A formal support for incremental behavior specification in agile development

Anne-Lise Courbis[1], Thomas Lambolais[1], Hong-Viet Luong[2], Thanh-Liem Phan[1], Christelle Urtado[1], and Sylvain Vauttier[1]

[1]LGI2P, école des mines d'Alès, Nîmes, France, First.Last@mines-ales.fr
[2]Laboratoire Ampère, UMR 5005, INSA-Lyon, Lyon, France, Hong-Viet.Luong@insa-lyon.fr

Abstract

Incremental development is now state of the practice. Indeed, it is promoted from the rational unified process to agile development methods. Few methods however guide software developers and architects in doing so. For instance, no tool is proposed to verify the non-regression of functionalities, modeled as behavior specifications, between increments. This work helps to incrementally specify software functionalities using UML state machines. It provides an on-the-fly evaluation of a specified behavior as compared to that of previous increments. The proposed contribution is based on two formally specified relations that are proved to preserve refinement when composed. Architects and developers are free to choose their preferred behavior specification strategy by iteratively applying them, so as to develop the required functionalities, having at each step the benefit of a formal non-regression checking to guide the global specification process. Our proposal is implemented in a proof-of-concept tool and illustrated by a didactic case-study.

***Keywords**: UML state machines, incremental development, agile methods, state machine verification, conformance relations, refinement.*

1. Introduction

The evolution of software system development processes currently follows two apparently contradictory main trends. *Agile and extreme programming* promote fast development of small increments that will altogether constitute the expected complete software [3]. These development methods are very popular as they are concrete, foster the sense of achievement among development teams and best satisfy clients as well as stakeholders by early, fast and regular deliveries of usable and valuable software that incrementally integrates all the required functionalities. However, the lack of a big picture to guide the development process towards well defined goals may lead to harmful inconsistencies such as regressions or substitution mismatches. *Model driven engineering* (MDE) promotes models as the main artifacts to capture both requirements and the designed solution. They are used, via automated or assisted transformations, to create the implementation of the system. MDE concentrates developers' efforts on the early development steps, trying to specify once and then generate implementations to various target technologies or execution frameworks, skipping, as much as possible, tedious repetitive design or coding tasks. MDE provides an effective support to capture the big picture of a specification and reason about design decisions. However, MDE does not yet fully support disciplined incremental development. Indeed, non regression is often prevented by the means of tests [7]. MDE lacks formal tools to perform behavioral model verifications.

This paper advocates that it is possible to combine the advantages of both trends by providing tools to compare the behavior specifications of increments and evaluate the existence of refinement relations in order to verify the global consistency of the development process. This enables incremental but disciplined development processes, supported by tools that provide guidance to enforce consistency. UML state machines are used as a uniform means to model behaviors throughout the development process, from initial, partial and abstract specifications, that interpret requirements as functionalities, to detailed designs, that fully define system dynamics. This way, incremental behavior specification and design schemes can be proposed thanks to three relations, that we have adapted from process algebra literature to state machines in previous work [17]:

- the behavior *extension* relation (noted ext) captures the fact that a machine adds behaviors to another one,

without impeding existing mandatory behaviors.

- the behavior *restricted reduction* relation (noted redr) captures the fact that a machine does not add extra observable behaviors and that mandatory behaviors are preserved: non observable behaviors may be detailed and optional behaviors may be removed.
- the behavior *refinement* relation (noted refines) links an abstract machine to a more concrete one and enforces that all the mandatory behaviors specified in the abstract machine are preserved in the refined one. Some optional behaviors may be removed, while new observable ones may be added, provided they do not conflict with existing ones.

These relations are going to serve as a basis for the incremental development of behavior models. The idea of the paper is that they altogether form a formal yet not constraining means to evaluate the consistency of artifacts produced when using agile development processes.

Whichever relations are composed, the latter machines are guaranteed to be conform implementations of the formers. When none of theses relations can be asserted between two successive machines, a rupture is detected in the refinement process. This paper advocates for a guided revision mechanism that helps analyze the cause of the inconsistency and decide which machine should be modified: either the proposed implementation may be erroneous, or the abstract machine may be over-specified and impossible to be properly implemented. Once this ambiguity is resolved, the system might also help designers propagate involved corrections to other machines so as to establish the required relations. These two cases show how composing the restricted reductions and extensions might constitute an agile but disciplined method for specifying the behavior of systems.

The remainder of the paper is structured as follows. Section 2 describes a didactic motivating example that is used as an illustration throughout the paper. Section 3 presents our proposal. It describes the three proposed relations we choose to support state machine development and then presents how they can be used to support an agile development scenario. Section 4 discusses our approach against state of the art before concluding in Section 5 with some perspectives.

2. Motivating example

Informal specification of a Vending Machine. The specification a Vending Machine (Figure 1) contains mandatory parts (refund the customer unless goods are obtained), as well as optional parts (maintenance, cookies).

Successive UML state machines. In order to progressively design the behavior of this vending machine, the designer produces several intermediate state machines (Figure 2). He starts from mandatory behaviors, considering coin and cancel signals only. Hence, the Minimal Machine is a rather stupid machine, which specifies that after any amount of coins, the user can be refunded and that is the only thing he can ask for. The drink signal is always ignored. In the second NeverEmpty Machine, the designer adds the ability to react to the drink signal, in *some cases* after the coin signal. At that time, the user can still be refunded. If he chooses a drink, the machine will eventually distribute it and give him his money back. Note that this machine is *nondeterministic*[1].

> *The system delivers goods after the customer inserts the proper credit into the machine. Goods are drinks, but could also be cookies. Optionally, a technician can shutdown the machine with a special code. When used by a customer (not a technician), the system runs continuously. An important feature is that the system must not steal the user: if the customer has not inserted enough money, changes his mind or if the system is empty, the system refunds the user.*

Figure 1: Informal specification

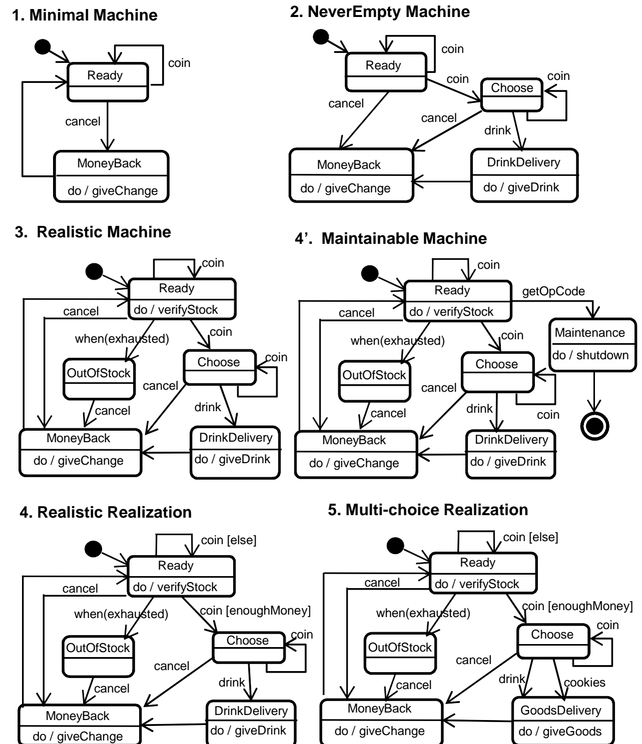

Figure 2: Incremental development proposing several UML state machines

The third Realistic Machine considers the fact that the

[1]Although nondeterminism is not allowed in UML, we consider that final models have to be deterministic, but that initial and intermediate models may be nondeterministic.

machine may be empty which should leave solely the cancel action to the user. This machine describes every *mandatory* part of the above informal specification. Other features (cookies and maintenance) are options. Hence, it can lead to the concrete state machine (Realistic Realization) which fixes the nondeterministic points and can be used as a basis for a first implementation. This concrete machine can then be extended (Multi-choice Realization) to add the second choice of goods with the cookies signal. Alternatively, from the third Realistic Machine, we could also extend the behaviors and consider the getOpCode signal in the Ready state. These development sequences show that concrete machines can be derived from intermediate abstract models, which only describe the most important features. In that sense, they obey an agile development process where simplified products are quickly produced. Machines 4 and 5 are called *realizations* since they are concrete, deterministic machines, which describe all the mandatory behaviors of the informal requirements of Figure 1.

Verification needs. Having such development scenarios in mind, we focus on the following properties:

r_1 : *Implementation.* At any step of a development process, the resulting machine has to fulfill the requirements expressed by the first specification model.

r_2 : *Liveness preservation*. Liveness properties state a system has to react to some signals after precise signal sequences. At any time, our example system must react to the cancel button and refund the user; after a given sequence of coin signals and if the machine is not empty, it must react to the drink button.

r_3 : *Progress*. During the development process, when an M' machine is supposed to be *an extension* of an M machine, we want to be able to verify that any behavior offered by M is also offered by M' in the same conditions.

r_4 : *Safety preservation*. Safety properties state that some actions are forbidden. When an I machine is *a restricted reduction* of an M machine, we need to guarantee that I does not implement behaviors not described by M. On the example, delivering a product that has not been paid for or delivering two products instead of one are such forbidden actions.

r_5 : *Composability.* When chaining extensions, we want the result to be an extension of the initial machine, and similarly for reductions. When combining extensions and reductions, we need to know the relation between the resulting machine and the initial one.

3. Relations to support incremental development processes

The verification technique we choose to satisfy these properties is to compare models between them. This excludes the developer to separately describe liveness and safety properties in an another language, as in [10]. At first, we mainly focus on properties r_1, r_2, r_3 and r_4.

3.1. Behavior conformance relation

Conformance testing methodologies proposed by ISO [13] compare an implementation to a standard specification. Recommendations define mandatory and optional parts. An implementation is in conformance to a specification if it has properly implemented all the *mandatory parts* of that specification [19]. We consider conformance as our reference behavior *implementation* relation.

Formalizing the conformance relation [5] consists in comparing the event sets that *must be accepted* by the compared models, after any trace of the specification model. A *trace* is a partial *observable* sequence of events and/or actions that the machine may perform. The set of traces of an M machine is noted $\mathrm{Tr}(M)$. An implementation model conforms to a specification model if, after any trace of the specification, any set of events that the specification *must* accept, *must* also be accepted by the implementation model. We refer to [14] for a study of this relation on Labeled Transition Systems (LTSs), and to our works [17] for an implementation technique and a translation from UML state machines to LTSs.

In the example of Figure 2, $\mathrm{Tr}(\mathsf{Minimal\ Machine}) = \{\mathsf{coin}^*, (\mathsf{coin}^*.\mathsf{cancel})^*\}$. Minimal Machine must accept coin and cancel events after any trace $\sigma \in \mathrm{Tr}(\mathsf{Minimal\ Machine})$. This property is satisfied by the NeverEmpty Machine, which consequently conforms to Minimal Machine.

$$\mathsf{NeverEmpty\ Machine\ conf\ Minimal\ Machine}$$

The conformance relation is suited for implementation (r_1 property) and liveness (r_2 property). However, it is too weak to guarantee progress and safety properties (r_3, r_4), and, since it is not transitive, it cannot answer property r_5. It thus cannot be considered as a refinement relation.

3.2. Behavior refinement relation

Considering conf as an implementation relation, the refinement relation (refines) is defined as the largest relation satisfying the following *refinement* property: For all machines M_1 and M_2,

$$M_2 \ \mathsf{refines}\ M_1 \Longrightarrow \forall I, I \ \mathsf{conf}\ M_2 \Rightarrow I \ \mathsf{conf}\ M_1. \quad (1)$$

The refines relation has the following properties:

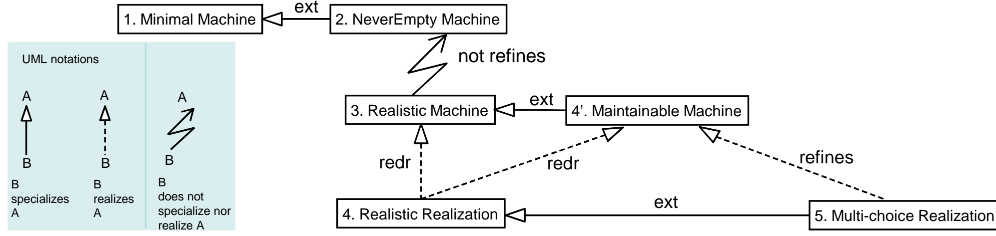

Figure 3: Synthesis of relations

- refines $\subseteq$ conf: refines can be used as an implementation relation (property r_1) and inherits properties of the conf relation, such as liveness preservation (property r_2);
- it is transitive;
- If M_2 refines M_1, for any trace σ of M_2 which is not a trace of M_1, M_2 must refuse everything after σ.

This definition of refinement is large enough to encompass both notions of classical refinement [2] and incremental constructions (extension).

3.3. Specialized behavior refinement relations

Extension. The *extension* relation is defined by ext $=_{def}$ refines $\cap \supseteq_{Tr}$, where, for two machines M_1 and M_2, $M_2 \supseteq_{Tr} M_1 =_{def} \text{Tr}(M_2) \supseteq \text{Tr}(M_1)$. The ext relation inherits implementation and liveness preservation property from refines (properties r_1, r_2). It is moreover defined to satisfy the progress property (r_3). The ext relation is a refinement relation that reduces partiality and nondeterminism. In Figure 2, the NeverEmpty Machine offers the possibility to ask for a drink, without preventing the user from doing something he could do with the Minimal Machine:

$$\text{NeverEmpty Machine ext Minimal Machine} \quad (2)$$

Restricted reduction. To overcome the fact that ext does not preserve safety (one cannot know whether the drink action, which is new, is safe or not), restricted reduction is defined by redr $=_{def}$ refines $\cap \subseteq_{Tr}$. redr inherits properties from refines and adds safety preservation property (r_4). redr is very similar to classical refinement (it reduces abstraction and nondeterminism). In Figure 2, the RealisticRealization is a reduction of the Maintainable Machine: the getOpCode signal can be refused by Maintainable Machine after any trace in $\{\varepsilon, \text{coin}^*\}$;

$$\text{RealisticRealization redr Maintainable Machine} \quad (3)$$

To summarize, we keep three refinement relations which are transitive and preserve liveness properties: refines is the largest one, ext is the subset of refines ensuring progress and redr is the subset of refines ensuring safety preservation. Our goal now is to study the composition (property r_5) between these three relations.

Analysis on the example. It appears that:

$$\text{not(Realistic Machine refines NeverEmpty Machine)} \quad (4)$$

The Realistic Machine may refuse coin after the empty trace ε, whereas NeverEmpty Machine must always accept coin. Considering the development process proposed in Figure 2, result (4) gives information to the designer. He has to answer the question whether the coin event must always be accepted initially or may be refused, by forcing him to ask for the cancel event. This point is not clear in the informal requirements (Figure 1). If the developer considers the coin action is mandatory, he must correct Realistic Machine by adding for instance a self-transition triggered by coin on state OutOfStock. Otherwise, if coin can be refused when the machine is empty, Realistic Machine is to be considered as the new reference specification.

Figure 3 sums up relations ext, redr, and refines on the six proposed machines. In an agile development process, having proposed Realistic Realization model, the designer can come back to Maintainable Machine, checking that the following result (5) is satisfied. Then, knowing result (3), he can propose Multi-choice Realization and verify relation (6):

$$\text{Maintainable Machine ext Realistic Machine} \quad (5)$$

$$\text{Multi-choice Realization ext Realistic Realization} \quad (6)$$

3.4. Composing relations to support agile development processes

We call *strategy* the successive steps that a designer chooses to achieve a development. With our disciplined framework, this amounts to choose to add behaviors (horizontal refinement) or details (vertical refinement) to the current behavior model, producing a new behavior model that must verify an ext or a redr relation. Figure 4 shows how an Agile development process can be managed as an instantiation of such a strategy. From partial and abstract requirements (S_0), a specification of the first increment to implement is defined (S_1). This specification is detailed through several design steps to produce eventually an implementation model (S_3). Then, a new increment is defined (S_4), extending the previous, and so on, until all requirements are implemented. Thereafter, an evolution of the

software, based on revised requirements (S_7), can be developed as a new development process. A crucial issue is to guarantee that the development process leads to conformant implementations (for instance S_3 as compared to S_1 or S_0) whereas only local consistency is stated by the refinement relations that are built between successive models. This implies to verify that refinement relations can be composed into implementation relations.

Local composition. Locally, two ext and redr relations commutatively compose refines relations:

$$\mathsf{redr} \circ \mathsf{ext} = \mathsf{ext} \circ \mathsf{redr} = \mathsf{refines} \quad (7)$$

This result comes from the definition of redr and the properties of conf (see 3.2 and 3.3). As redr and ext relations are easier to check, it is interesting to deduce refines relations from them.

Global composition. Globally, ext and redr relations compose with refines relations as follows:

$$\mathsf{refines} \circ \mathsf{ext} = \mathsf{ext} \circ \mathsf{refines} = \mathsf{refines} \quad (8)$$

$$\mathsf{refines} \circ \mathsf{redr} = \mathsf{redr} \circ \mathsf{refines} = \mathsf{refines} \quad (9)$$

These two properties derive from the fact that for any preorder A and any relation $X \subseteq A$, we have: $A \circ X = X \circ A = A$.

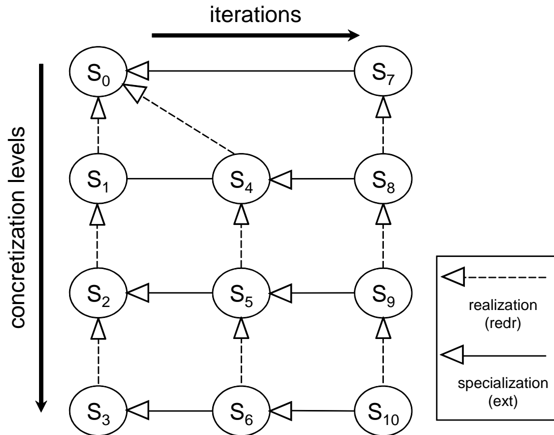

Figure 4: Instantiation of an Agile development process

Applied to the example, result (10) is inferred from computed relations (3), (6) and by property (7):

$$\text{Multi-choice Realization refines Maintainable Machine} \quad (10)$$

With property (9), we also conclude that:

$$\text{Multi-choice Realization refines Realistic Machine} \quad (11)$$

Such calculations provide an effective support to agile development. At any step, designers can freely choose their preferred strategy and leverage inferred relations to carry on development.

4. Discussion and related works

Process algebra implementation and refinement relations. In order to satisfy property r_1, we find a large number of behavior model comparison relations in the context of process algebras and LTSs. Milner's observational equivalence and congruence, based on bisimulations [18], are well known relations to compare a detailed implementation model to an abstract specification model. Observational congruence can be considered as an implementation relation in a strong sense, where mandatory as well as optional behaviors must be present in the implementation model. They have been implemented in several toolboxes such as [10]. Milner's observational congruence preserves safety and liveness. However, it does not satisfy the r_3 property: observational congruence cannot be used in an incremental process.

An interesting result is that conformance is weaker than Milner's observational congruence: any observationaly congruent models are also conformant. Hence, the refines relation still distinguishes dangerous from harmless livelocks, as in Milner's theory, which is not the case of Hoare's CSP refinement relations [12].

Incremental construction versus refinement. *Refinement* has to be discussed since it has various interpretations. It is a well-known and old concept [22] used in some reference works about state machine refinement [1] or specification refinement [2], where it is considered as a relation for comparing two models in order to reduce non determinism and abstraction. This relation corresponds to a reduction: it consists in introducing details into models in order to get an implementation model. It has been implemented in languages such as B [2] and Z [8]. From our point of view, founding a development process on such a relation is restrictive. We prefer the definition given by [4], in accordance with definition 1 of section 3.2. Note that this relation is called *consistency* in [14] and some researchers of the UML community prefer this term rather than *refinement*. This definition is interesting because it includes the conventional refinement based on reduction but does not exclude the extension of initial specifications. The benefits of our approach compared to the conventional refinement processes are manifold. It is close to the way of reasoning and to the practice of designers to finalize complex models. It is close to agile processes to allow rapid delivery of high-quality software meeting first requirements of customers. It attests the feasibility of a first implementation model before enhancing it to get the final one. Finally, it might help to support evolution because "systems, and consequently their design, are in perpetual evolution before they die" [15].

Related approaches for analyzing state machine consistency. Few works deal with incremental development of

state machines. [6] addresses the problem at architecture level (state machine assembly). [9] does alike and guides assembling with rules.

Works about state machines verification have to be mentioned despite their different objectives, as they focus on consistency between a software and its specification. Many works are based on model checking techniques. UML is thus transformed into the modeling language of the model checking tool: it can be PROMELA to use SPIN as it is done in [16] or LTSs to use JACK as in [11]. Some works analyze consistency using pre and post-conditions as it is done in [4] using the Z formalism. Lastly, consistency can be expressed through transformations as it is done for refactoring in [21]. Such techniques require to explicitly express liveness properties.

5. Conclusions and future works

In this paper, we address the issue of the incremental construction of state machines to support agile development processes. It implies a composition of successive vertical and horizontal refinements that must globally achieve a consistent implementation of the initial software specification. The study of existing works points out that these two aspects are never considered as a whole, despite they are key points to define development strategies.

We demonstrated the computational feasibility of our proposal by developing a JAVA tool named IDCM (Incremental Development of Conforming Models). It implements the verification of conf, ext, redr and refines relations [17] by transforming UML state machines into LTSs and analyzing their relations. Analysis provides designers with feedback about detected warnings or errors.

Beyond the several experimented case studies, we plan to evaluate our proposal and tool on full size projects. We also currently study the adaptation of this work to component-based architectures, in other words to coarse-grained, reuse-centered development approaches, to address complexity and scalability issues.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Logic in Computer Science*, pages 165–175, 1988.

[2] J. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] S. W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0.* 3rd edition, 2004.

[4] E. Boiten and M. Bujorianu. Exploring UML refinement through unification. In *Critical Systems Development with UML*, LNCS, page 47—62, 2003.

[5] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Twente University of Technology, Department of Informatics, Enschede, Netherlands, Dec. 1986.

[6] S. Burmester, H. Giese, M. Hirsch, and D. Schilling. Incremental design and formal verification with UML/RT in the FUJABA Real-Time tool suite. In *SVERTS*, 2004.

[7] A. Cicchetti, D. D. Ruscio, D. S. Kolovos, and A. Pierantonio. *A test-driven approach for metamodel development*, chapter Emerging Technologies for the Evolution and Maintenance of Software Models, pages 319–342. IGI Global, 2012.

[8] J. Derrick and E. Boiten. *Refinement in Z and object-Z: foundations and advanced applications*. Springer-Verlag, 2001.

[9] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. 6th Int. Conf. on Integrated Design and Process Technology*, 2002.

[10] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In P. Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.

[11] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, Apr. 2001.

[12] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall, June 2004.

[13] ISO/IEC 9646-1. Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts, 1991.

[14] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23—41, 1992.

[15] M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[16] J. Lilius and I. Paltor. Formalising UML state machines for model checking. In *UML conf.*, 1999.

[17] H. Luong, T. Lambolais, and A. Courbis. Implementation of the Conformance Relation for Incremental Development of Behavioural Models. *Models 2008, LNCS*, 5301:356–370, 2008.

[18] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[19] S. Moseley, S. Randall, and A. Wiles. In Pursuit of Interoperability. In K. Jakobs, editor, *Advanced Topics in Information Technology Standards and Standardization Research*, chapter 17, pages 321–323. Hershey, 2006.

[20] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[21] G. Sunyé, D. Pollet, Y. L. Traon, and J. Jézéquel. Refactoring UML models. In *UML conf.*, pages 134–148, 2001.

[22] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.