

A Formal Architectural Design Patterns-Based Approach to Software Understanding ^{*}

P.S.C. Alencar, D.D. Cowan, Thomas Kunz, and C.J.P. Lucena
Department of Computer Science
University of Waterloo
Waterloo, Ont, Canada N2L 3G1

Abstract

Mastering the complexity of programs and systems, particularly distributed systems, should lead to significant improvements in program and system understanding. In this paper we present a formal approach for (distributed) software understanding based on abstraction hierarchies represented by architectural design patterns. This approach allows us to model the distributed software applications through a formal model representing the underlying structure. The representation uses instances of architectural design patterns which can hide details that may be irrelevant in specific situations. The formal models which are produced could be used as a basis for reasoning, code generation, and measuring the "goodness" of a design.

1 Introduction

Complexity is a substantive barrier to program and system understanding, which becomes even more significant when we consider distributed applications. Many of the features of distributed applications, including the complex interprocess interoperability and communication mechanisms, parallelism, non-deterministic execution, lack of global components such as clock and memory, communication delays, and sheer size, contribute to the difficulty of understanding. In addition, the fact that the trace of a thread of execution of a system of interconnected objects is complicated by the callback-driven flow of control add extra dimensions to be considered.

In this paper we present a formal approach to distributed software understanding based on formal architectural design patterns [1, 2] in order to manage

the underlying numerous challenges. This formal top-down approach to software understanding emphasizes thus that abstraction hierarchies are used to minimize the complexity involved. In this context a software application is viewed at a very high level of abstraction and, in the course of the understanding process, more difficult or interesting parts of the application are isolated and examined at a lower level of abstraction. Related tools aim to reduce the complexity of the distributed application by grouping processes into process clusters represented by formal distributed architectural design patterns. Process clusters represent subsystem structures, exposing the overall architecture of the subject system, and are invaluable in recovering the original design decisions. The formal abstractions can be automatically derived from the structuring concepts provided by a programming language.

We adopt the view that automatic process clustering can be based on reverse engineering ideas: the reconstruction of a possible design of an application by using the given software system as the only information source. The reconstructed abstract view does not necessarily correspond to that of the application designer, although it is in fact a plausible analysis of the application. This makes reverse engineering particularly valuable in case of a mismatch between specification and implementation. In fact, this is part of our view of the software construction process as we shall discuss later. We are based on the tested hypothesis that reverse engineering tools can be powerful enough to construct abstract hierarchies from a given source [3, 4].

In this way we consider a catalogue of simple and basic sub-structures or recurring high-level patterns for software construction and organization in-the-large (architectural design patterns) as the main vehicle for representing and understanding the software applications. Recent work that emphasizes the effort in deter-

^{*}The work described here has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the National Research Council of Brazil (CNPq), and WATCOM.

mining the high-level organization in software is evidenced by the interest in architectural styles or idioms, and in the design and object-oriented communities in design patterns. We advocate that software understanding at a higher abstraction level is enhanced if you know these basic software architecture building blocks. These building blocks are considered here as formal architectural design patterns based on a reuse model.

Design patterns can be defined as a means to achieve large-scale reuse by capturing successful software development design practice within a particular context [5, 6, 7]. Patterns should not be limited in what they can describe and can be used to encapsulate good design practices at both the specification and implementation levels. Thus, design patterns can be applied at many different levels of abstraction in the software development life-cycle, and can focus on reuse within architectural design as well as detailed design and implementation. In fact, a system of patterns for software development should include patterns covering various ranges of scale, beginning with patterns for defining the basic architectural structure of an application and ending with patterns describing how to implement a particular design mechanism in a concrete programming language. Design patterns that represent the basic architectural structure of a distributed software application are the ones we are considering here. These patterns reflect typical programming (or programming-in-the-large) paradigms for distributed software system application development.

Most published research [5, 7] in design patterns has been described in a structured but informal notation, and has focused on implementation-oriented patterns rather than architectural ones. For example, one publication [5] contains descriptions of patterns using text and diagrams, and has grouped patterns into three major categories. These descriptions can be viewed as an informal recipe or process for producing instantiations of specific patterns in languages such as Smalltalk or C++. Even when architectural issues are considered [6], the software architectural design is expressed only through informal patterns. An architectural pattern is based on selected types of components and connectors, together with a control structure that governs their execution.

We use a formal approach to design patterns [1] which encompasses patterns at different levels of granularity, ranging from architectural to implementation descriptions. There are two aspects to design patterns that are considered in this formal presentation: the process of producing specific instantiations of a design

pattern, and the use of formally defined components or objects to substitute in these instantiations.

If the process is defined through a process programming language with formal syntax and semantics, then any ambiguities in the process of design pattern instantiation should be eliminated. Reducing or even eliminating ambiguity should make it easier to derive code consistently and perhaps even lead to some automation of the code production for the particular instantiation of a design pattern [2]. Substituting formally defined components into an instantiation could permit a formal reasoning process about the resulting system. We currently have established two different frameworks for reasoning about designs [8, 9] of this type.

Recent investigations [9] have shown how both a formal model and a prototype can be derived from a single component-based specification, thus providing a strong link between formalism and implementation.

The formally defined components are based on the Abstract Data View (ADV) approach [10, 11, 12] which uses a formal model [8, 13] to achieve separation by dividing designs into two types of components: objects and object views, and by strictly following a set of design rules. Specific instantiations of views as represented by Abstract Data Views (ADV) and objects called Abstract Data Objects (ADO) are substituted into the design pattern realization while maintaining a clear separation between view and object. Currently the ADV and ADO components are specified using temporal logic and the interconnection between components is described in terms of category theory. Each design pattern has an associated process program that describes how to substitute these components to create a specific instantiation. In fact, this framework can be seen as a formal approach for a system of design patterns.

The basic idea behind the proposed formal approach to distributed software understanding is to reverse engineer applications to extract the various levels of architectural design patterns that have been used and/or to produce a similar but better design based on an architectural pattern structure. In this sense, this paper presents an architectural design pattern-based approach to reverse engineering applied to the domain of distributed applications. This exercise would not only assist us in understanding the application, but also in evaluating and enhancing the design, proving formal properties, and through some form of simulation verifying the model against the existing application. We should point out that tools have been developed, implemented, and tested to facilitate the

understanding of the execution of large and complex distributed applications coded in the particular programming language Hermes [4, 3, 14]. However, while the static source analysis components of the tools are language-specific, we believe that the general approach described here is directly applicable in other target environments. In this way, the presented approach can be seen as a formal counterpart of the informal approach to process clustering presented in [4].

In addition to providing a basis for code generation and reasoning, the underlying formal approach to architectural design patterns can be used to address several other important issues. This approach can indicate possible steps towards the definition of a unique process language vocabulary that can describe the interconnection mechanisms at the object, module, and architectural description levels. It can also provide a foundation for the definition of an integrated formal approach to software system specification and design that considers various levels of abstraction, including the architecture and design. The formal components can be specified at different levels of detail and thus various program/system/architecture features such as data, control, functionality, behavior, communication, concurrency/distribution/timing concerns can be included in the description as it is developed. Hopefully a uniform formal approach such as the one described in this paper can enhance the quality of the architectural descriptions.

Furthermore, a formal approach that links software understandability and software architecture analysis has a number of benefits. It might allow us to decide which of some alternate designs is better suited in particular cases by considering architectural design metrics. It also allows design inspections and design activity in a higher-level of abstraction. Besides helping in the understanding of the software application, this approach also enhances communication by the use of suitable abstraction hierarchies. Finally, the approach might also lead to tools that can assist the designer to introduce non-functional qualities to the software system application.

2 A Design Pattern-Based Reverse Engineering Process

The formal framework for software understanding is based on the assumption that the construction of a software system can be viewed as in Figure 1 where the formal system or the abstract architectural pattern-based description is the basic design from which we de-

rive an implementation and a reasoning model. Thus, ideally we can maintain one description of the system from which we both reason about properties and produce operating software.

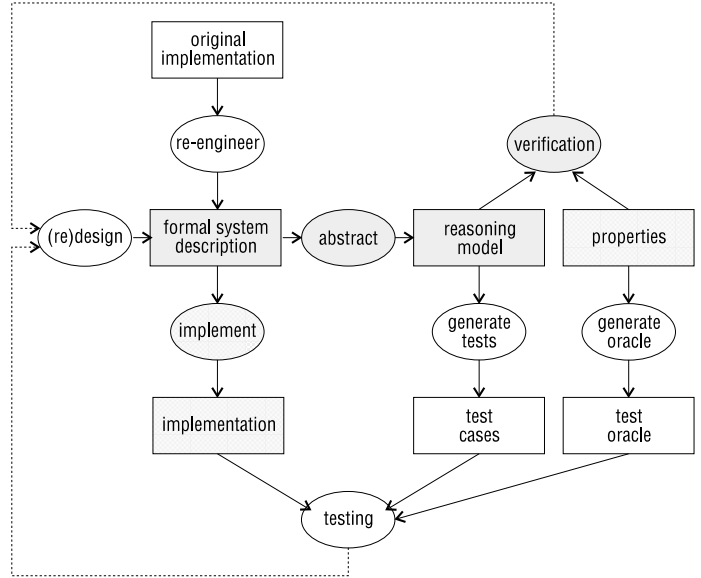


Figure 1: The Process of Software System Construction

This paper presents a formal version of the tool-based approach [4] developed to derive abstraction hierarchies automatically based on a process view. This view focusses on the process structure of a distributed application, describing which processes communicate with one another. The tools combine run-time information with a static source analysis to group processes and clusters into higher-level clusters. Cluster membership is determined by rules based on programming paradigms. In our case, some typical programming paradigms are captured by formal architectural design patterns that represent the process clusters. We will give a detailed description of these formal patterns and describe their associated cluster rules. In other words, process abstraction allows groups of processes to be combined into clusters, eliding the interaction among processes in the cluster. Combining clusters into higher-level clusters leads to an abstraction hierarchy. Together with existing work on how to display arbitrary process clusters correctly (by visualization techniques as in [4]) this is sufficient to display the abstract behavior of a distributed software application under a process view.

The basic idea behind the process clustering or architecture design pattern-based approach is as follows. The process structure of a distributed software appli-

cation is determined by the architectural design patterns applied during the development of the application. The clustering tool uses rules that describe process structures likely to result from the application of specific paradigms. Each of these rules is of the following form: processes with such-and-such semantic description and this-and-that interprocess communication structure should be grouped together, forming a cluster with a specific semantic description. Assigning clusters a semantic description by rules allows for the repeated application of the clustering rules, potentially deriving a hierarchy of process clusters or distributed architectural design patterns. The initial set of semantic descriptions for each application process is obtained by a static source code analysis tool. The inter-process communication structure at run-time is deduced from the event stream generated during the software execution. In the case more than one cluster rule can be applied at one step, the conflicting clustering alternatives are resolved by a quantitative cluster evaluation measure added to the tool [15]. The measure determines the coupling and cohesion of each potential process cluster and is primarily used in the resolution of clustering conflicts. Only the cluster with the best evaluation is built.

The formal design pattern system representation model is obtained by reverse engineering the existing application (implementation) you want to understand or analyze. The models generated from the pattern-based formal descriptions according to Figure 1 could also be used to aid in the testing process either by serving as a basis for test case generation or by providing means for measuring test coverage. The verification and test processes can result in system redesigns for various reasons as, for example, in the case that the most suitable architectural design pattern has not been used. In this paper, however, we concentrate on the reverse engineering process and the formal pattern-based system descriptions.

3 Formal Architectural Design Pattern Description

In this section we present a formal description of distributed architectural design patterns and describe the underlying ADV reuse model and its associated formal schemas.

3.1 A Formal Description of Design Patterns

The ADV model supports reuse since it divides an application into a set of specialized objects (separation of concerns) each of which may be used in other designs. However, we would like to “glue” these objects into reusable systems, that is, systems which are easily maintained over time. Design patterns as proposed in [5] support this form of reuse. Each design pattern is a meta-description of a solution for a problem that occurs frequently in software design. The application of the meta-description results in a collection of a few objects that form a specific instantiation of such a design problem.

The acceptance of reusable descriptions, such as design patterns, is highly dependent on easily comprehensible definitions and unambiguous specifications. We address both issues in a single formalism for design pattern application.

In order to formalize the application of design patterns, we introduce development constructors which are based on schemas that indicate how to apply a pattern. We define design pattern constructors to consist of a language-independent part and a product text specification, where a specific language is adopted; this approach is similar to that described in [16].

The language-independent part of the structure should clearly define the characteristics of a design pattern. According to [5], a pattern is composed of four essential elements: *pattern name*, *problem statement*, *solution*, and *consequences*.

Appropriate pattern names are usually important factors to assist developers in the specification of a system. In the case of reusable modules, the vocabulary of patterns could be one way of guiding the user to choosing suitable modules for the solution of particular problems.

A problem statement is a description of the circumstances in which to apply a design pattern, and clarifies the pattern objectives. In the development constructor structure shown in Figure 2, such a statement is described by an *Objective* section.

Applying a pattern in the context of a specific problem requires a process description, and so we specify this process in terms of primitive development constructors and parameters. The primitive constructors applied to pattern construction are organized in a section of the schema called *Subtasks*, while input parameters used in this process are declared in the *Parameters* section.

The consequences of an application of a pattern provide a description of the results of using such struc-

```

Operator Pattern Name
  Objective
    description of the intent of the pattern
  Parameters
    external elements used in the pattern definition
  Subtasks
    description of pattern in primitive constructors
  Consequences
    how the pattern supports its objective
  Product Text
    language-dependent specification of pattern
End Operator

```

Figure 2: Development constructor structure for a design pattern.

ture in a software system. The roles of the components within the pattern objectives are also illustrated. This section may be helpful in evaluating the suitability of a pattern in a specific context. These ramifications are specified in the *Consequences* section of a pattern schema.

The language-dependent part of the pattern constructors (the product text part) describes the result of the application of a pattern as a specific formal representation. Since design patterns are solution abstractions, a template of the pattern should be a helpful instrument in guiding the user to a particular specification. Such templates are illustrated in the pattern development constructors using the formalism of ADV/ADO schematic representations described in the following section.

We assume a formal object-oriented reuse model [8, 13] that is based on the Abstract Data View (ADV) component model [10, 11, 12] coupled with a process for composing designs. The ADV model which decomposes designs into objects (Abstract Data Objects or ADOs) and views of objects (ADVs) allows us to address issues such as abstraction, encapsulation, separation of concerns, coupling and cohesion. The formal architectural system description is produced by composing a set of objects (Abstract Data Objects or ADOs) and views of those objects (Abstract Data Views or ADVs) through design patterns.

Essentially, the schemas contain the formal counterparts of all the three OMT [17] object informal descriptions: the object model, the functional model and the dynamic model. We should also note that ADVs and ADOs can be represented in various abstraction levels: from abstract specifications to implementations in a particular concrete programming language. The specification syntax of the whole schema which is based on ones described in [18] is presented

```

ADV ADV_Name for_ADO ADO_Name
  Declarations
    Data Signatures
      sorts and functions
    Attributes
      observable properties of objects
    Causal Actions
      list of possible input actions
    Effectual Actions
      list of possible effectual actions
    Nested ADVs
      allows composition, inheritance, sets, ...
  Static Properties
    Constraints
      constraints in the attributes values
    Derived Attributes
      non-primitive attribute descriptions
  Dynamic Properties
    Interconnection
      communication process description
    Valuation
      the effect of events on attributes
    Behavior
      behavioral properties of the ADV
End ADV_Name

```

Figure 3: A descriptive schema for ADVs.

essentially through a temporal logic formalism [13, 19]. Every ADV or ADO structure is subdivided into three sections. A declaration part contains a description of all the elements that compose the object including sorts, functions, attributes, and actions. The static properties part defines all the properties which do not affect the state of the object. Dynamic properties establish how the states and attribute values of an object are modified during its lifetime.

Figure 3 shows the structure of the schemas to be used in the specification of ADVs. Causal Actions correspond to input events while Effectual Actions correspond to method invocations. ADO schemas, which are not illustrated here, are structured similarly to ADVs, except that ADOs do not support causal actions.

4 Formal Distributed Architectural Design Patterns

In this section we provide a formal description of distributed architectural design patterns that can be used to represent the abstraction hierarchies that help us to understand the distributed software applications. For completeness, we also provide a formal description of design patterns at the program design level.

```

Operator Design Pattern Adapter
Objective
  Modify the interface of an object
Parameters
  Objects: ADAPTEE, TARGET;
Subtasks
  1 - Specify Adaptation Object:
    ADAPTEE, TARGET → ADAPTER
    1.1 - Create Object: → ADAPTER
    1.2 - Compose Objects:
      ADAPTEE, ADAPTER → ADAPTER
    1.3 - Inherit Objects:
      TARGET, ADAPTER → ADAPTER
    1.4 - Specify Links:
      ADAPTEE, ADAPTER → ADAPTER
Consequences
  ADAPTER will contain most of ADAPTEE function-
  ality available through the TARGET object interface
Product Text
  ADV/ADO ADAPTER
  Declarations
    ...
    Nested ADVs/ADOs
      Compose ADAPTEE;
      Inherit TARGET;
    ...
  Dynamic Properties
    ...
  Interconnection
    With ADV/ADO ADAPTEE
      TargetActions ↦ AdapteeActions;
  End ADAPTER
End Operator

```

Figure 4: Specification of adapter pattern constructor.

We have chosen the adapter design pattern to illustrate formal descriptions of patterns used to support program designs and we also formally describe some typical distributed design patterns that are used to describe system architectures. In this way we illustrate how the same basic formalism may be used to specify software at two different levels of abstraction. We present the program patterns first, since they have appeared elsewhere [5], and the interested reader may wish to compare the two different styles of presentation.

4.1 Program Design Patterns

We have chosen the adapter pattern specifically to illustrate how the pattern constructor described in Figure 2 is applied. A more complete set of program design patterns is given in [2]. There is a substantial difference between the pattern specifications presented in [5], and the specifications introduced in this paper. The patterns in [5] are based on OMT diagrams, infor-

mal descriptions in English and C++ templates, and are much closer to the implementation level than the version of the pattern descriptions we present in this paper. Our version of the patterns is based on the pattern constructors and the specification formalism associated with the ADV design approach.

In the following example the pattern specifications provide explanations and directions to instantiate a design through the use of well-defined development operators and incomplete object schemas. The development operators are sequentially numbered in a section called *Subtasks*, while the object schemas are defined in the language-dependent section called *Product Text*, which provides reusable patterns for the program design. Other sections complete the design pattern specifications by providing additional information.

The *adapter* specification schema is shown in Figure 4. The objective of the *adapter* is to modify the interface of a given object to conform to the needs of a client object. It is generally used to produce compatibility between two objects. The adapter might be seen as an object which is a wrapper for another object, and it can be used for adaptations of interface objects (ADV) as well as application objects (ADO). An adapter object could also be regarded as a view (ADV) for an application object (ADO).

The subtasks which specify how to instantiate a pattern are given in a task or function notation of the form: $f : x \rightarrow y$ where f is a function, x is a list of parameters for the function f , and y is the result of applying the function f . In the *adapter* pattern the function “Create Object” returns a copy of the ADAPTER Product Text while the function “Inherit Objects” takes the two arguments TARGET and ADAPTER and returns the modified Product Text for ADAPTER. In the context of C++ the ADAPTER text would have the words “inherit from TARGET” inserted in the appropriate location.

The current specification approach differs from the design proposed in [5], in that the link between a view and its application object is represented by an ADV *mapping* design mechanism. This approach does not describe the implementation of the link, but indicates a mapping or morphism between elements of the objects involved. In contrast, the design described in [5] proposes a design technique that is closer to the implementation than the proposed *mapping*.

4.2 Distributed Architectural Design Patterns

In order to understand or analyze the distributed software applications the process clusters or dis-

tributed architectural design pattern-based software descriptions are developed by using semantic information characterizing individual processes and the relationship among processes. A process can be characterized, for instance, by its type (worker or server), and its complexity (simple or complex). Server processes offer one or more services in an endless loop. All other processes (i.e., those with no such outer infinite loop) are workers, advancing the state of the computation.

The derivation of appropriate clustering rules is based on the following idea. Programming paradigms are collections of conceptual patterns that influence the design process and ultimately determine the structure of a software system. By analyzing programming paradigms used for distributed programming, potential program or software structures can be identified. Using rules that describe these structures, a tool can analyze the runtime behavior of a distributed application and automatically generate a hierarchy of process clusters.

In this section we describe how typical program paradigms for distributed computing can be represented as formal architectural design patterns. We also describe their associated process cluster rules. The process clustering rules capture the essential structural aspects of the programming paradigms being considered. A more detailed description of these cluster rules can be found in [3, 4]. For more details and the formal description of other distributed patterns, e.g. the layered system, the client server, the complex server, the administrator, the compute-aggregate-broadcast, the pipeline and filters, the peer groups, the iterative relaxation, and the divide conquer, see [20].

The distributed architectural design pattern we formally describe here is the master-slave. This pattern consists of a master component and a set of slave components (at least two). The slaves are independent components that each provide the same service. The master is the only component to which the clients of the service communicate, and the service is only accessible through the master's public interface. The master is responsible for invoking the slaves and for producing the final result which is computed from the results returned by the slaves. Thus, the master does not provide the service directly, but delegates the task (or parts of it) to several independent suppliers and then returns the selected result to its clients.

From this discussion we can see that the Master-Slave design pattern has three kinds of participants: the client, the master and the slaves. The client re-

quires a certain service in order to complete its own task. The master organizes the invocation of replicated services and decides which of the results returned by its slaves is to be passed to its clients. The slaves are the ones that implement the critical service. In this case a pool of worker or slave processes stands ready to solve the subproblems as they are created and distributed by the master. Contrary to the divide and conquer paradigm, the subordinate problems are not necessarily homogeneous. The Master-Slave Distributed Design Pattern is specified in Figure 5.

The associated process cluster rule scans all processes for simple workers that are called from exactly one other process and clusters these processes with the calling process. The underlying notion is that the calling process passes on some of its work to the worker being called. The calling process can be of type server or worker. The resulting cluster is assigned the name and type of the calling process.

5 Conclusions

Reuse and maintenance are two activities that require the understanding of the application software systems. The high complexity of distributed applications makes the software understanding process very hard. To manage complexity, we have proposed a formal top-down approach to software understanding that uses suitable abstraction hierarchies based on distributed architectural design patterns. This work can be seen as a formal counterpart of a tool-based approach to derive suitable abstraction hierarchies automatically and also allows abstract visualization of distributed executions [4]. Given the complexity of the distributed applications, deriving such abstraction hierarchies manually is too tedious and error prone.

Our work addresses the development of theoretical foundations and the exploration of the practical implications of automatic abstraction tools for concurrent and distributed systems. We have adopted a reverse engineering formal approach in which the only available sources of information are the application source and the information about the runtime behavior. Although the automatic abstraction tools was first presented for distributed applications written in Hermes, we should note that the emphasis of this work is on the theoretical foundation for the process clustering activities and that both the programming paradigms and the related process rules have only been informally described until now [3, 4].

In this article we have described a formal architectural design patterns-based approach to software

understanding. According to this formal approach we can reverse engineer distributed software applications to extract the various levels of architectural design patterns that have been used or produce a similar design based on alternative pattern structures. This exercise assists us to understand the software application, and might assist us to evaluate and enhance the design, prove formal properties, and through some form of simulation verify the model against the existing application.

We have also described a formal approach to achieve large-scale reuse by capturing successful software designs expressed by design patterns. We believe we have demonstrated our techniques can be used to describe design patterns at both the architectural and program design levels of software design [1]. Our process programming approach to design patterns allows us to define primitive design pattern (sub)tasks or constructors that can be used in the development of specific instantiations. We believe that this approach clarifies both the application and structure of the design patterns. In addition, the architectural patterns have been specified so as to incorporate the concept of objects (ADOs) and object views (ADVs), a goal of subjectivity [21, 22].

Using this formal approach including objects and object views directs us toward several important results. By using formally defined components we are able to reason about the design (the formal design patterns-based architectural descriptions) and prove properties as shown in [8, 9]. Of course systems often do not yield to formal approaches because of their size and complexity. However, the formal approach could still produce useful results in that the models generated could be used to aid in the testing process [9] by serving as a basis for test case generation [23], or by providing a means for measuring test coverage.

Another ongoing research topic is the investigation of code generation from design patterns incorporating ADO and ADV schemas. Experiments with the process program description has shown that design patterns can yield corresponding C++ schemas which can be completed by the designer through an interactive dialogue. In fact, we are currently experimenting with C++ code generation and we are constructing an interactive tool to generate and complete the schemas [24]. Thus, we should be able to produce a single design representation from which we can both reason about formally specified properties and generate code. This might also allow us to formulate the code generation process of the formal architectural pattern-based software system descriptions. Similar work is reported

for example in [25, 26], where a formal description of the module interconnections drives the generation of appropriate “glue” code. However, important differences exist in our work, in that we provide not only source-code generation, but also a formal reasoning model based on these formal descriptions. Furthermore, the cited work concentrates on the instantiation of complete modules, whereas our formalism allows the specification of design patterns that cover only parts of a distributed application, via the Parameters section.

We also believe that with the correct program library or repository of architectural level objects and corresponding architectural design patterns we might be able to extend this tool to produce “running” architectures. Furthermore, this repository might support the retrieval of reusable components by structure using design patterns. This might be implemented by introducing a process language based on predicates that allows us to represent various types of interconnections including objects, MILs and architectures. This process language is also currently under investigation.

Finally, we have also mentioned in the paper how the ADV model may be viewed as a formalism that supports the integration of design patterns and subjectivity. This relationship of objects and object views to subject-oriented programming [21, 22] has been explored in [27], where ADOs and ADVs could be interpreted in terms of subjects, and subject activations. Thus, we believe we can link the component-based reuse approach embodied in subjectivity to the process-based reuse of designs as exemplified in architectural design patterns.

References

- [1] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena, “A Formal Approach to Architectural Design Patterns”, Tech. Rep. CS-95-38, University of Waterloo, Waterloo, Ontario, Canada, August 1995.
- [2] P.S.C. Alencar, D.D. Cowan, D.M. German, K.J. Lichtner, C.J.P. Lucena, and L.C.M. Nova, “A Formal Approach to Design Pattern Definition & Application”, Tech. Rep. CS-95-34, University of Waterloo, Waterloo, Ontario, Canada, August 1995.
- [3] T. Kunz, “Abstract Behaviour of Distributed Executions with Applications to visualization”, Phd thesis, University of Darmstadt, Computer

- Science Department, Darmstadt, Germany, May 1994.
- [4] T. Kunz and J. P. Black, "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, 1995.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
- [6] J.O. Coplien and D.C. Schmidt, Eds., *Pattern Languages of Program Design*, Addison-Wesley, 1995.
- [7] P. Coad, *Object Models: Strategies, Patterns & Applications*, Yourdon Press, 1995.
- [8] P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena, "A Logical Theory of Interfaces and Objects", Tech. Rep. CS-95-15, University of Waterloo, Waterloo, Ontario, Canada, 1995.
- [9] P. Bumbulis, P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena, "Combining Formal Techniques and Prototyping in User Interface Construction and Verification", in *2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'95)*. 1995, Springer-Verlag Lecture Notes in Computer Science, to appear.
- [10] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien, "Abstract Data Views", *Structured Programming*, vol. 14, no. 1, pp. 1-13, January 1993.
- [11] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien, "Application Integration: Constructing Composite Applications from Interactive Components", *Software Practice and Experience*, vol. 23, no. 3, pp. 255-276, March 1993.
- [12] D.D. Cowan and C.J.P. Lucena, "Abstract Data Views: An Interface Specification Concept to Enhance Design", *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 229-243, March 1995.
- [13] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova, "Formal specification of reusable interface objects", in *Proceedings of the Symposium on Software Reusability (SSR'95)*. 1995, pp. 88-96, ACM Press.
- [14] T. Kunz, "Process Clustering for Distributed Debugging", *ACM SIGPLAN Notices*, vol. 28, no. 12, December 1993.
- [15] T. Kunz, "Developing a Measure for Process Cluster Evaluation", Technical Report TI-2/93, University of Darmstadt, Computer Science Department, Waterloo, Ontario, February 1993.
- [16] N. Levy and G. Smith, "A Language Independent Approach to Specification Construction", in *Proceedings of the SIGSOFT'94*, New Orleans, LA, USA, December 1994.
- [17] J. Rumbaugh et al., *Object-Oriented Modelling and Design*, Prentice-Hall, Englewood Cliffs, 1991.
- [18] Martin Gogolla, Stefan Conrad, and Rudolf Herzig, "Sketching Concepts and Computational Model of TROLL Light", in *Proceedings of Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, Berlin, Germany, March 1993, Springer.
- [19] Zohar Manna and Amir Pnueli, *The temporal logic of reactive systems: Specification*, Springer-Verlag, 1992.
- [20] P. S. C. Alencar, D. D. Cowan, T. Kunz, and C. J. P. Lucena, "A Formal Architectural Design Patterns-Based Approach to Software Understanding", Technical Report 95-42, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, August 1995.
- [21] William Harrison and Harold Ossher, "Subject-Oriented programming (A Critique of Pure Objects)", in *OOPSLA '93*. 1993, ACM.
- [22] William Harrison, Harold Ossher, Randal B. Smith, and Ungar David, "Subjectivity in Object-Oriented Systems, Workshop Summary", in *OOPSLA '94*. 1994, ACM.
- [23] Bogdan Korel, "Automated software test data generation", *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870-879, Aug. 1990.
- [24] P.S.C. Alencar, D.D. Cowan, K.J. Lichtner, C.J.P. Lucena, and L.C.M. Nova, "Tool Support for Formal Design Patterns", Tech. Rep. CS-95-36, University of Waterloo, Waterloo, Ontario, Canada, August 1995.

- [25] Keng Ng and Jeff Kramer, “Automated support for distributed software design”, in *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering*, Toronto, Ontario, Canada, July 1995, pp. 381–390.
- [26] Jeff Kramer, Jeff Magee, Morris Sloman, and Naranker Dulay, “Configuring object-based distributed programs in REX”, *Software Engineering Journal*, vol. 7, no. 2, pp. 139–149, Mar. 1992.
- [27] P.S.C. Alencar, D.D. Cowan, and Lucena C.J.P., “Abstract Data Views as a Formal Approach to Subject-Oriented Programming”, Tech. Rep. CS-95-19, University of Waterloo, Waterloo, Ontario, Canada, May 1995.

Operator Design Pattern Master-Slave

Objective

Handle the computation of replicated services

Parameters

Objects: CLIENT;

Subtasks

1 - Specify Master Object: CLIENT \rightarrow MASTER

1.1 - Create Object: \rightarrow MASTER

1.2 - Compose Objects:

CLIENT, MASTER \rightarrow MASTER

1.3 - Specify Links: MASTER \rightarrow MASTER

2 - Specify Slave Objects:

MASTER \rightarrow SLAVE_{*i*} $i = 1, \dots, N$

2.1 - Create Objects: \rightarrow SLAVE_{*i*} $i = 1, \dots, N$

2.2 - Compose Objects:

MASTER, SLAVE_{*i*} \rightarrow MASTER $i = 1, \dots, N$

2.3 - Specify Links: MASTER \rightarrow MASTER

Consequences

The replicated service provided by the SLAVE_{*i*} (the slaves) is offered to CLIENT through the MASTER after selection

Product Text

ADV/ADO CLIENT

Declarations ...

Attributes

ClientAttributes;

Actions

ComputeTask;

End CLIENT

ADV/ADO MASTER

Declarations ...

Attributes

MasterAttributes;

Actions

OutCom; Service; $\overline{\text{Service}}$;

SelectResult(Res₁, ..., Res_{*N*});

Dynamic Properties ...

Interconnection

With ADV/ADO CLIENT

Service \mapsto ComputeTask;

With ADV/ADO SLAVE_{*i*} $i = 1, \dots, N$

Service \mapsto ServiceSlave_{*i*};

...

$\overline{\text{Service}}$ \mapsto ServiceSlave_{*N*};

Behavior

$s_1 \wedge \overline{\text{Service}} \rightarrow s_2$

$s_2 \wedge \overline{\text{Service}} \rightarrow s_3$

End MASTER

ADV/ADO SLAVE_{*i*} $i = 1, \dots, N$

Declarations ...

Attributes

Slave_{*i*}Attributes;

Actions

ServiceSlave_{*i*};

End SLAVE_{*i*}

Figure 5: Specification of the Master-Slave pattern constructor.