Architectural Patterns for Problem Frames

Christine Choppy[‡], Denis Hatebur[§] and Maritta Heisel[¶]

October 25, 2004

Abstract

Problem frames provide a characterisation and classification of software development problems. Fitting a problem into an appropriate problem frame should not only help to understand it, but also to *solve* the problem (the idea being that, once the adequate problem frame is identified, then the associated development method should be available). We propose software architectural patterns corresponding to the different problem frames that may serve as a starting point for the construction of the software solving the given problem. These architectural patterns exactly reflect the properties of the problems fitting into a given frame, and they can be combined in a modular way to solve multi-frame problems.

1 Introduction

Pattern-orientation is a promising approach to software development. Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure.

Patterns have been introduced on the level of detailed object oriented design [10]. Today, patterns are defined for different activities. *Problem Frames* [14] are patterns that classify software development *problems*. *Architectural styles* are patterns that characterize software architectures [18, 1]. They are sometimes called "architectural patterns". *Design Patterns* are used for finer-grained software design, while *frameworks* are considered as less abstract, more specialized. Finally, *idioms* are low-level patterns related to specific programming languages [5], and are sometimes called "code patterns".

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time.

It is acknowledged that the first steps of software development are essential to reach the best possible adequation between the expressed requirements and the proposed software product, and to eliminate any source of error as early as possible. Therefore, we propose to use patterns already in the requirements elicitation phase of the software development lifecycle.

M. Jackson [13, 14] proposes the concept of *problem frames* for presenting, classifying and understanding software development problems. A problem frame is a characterization of a class of problems in terms of their main components and the connections between these components. Once a problem is successfully fitted into a problem frame, its most important characteristics are known.

Gaining a thorough understanding of the problem to be solved is a necessary prerequisite for solving it. However, when using problem frames, one can even hope for more than just a

[‡]LIPN, UMR CNRS 7030, Institut Galilée - Université Paris XIII, France, email: Christine.Choppy@lipn.univ-paris13.fr

[§]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: denis.hatebur@uni-duisburg-essen.de and Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

[¶]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: maritta.heisel@uni-duisburg-essen.de

full comprehension of the problem at hand. Since problem frames are patterns, they represent problem structures that occur repeatedly in practice. Hence, it is worthwhile to look for solution structures that match the problem structures represented by problem frames.

The construction of the solution of a software development problem should begin with the decision on the main structure of the solution, i.e., a decision on the software architecture. Our aim is to exploit the knowledge gained in representing a problem as an instance of a problem frame in taking that decision. For each problem frame, we propose a corresponding architectural pattern that takes into account the characteristics of the problems that fit into the given problem frame. As problem frames, these architectural patterns must be instantiated to develop a solution for a concrete problem. The structuring provided by the architectural pattern constitutes a concrete starting point for the process of constructing a solution to a problem that is represented as an instance of a problem frame.

The rest of the paper is organised as follows: After introducting the basic concepts of our work in Section 2, we discuss related work in Section 3. In Section 4, we present the architectural patterns associated with each problem frame. In Section 5, we present an example and show how the different solutions of multiframe problems can be combined. In Section 6, we conclude with a discussion of our approach and directions for future research.

2 Basic Concepts

In this paper, we relate architectural patterns to problem frames. As a notation for our architectural patterns, we use composite struture diagrams of UML 2.0. In the following, we give brief descriptions of these three ingredients of our work.

2.1 Problem Frames

Jackson [14] describes problem frames as follows:

> A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.

Solving a problem is accomplished by constructing a "machine" and integrating it into the environment whose behaviour is to be enhanced.

For each problem frame a diagram is set up (cf. e.g. left-hand sides of Figures 2 and 3). Plain rectangles denote application domains (that already exist), rectangles with a double vertical stripe denote the machine domains to be developed, and requirements are denoted with a dashed oval. They are linked together by lines that represent interfaces, also called shared phenomena. Jackson distinguishes *causal* domains that comply with some laws, *lexical* domains that are data representations, and *biddable* domains that are people. Jackson defines five basic problem frames (*Required Behaviour, Commanded Behaviour, Information Display, Workpieces* and *Transformation*), and we consider them together with one variant (*Commanded Information*). In order to use a problem frame, one must instantiate it, i.e., provide instances for its domains, interfaces and requirements.

2.2 Architectural Styles

According to Bass, Clements, and Kazman [1],

> the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectural styles are patterns for software architectures. A style is characterized by [1]:

- a set of component types (e.g., data repository, process, procedure) that perform some function at runtime,

- a topological layout of these components indicating their runtime interrelationships,
- a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it),
- a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

Important architectural styles are the following [18]:

- **Data-Centered** with substyles *Repository* and *Blackbord*
- **Data Flow** with substyles *Batch Sequential* and *Pipe-and-Filter*
- **Virtual Machine** with substyles *Interpreter* and *Rule- Based Systems*
- **Call-and-Return** with substyles *Main Program and Subroutine*, *Layered*, *Object-Oriented* or *Abstract Data Types*
- **Independent Components** with substyles *Communicating Processes* and *Event Systems* (implicit/explicit invocation)

When choosing an architecture for a system, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. In the following, we propose to associate a given architecture with each basic problem frame so as to facilitate the design. We chose to use UML 2.0 composite structure diagrams (see Section 2.3) to show the layout as done for instance in Figure 1 for the *Layered* (on the right-hand side) and the *Repository* (on the left-hand side) architectural styles.

Figure 1: Layered and Repository architectural styles

2.3 Composite Structure Diagrams

Composite structure diagrams [20] are a means to describe architectures. They contain named rectangles, called *parts*. Parts may have *ports*, denoted als small rectangles. Ports denote interaction points of a part with its environment. A line between two ports is a *connector*. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation, but that notation will not be used in this paper.

3 Related Work

Since patterns were introduced, the question of how to make the best use of them in the software development process inspired a number of research works. We mainly consider here those related with the use of problem frames, also in relationship with architectural styles.

Along the idea to integrate problem frames in a formal development process, Choppy and Reggio [8] showed how a formal specification skeleton may be associated with some problem frames (Translation and IS as named in [13]), using CASL[3] and its CASL-LTL extension [17] as target formal specification languages.

Choppy and Heisel showed in [6, 7] that this idea could be extended to other formal specification languages, such as LOTOS [4] or Z [19]. In that work, they also provided a transition from problem frames to architectural styles, so as to provide finer structures when moving from the requirements specification to the detailed specification or the design phases. In [6], they gave criteria for (i)

helping to select an appropriate basic problem frame, and (ii) choosing between architectural styles that could be associated with a given problem frame. In [7], a proposal for the development of information systems is given. The system decomposition is done along different identified use cases.[1] Then, to each use case related with updates or queries is associated an update or query problem frame (specifically proposed for information systems). A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts.

The approach developed in [11, 16] is quite complementary since the ideas developed there are to introduce architectural concepts with problem frames (introducing "AFrames") so as to benefit from existing architectures. In [11], the applicability of problem frames is extended to include domains with existing architectural support, and to allow both for an annotated machine domain, and for annotations to relate/discharge the frame concern. In [16], "AFrames" are presented corresponding to the architectural styles Pipe-and-Filter and MVC, and applied to transformation and control problems.

Finally, let us mention Lavazza and Del Bianco [15] who do not look for architectures, but provide a description of commanded and required behaviour problem frames in UML-RT focussing on active objects or "capsules" communicating through ports (defined by protocols), and they provide a real time version of OCL, called OTL.

4 Architectural Patterns

We now present the six most important problem frames and give their corresponding architectural patterns. These architectural patterns are not pure instances of some architectural style, but they combine elements of different architectural styles to adequately reflect the problem characteristics as given by the respective problem frame.

The quotations at the beginnings of the subsections are from Jackson [14].

4.1 Required Behaviour

The following problems fit into the problem frame *Required Behaviour*:

> There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.

The corresponding frame diagram is shown on the left-hand side of Figure 2. The "C" in the frame diagram indicates that the *Controlled domain* must be causal. The machine is always a causal domain (so an explicit "C" is not needed). The notation "CM!C1" means that the causal phenomena *C1* are controlled by the Control machine *CM*. The dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

This problem frame is appropriate for *embedded systems*, where the machine to be developed is embedded in a physical environment that must be controlled. The communication between the machine and the physical environment takes place via *sensors* and *actuators*. Thus, only by virtue of sensors and actuators can there be shared phenomena between the machine and its environment. Sensors realize the phenomena $C2$ of the frame diagram, i.e., the phenomena controlled by the environment but observable by the machine. Actuators realize the phenomena $C1$ of the frame diagram, i.e., the phenomena controlled by the machine and observable by the environment.

For example, we might want to build a machine that keeps the temperature of some liquid between given bounds. Then, the temperature of the liquid would be a shared phenomenon controlled by the environment. The corresponding sensor would be a thermometer. Another shared phenomenon would be the state of a burner. That state would be controlled by the machine, i.e., the machine is able to switch the burner on or off.

[1] In [11], an example of problem decomposition is based on different requirements statements, which may be a similar idea unless some requirements statements refer to a same use case.

Figure 2: Required Behaviour Frame Diagram and Architecture

Practical work in developing embedded systems has shown that a layered architecture is appropriate for these systems [12]. Such an architecture[2] is shown on the right-hand side of Figure 2, expressed as a composite structure diagram in the notation of UML 2.0.

The lowest layer is the *hardware abstraction layer* (HAL). This layer covers all interfaces to the external components in the system architecture and provides access to these components independently of the used controller or processor. For porting the software to another hardware platform, only this part of the software needs to be replaced.

The hardware abstraction layer is used by the *interface abstraction layer* (IAL). This layer provides an abstraction of the (low-level) values yielded by the sensors and actuators. For example, a frequency of wheel pulses could be transformed into a speed value. Thus, in the interface abstraction layer, values for the monitored and controlled variables (see [9]) of the system are calculated. It is possible that these variables have to be calculated from the values of several hardware interfaces. For safety-critical software components, the interface abstraction layer will usually make use of redundant arrangements of sensors and actuators.

The highest layer of the architecture is the *application layer*. This layer only has to deal with variables from the problem diagram. Therefore, the system requirements can be directly mapped to the software requirements of the application layer, as described by Bharadwaj and Heitmeyer[2].

Thus, the architecture shown on the right-hand side Figure 2 represents an adequate structure for the *ControlMachine* of the left-hand side Figure 2. For special kinds of embedded systems, that architecture could be refined. However, a refinement of the architecture would also correspond to a refinement of the corresponding problem frame. The architecture shown here has the same degree of generality as the problem frame.

4.2 Commanded Behaviour

The following problems fit into the problem frame *Commanded Behaviour*:

> There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.

The corresponding frame diagram is shown on the left-hand side of Figure 3. The "B" indicates that the domain *Operator* is a biddable domain, and the phenomena *E4* are the operator commands.

Figure 3: Commanded Behaviour Frame Diagram and Architecture

As can be seen from the frame diagram, the distinguishing feature of the *Commanded Behaviour* frame as compared to the *Required Behaviour* frame is the presence of an operator. That distiction is reflected in the corresponding architecture shown on the right–hand side of Figure 3. The operator commands and the corresponding feedback are handled by a dedicated component *User Interface*.

Since each architecture corresponding to a problem frame containing an operator domain will contain a user interface component, we give the structure of such a component in more detail

[2]In the following, we use the word "architecture" instead of "architectural pattern" for reasons of readability. It is clear, however, that the components shown in the architectural diagrams have to be instantiated in order to obtain a concrete software architecture.

Figure 4: Detailed architecture for user interface

(Figure 4). For reasons of practicality, the user interface component contains not only a sub-component that serves to read user input via some device. In most cases, a sub-component will also be needed that provides some kind of feedback to the user via a display. The physical input arriving at the interface at the bottom of the component is transformed into the more abstract phenomena $E4$ by the sub-component *User Input/Output Interface*.

4.3 Information Display

The following problems fit into the problem frame *Information Display*:

> There is some part of the physical world whose states and behaviour is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.

The *Information Display* problem frame offers a structure for applications devoted to the display of real world physical data. The corresponding frame diagram is shown on the left-hand side of Figure 5. The "C" indicates that the *Real World* and *Display* domains are causal. The interface between the information machine and the real world contains only phenomena $C1$ that are controlled by the real world. This means that the machine cannot influence the real world. Its purpose is only to display things that happen in the real world.

Figure 5: Information Display Frame Diagram and Architecture

Accordingly, the architecture given on the right-hand side of Figure 5 does not contain any components for handling actuators, but only components for handling sensors. There is no operator, but a display is needed. Hence, the architecture contains a display interface.

The architecture shown in Figure 5 is very simple. It only works if the phenomena of the real world can directly be transformed into a representation suitable for display. Often, some processing of the information yielded by the sensors will be necessary. In this case, an application layer must be added to the architecture, as shown in Figure 6.

Figure 6: Architecture for information display with application

4.4 Commanded Information

The *Commanded Information* problem frame (Figure 7) is derived from the *Simple IS* frame [13]. In [14], the *Commanded Information* frame is presented as a variant of the *Information Display* frame, where an operator is added. The *Commanded Information* frame is very similar to a *Database Query* frame [7], the only difference being that the domain to be displayed need not be causal, but can also be a lexical or a model domain.

The architecture we propose for machines that solve a *Commanded Information* problem is shown on the right-hand side of Figure 7. To take the presence of an operator into account, the *Display Interface* component of the architecture for the *Information Display* frame is replaced by a user interface component.

Moreover, to cover cover database applications as well as the operator-controlled display of physical data, the architecture we propose contains a *Data Storage* component. Of course, this component can be left out if it is not needed to solve the problem. In that case, there would only

Figure 7: Commanded Information Frame Diagram and Architecture

be one (or even none) application component. On the other hand, for pure database applications, the sensor-handling components of the architecture will not be needed.

4.5 Workpieces

The following problems fit into the problem frame *Workpieces*:

> A tool is needed to allow a user to create and edit a certain class of computer processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.

The "X" indicates that the *Workpieces* domain of the frame diagram shown in Figure 8 is a lexical (inert) domain. The *Workpieces* problem frame is very similar to a *Database Update* frame [7].

Figure 8: Workpieces Frame Diagram and Architecture

The architecture shown on the right-hand side of Figure 8 contains a user interface component, because the problem frame diagram contains a user. The data storage component of the architecture corresponds to the *Workpieces* domain of the frame diagram. The *Application* component is responsible for manipulating the data storage according to the user commands. This architecture is closely related to the *repository* architectural style, see Figure 1. Note that there is only one interface with the environment – namely the interface with the user – because the lexical workpieces domain is part of the machine. This holds true also for the input and output domains of the architecture for transformation problems (see Section 4.6).

As a generalisation that will be needed for practical applications involving distributed systems, we propose the architecture given in Figure 9. Here, remote access to the data storage is possible via a network.

Figure 9: Architecture for remote access to data storage

4.6 Transformation

The following problems fit into the problem frame *Transformation*:

> There are some computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.

Again, the architecture shown in Figure 10 exactly reflects the domains of the frame diagram. Inputs and outputs are stored in data storage components, and the application component is responsible for transforming inputs into outputs.

Often, pipe-and-filter architectures are useful for solving transformation problems. In this case, the *Application* component of our architecture would take the form of a pipe-and-filter structure.

Figure 10: Transformation Frame Diagram and Architecture

5 Multiframe-Problem Automatic Teller Machine (ATM)

As an example, we consider an automatic teller machine (ATM). The mission of an ATM is to provide customers with money, provided that they are entitled to withdraw the desired amount.

The ATM is an example for a multiframe problem, i.e., it consists of several subproblems that can be fitted into different problem frames. In the following, we will identify the subproblems, fit them into an appropriate problem frame, and derive the corresponding software architectures according to the patterns given in Section 4. Finally, we will compose the architectures yielded by the different subproblems to obtain an architecture for the whole ATM.

5.1 Money-Case-Control Subproblem

The first subproblem of the ATM controller problem concerns the control of the money case. To deliver the money to the customer, the ATM has a case where it puts in the requested banknotes. This case has a shutter visible by the customer. To prevent that an other person from taking forgotten money, the ATM will only open the case for a limited amount of time. When that time is over, the ATM retracts the money from the case. The instantiated frame diagram for *Commanded Behaviour* is shown in Figure 11. The corresponding architecture is shown in Figure 12.

Figure 11: Problem diagram for money-case-control

C1: $\{put\_banknote\_to\_case, start/stop\_open\ case, take\_banknotes\_from\_supply, start/stop\_close\_case, retract\_banknotes\_from\_case\}$

C2: $\{case\_is\_open, case\_is\_closed, banknotes\_removed\}$

C3: $\{banknotes\_in\_case\}$

E4: $\{insert\_card, remove\_card\}$

E5: $\{enter\_request, enter\_pin\}$

Figure 12: Architecture for money-case-control subproblem

5.2 Card-Reader-Control Subproblem

The second subproblem concerns controlling the card reader. The ATM should have control over the card. Once a user has inserted his or her card, it is under control of the ATM. When the user has successfully withdrawn the money, the ATM ejects the card. In case of an invalid card or three unsuccessful attemps of authentication, the card will be kept by the ATM. After the ATM has ejected the card, the user must remove it within some time. Otherwise it will be retracted again to prevent somebody else from taking it.

see 13

C6: $\{card\_inside, no\_card\_inside\}$

C7: $\{eject\_card, retect\_card\}$

C8: $\{eject, retract\}$

see Figure 14

Figure 13: Problem diagram for for card-reader-control

Figure 14: Architecture for card-reader-control Subproblem

5.3 Log-File Subproblem

Commanded Information: Adminitstrator reads log-file with all recognized actions of user and system
see Figure 15

Figure 15: ...

see Figure 16

5.4 Update-Account Subproblem

Workpieces: request account balance, update account balance
see Figure 17

5.5 Composed Architecture

all architectures together
see Figure 18

6 Conclusion

In the present paper, for each basic problem frame (together with the commanded information variant), an associated architecture is designed that is expressed using the composite structures of UML 2.0, thus taking advantage of the latest features available to show how connections are planned. The contributions of our approach are the following:

- We provide a way to start describing, through a given architectural pattern, solution packages associated with basic problem frames.
- To this aim, we designed new architectural patterns specifically elaborated to reflect the problem frame parts and their intended refinement when moving to the solution design.
- The components of problem frames are reflected in the architectures, and a clear correspondance between domains of frame diagrams and components of architectures is extablished.
- The provided architectural patterns have same abstraction level as problem frames; both can be refined to accomodate more specific kinds of problems.
- The architectural patterns enable recomposition of solutions developed for different subproblems of multi-frame problems.

While in [11, 16] a related approach is taken to extend problem frames with architectural concepts, so as to introduce more knowledge and information in problem frames, our approach is to propose architectural patterns dedicated to the basic problem frames, that can be used when moving to the design phase (or even in the detailed specification phase) of the software development as advocated in [6, 7]. We see these as complementary since while [11, 16] provide a

Figure 16: Commanded Information: Adminitstrator reads log-file with all recognized actions of user and system

Figure 17: Workpieces: request account balance, update account balance

way to incorporate/reuse domain knowledge in the problem description, we focus on proposing a way to move from the problem description to the solution description.

Although the work presented here is independent of any formal specification language, if desired, it would be possible to accompany this with a formal specification development along the ideas of [8, 6, 7] mentioned in Section 3.

In the future, we intend to extend work in several directions. We would like to describe the behavioural aspects associated with the architectural patterns, and to specify in more detail the communication between the different architecture components, for instance using communication patterns. As we rely on a problem decomposition, e.g. based on the different identified use cases, we need to develop in more detail on how should the recomposition be achieved.

Moreover, since our approach aims at a guided and integrated use of several techniques and several patterns, we would like to explore how to integrate the use of design patterns in this development.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] R. Bharadwaj and C. Heitmeyer. Hardware/software co-design and co-validation using the scr method. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99)*, 1999.

[3] M. Bidoit and P. D. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language*. Number 2900 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems, North-Holland*, 14:25–59, 1987.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[6] C. Choppy and M. Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. In M. Wirsing, R. Hennicker, and D. Pattinson, editors, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, LNCS 2755, pages 205–220. Springer Verlag, 2003.

[7] C. Choppy and M. Heisel. Une approache à base de "patrons" pour la spécification et le développement de systèmes d'information. In *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2004*, pages 61–76, 2004.

[8] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers*, LNCS 1827, pages 104–123. Springer Verlag, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

Figure 18: Composed Architecture

[9] J. M. D. L. Parnas. Functional documents for computer systems. In *Science of Computer programming*, volume 25, pages 41–61, 1995.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[11] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating Software Requirements and Architectures using Problem Frames. In *Proceedings of IEEE International Requirements Engineering Conference (RE'02)*, Essen, Germany, 9-13 September 2002.

[12] M. Heisel and D. Hatebur. A model-based development process for embedded systems. Submitted for publication, 2004.

[13] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[14] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[15] L. Lavazza and V. D. Bianco. A UML-Based Approach for Representing Problem Frames. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEE Press, 2004.

[16] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture Driven Problem Decomposition. In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04*, Kyoto, Japan, 6-10 September 2004.

[17] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL extension for dynamic reactive systems – version 1.0 – summary. Technical Report DISI-TR-03-36, Università di Genova, Italy, 2003.

[18] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[19] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[20] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.