

# On Specification Carrying Software, its Refinement and Composition

Matthias Anlauff and Dusko Pavlovic  
Kestrel Institute, Palo Alto  
{*ma,dusko*}@kestrel.edu

## ABSTRACT:

In this paper, we present the framework of evolving specifications (specs), implementing, in the categorical setting of algebraic specifications, a logical view of state, known from Hoare logic to abstract state machines (evolving algebras). The categorical support for both top-down and bottom-up development is thus extended from the refinement and composition of the structure of programs, to the refinement and composition of their behaviors.

While they were originally defined as specifications carrying state machines, specs can also be viewed from another angle, as software components carrying their specifications. As first-class citizens of software systems, specifications are thus made available both statically, as generalized interfaces, and dynamically, as the carriers of adaptability. From this point of view, specs seem particularly suitable for capturing and analyzing the dynamic aspects of architectural composition. In this expository note, we shall survey the main ideas, and outline some examples, including a summary of a method for analysis and transformation of security protocols, where runtime architectural changes result from the internal dynamics of connectors, or components. In such cases, an architectural view with the abstraction level predetermined by the features of the chosen architecture description language, may conceal the essence, whereas the practical application requires versatility.

## I. INTRODUCTION

### A. “Software Philosophy”

Software has become an essential part of the mental tissue of western society. The global market of goods and informations is now implemented as a global network of software systems and agents, pervading our homes and offices. They interact with us, and depend on human guidance as much as our science, communication, banking, even our ability to move in space, essentially depend on them.

As carriers of our evolution, software systems and agents have been evolving together with our society. However, their adaptability to new tasks and new environments (often created by their own modifications) has been a fact of life before it became a subject of science. While generations of software systems lived and died in front of every computer user, computer scientists patiently analyzed some static mathematical models of programming structures. Be-

sides the various technical reasons, a possible conceptual reason of this may be that the software evolution did not have an intrinsic carrier, “software DNA”, but was imposed from outside, by the developers, reluctantly passing to each other the evolutionary invariants embedded in “project documentation”.

The research agenda outlined in the present paper pursues a semantically based approach to dynamics, evolution and adaptability of software. An obvious prerequisite is a mathematical model of software components, and their composition, that should display their evolution and adaptability as a process and a property intrinsic to software itself, as it plays its role of a fundamental carrier of everyday life — rather than just an aspect of software development and maintenance, or a factor in its economy. We contend that the technical and conceptual prerequisites, necessary for studying the complex and *natural* processes of software evolution and adaptation as a crucial aspect of modern computation, have become available through recent technical advances in semantics of computation, software science, and engineering.

### B. “Software Engineering”

In recent years, software engineering has largely abandoned the long-standing waterfall model of the development of software systems, obeying the strict rhythm of analysis-design-implementation-maintenance. Nowadays, the main challenge is to find the methods that would help us comprehend and support the multifaceted dynamics of the software systems, always open to change, whether they have been long deployed, or are still under development; to accommodate, rather than try to dam the evolutionary changes in the real systems, old and young.

In the traditional approach, the formalization of the requirements has been seen as the hardest problem of software development for a long time; a view being most convincingly propounded by David Parnas [12]. The requirements hardly ever exists as a static entity: formal or not, they change during the lifetime of the system. This shift of focus is also reflected in some newer publications in software development approaches, for instance the *continuous software engineering* approach by H. Weber et. al. in [21], [19] and [22]. In this approach, the development of a piece of software is seen as a continuous circle of engineering/reverse-engineering steps triggered by chang-

ing requirements.

The relation between requirements and the software realizing them is somewhat more complex than a simple dependency in one or in the other direction, because the requirements interact and dynamically change with their realizations. In our view, requirement specifications and the implementations are inseparable as the constituents of software systems. They interact with the environment through each other, and evolve supporting and carrying each other.

Changing software during its lifetime precludes all software development approaches that don't provide means for accessing at least some internal structure of the system, because then it would be impossible to change or extend its functionality. A prominent reason for the recent triumphal procession of component-based approaches across many fields of software engineering is mainly based on the fact, that this paradigm provided a framework for providing this kind of information without revealing implementation details. Moreover, in many approaches, components are runtime artifacts that can be assembled dynamically.

### C. Specification Carrying Software

However, the use of components alone doesn't solve the problem of *correctly* (re-)constructing software systems. In most cases, the quantity and quality of information available for the components is not enough to assure satisfaction of the desirable, or even essential functionality, safety, reliability, performance, security, or other system properties. Also, the specification of a component — even if it exists and is expressive enough to be used in a meaningful way while composing software — is often not accessible when it is actually needed, because it resides in some document that has not been shipped with the component itself for one or the other reason, leave alone those cases, where a component specification doesn't exist at all.

The general approach to this problem, that we shall present in this paper, is based on the idea of *specification carrying software*. Very roughly, software components should carry the specifications they implement, just like organisms carry the genes that they embody, and are carried by them; just like genes and organisms, specifications and components should evolve together. The main point of this metaphor is that the compatibility of components needs to be resolved on their specifications, just like the compatibility of organisms is resolved on their genes. Specifications, as the first class citizens of software, available at runtime, are the carriers of both the preservation and the variation in a population of components, just like the genes are in a population of organisms.

In the dynamic world of the Web, data are used in always different, unpredictable ways, and therefore cannot be structured in advance. The need for *semistructured, self-describing* data[1] is addressed by markup languages like XML.

In the same dynamic world, software components are

also used in always different, unpredictable ways. The idea of structuring modules in advance has become unfeasible. The OO-development, with prefabricated class templates spawning all objects that constitute systems, has become too rigid. Whichever form the solution may take, it will have to be based on semi-structured, self-describing software components, carrying their mutable specifications like XML data carry theirs.

In this way, software adaptation, as the precondition of software composition, and the essence of the paradigm of components [11], is hoped to extend beyond COTS reuse, based on glue and system designers' interventions, and to enable runtime reconfigurations, possibly even involving on-the-fly source code regeneration and recompilation [17].

### D. Outline

In Section II, we will present *especs*, a formal framework of evolving specifications implementing a logical view of state known from Hoare local to Abstract State Machines (Evolving Algebras). Section III contains a description how we represent refinements in our framework. Section IV introduces the colimit algorithm which is used to model composition of specifications. Section V gives an example explaining some of the concepts presented in the paper; Section VI contains final remarks and provides an outlook on future work.

## II. EVOLVING SPECIFICATIONS

In this section, we introduce *especs* as a convenient framework for describing, refining, adapting and composing specification carrying components.

### A. From ASMs to *especs*

The shortest description of *especs* (evolving specifications) is probably that they are a version of Gurevich's abstract state machines (evolving algebras) — deployed in a categorical framework. The point of such a framework is that it supports

- refinement, or top-down development, in the form of morphisms between specifications,
- effective, automated composition operations, for bottom-up development, in the form of colimits,
- rich encapsulation and stratification mechanisms, for abstraction, and building and transforming systems, and systems of systems, in the form of diagrams and hereditary diagrams.

A bit more formally, an *espec* is a particular sort of diagram in a suitable category of algebraic specifications. Its states are *specs*, and its transitions are the interpretations between the *specs*, but in the opposite direction. This is, of course, based on the view of abstract state machines as programming language based on algebraic specifications and function updates. The advantages and disadvantages of programming in such a framework, as compared with the OO and structured programming, have been thoroughly

analyzed in [10], [11]. The function updates of algebraic specs provide the combined power of references and assignments, the convenience of jumps and mutable state, while minimizing the familiar practical and semantical dangers, arising from combining the two.

To all this, especs are adding just the simple observation that *function updates are in fact morphisms between algebraic specifications*. In a sense, algebraic specifications with function updates between them are nothing but the category of algebraic specifications. While a function update  $f(u) := t$ , as a runtime operation, is fairly unusual among the languages from Lisp to C++, reinterpreting a function by a term  $f(u) \mapsto t$  is the basic operation in specification development, just an item in the definition of a spec morphism.<sup>1</sup>

## B. State machines

State machines consist of states and transitions. They have been studied as abstract models of computation ever since Turing [20], [9]. Nowadays, though, they are often encountered as useful tools of engineering practices: several state machine based languages and frameworks, such as Statecharts [8], Esterel [2], abstract state machines (ASMs)[7] etc., have found many practical applications.

The transitions correspond to the computational steps between the states of the computer. In a way, programming generally consists of specifying transitions, while the states are usually left implicit, or recorded as comments. The semantical analyses of programs often consist of reconstructing states generated in the execution. E.g., in the Floyd-Hoare logic, and axiomatic semantics based on it, the states are annotated by some predicates, recognized as valid at each of them. Starting from this paradigm of *states as predicates*, Dijkstra [5] has then analyzed *transitions (commands) as predicate transformers*. Programming by state machines goes far back and deep into the roots of semantics of computation [16], [4]. From this angle, perpendicular to the categorical view, especs can be construed as

- Dijkstra’s language of guarded commands, with
- explicit Floyd-Hoare annotations<sup>2</sup>, and
- procedural encapsulation.

### B.1 States (modes) as specs

Intuitively, a *state* is a snapshot of a process. For a computational process, the idea is that you “freeze” the computer at some point. In that snapshot, each variable of computation has some value; all together, they constitute a mathematical structure, consisting of the operations and constants involved in the computation. This structure is surely a model of any specification, *viz* logical theory specifying the requirements of computation.

<sup>1</sup>An update of a finite number of values, typical for ASMs, corresponds to reinterpreting the function by a case-expression.

<sup>2</sup>not the weakest preconditions

A state of computation can thus be viewed as a model of the specification of that computation. Since this specification is satisfied at *all* states, it displays some set of the *invariants* of computation.

Besides the conditions invariant under the computational steps, some of which are displayed in the specification, each state-as-model satisfies some conditions that are not preserved under the computational steps, and are thus not satisfied at other states. In order to capture a state in a specification tool, we can display some such particular conditions, only satisfied at that state. While we cannot capture state-as-model in the world of specifications, *viz* logical theories describing models, we can *approximate* a state by specifying some of the conditions satisfied at that state, and not at others.

A state of computation can thus be viewed as a specification of which it is a model. This *local* specification extends the *global* specification, consisting of some conditions satisfied throughout computation.

Soundness of this approximation is underlined by the full and faithful embedding of the category  $\text{Mod}(T)$  of models of the spec  $A$  in category  $T/\text{Spec}$  of the refinements of  $T$ . It maps each model  $M$  of  $T$  to theory  $\Theta_M$ , spanned by all formulas satisfied by  $M$ . Obviously, the fact that  $M$  is a model of  $T$  implies that there is an inclusion  $T \hookrightarrow \Theta_M$ . This is the image of  $M \in \text{Mod}(A)$  in  $T/\text{Spec}$ . In fact, models of  $T$  can be identified with the maximal refinements of  $T$ , i.e. the interpretations  $T \rightarrow \Theta$  which cannot be nontrivially refined. This corresponds to the well known correspondence of models of a theory with the maximal filters<sup>3</sup> containing that theory.

**Procedures.** Specs just APPROXIMATE states/models. To support our view of computation as a path from state to state, we idealize stads, and imagine that they are actual states, ignoring the fact that they are not really snapshots, but can extend in time, and hide a lot of action.

An orthogonal view of stads is that they are the basic *encapsulation* mechanism in especs: a way to subdivide an undetermined program into procedures. If we don’t ignore, but *emphasize* the fact that stads are extended in time, then they become a natural subdivision of program into procedures.

A mode can thus be construed as a state, or as a procedure template.

Espec refinement becomes procedural refinement.

### B.2 Transitions as reverse interpretations

Let  $M$  and  $N$  be two states in computation. A computation step  $M \xrightarrow{c} N$  derives the values of the variables at  $N$  from the values of the variables at  $M$ . In other words, the specification of  $N$  is assigned an interpretation in terms of the specification of  $M$ . A *transition*  $M \xrightarrow{c} N$  is an

<sup>3</sup>classically, this is ultrafilters; intuitionistically, it’s perfect filters

interpretation  $\Theta_M \xleftarrow{u} \Theta_N$ .

So the idea that states and transitions are specs and interpretations<sup>op</sup> suggests that *state machines* are diagrams in  $\text{Spec}^{op}$ . But, as explained above, if we recognize a theory  $T_A$ , describing some conditions invariant under all state changes in computations of a state machine  $A$ , then the states of this state machine will not be just any specs, but will come as refinements of  $T_A$ . In other words, each state  $M$  of  $A$  comes with an interpretation  $T_A \rightarrow M$ . So a state of  $A$  is not a mere object of  $\text{Spec}$ , but an object of the co-slice category  $T_A/\text{Spec}$ , of arrows from  $T_A$  in  $\text{Spec}$ .

#### Global spec: immutable data

A spec of a recursive function is not the same thing as the corresponding program, e.g. presented as espec.

```

spec Mult
  import Nat
  op multOp: Nat, Nat  $\rightarrow$  Nat

  ax multOp(0, n) = 0
  ax multOp(m+1, n) = multOp(m, n) + n
  ;: ( $\forall m, n$ ) implicit
end-spec

espec multProc (in m, n:Nat) (out z:Nat)
  import Nat

  prog

    step
      z := 0
    cond
      m = 0
    end-step

    step
      z := multProc(m-1, n) + n
    cond
      ( $\exists m'$ ) m = m'+1
    end-step

  end-prog
end-espec

```

The main point of the difference is that the operation `multOp` is immutable, together with all of the signature displayed in the spec, whereas the procedure (function) `multProc` is mutable: each recursive call updates some of its values: the procedure call in the step is replaced by a copy of the state machine of the same `espec mult`, initialized with (*m*-1, *n*).

### B.3 Guards as instantiations

If states are models, a transition between the specs  $S \rightarrow T$  should thus correspond to a functor  $\text{Mod}(S) \rightarrow$

$\text{Mod}(T)$ . This functor should, moreover, be effectively computable, by updating the terms of  $T$  in terms of  $S$ . When  $S \xrightarrow{c} T$  is an interpretation  $S \xleftarrow{c} T$ , this is indeed the case: the interpretation induces the functor  $c^* : \text{Mod}(S) \rightarrow \text{Mod}(T)$  which does the job: if  $c$  interprets each term of  $f \in T$  by some term  $t = c(f) \in S$ , we can take the meaning of  $t$  to be the meaning of  $f$ . The result will be a model of  $T$  because of the requirement that  $c$  preserves theorems: all properties of  $f$  provable in  $T$  must be provable about  $t = c(f)$  in  $S$ .

So we have the correspondence

$$\frac{S \xleftarrow{c} T}{\text{Mod}(S) \xrightarrow{c^*} \text{Mod}(T)}$$

$$S \xrightarrow{c} T$$

But what if the transition is *guarded*? The intended meaning of a guarded update  $S \xrightarrow{\Gamma, \zeta} T$  is that it should be executed only if the condition  $\Gamma \in S$  is satisfied. More precisely, if we are at the state  $S$ , i.e. given a model  $M \in \text{Mod}(S)$ , we should

- execute  $c$  if  $M$  also satisfies  $\Gamma$ , i.e.  $M \in \text{Mod}(S \wedge \Gamma)$ , and
- do nothing otherwise.

This means that a transition from  $S$  to  $T$ , guarded by  $\Gamma$ , actually corresponds to a functor  $\text{Mod}(S \wedge \Gamma) \rightarrow \text{Mod}(T)$ . The correspondence is now

$$\frac{S \rightarrow S \wedge \Gamma \xleftarrow{c} T}{\text{Mod}(S) \leftarrow \text{Mod}(S \wedge \Gamma) \xrightarrow{c^*} \text{Mod}(T)}$$

$$S \xrightarrow{\Gamma, c} T$$

The category  $\text{Spec}_\perp$  of specs and *guarded transitions* consists of

*objects*: are specs  $S, T \dots$

*morphisms*: a morphism  $S \xrightarrow{\Gamma, \zeta} T$  is an opspan

$$S \rightarrow S \wedge \Gamma \xleftarrow{c} T$$

in the category  $\text{Spec}$ , where the first component  $S \rightarrow S \wedge \Gamma$  is identity on the signature;

The composition is

$$\frac{R \xrightarrow{\Phi, d} S \quad S \xrightarrow{\Gamma, \zeta} T}{R \xrightarrow{\Phi \wedge d(\Gamma) + doc} T}$$

The identities are the identity opsans.

This category, and the general construction on which it is based, are described in [14].

## B.4 Summary

A state machine  $A$  is thus a diagram in the category  $(T_A/\text{Spec})^{op}$ , which fully embeds, and in a fundamental sense generalizes the category  $\text{Mod}(T_A)$  of  $T_A$ -models.

Thus, the basic ideas of especs can be summarized by the following statements:

states (modes)	$\equiv$	specs (theories)
transitions (updates)	$\equiv$	(interpretations) <sup>op</sup>
guards (conditions)	$\equiv$	instantiations
state machine	$\equiv$	diagram of specs and guarded updates

## C. Example

As a simple introductory example, we will present the greatest-common-denominator using especs.

The following espec  $GCD\_base$  defines the basic operators and axioms that form the basis for subsequent refinements of the gcd specification.

```

espec  $GCD\_base$  is
  spec
    op  $divides : PosNat * PosNat \rightarrow Boolean$ 
    axiom  $gcd\_spec$  is
       $gcd(x, y) = z$ 
       $\Rightarrow (divides(z, x) \wedge divides(z, y)$ 
         $\wedge \forall w: PosNat$ 
           $(divides(w, x) \wedge divides(w, y)$ 
             $\Rightarrow w \leq z))$ 
    end-spec
  end-espec

```

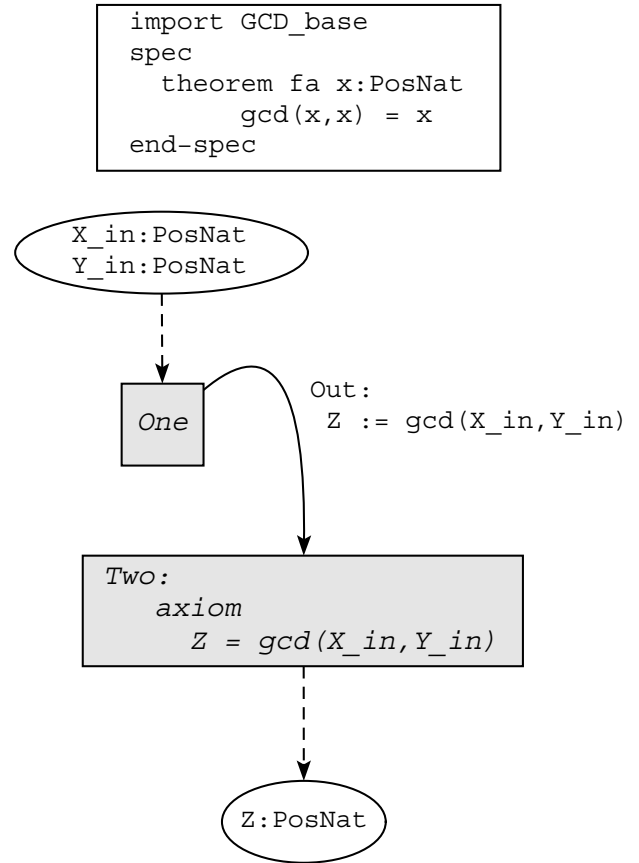
The espec in this case only consists of the “spec” part, the logical specification. The state machine itself is in this case trivial, with a one state and one transition.

The espec  $GCD\_0$  defines the gcd of two natural numbers and the state machine specifies the required behavior of a greatest-common-denominator computation.

$GCD\_0$  is shown in a graphical notation where the white box on top represents the spec-part, the grey boxes the stads, and the arrows the steps. The oval nodes contain input and output parameters; an arrow from a parameter node  $p$  to a stad node  $s$  expresses two things: the parameters contained in  $p$  are input parameters, and  $s$  is an initial stad. Analogously, an arrow from a stad node  $s$  to a parameter node  $p$  expresses that  $s$  is a final stad and that the parameters defined in  $p$  are output parameters of the espec. The contents of the stad boxes are the local specs, the labels of the arrows are the update rules representing the transformation from one stad to its successor. In general, a guard term can also be part of a step label specifying the condition under which the step is activated in a given state. Initial states are represented by node with no incoming step edges; final state by nodes with no outgoing edges.

Espec  $GCD\_1$ , below, refines  $GCD\_0$ . The state graph expresses the classical GCD algorithm, which might have

$E\text{Spec} : GCD\_0$



been generated by a design tactic.  $GCD\_1$  extends the logical spec of  $GCD\_0$  with an axiom that serves as the key loop invariant of the program:  $X$  and  $Y$  change under the transitions, but always so that their GCD is the same as the GCD of the input values  $X\_in$  and  $Y\_in$ .

The correctness of this refinement essentially amounts to showing that the  $GCD\_0$  axiom in stad **Two** translates to a theorem in the  $GCD\_1$  Stad **Two**. Note that the possible behaviors of  $GCD\_1$  are a proper subset of the possible behaviors of  $GCD\_0$ .

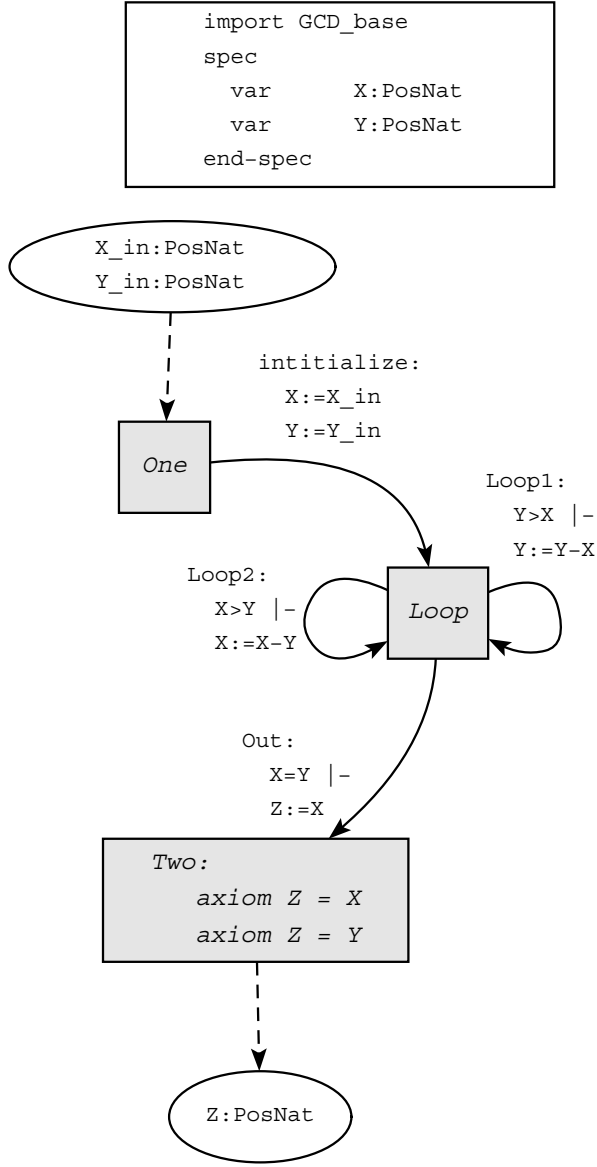
## D. Especs defined

Towards a formal definition of especs, we begin from a category  $\text{Spec}$  of specifications, e.g. as developed through [3], [6], [18].

An evolving specification  $A$  consists of

- a static, *global specification*  $spec_A$ , capturing the globally visible *immutable* structure,
- a program, or state machine  $prog_A$ , represented as a diagram of local specifications, or *state descriptions*, capturing the structure and properties visible at some states and not at others.

*Definition II.1:* A *graph*  $s$  consists of two sets  $edge_s$  and  $node_s$ , and two functions,  $dom_s$  and  $cod_s$  from  $edge_s$  to



node<sub>s</sub>.

A *shape* is a graph  $s$ , which is moreover

- reflexive, in the sense that there is a function  $\text{id}_s : \text{node}_s \rightarrow \text{edge}_s$ , which assigns a distinguished loop to each node;
  - distinguished initial node  $i$ , and a set  $O$  of final nodes  $o$ ;
- Together with the morphisms preserving all displayed structure, shapes form the category **Shape**.

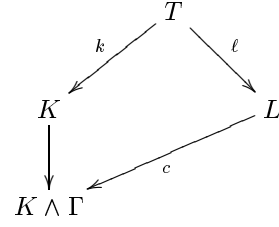
**Definition II.2:** Given a spec  $T$ , the category  $\text{Ext}_T$  is defined as follows:

*objects* are the extensions of  $T$ , i.e. the morphisms out of  $T$  in the category **Spec**;

*morphisms* are the guarded transitions preserving  $T$ :

Given extensions  $k : T \rightarrow K$  and  $\ell : T \rightarrow L$  an

$\text{Ext}_T$ -morphism is a guarded transition  $K \xrightarrow{\Gamma, \varsigma} L$ , making the following diagram commute.



**Remark.** Equivalently,  $\text{Ext}_T$  could be defined as the full subcategory, spanned in the slice category  $\text{Spec}_\perp / T$  by the unguarded transitions into  $T$ . Recall that  $\text{Spec}_\perp$  is the category of specs and guarded transitions, described in section II-B.3, and in more detail in [14]. The unguarded transitions into  $T$  are, of course, just the **Spec**-morphisms out of  $T$ , i.e. the interpretations of  $T$ .

**Definition II.3:** An *evolving spec*, or **espec**  $A$  consists of

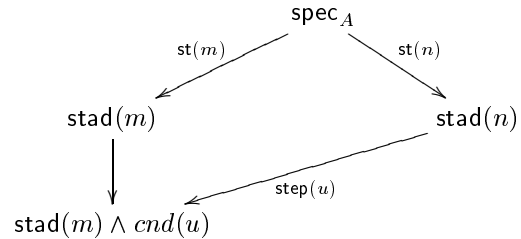
- a spec  $\text{spec}_A$ , and
- a program  $\text{prog}_A$ , presented by
  - a shape  $\text{shape}_A$ ;
  - a reflexive graph morphism (diagram)  $\text{st}_A : \text{shape}_A \rightarrow \text{Ext}_A = \text{Ext}_{\text{spec}_A}$

**Comments, conventions and notations.** The last item is, of course, based on the obvious fact that every category can be viewed as a reflexive graph, with the identities as the distinguished self-loops.

If  $n$  is a node of  $\text{shape}_A$ , the codomain of  $\text{st}_A(n)$  is written as  $\text{stad}_A(n)$ . If  $u : m \rightarrow n$  is an edge of  $\text{shape}_A$ , its image  $\text{st}_A(u)$  is usually written as  $\text{step}_A(u)$ . In summary,

- $\text{stad}$  assigns to each shape-node  $n$  a *state description*  $\text{stad}(n)$ , which comes with the interpretation  $\text{st}_A(n) : \text{spec}_A \rightarrow \text{stad}(n)$ ;
- $\text{step}$  assigns to each shape-edge  $u : m \rightarrow n$ 
  - the guard  $\text{cond}(u)$ , which is a formula in the signature of  $\text{stad}(m)$ , and
  - the interpretation  $\text{step}(u) : \text{stad}(m) \wedge \text{cond}(m) \leftarrow \text{stad}(n)$

which together make the following diagram commute.



In order to make **especs** into a usable development framework, a lot of syntactic sugar needs to be added, to abbreviate the various frequently used structures and patterns. The first such addition are the *input and output interfaces*,

presented as parameter subtheories<sup>4</sup>  $\text{Xi} \leftrightarrow \text{stad}(\text{i})$  and  $\text{Xo} \leftrightarrow \text{stad}(\text{o})$ , of the initial and the final states, respectively. The input and the output interfaces are usually written  $\text{stad} \langle \text{name} \rangle \text{init}[\text{Xi}]$  and  $\text{stad} \langle \text{name} \rangle \text{fin}[\text{Xo}]$ .

### III. REFINEMENTS

If an espec consists of a spec and a program, then an espec refinement must consist of a spec refinement and a program refinement.

Spec refinement is a fairly simple concept: it interprets the terms of the source spec in terms of the target spec, making sure that it has at least as much structure as the former, and possibly more.

But what is program refinement? In practice, programmers refine by running code some abstract templates, which specify structures and components that will be used. E.g., a virtual function, a class template, or a module with some unimplemented components, are all abstract programs, that need to be refined. Program refinement is the further implementation of unimplemented components.

Typically, an abstract program encapsulates unimplemented parts into a module template; the corresponding description specifies computation that needs to be subsequently implemented.

In especs the encapsulation mechanism is realized by MODES. They approximate states, but also encapsulate modules, or procedures.

The refinement “opens up” our abstract state, as a black box where the unimplemented computation was hidden, and replaces it by a state machine, which performs the needed computation. Of course, some details of this new state machine may still be unimplemented, and some of its states may need to be further refined.

State is here thus better understood as a *procedure template*. Espec refinement is procedural refinement.

*Definition III.1:* Given especs  $A$  and  $B$ , a refinement  $f : A \rightarrow B$  consists of:

$$\begin{array}{c}
 A = \langle \text{spec}_A, \text{shape}_A \xrightarrow{\text{st}_A} \text{Ext}_A^{\text{op}} \rangle \\
 \downarrow f \\
 B = \langle \text{spec}_B, \text{shape}_B \xrightarrow{\text{st}_B} \text{Ext}_B^{\text{op}} \rangle
 \end{array}
 \quad
 \begin{array}{ccc}
 \text{shape}_A & \xrightarrow{\text{st}_A} & \text{Ext}_A^{\text{op}} \\
 \uparrow f_1 & \searrow f_2 & \uparrow f_0^* \\
 \text{shape}_B & \xrightarrow{\text{st}_B} & \text{Ext}_B^{\text{op}}
 \end{array}$$

- a *structure map* (or translation)  $f_0$
- a *behavior map* (or simulation)  $f_{\text{prog}} = \langle f_1, f_2 \rangle$ , where
  - $f_1$  is a reflexive graph morphism, preserving the initial and the final nodes,
  - $f_2$  is  $\text{spec}_A$ -preserving natural transformation; this naturality and preservation amount to the commutativity of Figure 1 for every  $v : k \rightarrow \ell$  in  $\text{shape}_B$  (see notes

<sup>4</sup>By definition, the parameter  $X \subseteq S$  of a parametric specification  $S[X]$  can be freely instantiated, without causing any inconsistencies in the parameterized specification [15]. This also captures the idea of interface.

below), where we have used the following abbreviations:  $f_1(k) = k'$ ,  $f_1(v) = v'$ , and  $f_1(\ell) = \ell'$ .

- Together,  $f_0$  and  $f_{\text{prog}}$  must also satisfy the *guard condition*: for every edge  $v : k \rightarrow \ell$  in  $\text{shape}_B$  and edge  $v' = f_1(v)$  in  $\text{shape}_A$

$$\text{stad}_B(k) \wedge \text{cond}_B(v) \vdash f_0(\text{cond}_A(v'))$$

- The inverse-image functor  $f_0^*$  acts on the category of extensions:  $f_0^*(e) = e \circ f_{\text{spec}}$ .

Clearly, especs and refinements form a category, which we shall denote ESPEC.

**Intuition.** The last diagram tells that the components of  $f_2$  coherently extend  $f_0$  from the global specs  $\text{spec}_A$  and  $\text{spec}_B$  to their extensions  $\text{stad}_A$  and  $\text{stad}_B$ . Just like  $\text{spec}_B$  refines  $\text{spec}_A$  because it proves all formulas in the image  $f_0[\text{spec}_A]$ , each  $\text{stad}_B(n)$  refines  $\text{stad}_A(f_1(n))$  because it proves all formulas in the image  $f_2(n)[\text{stad}_A(f_1(n))]$ . The *structural* refinement is thus extended from  $f_0 : \text{spec}_A \rightarrow \text{spec}_B$  to  $f_2 : \text{stad}_A \rightarrow \text{stad}_B$ . Its naturality ensures that each transition  $\text{step}_B(v)$  of  $B$  extends the transition  $\text{step}_A(f_1(v))$  of  $A$ .

The guard condition ensures that every behavior of  $B$  maps to a behavior of  $A$ . There are stronger versions of the guard condition that also ensure that  $B$  simulates all of  $A$ 's behaviors, and others that eliminate nondeterminism. Rather than commit to one such definition, we use several, but the guard condition above is sufficient for the purposes of this paper.

Let us return to the example in Section 2. In the refinement from GCD-0 to GCD-1,  $f_0$  is a simple inclusion, and  $f_1$  is given by the stad map

$$\begin{array}{l}
 \text{One} \mapsto \text{One} \\
 \text{Loop} \mapsto \text{One} \\
 \text{Two} \mapsto \text{Two}
 \end{array}$$

and the step map

$$\begin{array}{l}
 \text{initialize} \mapsto \text{id}_{\text{One}} \\
 \text{Loop1} \mapsto \text{id}_{\text{One}} \\
 \text{Loop2} \mapsto \text{id}_{\text{One}} \\
 \text{Out} \mapsto \text{Out}
 \end{array}$$

Three of the steps map to the identity step on stad One in GCD-0 because they only change the local variables  $X$  and  $Y$ , corresponding to identity steps in GCD-0 (sometimes called stuttering steps). Checking the components of the natural transformation is straightforward – the proof obligations include showing that  $f_2(k)$  is a translation for all nodes  $k$  in  $\text{shape}_A$ ; e.g. that the axioms of stad One in GCD-0 translate to theorems in stad Loop in GCD-1. Checking the guard condition is also straightforward; e.g. for step Loop1 in GCD-1, the guard condition instantiates to

$$\text{Loop} \wedge X > Y \vdash \text{true}$$

in GCD-0.

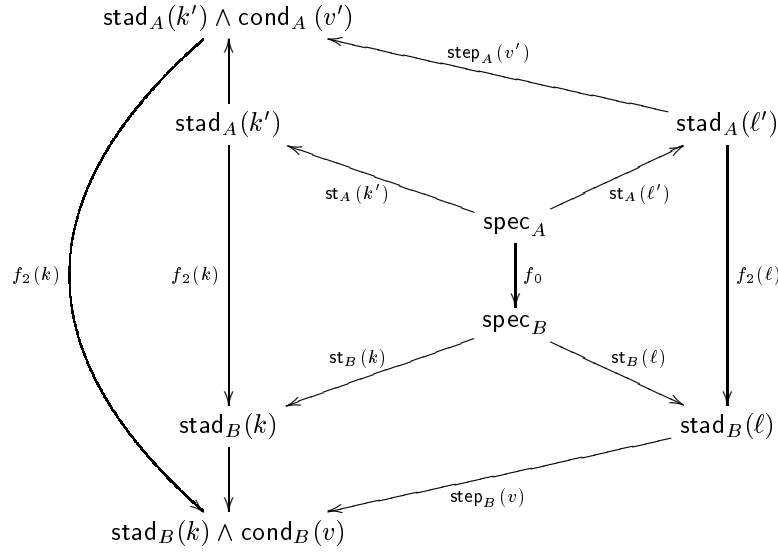
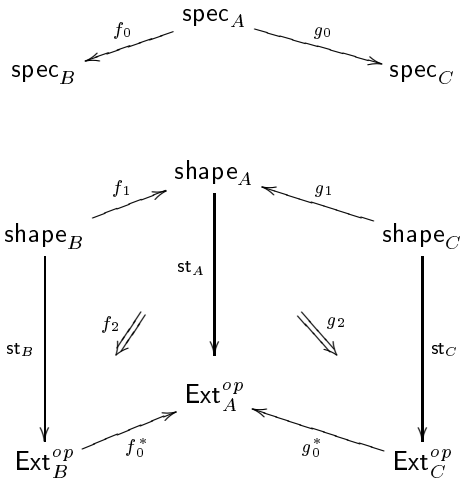


Fig. 1. Naturality Condition of a Refinement

#### IV. COLIMITS

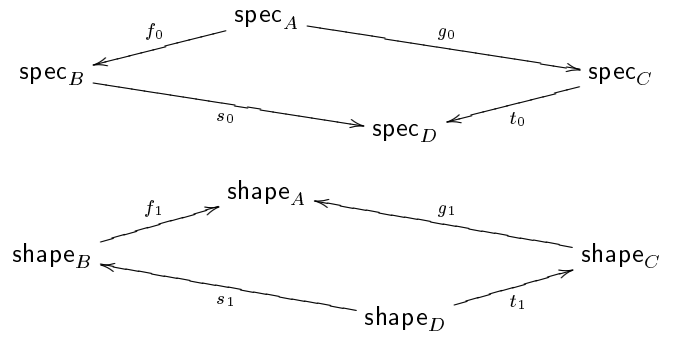
Composition of especs is carried out by the colimit operation. Colimits in ESPEC are constructed from the colimits in SPEC, the limits in SHAPE, plus some wiring to connect them in CAT. First of all, recall that all colimits can be derived from the initial object and the pushouts. Of course, the initial espec consists of the empty spec, and a one-state-one-step program (with the state represented by the empty spec).

To describe the pushout of especs, suppose we are given a span of especs

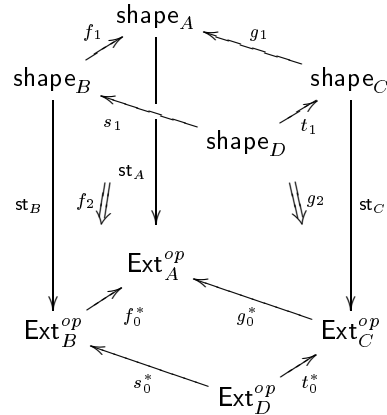


To compute the pushout, we first compute the correspond-

ing pushout of specs and the pullback of shapes.

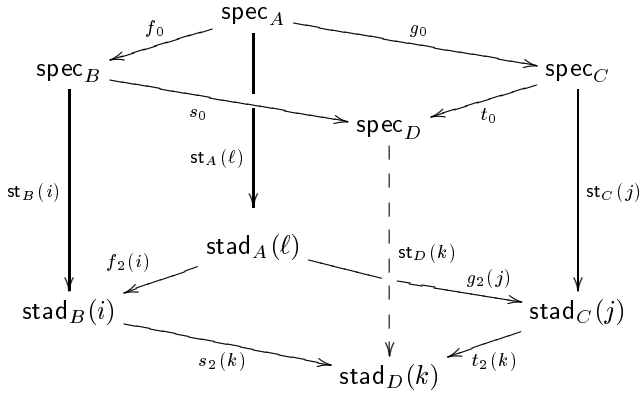


It is easy to see that  $M : \text{Spec}^{op} \rightarrow \text{Cat}$  maps the upper pushout to the pullback at the bottom of the induced cube.



If  $f_2$  and  $g_2$  were identities, i.e. if the two back faces of the cube were commutative, the fact that the bottom face is a pullback would induce a functor  $\text{st}_D : \text{shape}_D \rightarrow$

$(\text{spec}_D/\text{Spec})^{op}$ . Since they are not, this functor must be constructed taking  $f_2$  and  $g_2$  into account. The image  $\text{st}_D(k)$  of a node  $k$  of  $\text{shape}_D$  is now obtained as the unique arrow from the pushout at the top to the pushout at the bottom of the following cube.



Since  $\text{shape}_D$  is the pullback of  $f_1$  and  $g_1$ , the node  $k$  corresponds to a pair  $\langle i, j \rangle$  of the nodes from  $\text{shape}_B$  and  $\text{shape}_C$ , identified in  $\text{shape}_A$  as the node  $\ell = f_1(i) = g_1(j)$ . Of course,  $i = s_1(k)$  and  $j = t_1(k)$ .

This construction gives the node part  $\text{stad}_D$  of  $\text{st}_D : \text{shape}_D \rightarrow \text{Ext}_D^{op}$ , as well as the components of  $s_2$  and  $t_2$ . The arrow part  $\text{step}_D$  is induced by the fact that the bottom of the cube is a pushout, using the naturality of  $f_2$  and  $g_2$ . This also yields the naturality of  $s_2$  and  $t_2$ . Finally we construct the guards for the edges of  $\text{shape}_D$ . Given an edge  $w : k \rightarrow k'$  of  $\text{shape}_D$  define

$$\begin{aligned} \text{cond}_D(w) &= s_0(\text{cond}_B(s_1(w))) \\ &\quad \wedge t_0(\text{cond}_C(t_1(w))) \end{aligned}$$

A proof that this construction yields an  $\text{espec}$  with the desired universal properties may be found in [13].

**Explanation.** The pushout of specs is clear enough: the languages get joined together, and identified along the common part. The pullback of shapes produces the *parallel composition* of the behaviors they present. This is particularly easy to see for products, i.e. pullbacks over the trivial shape, with a single node. For example, a product of any shape with the two-node shape  $\bullet \rightarrow \bullet$  consists of the cylinder, with the two copies of shape, and each two of their corresponding nodes connected by an edge. A product with the three-node shape  $\bullet \rightarrow \bullet \rightarrow \bullet$  consists of three copies, similarly connected.

In general, the product of any two shapes  $\text{shape}_B$  and  $\text{shape}_C$  can be envisaged by putting a copy  $S_n$  of  $\text{shape}_B$  at each node  $n$  of  $\text{shape}_C$ , and then expanding each edge  $m \xrightarrow{u} n$  of  $\text{shape}_C$  into a cylinder from  $S_m$  to  $S_n$ , i.e. a set of parallel edges, connecting the corresponding nodes. The initial node is the pair of the initial nodes of  $\text{shape}_B$  and  $\text{shape}_C$ , whereas the final nodes are the pairs of final nodes.

In the resulting shape  $\text{shape}_B \times \text{shape}_C$ , each edge either comes from a copy of  $\text{shape}_B$  placed on a node of  $\text{shape}_C$ , or from an edge of  $\text{shape}_C$  copied to connect two particular copies of a node of  $\text{shape}_B$ ; so it is either in the form  $\langle \text{node of } \text{shape}_C, \text{edge of } \text{shape}_B \rangle$ , or in the form  $\langle \text{node of } \text{shape}_B, \text{edge of } \text{shape}_C \rangle$ . A moment of thought shows that each path through  $\text{shape}_B \times \text{shape}_C$  corresponds to a shuffle of a path through  $\text{shape}_B$ , and a path through  $\text{shape}_C$ ; and that every such path comes about as a unique path in  $\text{shape}_B \times \text{shape}_C$ . In this sense,  $\text{shape}_B \times \text{shape}_C$  is the *parallel composition* of  $\text{shape}_B$  and  $\text{shape}_C$ .

A pullback extracts a part of such product, identified by a pair of shape morphisms  $\text{shape}_B \rightarrow \text{shape}_A \leftarrow \text{shape}_C$ . Since the initial node must be preserved, the initial node of the product will surely be contained in the pullback. The set of final nodes may be empty in general.

For each pair of nodes  $\langle i, j \rangle$ , contained in the pullback  $\text{shape}_D$  as the node  $k$ , the corresponding state description is constructed as the pushout  $\text{stad}_D(k)$  of  $\text{stad}_B(i)$  and  $\text{stad}_C(j)$  on the above diagram. As a theory, this state description may be inconsistent. Indeed, if  $B$  and  $C$  are not independent, but have a shared part  $A$ , their parallel composition may be globally inconsistent, in the sense that  $\text{spec}_D$  may be inconsistent; or some of the pairs of states that may come about in shuffling their computation paths may be inconsistent, which makes such paths computationally impossible. Inference tools can be used to eliminate inconsistent/unreachable stads from the colimit  $\text{espec}$ .

Despite the apparent complexity of the description of the colimit, the actual computation is relatively simple. There are just three steps:

- pullback of shapes;
- pushout of specs; (the guards can be directly computed at this point)
- the pushout extensions of stads and steps.

The first two steps are simple and well known. The third one amounts to computing a pushout of theories for each stad, and using the universality of each such pushout to generate the steps from it –  $\text{Espec}$ 's colimit algorithm returns both the cocone and a generator of translations that witness the universality of the apex.

## V. EXAMPLE: AN ELEVATOR SPECIFICATION

In this section, we will illustrate some of the concepts introduced above using an example of a simple elevator system.

The spec "Floor" contains an abstraction of the representation of floors in the elevator system; the axioms describe the meaning of the declared operators.<sup>5</sup>

```
spec Floor is
  sort Floor
  op firstFloor : Floor
```

<sup>5</sup>For the sake of brevity, the obvious axioms specifying the meaning of the infix operators have been omitted.

```

op uppermostFloor : Floor
op lowermostFloor : Floor
op succFloor : Floor → Floor
op predFloor : Floor → Floor

op > : Floor * Floor → Boolean
op < : Floor * Floor → Boolean
op ≥ : Floor * Floor → Boolean
op ≤ : Floor * Floor → Boolean

axiom
  uppermostFloor ≥ lowermostFloor
axiom
  (firstFloor ≤ uppermostFloor) ∧
  (firstFloor ≥ lowermostFloor)
axiom
  ∀(f1 : Floor, f2 : Floor)
    (succFloor(f1) = f2) ⇒ (f1 < f2)
axiom
  ∀(f1 : Floor, f2 : Floor)
    (predFloor(f1) = f2) ⇒ (f1 > f2)
axiom
  ∀(f : Floor) (f ≥ lowermostFloor) ∧
    (f ≤ uppermostFloor)
end-spec

```

The following espec represents the common data used in the different parts of the model. It contains basic sorts, ops and constant definitions that are used throughout the specification. The meaning should be self-explanatory.

```

espec elevData is
spec
  import Floor

  sort Direction = | NotMoving | GoingUp | GoingDown
  sort Request = | NoRequest | FloorRequest Floor

  op currentFloor : Floor
  op nextFloor : Floor
  op nextRequest : Request
  op direction : Direction

end-spec
end-espec

```

The "elevMovement" espec contains the logic determining the behavior of the elevator depending on the stop requests.

```

espec elevMovement is
  import elevData

  The spec-part of the "elevMovement" espec defines two
  auxiliary operators for determining whether the next
  request is a request for going up or for going down.

spec (* of espec elevMovement *)
  op isGoingUpRequest : Request → Boolean
  def isGoingUpRequest(req) =
    case req of
      | NoRequest → false
      | FloorRequest n → (n > currentFloor)
  op isGoingDownRequest : Request → Boolean
  def isGoingDownRequest(req) =
    case req of

```

```

      | NoRequest → false
      | FloorRequest n → (n < currentFloor)
end-spec

```

It is assumed that initially the elevator is at the first floor and is not moving. The following mode is entered initially and the values of "currentFloor" and "direction" are set accordingly.

```

initial stad InitElevator (* of espec elevMovement *)
is
  step Initialize : → AtFloor do
    currentFloor := firstFloor
    direction := NotMoving
  end-step
end-stad

```

The stad "AtFloor" described the state in which the elevator is not moving and standing at the floor "currentFloor". The stad will be changed to "Moving", if a request event is detected, which is modeled by the (external) variable "nextRequest". In this espec, we do not specify the concrete representation of the "nextRequest" function; this will be done by a separate espec. Depending on whether the next request – which is usually triggered by either pressing a button inside the elevator or outside on a certain floor – sends the elevator to a floor above or below the current position, the direction is set to "GoingUp" or "GoingDown" accordingly.

```

stad AtFloor (* of espec elevMovement *)
is
  axiom direction = NotMoving
  step StartGoingUp : → Moving
    when isGoingUpRequest(nextRequest) do
      nextFloor := succFloor(currentFloor),
      direction := GoingUp
  end-step
  step StartGoingDown : → Moving
    when isGoingDownRequest(nextRequest) do
      nextFloor := predFloor(currentFloor),
      direction := GoingDown
  end-step
end-stad

```

The stad "Moving" describes the behavior of the elevator when it is moving from one floor to the other. The stad will only be changed back to "AtFloor", if a floor is reached that matches the next request. Otherwise the next floor is set according to the current direction.

```

stad Moving (* of espec elevMovement *)
is
  axiom
    ((direction = GoingUp)
     ∧ (direction = GoingDown))
     ∧ (nextRequest ≠ NoRequest)

  op getFloor : Request → Floor
  def getFloor(FloorRequest n) = n

  step Stop : → AtFloor
    when getFloor(nextRequest) = nextFloor do
      currentFloor := nextFloor
  end-step

  step GoUp : → Moving

```

```

    when direction = GoingUp do
      nextFloor := succFloor(nextFloor)
    end-step

    step GoDown: → Moving
      when direction = GoingDown do
        nextFloor := predFloor(nextFloor)
      end-step
    end-stad

end-espec (* elevMovement *)

```

### Refinement

The refinement of the above specification adds the modeling of opening and closing the elevator door when it stops at a floor responding to a request. The refined espec "elevMovementRefined" imports the espec "elevMovement" meaning that both the spec part and the state-machine part is imported.

```

espec elevMovementRefined is
  import elevMovement

```

In the spec part of the refined espec the sort and variable used to model the elevator door actions are declared:

```

spec (* of espec elevMovementRefined *)
  sort DoorStatus = | Open | Closed
  op door : DoorStatus
end-spec

```

The refined espec doesn't introduce new modes, but it refines the imported ones. The following stad description adds new conditions to the step guards of the steps "StartGoingUp" and "StartGoingDown": these steps only "fire", if the door is closed. The additional step takes care of the state when the door is actually closed and sets the variable "door" accordingly. In subsequent refinement steps the guard of this step might be augmented with predicates modeling the fact that the door has to stay open for a certain time period.

```

stad AtFloor (* of espec elevMovementRefined *)
is
  step StartGoingUp: when door = Closed
  end-step
  step StartGoingDown: when door = Closed
  end-step

  step CloseDoor: → AtFloor when door = Open do
    door := Closed
  end-step
end-stad

```

Analogously, the guards of the steps leaving stad "Moving" are extended with the conditions; the stad "AtFloor" is only reached from step "Stop", if the door is open, which has been done in the newly introduced step "OpenDoor".

```

stad Moving (* of espec elevMovementRefined *)
is
  step GoUp: when door = Closed
  end-step
  step GoDown: when door = Closed

```

```

end-step

step Stop: when door = Open
end-step

step OpenDoor: → Moving
  when ( getFloor(nextRequest) = currentFloor )
    ∧ ( door = Closed ) do
    door := Open
  end-step
end-stad

end-espec (* espec elevMovementRefined *)

```

The refinement morphism is then given by the following stad map:

```

InitElevator ↦ InitElevator
AtFloor ↦ AtFloor
Moving ↦ Moving

```

and the step map

```

Initialize ↦ Initialize
StartGoingUp ↦ StartGoingUp
StartGoingDown ↦ StartGoingDown
GoingUp ↦ GoingUp
GoingDown ↦ GoingDown
OpenDoor ↦ idMoving
CloseDoor ↦ idAtFloor

```

Further refinements could be introduced by composition, i.e. using the espec colimits. For instance, to take into account the possibility that, while the elevator is going up, a request comes in for it to go down, one could write a one-state espec, connecting the "nextRequest" variable with newly introduced "button" variables. The colimit construction would then combine the new espec with the "elevMovementRefined" espec.

## VI. CONCLUSION

Component-based software development is a widely used technique, which has proven to be useful and efficient in many different settings. However, most of the existing component-based approaches have decisive shortcomings: they only handle implementation components, so that composition is defined only in terms of external compatibility; abstract specifications – if they exist – are not taken into account. Architectural approaches address this issue and define composition not on the implementation but on the design level. They usually define their own architectural description language, which represent composition in terms of signature matching of the import/export services of the components. In most cases, only syntactic information is used to decide, whether a given configuration is valid or not. The behavior of particular components is usually not a part of such frameworks, but deferred to the extensions.

Especs go a step further: they define a formal framework for specifying structure and behavior of components, for refining, composing and transforming them. Specifications

are expressed in a categorical framework based on the well-known category of specifications. Functionality and behavior of a component can be specified using algebraic specifications and/or control flow graphs. Refinement is formally expressed in the automatically supported categorical framework. Component composition is captured by the categorical colimits of specs, thus extending the well-known correspondence of specification composition and colimits.

Especs thus provide a formal framework to express all relevant aspects of component-based software development. When sufficiently determined, composite components and systems can be automatically calculated, rather than put together by hand. This opens a new perspective on how software components can be used to develop safety-critical systems, as safety properties of the composite can be derived from the safety properties of the components. On the other hand, dealing with the noncompositional properties, that may emerge or vanish in development, is also facilitated by the provided semantical and formal support. Basic transformation techniques can be used to automatically generate executable code for the composed system. The concept of specification-carrying-components is thus resurrected by specs in its original meaning, because components carrying their (e)specifications along can be combined with a greater confidence, and less effort.

#### REFERENCES

- [1] S. Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.
- [2] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, Berlin, 1985.
- [3] R.M. Burstall and J.A. Goguen. Putting together theories to make specifications. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 1045–1048. Cambridge, UK, 1977.
- [4] W.P. de Roever. Dijkstra’s predicate transformer, non-determinism, recursion, and terminations. In *Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 472–481. Springer-Verlag, Berlin, 1976.
- [5] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.
- [6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification, volumes 1 and 2*, volume 6, 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985, 1990.
- [7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [8] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [9] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
- [10] Martin Odersky. Programming with variable functions. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 105–116. ACM, 1998.
- [11] Martin Odersky. Objects + views = components. In Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors, *Proceedings of the International Workshop on Abstract State Machines (ASM2000)*, volume 1912 of *LNCS*. Springer, 2000.
- [12] D. L. Parnas. The use of precise specifications in the development of software. In *Proceedings of IFIP Congress 1977*. North-Holland Publishing Company, 1977.
- [13] Dusko Pavlovic. Epoxi. Technical report, Kestrel Institute, January 2001.
- [14] Dusko Pavlovic and Douglas R. Smith. Guarded transitions in evolving specifications. Technical report, Kestrel Institute, February 2002.
- [15] Duško Pavlović. Semantics of first order parametric specifications. In J. Woodcock and J. Wing, editors, *Formal Methods ’99*, volume 1708, pages 155–172. Springer Verlag, 1999.
- [16] G.D. Plotkin. Dijkstra’s predicate transformers and smyth’s power domains. In D. Bjorner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 527–553. Springer-Verlag, Berlin, 1980.
- [17] Paul Robertson, Howie Schrobe, and Robert Laddaga, editors. *Proceedings of the Workshop on Self-Adaptive Software*, volume 1936. Springer Verlag, 2001.
- [18] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. In G. Levi, H. Ehrig, R. Kowalski, and U. Mantanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT ’87). Volume 1: Advanced Seminar on Foundations of Innovative Software Development 1 and Colloquium on Trees in Algebra and Programming (CAAP’87)*, volume 249, pages 96–110. Springer-Verlag, Berlin, Germany, 1987.
- [19] A. Sünbül. *Architectural Design of Evolutionary Software Systems in Continuous Software Engineering*. Der Andere Verlag, 2001. Ph.D. Thesis, Technical University Berlin, Germany.
- [20] A. M. Turing. Computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1937.
- [21] H. Weber. Continuous engineering of information and communication infrastructures (extended abstract). In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering FASE’99 Amsterdam Proceedings*, pages 22–29, spraddr, 1999. spr.
- [22] H. Weber, A. Sünbül, and J. Padberg. Modelling information and communication infrastructures for evolutionary design of business processes using component technologies. Society for Design and Process Science, IEEE International Conference on Systems Integration, June 2000.