising from a given set of components, or user-oriented in which each possible composition is validated or discarded by the user. In any case the problem is to identify pairs of functions, in different components, whose range and domain match, according to some matching criteria detailed below. Note that the collection of functions in a component abstract API models the available fine-grain services.

As shown above in the respective specification (Fig. 4), a module records interface information for each function in its function set. Such information amounts basically to the *signature*, *i.e.*, a type declaration of its arguments and result. It becomes clear that a uniform specification of the datatypes used in all components registered in CompSrc is a key issue in the design of the repository. Therefore, when an *abstract API* for a submitted component is built, each *extractor* engine analysis the available type information to build a correspondent abstract representation in the form of (instances of) *polynomial functors*. A brief explanation is now in order.

A function

$$f : I \longrightarrow O$$

models a computational process as a transformation rule between two structures I and O, *i.e.*, as a recipe (a tool, a technology) to build 'gnus' from 'gnats'.

Types I and O may be 'primitive' (*i.e.*, defined as such at the programming language level). Often, however, such is not the case. For example, one may know how to produce 'gnus' from 'gnats' but not in all cases. This is expressed by observing the output of f in a more refined context: O is replaced by $\mathbf{1} + O$ and f is said to be a *partial* function. In other situations one may recognise that there is some environmental (or context) information about 'gnats' that, for some reason, should be hidden from input. It may be the case that such information is too extensive to be supplied to f by its user, or that it is shared by other functions as well. It might also be the case that building gnus would eventually modify the environment, thus influencing latter production of more 'gnus'. For U a denotation of such context information, the signature of f becomes

$$f : I \longrightarrow (O \times U)^U$$

In both cases f can be typed as

$$f : I \longrightarrow \mathsf{F}\, O$$

for $\mathsf{F}X = X + \mathbf{1}$ and $\mathsf{F}X = (X \times U)^U$, respectively. Informally, F can be thought of as a type transformer providing a *shape* for the output of f. Technically, F is a *functor* [2] The notation used above (exponentiation and $+$) stand for some basic datatype (and functor) constructors which express the ways in which 'types' (information, in general) can be composed. Such basic constructors are

- *Cartesian product* $(A \times B)$ for aggregation in the spatial axis;
- *sum* $(A + B)$, for choice (i.e., aggregation in the temporal axis);
- *exponentiation*, or function space, (A^B) for functional dependence;
- *constants*, like the exception type $\mathbf{1}$ or, in general, any primitive type; and
- *powerset* $(\mathcal{P}A)$ and *sequences* (A^*) related to non deterministic and deterministic collections of data, respectively.

These constructors can be found almost directly in high-level languages (such as, *e.g.*, HASKELL or the VDM meta-language) and inferred in a systematic way from other programming notations. Functors built from (and closed by) such constructors plus functor composition are called (extended) *polynomial* and extensively used in the repository to record functions' signatures.

In general, each function in the abstract API is specified as

$$\mathsf{F}\, I \;\rightarrow\; \mathsf{G}\, O$$

where I and O are the *import* (respectively, *export*) datatypes embedded in a behavioural context F (respectively, G) represented as a polynomial functor. For example a non deterministic service may be modelled by a function $\mathsf{serv} : I \longrightarrow \mathcal{P}O$, where $\mathcal{P}$ stands for the (finite) powerset functor. Similarly a service in the form of a partial transducer may take the form of $\mathsf{machine} : I \longrightarrow \mathbf{1} + (O \times I)^K$.

[2] A concept borrowed from category theory (see *e.g.*, [7, 8] or [3] for a computer science perspective) capturing a uniform transformation of both 'types' and 'type-preserving' operations.

This sort of embedding through a functor entails the need to equip CompSrc with the ability to *compare* functors involved in the datatypes of function signature's. Functor (structural) equality and functor instantiation order are achieved by the comparison functions listed in appendices A and B.

Based on such functions the repository is able to perform different types of 'functional' composition, besides the obvious one between functions sharing the domain of one with the codomain of the other. In particular the following 'extra' *pipelining* composition patterns are considered:

- *Curry insensitive*, which allows to perform curry or uncurry on a pair of otherwise no composable services.
- *Monadic*, whenever the context information is captured by a *monad* (which, as shown in [1] is often the case) [3]. Monadic composition includes both the usual Kleisli composition of monadic functions used in functional programming, and a simpler scheme amounting to the monadic embedding of a 'plain' function. Both cases are illustrated in Figure 5. Kleisli composition of functions f and g is achieved by flattening (with the corresponding multiplication μ) the result of the top line composition scheme. The bottom line scheme represents monadic embedding.

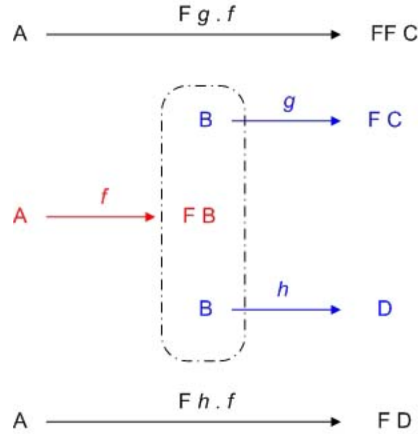

Fig. 5. Monadic Composition

[3] A *monad* is also a concept borrowed from category theory. Computationally, it stands for a functor, representing some sort of 'computational effect', equipped with an embedding function ($\eta : X \longrightarrow \mathsf{F}X$) to 'see' simple values as F-computations and a multiplication ($\mu : \mathsf{FF}X \longrightarrow \mathsf{F}X$) to 'flatten' computational effects (see, eg, [16])

The specification of the wiring process is based on the structural comparison functions mentioned above to relate, up to the intended composition pattern, functions input/output contexts.

5 An Example

The example shown in this section illustrates the identification of composable functions in two rather different components, developed in the context of the Ikf project, by different groups and using different technologies. This example is fully documented in [12], to which the interested reader is referred to.

The first component is a Vdm-SL specification of a robot which manages box storing inside a generic warehouse. The component provides functionality to, *e.g.*, find the best fit of a box inside the warehouse storing space, remove a box, rearrange the warehouse in order to get the biggest amount of free space, etc.

The second component is a web-publisher generator from any sort of information organised as a *leaf tree* (*i.e.*, a binary tree with all information stored on the leafs), developed in Haskell. This component provides, in particular, a function *y2html* to generate the Html representation of a leaf tree value.

In appendix C part of the abstract interfaces generated for CompSrc from the sources of these two components are shown. As a remark note the type information derived, in the second component, from the following Haskell declaration:

```
data Y a b i = Leaf (Unit a b) | Node (Mode i, (Y a b i, Y a b i))
deriving Show

data Mode i = Hr i | Hl i | Vt i | Vb i deriving Show

data Frame i = Frame i i deriving Show

data Sheet a b i = Rect (Frame i) (Y a b i) deriving Show
```

Applying some of the composition test suites defined in the CompSrc leads to the identification of several possible compositions between (the abstract interfaces of) these two distinct modules. The repository is able, in particular, to determine, in an automatic way, that the *functors* underlying type *Space* in the robot component and type Y in the web-publisher one are structurally identical. This fact opens the possibility of composing function *y2html* in the latter component with any function returning values of type *Space* in the former. Such is the case, *e.g.*, of functions *freeSpace*, *defragment* and *whichBoxes*. The composition of *y2html* with any of these functions provides, *for free*, generators of Html interfaces for the warehouse Vdm prototype.

Such *wiring* possibility is detected by the application of function *compareFunctor* (in appendix B) whose result in the Vdm ToolBox syntax, is

```
{ mk_( "y2html",{ "freeSpace","defragment","whichBoxes" } ) }
```

This identifies a functional wiring scheme between *y2html*, on one hand, and functions *freeSpace*, *defragment* and *whichBoxes*, on the other. It also tells that the order of application for this interaction is *y2html* following *freeSpace*, *defragment* or *whichBoxes*.

6 Conclusions and Further Work

Software development by component assembly is most likely to become the main stream in software engineering in the near future. This will lead to a broader understanding of what a software component is (virtually any sort of content can be encapsulated in a reusable entity with well-defined interfaces and able to be connected at runtime) and to the emergence of standardised component frameworks able to integrate heterogeneous components and deal with other such systems in a cooperative manner. The ability to deal with 'non native' components has been recognised as the hallmark of the so-called 'second-generation' component systems [13], a step ahead of what is currently achieved in Corba, JavaBeans or Objectspace Voyager.

The basic lesson learnt from the development of CompSrc is the potential of formal, model-oriented, methods in guiding the design of such platforms. We believe this exercise can be further extended to cope with some issues not covered in the present version, namely, dynamic instantiation of components, location and mobility. Those are fundamental issues for modelling *distributed* component frameworks.

The CompSrc prototype has been used, not only in the context of the project in which it was originally developed, but also to organise software components arising from a massive re-engineering effort of legacy code undertaken by the software company to which the first author is affiliated.

Such an architectural re-engineering effort aimed to identify service components orthogonal to the basic development layers considered in the design practice (*i.e.*, database definition, middleware and GUI). For each identified component an abstract interface, as described above, has been written and directly submitted to CompSrc together with a .Net script to navigate in the (monolithic) legacy code (which remains unchanged) and generate the actual executable code corresponding to the new abstract API. In a subsequent stage new software products incorporating the recovered components have been generated within CompSrc.

Future work on the the CompSrc prototype is foreseen in two main directions:

- The scaling up CompSrc to act over the web in a *transparent* way, instead of relying on a localised component source database, as well as making it able to produce new component connection at runtime, is a main challenge from a technological point of view.
- Conceptually more demanding is the addition of facilities to cope with heterogeneity not only at the 'linguistic' and component-style levels (*e.g.*, the integration of object and functional models), but also at the level of the *interaction style* (aiming at the integration of, *e.g.*, method invocation, dataflow stream processing and event-based interaction). Some preliminary work on a similar topic is documented in [6].

Acknowledgements.

The work reported in this paper was supported by the Ikf project (*Information and Knowledge Fusion*), IKF-IPTG-CW (*ComponentWare*) under contract E!2235.

References

1. L. S. Barbosa. Towards a calculus of state-based software components. In *Selected Papers from the 7th Brazilian Symposium on Programming Languages (to appear in the* Jour. of Universal Computer Science*)*, Ouro Preto, Brasil, June 2003.

2. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82.1, Warsaw, April 2003.
3. R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.
4. J. Fitzgerald and P. G. Larsen. *Modelling Systems: Pratical Tools and Techniques in Software Development.* Cambridge University Press, 1998.
5. Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.
6. K.-P. Lohr. Towards automatic mediation between heterogeneous software components. volume 65.4. Elect. Notes in Theor. Comp. Sci., Elsevier, 2002.
7. S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.
8. C. McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides.* Clarendon Press, 1992.
9. B. Meyer and C. Mingins. Component-based development: From buzz to spark. *IEEE Computer*, 32(7):35–37, 1999.
10. O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall International, 1995.
11. N. Rodrigues. Abstract interfaces. Technical report, Sidereus and DI (U. Minho), 2003.
12. N. Rodrigues. Formal methods laboratory: the component repository specification. Technical report, Univ. Minho, DI (*document and repository prototype available from* `nunorodrigues@di.uminho.pt`), 2003.
13. K. Schmaranz. On second generation distributed component systems. *Journal of Universal Computer Science*, 8(1):97–116, January 2002.
14. J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.
15. C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
16. P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming.* Springer Lect. Notes Comp. Sci. (925), 1995.
17. P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Report on the Dagstuhl Seminar 99081, February 1999.

A CompareFunctor Function

```
compareFunctor : DataType × DataType → 𝔹
compareFunctor (fdt, dt) ≜
  cases fdt :
    mk-Plus (s) →
        is-Plus (dt) ∧
        len s = len dt.P ∧
        dand ([compareFunctor (s (i).#2, dt.P (i).#2) | i ∈ inds s]),
    mk-Times (s) →
        is-Times (dt) ∧
        len s = len dt.T ∧
        dand ([compareFunctor (s (i), dt.T (i)) | i ∈ inds s]),
    mk-Power (b, e) →
        is-Power (dt) ∧
        compareFunctor (b, dt.Base) ∧
        compareFunctor (e, dt.Exponent),
    mk-Map (l, r) →
        is-Map (dt) ∧
        compareFunctor (l, dt.Left) ∧
        compareFunctor (r, dt.Right),
    mk-Set (d) → is-Set (dt) ∧
                compareFunctor (d, dt.Of),
    mk-Seq (d) → is-Seq (dt) ∧
                compareFunctor (d, dt.Of),
    mk-Curry (s) →
        is-Curry (dt) ∧
        len s = len dt.C ∧
        dand ([compareFunctor (s (i), dt.C (i)) | i ∈ inds s]),
    mk-Var (-, -) → is-Var (dt),
    REC → dt = REC,
    NIL → dt = NIL
  end;
```

B EqualsLessFunctor Function

```
equalsLessFunctor : DataType × DataType → 𝔹
equalsLessFunctor (fdt, dt) ≜
  if fdt = dt
  then true
  else cases fdt :
        mk-Plus (s) → dor ([equalsLessFunctor (s (i).#2, dt) | i ∈ inds s]),
        mk-Times (s) → dor ([equalsLessFunctor (s (i), dt) | i ∈ inds s]),
        mk-Power (b, e) → equalsLessFunctor (b, dt) ∨
                          equalsLessFunctor (e, dt),
        mk-Map (l, r) → equalsLessFunctor (l, dt) ∨
                        equalsLessFunctor (r, dt),
        mk-Set (d) → equalsLessFunctor (d, dt),
        mk-Seq (d) → equalsLessFunctor (d, dt),
        mk-Curry (s) → dor ([equalsLessFunctor (s (i), dt) | i ∈ inds s]),
        mk-Var (-, t) → is-Var (dt) ∧
                        dt.Type = t,
        REC → dt = REC,
        NIL → dt = NIL
      end;
```

C Component Source Values

$BoxInfo : DataType = \mathsf{mk\text{-}}Var\,(\mathsf{nil}\ , "real");$

$Width : DataType = \mathsf{mk\text{-}}Var\,(\mathsf{nil}\ , "real");$

$Box : DataType = \mathsf{mk\text{-}}Times\,([BoxInfo, Width]);$

$Space : DataType = \mathsf{mk\text{-}}Plus\,([\mathsf{mk\text{-}}\,("Left", Box),$
$\qquad \mathsf{mk\text{-}}\,("Right", \mathsf{mk\text{-}}Times\,([\textsc{Rec}, \textsc{Rec}]))]);$

$wbOutput : DataType = \mathsf{mk\text{-}}Set\,(BoxInfo);$

Fig. 6. VDM-SL Warehouse Storing Robot DataTypes

```
robotFuncs : Function-set = {mk-Function ("whichBoxes",
                                          wbOutput,
                                          "Delivers the information of all boxes
                                          in the palette.",
                                          {},
                                          mk-CmpSrc`FlatFunc (Space, nil , nil )),
                             mk-Function ("freeSpace",
                                          Width,
                                          "Yields the width of the widest free
                                          space in the palette.",
                                          {},
                                          mk-CmpSrc`FlatFunc (Space, nil , nil )),
                             mk-Function ("insertBox",
                                          Space,
                                          "Best fit insertion of a box, if
                                          possible.",
                                          {},
                                          mk-CmpSrc`FlatFunc
                                              (mk-Times ([Box, Space]), nil , nil )),
                             mk-Function ("removeBox",
                                          Space,
                                          "Removes a box from the warehouse.",
                                          {},
                                          mk-CmpSrc`FlatFunc
                                              (mk-Times ([Box, Space]), nil , nil )),
                             mk-Function ("defragment",
                                          Space,
                                          "collapse all empty subareas as much
                                          as possible.",
                                          {},
                                          mk-CmpSrc`FlatFunc (Space, nil , nil ))};

moduleRobot : ModuleData = mk-ModuleData (mk-token ("Robot"),
                                             {},
                                             {↦},
                                             robotFuncs,
                                             MODULED,
                                             "VDM-SL",
                                             mk-token (nil ),
                                             {},
                                             mk-CmpSrc`SpecFAPI (nil ));
```

Fig. 7. Warehouse Storing Robot

```
Str : DataType = mk-Var (nil , "String");
Unit : DataType = mk-Times ([Str, Str]);

Mode : DataType = mk-Plus ([mk- ("Hr", mk-Var (nil , "int")),
                            mk- ("Hl", mk-Var (nil , "int")),
                            mk- ("Vt", mk-Var (nil , "int")),
                            mk- ("Vb", mk-Var (nil , "int"))]);

Y : DataType = mk-Plus ([mk- ("Leaf", Unit),
                         mk- ("Node", mk-Times ([REC, REC]))]);

Frame : DataType = mk-Curry ([mk-Var (nil , "int"), mk-Var (nil , "int")]);

Sheet : DataType = mk-Curry ([Frame, Y]);

journalFuncs : Function-set = {mk-Function ("y2html",
                                            Y,
                                            "Generates de HTML representation
                                            of a Y value.",
                                            {},
                                            mk-CmpSrc`FlatFunc
                                                (mk-Var (nil , "String"), nil , nil ))}

moduleHJournal : ModuleData = mk-ModuleData (mk-token ("Journal Generator"),
                                             {},
                                             {↦},
                                             journalFuncs,
                                             IMPLEMENTED,
                                             "Haskell",
                                             mk-token (nil ),
                                             {},
                                             mk-CmpSrc`SpecFAPI (nil ));
```

Fig. 8. Web-publisher Generator