Composition and Refinement of Evolving Specifications

Matthias Anlauff and Dusko Pavlovic and Douglas R. Smith[*]

Kestrel Institute,
Palo Alto, California 94304, USA

Abstract. We represent state machines in the category of specifications, where assignment statements correspond exactly to interpretations between theories [6, 9]. However, the guards on an assignment require a special construction. In this paper we raise guards to the same level as assignments by treating each as a distinct category over a shared set of objects. A guarded assignment is represented as a pair of arrows, a guard arrow and an assignment arrow. We give a general construction for combining arrows over a factorization system, and show its specialization to the category of specifications. This construction allows us to define the fine structure of state machine morphisms with respect to guards. Guards define the flow of control in a computation, and how they may be translated under refinement is central to the formal treatment of safety, liveness, concurrency, and determinism.

1 Introduction

1.1 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. Besides 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.

1.2   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 multi-faceted 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 [8]. 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 [15,14] and [16]. In this approach, the development of a piece of software is seen as a continuous circle of engineering/reverse-engineering steps triggered by changing 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.

1.3   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 whenit 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 mainpoint 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 [7], 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 [11].

2   Evolving Specifications

In previous work [9] we introduced *Evolving Specifications* (abbreviated to especs) as a framework for specifying, composing and refining behavior. The point of such a framework is, at the very least, to help us cross the path from ideas to running code. Programming languages are designed to support us at the final

sections of that path. On one hand, *especs are evolving specifications*: diagrams of specs, displaying how the conditions, satisfied by the variables of computation, change from state to state. On the other hand, *especs are specification-carrying programs*: pieces of code, given with some global requirements and invariants, as well as annotated with some local conditions, *state descriptions*, satisfied at some states of computation and not at others. They can be construed as formalized comments, or Floyd-Hoare annotations, but made in to the first-class citizens of code, i.e. available at runtime.

While such global and local specifications of the intent of computation are hard to reconstruct if the design record shave been lost or thrown away, they are *easy to verify* if the design records are carried with the code.

2.1   State machines and algebraic specifications

Originally, state machines were introduced and studied (by Turing, Moore, Mealy, and many others) as abstract, mathematical models of computers. More recently, though, software engineering tasks reached the levels where *practical* reasoning in terms of state machines has become indispensable: designing reactive, hybrid, embedded systems seems unthinkable without the various state modeling tools and languages, like Esterel, or Statecharts. Verifying high assurance systems by model checking is based on such state machine models. Moreover, one could argue that the whole discipline of object oriented programming is essentially a method for efficient management of state in software constructs.

However, there seems to be a conceptual gap between state machines as theoretical versus practical devices. A notable effort towards bridging this gap are Gurevich's Abstract State Machines [4]: on one hand, they are a foundational paradigm of computation, explicitly compared with Turing machines; on the other hand, they have been used to present practically useful programming languages, capturing semantical features of C, Java, and others. However, the absence of powerful typing and structuring (abstraction, encapsulation, composition...) mechanisms makes them unsuitable for the development and management of large software systems.

We wish to investigate a representation of state machines in a framework for large-scale software specification development ("from-specs-to-code"). Previous work at Kestrel Institute has implemented the Specware/Designware framework for the development of functional programs that is based on a category of higher-order logical specifications, composition by colimit, and refinement by diagram morphisms [12, 13]. The current work builds on and extends this framework with behavioral specifications (especs), representing state machines as diagrams of specifications, and again using composition by colimit and refinement by diagram morphism. Related approaches to representing behavior in terms of a category of specifications include [3, 5].

The goal is to build a practical software development tool, geared towards large, complex systems, with reactive, distributed, hybrid, embedded features, andwith high assurance, performance, reliability, or security requirements, all on a clean and simple semantical foundation.

2.2   Evolving Specifications

There are four key ideas underlying our representation of state machines as evolving specifications (especs). Together they reveal an intimate connection between behavior and thecategory of logical specifications. The first two are folklore, the third one is due to Gurevich, and lies in the foundation of his Abstract State Machines [4], and we have contributed the fourth one,as a way to represent state machines and behaviors in a category of specifications.

1. *A state is a model* – A state of computation can be viewed as a snapshot of the abstract computer performing the computation. The state has a set of named stores with values that have certain properties.

2. *A state transition is a finite model change* – A transition rewrites the stored values in the state.

3. *An abstract state is a theory* – Not all properties of a state are relevant, and it is common to group states into abstract states that are models of a theory. The theory presents the structure (sorts, variables, operations), plus the axioms that describe common properties (i.e. invariants). We can treat states as static, mathematical models of a global theory thy_A, and then all transitions correspond to model morphisms. Extensions of the global theory thy_A provide local theories for more refined abstract states, introducing local variables and local properties/invariants.

4. *An abstract transition is an interpretation between theories* – Just as we abstractly describe a class of states/models as a theory, we abstractly describe a class of transitions as an interpretation between theories [6,9]. To see this, consider the correctness of an assignment statement relative to a precondition P and a postcondition Q; i.e. a Hoare triple $P\{x := e\}Q$. If we consider the initial and final states as characterized by theories thy_{pre} and thy_{post} with theorems P and Q respectively, then the triple is valid iff $Q[e/x]$ is a theorem in thy_{pre}. That is, the triple is valid iff the symbol map $\{x \mapsto e\}$ is an interpretation from thy_{post} to thy_{pre}. Note that interpretation goes in the *opposite* direction from the state transition.

   Thus interpretations between theories correspond exactly to (parallel) assignment statements. However, to model guarded assignments, we introduced the notion of guarded interpretation [10]. The idea is that guards and assignments can each be regarded as arrows in their own categories, that together form a factorization system. Theguards areepis andtheassignments are the monics. A general construction that combines an epi and a monic into a single morphism in a cocomplete category is given in [10]. The category of guarded assignments/interpretations falls out as a special case.

The basic idea of especs is to use specifications (finite presentations of a theory) as state descriptions, and touse guarded interpretations to represent transitions between state descriptions.

The idea that abstract states and abstract transitions correspond to specs and interpretations suggests that state machines are diagrams over Spec^{op}. Furthermore, state machines are composed via colimits, and state machines are refined via diagram morphisms [9].

2.3 Example
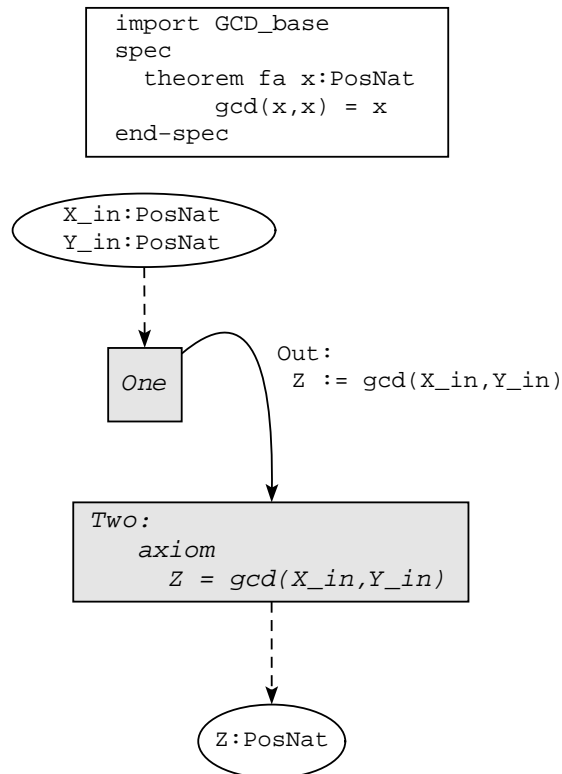
As a simple introductory example, we will present the greatest-common-denominator using especs. The following espec *GCD_base* defines the basic operators and axioms that form the basis for subsequent refinements of the gcd specification.

```
espec GCD_base is
  spec
    op divides: PosNat * PosNat -> Boolean
    axiom gcd_spec is
        gcd(x,y) = z
        => (divides(z,x) & divides(z,y)
            & fa w:PosNat
              (divides(w, x) & divides(w,y)
               => w <= z))
    end-spec
end-espec
```
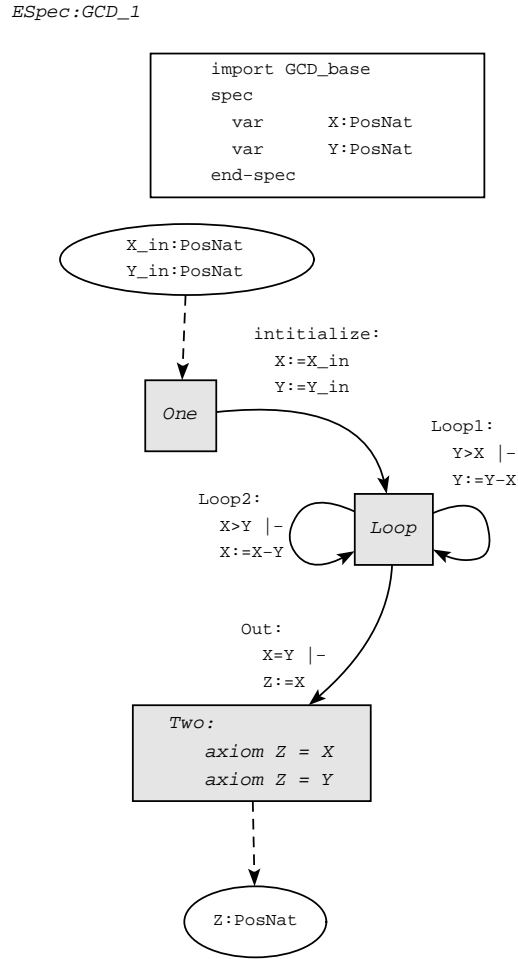
ESpec:GCD_0

```
import GCD_base
spec
  theorem fa x:PosNat
        gcd(x,x) = x
end-spec
```

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.

ESpec:GCD_1

```
        import GCD_base
        spec
          var      X:PosNat
          var      Y:PosNat
        end-spec
```

X_in:PosNat
Y_in:PosNat

intitialize:
    X:=X_in
    Y:=Y_in

One

Loop1:
    Y>X |-
    Y:=Y-X

Loop2:
    X>Y |-
    X:=X-Y

Loop

Out:
    X=Y |-
    Z:=X

Two:
    axiom Z = X
    axiom Z = Y

Z:PosNat

GCD_0 is shown in a graphical notation where the whitebox 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 toits 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 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.

3   Implementation and Application Areas

3.1   Planware

Planware is a domain-specific generator of high-performance schedulers. The system supports a rich modeling language, based on especs, so that users can express complex multi-resource problems. For example, a transportation organization might want a scheduler to simultaneously handle its aircraft, crews,fuel, and airport load/unload facilities. Each resource has its own internal required patterns of behavior and may have dependencies on other resources.

The key to a domain-specific language for specifying scheduling problems lies in abstract but semantically precise behavioral models of tasks and resources. Fortunately we found that especs are well-suited for both.

The semantics of are source is the set of possible behaviors it can exhibit. We treat these behaviors as (temporal) sequences of activities which we model as modes/stads.Each activity has mode variables (e.g. start-time and duration) and any services that it offers (e.g.the flying mode of an aircraft offers transportation service) and services that it requires (e.g. the flying mode of an aircraft requires the services of a crew). A formal theory of a resource should have as models exactly the physically feasible behaviors of the resource. The axioms serve to constrain the values that mode variables can take on instates (e.g. the weight of cargo cannot exceed a maximum bound during the flying mode of an aircraft). The transitions serve to constrain the evolution of the mode variables (e.g. the finish time of one activity must occur no later than the start time of the next activity).

A task is also expressed formally as an espec. The main difference from a resource espec is that a task espec offers no service, it only requires services of resources. For example, a cargo container requires transportation service.

The Planware design process has the following steps:

1. *Requirement Acquisition* – The user supplies a model of a scheduling problem in terms of especs for the kinds of tasks and resources that are of concern. The problem model is formalized into a specification that can be read abstractly as follows: given a collection of task instances (that accord with the task especs) and a collection of resource instances, find a schedule that accomplishes as many of the tasks as possible (or (approximately) optimizes the given cost function), subject to all the constraints of the resource models and using only the given resources.

   The required and offered services of are source express the dependencies between resource classes. Planware analyzes the task and resource models todetermine a hierarchy of service matches (service required matched with service offered) that is rooted in a task model.

2. *Algorithm Design* – The problem specification is used to automatically instantiate program schemes that embody abstract algorithmic knowledge about global search and constraint propagation. The algorithm generation process follows the structure of the service hierarchy, resulting in a nested structure of instantiated search schemes.

3. *Datatype Refinement and Optimization* – Abstract data types are refined to concrete programming-language types, and other optimizations are applied.

4. *Code generation* – Finally code in a programming language (currently CommonLisp) is generated. In one recent example, we developed formal models for air cargo packages, cargo aircraft, aircrews, and port facilities (i.e. four espec models). In about one second Planware generates 6560 LOC in a our local MetaSlang language, and then translates it to 19088 LOC in Commonlisp comprising over 1780 definitions.

3.2   Evolving specifications of security protocols

Since security is an integral and improtant aspect of software, security engineering should be an integral part of software engineering. In practice, however, the two communities, and their methods, are largely disjoint: software engineers talk about architectures, components and processess, while security is usually treated separately. The reason is, of course, that dealing with complex systems is based on the abstraction, and separation of concerns. We can only capture as many dimensions as our methods, and analytic powers permit.

Security, however, does not obey our established abstraction routines. On one hand, the security properties mostly refer to system wide information flows, and thus concern architectures. Security protocols often add new, secure links to the system, and remove the insecure, or transient links (e.g. with a key server). While they can, like all protocols, be encapsulated as architectural connectors, security protocols can only be analyzed in the context of overall architectural dynamics.

The extant architecture description languages do not conveniently support this need, as they tend to encapsulate and abstract away the behavior of connectors and components (leave it to extensions, or to the hardwired process calculi). Especs, on the other hand, permit a *completely flexible level of abstraction*: any

aspect of a system can be captured (if it can be mathematically expressed), or abstracted away, as needed. Therefore, taking up the subtle task of reasoning about security protocols, in the context of architectural dynamics that they induce, seems like a suitable challenge, and an important test case.

A general method for representing security protocols as especs has been developed. Each agent is specified as a separate state machine, with the state descriptions listing respective agents' states of knowledge. In fact, for each fixed set of formalized security properties of interest, a generic espec, describing the suitable process calculus can be developed: all agents can then be obtained as instances of this generic spec. Together with the connector-especs, describing the relevant properties of the network, the principals of a protocol then constitute a diagram of especs, depicting the desired architecture of the protocol.[1]

In the first case study, we have analyzed a recent proposal of the key exchange for the Binding Update in Mobile IPv6protocol. Representing the proposed protocol as an evolving specification has enabled us to automatically calculate the authenticated version of this protocol, by applying a generic program transformation from the library. The resulting protocol (stripped of the derivation details) is presented in [2].

The current work extends to analyzing some of the recent proposals for thenew IPSec keyexchange protocols. The verification framework is based on game theory, which allows us to quantify the various forms of the Denial-of-Service resistance, and their tradeoffs, which is one of the crucial novelties among the requirements. The quantitative analysis is also needed for estimating the strength of authentication against the requirement of minimizing the need for the public key infrastructure.

3.3   Evolving tool: EPOXI

The essence of the proposed mathematical frameworkis, of course, that it can be automated, and implemented in a tool. Our starting point is Specware, a categorically - based specification environment, that is developed and maintained at Kestrel Institute. Evolving specifications are built as diagrams in the dual of the category of specifications. The categorical structure of especs is completely derived from the categorical structure of specs. Specs and their morphisms (interpretations) now appear in two different roles: as *global* specs and structural refinements on one hand, and as *local* state descriptions and transitions, or computational steps between them. Furthermore, the morphisms of especs are built by combining the spec morphisms (interpretations) with the suitable morphisms of state machines (capturing simulations), which on their own account, also boil down to families of spec morphisms.

So far, the category of especs and their morphisms has been developed enough for full support of refinement of evolving specifications — down to provably

---

[1] Protecting this architecture as a diagram ensures that the information boundaries between agents are preserved. Fusing the agents into a single evolving specification, depicting their joint runs, would allow refinements where, e.g., Alice directly reads Bob's variables.

correct, and genuinely imperative C-code. Some generic transformations and optimization of code can be performed on especs.

Current work is focused on providing automated support for composition of especs, and building systems of systems, i.e. on the colimits and hereditary diagrams in the category of especs.

References

1. Abiteboul, S., Buneman, P., and Suciu,D. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA94022, USA, 1999.
2. Datta, A., Mitchell, J., Muller, F., and Pavlovic, D. Authentication for mobile ipv6. Tech. rep., Kestrel Institute, Mar.2002. submitted to Mobicom2.
3. Errington, L. Notes on diagrams and state. Tech. rep., KestrelInstitute, 2000.
4. Gurevich, Y. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Boerger, Ed. Oxford University Press, 1995, pp. 9–36
5. J.L. Fiadeiro, and T. Maibaum. Interconnecting formalisms: supporting modularity, reuse and incrementality. In *Proc. 3rd Symposium on the Foundations of Software Engineering* (1995), G. Kaiser, Ed., ACM Press, pp. 72–80
6. Kutter, P. W. State transitions modeled as refinements. Tech. Rep. KES.U.96.6, Kestrel Institute, August 1996.
7. Odersky, M. Objects + views = components. In *Proceedings of the International Workshopon Abstract State Machines* (ASM2000) (2000), Y. Gurevich, P. Kutter, M. Odersky, and L.Thiele, Eds., vol.1912 of *LNCS*, Springer.
8. . Parnas, D. L. The use of precise specifications in the development of software. In *Proceedings of IFIP Congress 1977* (1977), North-Holland Publishing Company.
9. Pavlovic,D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Automated Software Engineering Conference* (2001), IEEE Computer Society Press, pp. 157–165.
10. Pavlovic, D., and Smith, D. R. Guarded transitions in evolving specifications. In *Proceedings of Algebraic Methods in Software Technology (AMAST)* (2002), Springer-Verlag LNCS.
11. Robertson, P., Schrobe, H., and Laddaga,R., Eds. *Proceedings of the Workshop on Self-Adaptive Software (2001)*, vol.1936 of *Lecture Notes in Computer Science*, Springer Verlag.
12. Smith, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.
13. Srinivas, Y. V., and Jüllig, R. Specware: Formal supportfor composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin,1995, pp. 399–422.
14. Sünbül, A. *Architectural Design of Evolutionary Software Systemsin Continuous Software Engineering*. Der Andere Verlag, 2001. Ph.D. Thesis, Technical University Berlin, Germany.
15. Weber, H. Continuous engineering of information and communication infrastructures *extendedabstract*. In *Fundamental Approaches to Software Engineering FASE'99 Amsterdam Proceedings*, (spraddr, 1999), J.-P. Finance, Ed., spr, pp. 22–29.

16. Weber, H., Sünbül,A., and Padberg, J. Modelling informationand communica-
tion infrastructures for evolutionary design of business processes using component
technologies. In *Integrated Design and Process Technology* (June 2000), Society
for Design and Process Science, IEEE International Conference on Systems Inte-
gration.