SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-04

Guiding Object-Oriented Design

Anthony MacDonald and
David Carrington

May 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Guiding Object-Oriented Design

Anthony MacDonald and David Carrington

Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia

email: {anti, davec}@cs.uq.edu.au

Abstract

In this paper we show how software architectural styles can be used to guide object-oriented design. Design guidance is important as the initial phases of software design significantly impact software quality. We use two different architectural styles to guide the design process from a formal specification to a design. Software architectural styles assist by providing a different level of reuse than currently practised in software design. A style provides a framework for top-level structure and guides selection of components and interfaces. In this paper we show the impact of software architectural styles on software design.

1 Introduction

We believe the initial phases of software design [4, 23] have significant impact on software quality. When considering these early phases of software design, we have two specific areas of interest: formal methods and software architecture. Firstly, we are interested in the impact of formal methods on the design process, especially the use of formal specifications to guide design. Secondly, we wish to investigate the effect of explicitly considering software architecture and how this impacts both the design process and quality.

The object-oriented community has been keen in their uptake of design patterns [9] for a level of reuse above the code reuse available through libraries, frameworks etc. Shaw [25], relates software architecture to patterns and shows how they both are seeking Alexander's [2] goal, "the quality without a name" [7]. Software architecture, while seeking the same goals as patterns, offers a different granularity of reuse to the software designer. Gamma et al. [8] say design patterns define something less than the complete application and are at best, micro-architectures. Monroe and Garlan [20] claim that software architecture offers a different granularity of reuse in their recent paper. They are interested in "(a) the use of components, connectors, and configurations as the basic vocabulary of reusable assets, and (b) the exploitation of architectural style to aid in the classification, retrieval, and instantiation of those assets". This guiding of design by software architecture could be called conceptual reuse, as opposed to concrete reuse.

In this paper we show how software architectural styles [1] can be used to guide object-oriented design. Meyer [16] states "Object-oriented design is the method which leads to software architectures based on objects [that] every system or subsystem manipulates". Software architectural styles enable software designers to improve seamlessness and traceability in design. This is achieved without sacrificing a clear separation of concerns between phases of the software life cycle.

In this paper, software architectural styles are used to guide the design and eventual implementation of two systems from a non-object-oriented formal specification. A controller for the production cell, a case study in safety-critical systems, was formally specified[1] in the Z notation [26, 3]. From this formal specification, two separate design paths are taken, each guided by a different software architectural style. The first is based on a standard abstract data type approach and the second is based on an event system. The software architectural style not only assists in guiding the design process, but also guides the designer to the relevant sections of the formal specification and hence helps bridge the gap between phases in the software lifecycle. The software architectural style assists the transition from a specification to a design. We aim to show that using architectural styles can give both conceptual reuse and design guidance.

Section 2 introduces the concept of software architecture and two architectural styles. Section 3 gives an overview of the production cell and the formal specification. The designs generated using each architectural style are discussed in Section 4. The paper summarises how explicit consideration of software architecture can aid the software designer in Section 5 and closes with conclusions.

2 Software architecture

The term, software architecture [10, 22], describes the high-level organisation of computational elements and the interactions between these elements, or simply the top-level structure of a software system. By abstracting to the architectural level, we can view the system as a graph consisting of a collection of components and connectors. Components are the primary computational elements of the system, while connectors are the interactions between these

[1]An alternative specification using Object-Z [6, 24] was also developed [15].

components. These interactions may be simple procedure calls, message passing, remote procedure calls, event-broadcasts, etc.

An architectural style defines a family of systems in terms of the top-level structure. Common architectural styles include abstract data type (ADT), event, blackboard, and pipe and filter architectures [10].

An individual system can be documented using its architecture to aid understanding and improve maintenance. An architectural style not only supports this aspect of software engineering, but it can also be used as a design guide. A style enables the designer to build a system around a defined and understood structure, improving understanding in all phases of the software lifecycle.

Design is a broadly used word; in this paper we are focusing on architectural design: the stage of design where both the connectors and components within the final system are chosen and the top-level structure is decided. The properties of an architecture can be viewed from two perspectives: at the system level where system-wide attributes are considered and at the module[2] level where module-specific attributes are considered. Using architectural styles, architectural design can be described as the process of mapping a particular specification onto an architectural style. This generates an instance of the architecture (a design) which meets the requirements captured in the specification.

Two architectural styles are used throughout this paper: an ADT-based architecture and, an event-based architecture. Each architecture is introduced and discussed in the following sections.

2.1 An ADT-based architecture

An architectural style is ADT-based when the core components are ADT modules and the connectors are routine calls (function and procedure calls).

The defining attribute of an ADT-based architectural style is at the module level and is the presence of encapsulated data. This data can only be accessed by procedure or function calls. The encapsulation of data is based on the principle of information hiding [21], as the physical representation and implementation of the module is irrelevant to users of the module (this includes the designer of the system containing the ADT initially). By encapsulating the data, the module is responsible for the integrity of the contained data.

At the system level, an ADT module can be viewed as an object containing state and operations to transform and query the state. A system based on an ADT architecture generally requires centralised control and all routines are explicitly invoked.

Object-oriented designs are generally based on ADT architectures. Meyer's [16] alternative definition for object-oriented design is "Object-oriented design is the construction of software

[2] Module is used here as a logical unit and is not programming- language-specific. In an Eiffel [18] context, *class* could be substituted.

systems as structured collections of abstract data type implementations".

Concrete reuse is possible at the individual module level. This type of reuse is generally as part as a library, like EiffelBase [19], or as part of a framework [11, 5].

2.2 An event-based architecture

An architectural style is event-based when the core components are modules (no particular defining features) and the connectors are events. The defining attributes of an event-based architectural style are at the system level and result from events being the predominant connectors.

The system can be viewed as a collection of events and registered responses to those events (there may be, and often are, multiple responses to a single event). The event manager is treated as part of the supporting system and hence the event-based system appears to have a decentralised or distributed control mechanism. This method of routine invocation is implicit and is the primary method of invocation in event-based systems. Explicit invocation still occurs, but is a result of events rather than a source of control. A consequence of implicit invocation is that modules cannot make assumptions about order of invocation or the state of other modules.

A module in an event-based system can be viewed as a collection of reactive routines. A characteristic of these routines is they must be independent and atomic, i.e., they can't rely on system state and expected behaviour, and they must leave the system in a stable state.

Event-based systems can be reused in two ways. The underlying event system (event manager and events) may be reused with new responses and overall behaviour, as is achieved with the X Window System. An event system can also be reused as a whole with event-based systems lending themselves to easy extension for new environments or extended behaviour.

3 A case study: the production cell

The original production cell is in a metal processing plant in Karlsruhe, Germany. Forschungszentrum Informatik (FZI), Karlsruhe, used the production cell as the basis for a study of formal methods for critical software systems [12]. The first step of the study generated a requirements document [12, Chapter 2] which attempts to capture, in plain language, the specification of a software controller for the production cell. To simplify the case study, the system specified is a simulation of the production cell and the simulator (see Figure 1) runs cyclically.

The production cell simulation (the system) takes a metal blank as input. The metal blank is transported by a conveyer belt (the feed belt) to an elevating rotary table. The elevating rotary table rotates and rises vertically to present the metal blank for the robot to pick up. The robot rotates and delivers the metal blank to the press. The press presses the metal blank and the robot retrieves the metal blank. The robot rotates and delivers the metal blank to

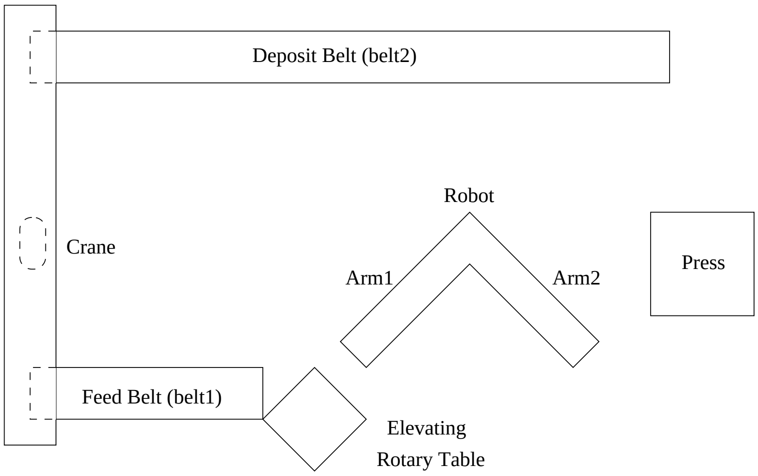

Figure 1: The production cell

a second conveyer belt (the deposit belt). The deposit belt transports the metal blank to a crane. The crane picks up the metal blank and transports the metal blank to the feed belt and completes the cycle. Throughout this paper, the crane is used as an example to explain details of the specification, the design and the design process.

The system actions are further complicated by a desire for speed and efficiency. To accommodate these aims, the robot is fitted with two arms. The arms are placed at right angles to each other and rotate together. The arms can extend/retract and load/unload independently. A second consequence of the aim for speed and efficiency is that the system should be able to handle multiple metal blanks in different parts of the system concurrently.

3.1 Production cell specification overview

Our Z specification [14, 15] was developed in a bottom-up manner. To simplify the specification, we developed an abstract model of the system that involved:-

- hiding the vertical movement of the crane's arm,
- allowing only discrete movement of machines and completely hiding conveyor belt movement, and
- restricting the conveyor belts to hold a maximum of one metal blank each.

The first part of the specification specifies each component independently. The system is subdivided into the following components: feed belt, elevating rotary table, robot, press, deposit belt and crane. The independent specification of each component captures the local state information and the allowable operations on the state.

The independent component specifications do not take into account relationships between different components. For example, the crane cannot pick up a metal blank from the deposit belt if no metal blank is there; however this information is not relevant to the independent specification of either the crane or the deposit belt. The independent specifications, each with their own local state, are combined to give the overall state of the system. Operations from the independent specifications are promoted to apply to the total state and are adjusted to take into consideration the relationships between components.

The system is modelled by the overall state and the corresponding operations, but the operations are partial and are only guaranteed to succeed (i.e., have a specified behaviour) when their preconditions are satisfied. The final section of the specification extends the partial operations to total operations. This is achieved by specifying possible error cases for each partial operation as an error schema. Joining the partial operation and the error schema via the schema calculus generates the final operation.

The specification clearly captures the system's behaviour and the constraints for safe operation. It is however a specification of the complete system, not a specification of a possible controller implementation and while providing the safe, total operations, it provides no indication of the order of use (scheduling) of operations. The complete specification, which includes textual description for each piece of mathematical specification, is 26 pages long. A section of the specification that models the behaviour of the crane is provided in Appendix A.

4 Production cell controller designs

The design problem requires producing a software controller for the production cell. The following sections outline two possible designs for the production cell controller. The designs are guided by software architectural styles; the first an ADT-based architectural style and the second an event-based architectural style. The discussion includes a description of the designs, the process involved (the transition from formal specification to design), and the transition from design to implementation.

4.1 ADT-based design

Designing a controller for the production cell using the ADT-based architectural style as a guide requires breaking the system into two parts. The first is the collection of abstract data types (ADT library) that define this style. The second is the control centre (main program) which coordinates the system, in this case a scheduler. Following this initial separation of the system into policy(the scheduler) and mechanism (the ADT library), the major design effort is developing the ADT library for the production cell controller.

Developing an ADT library for a specific problem requires the system state to be logically partitioned into cohesive sections. Due to the nature of the production cell and the specification used as the starting point of the design, the choice of logical partitions is simplified. An ADT module is developed for each component (the crane, feed belt, elevating rotary table, robot, press, and deposit belt) as shown in Figure 2.

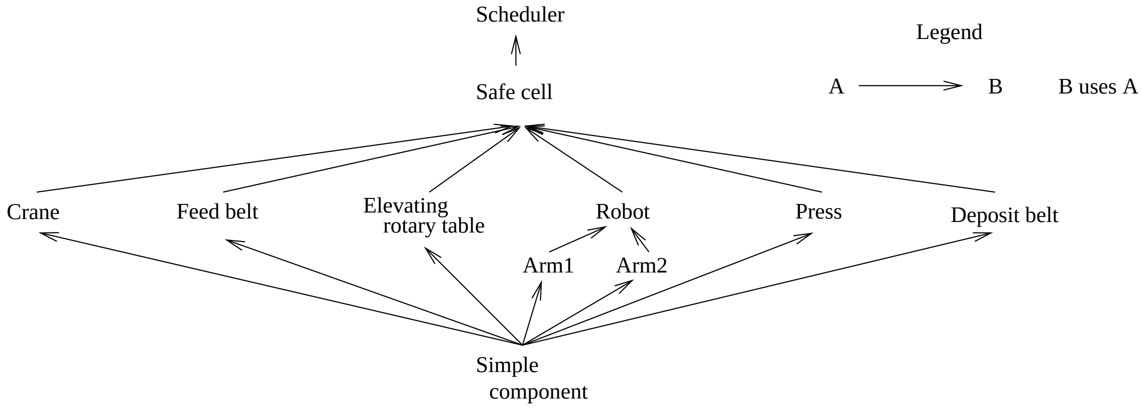

Figure 2: ADT-based design

As each module is responsible for the integrity of its encapsulated state, it is possible to schedule the production cell simulation using these modules as a collection. However, this ad hoc collection of modules does not consider inter-component interactions and hence makes scheduling difficult. To simplify the scheduler (and modification of the scheduling algorithm), the ADTs for each component are gathered into a subsystem, the *Safe_Cell*. The *Safe_Cell* provides an ADT interface to the scheduler[3]. The state encapsulated is the total system state and the operations are the allowable operations on the subsystem as a whole. The transitional operations in this interface are limited to the safe operations on the production cell, i.e., the inter-component interactions are captured. The inquisitive operations on each component are simply promoted to this level.

Using the crane as an example, the transformational operations needed (and defined in the specification) are *Move*, *Load*, and *Unload*. The inquisitive operations are *Loaded* and *Position*, which respectively return whether the crane is loaded and the current location of the crane.

From Figure 2, a subsidiary ADT has been included. This ADT,*Simple_Component*, provides basic state and behaviour required by each module.

The transition from this design to an Eiffel implementation was trivial. This transition was aided by the clarity of operation definition in the specification, especially the pre- and

[3]The *Safe_Cell* accesses each module in the subsystem via its public interface.

post-conditions when designing using “Design by Contract” [17].

Implementation in Eiffel forced a small, but unexpected change in the design. This resulted from the use of enumerated types in the specification and design. Enumerated types are not present in Eiffel and operations that rely on them must be changed. This affects both transformational and inquisitive operations and forces the use of several simple operations rather than one powerful operation. The transitional operation, *Move* was split into two smaller operations that performed specific moves. The inquisitive operation, *Position* was split into location-specific boolean operations. This implementation with a simple scheduler consists of 21 classes[4] and 1700 lines of code. The scheduler moves one metal blank sequentially through the system and is a single class consisting of 70 lines of code.

The ADT-based design provides an easy to understand/maintain system with a collection of simple ADTs composed into a subsystem. A side-effect of this compositional method is increasingly abstract and powerful transitional operations and many inquisitive operations to query the state.

The constraints of the ADT-based architectural style motivated the majority of the design decisions made. Though this relationship may seem trivial, with the specification structure mirroring the the design, it is not the case. This design can also be generated from a flat specification [13] of the system that focuses on the allowable transitions on a metal blank rather than focusing on the components in the production cell. With this alternative specification, the architecture provided invaluable guidance and traceability for the design process.

4.2 Event-based design

Designing a controller for the production cell using an event-based architectural style as a guide requires consideration of both events and the interactions between components in the system. Events are the defining attribute of this architectural style and designing routines (implicitly called) to handle events is the focus of design.

Partitioning the system into modules in an event-based system is not necessary for architectural reasons (as in the ADT-based architectural style), but is performed for software engineering reasons. The partitioning, as seen in Figure 3, is based on keeping related event handlers together. Related event handlers tend to share state and grouping them increases modular cohesion and decreases inter-module coupling. Due to the nature of the production cell, the modules chosen match the components present in the physical system.

Designing the interface (and hence the implicit state and behaviour) for each module is best explained using the crane as an example. The crane picks a metal blank up at the feed belt, retracts its arm, moves to the deposit belt, extends its arm, deposits the metal blank, retracts its arm, returns to the feed belt and re-extends its arm ready for the next metal blank.

[4]The short/flat form of the *CRANE* class is in Appendix B.1.

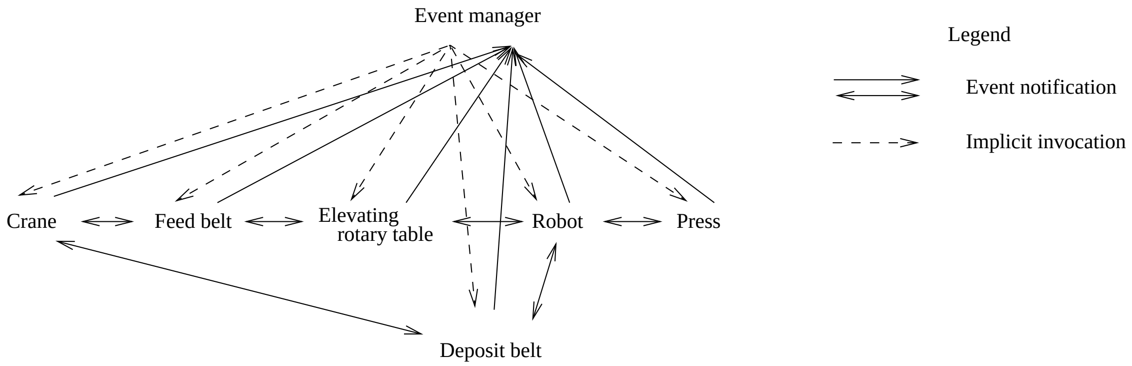

Figure 3: Event-based design

A module for the crane must[5] respond to the events: *over_deposit_belt*, *arm_extended_db*, *arm_retracted_db*, *over_feed_belt*, *arm_extended_fb* and *arm_retracted_fb*. Designing handlers for these events cannot proceed in isolation as a constraint on an event-based architecture is that no assumptions can be made about the state of the system as a whole. In this case, the crane is dependent on whether the feed belt is ready to load or whether the deposit belt is ready to unload. If either component is not ready, the crane must stall until the component is ready. It is the interplay between the components and the events that enables the control, in this case the scheduling, to be implicitly distributed throughout the system.

The interface for the crane module contains eight routines; two to stop horizontal movement, two for stopping retraction and extension at the feed belt, two for stopping retraction and extension at the deposit belt, and two routines to advise the crane that an adjacent component is ready to interact. The first six routines are implicitly called via the event-manager, but the last two are explicitly called by an adjacent component. The primary purpose of the first six routines is to handle a stop event[6]; however they also initiate the next stage of system behaviour. This may be starting the movement of the crane and registering for the corresponding stop event, or stalling the crane. When stalled, the crane waits for an adjacent component to call one of the routines that restarts the crane. Modules register event-handlers dynamically by calling the event-manager. Many event-based systems register events at initialisation; however this is not possible in the production cell. The production cell events are only of interest when they first occur. For example, the extension of the crane's arm over the feed belt is stopped when an *arm_extended_fb* event occurs, i.e., the arm has reached a specified extension. We do not wish to continually handle this event while the arm stays at

[5]These are also the only events directly related to the crane.

[6]Each event is generated by a moving component and the first action is to stop the component, hence stop events.

this extension, hence the event is only registered for when it is required.

The interactions between components are as important as the events themselves, but it is harder to ensure that all interactions are captured. The formal specification was invaluable for this phase of the design. The intermediate level of the specification, which promotes the individually specified components to the system level and adds constraints on their interactions, provided concise descriptions of the allowable interactions within the system. This included pre- and post-conditions on the interactions.

Translating this design to an Eiffel implementation was straight-forward, due largely to the quality and completeness of the design. The implementation of the underlying event management subsystem was a simple task. This subsystem was implemented using inheritance and polymorphism and is reusable (as a framework). The total system consists of 47 classes[7] and 1700 lines of code. The event subsystem is the major component of this with 39 classes and 1000 lines of code. As part of the event subsystem, there is a class for each event to which a component responds, i.e., there are 6 event classes for the crane. These event classes inherit from an abstract class event and redefine *trigger* and *reaction* operations. The core of the event system, the abstract class *EVENT* and the class *EVENT_MANAGER*, consists of 150 lines of code. This system schedules 7 metal blanks, with multiple components active simultaneously.

Designing an event-based controller for the production cell required more effort than the ADT-based controller. However, the comparison is misleading as the event-based system and the ADT-based system differ in two important areas. Firstly, the event-based system does not require an explicit scheduler, while the ADT-based system does. Much of the extra design effort for the event-based system was due to this fact. Secondly, the event-based system allows concurrency of the physical components, i.e., multiple components can be in motion with their stop events handled when they are raised. The ADT-based system does not allow this as the routines are responsible for the integrity of the system they act on. High-level atomic (safe) routines are provided instead. This second point highlights a difference in the suitable targets for the architectures. The ADT-based architectural style (and the formal specification) assume an abstract world where operations are instantaneous, whilst the event-based architectural style is sensitive to this real world issue. An example of this difference is the ADT-based design ignoring the vertical movement of the crane's arm within a load/unload operation and the event-based design having to make this movement explicit.

5 Design benefits of software architecture

In the preceding sections of this paper, we gave two examples of design guided by software architectural styles. The discussion of each design focussed on using software architectural

[7] The short/flat form of the *Crane* class is in Appendix B.2.

styles to drive the design process. A secondary topic in the discussion was the influence and assistance of a formal specification to architecturally-driven software design. In this section we summarise software architecture's contribution to design and then highlight future work which aims to improve software architecture's ability to guide the design process.

5.1 Current Contribution

In the introduction (Section 1), we state that software architectural styles improve seamlessness and traceability in design. Likewise we stated that software architectural styles bring a higher level of conceptual reuse to design than is currently present. In this section, we abstract from the example and highlight how these aims are currently supported by software architectural styles.

Using a specific software architectural style to guide design means choosing an architectural style which suits the problem. In this manner, an architectural style is like a pattern and provides conceptual reuse. As with patterns, the specific choice of style provides guidance. The architectural style clarifies the top-level structure of the system enabling the designer to focus on specifics of the problem. For example, in choosing an event-based system, the designer can concentrate on meeting the constraints of such a system and ignore the underlying architectural subsystem. Design becomes a process of mapping the specification (requirements) onto a framework.

In a similar fashion, the constraints on an architecture guide the style of a module and the granularity of the module interface. In the ADT-based design, the constraints on design include encapsulated data and the integrity of that encapsulated data. These constraints force an interface that contains routines to transform the encapsulated data and routines to query the encapsulated data. This process of using the constraints on an architectural style to guide the style and interface of a module is ad hoc and is the focus of future work.

The architectural style not only guides structure and individual module design, but can help in the transition from specification to design. For example, the design of the event-based system used the specification to ensure all inter-component interactions are handled by the design. This relationship between software architectural styles and specifications is not well understood and is also an area of active research.

Guiding design is the major focus of this work, but software architectural styles can provide more than guidance to the software engineer. The explicit choice of an architectural style eases initial understanding by placing the design within a family group. The system can then be understood relative to previous experience with the style or at least relative to a description of the style (as seen in Section 2). Design decisions can be justified relative to constraints on the architectural style and this improves traceability. Likewise seamlessness between specification and design can be improved by using the relationship between specifications and the architectural style. This relationship can be used to link the specification and the design.

5.2 Future work

Two main areas of future work are raised in the previous section: using the constraints on an architectural style to guide design, and discovering relationships between specifications and architectural styles.

Current descriptions of architectural styles provide an overview of the style, its structure, and the constraints on the architecture. This style of description is useful for understanding the style, and hence a family of systems, but does not capture explicitly an architecture's impact on the design and design process. We believe architectural style descriptions should be extended by providing a designer's view of an architectural style. A designer's view should provide:-

- an architecturally-specific design process,
- a collection of architectural constraints with guidelines for meeting them, and
- a discussion of the module structure and module interfaces expected.

We believe this view can be extended to include the relationship between a formal specification and the architectural style, in particular, directing the designer to relevant sections of a specification. This has two important implications: firstly, formal specifications can become more useful to the average designer, by integrating them into the design process and secondly, a documented relationship between a specification and a style can be used to link the specification to the design and add traceability.

6 Conclusions

In this paper we have shown how software architectural styles can be used to guide object-oriented design. Using two different architectural styles, we showed how to guide the design process from a formal specification to a design (subsequently implemented). From this example, we highlighted how software architectural styles can guide the design process. In particular, we demonstrated how software architectural styles can give a different granularity of conceptual reuse than is currently practised in software design. Using software architectural styles also improves seamlessness and traceability in the design process.

Acknowledgements

We wish to thank Andrew Hussey for clarifying design issues during the writing of this paper, and Andrew Martin for his thorough review of the final draft.

References

[1] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. In D. Notkin, editor, *SIGSOFT'93, Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 18(5) of *Software Engineering Notes*, pages 9–20. ACM Press, December 1993.

[2] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[3] S. M. Brien and J. E. Nicholls. Z base standard version 1.0. Technical Monograph PRG-107, Programming Research Group, Oxford University Computing Laboratory, November 1992.

[4] D. Budgen. *Software Design.* Addison-Wesley, 1994.

[5] L. P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, pages 57–71. Addison-Wesley, 1989.

[6] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Centre, Dept. of Computer Science, Univ. of Qld, Australia, 1994.

[7] R. P. Gabriel. The quality without a name. *Journal of Object-Oriented Programming*, 6(5):86–89, September 1993.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 406–431. Springer-Verlag, July 1993.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[10] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1995.

[11] R. E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–25, June 1988.

[12] C. Lewerentz and T. Lindner. *Case Study Production Cell: A Comparative Study in Formal Software Development.* Number 891 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[13] A. MacDonald. A Transition-based specification of the Production Cell. Working paper, Dept. Computer Science, Univ. of Qld, Australia, June 1995.

[14] A. MacDonald and D. Carrington. Z specification of the Production Cell. Technical Report TR94-46, Software Verification Research Centre, Dept. Computer Science, Univ. of Qld, Australia, November 1994.

[15] A. MacDonald and D. Carrington. Structuring Z Specifications: Some Choices. In J.P. Bowen and M. G. Hinchey, editors, *ZUM'95*, number 967 in Lecture Notes in Computer Science, pages 203–223. Springer-Verlag, September 1995.

[16] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[17] B. Meyer. Design by contract. In D. Manrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[18] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1994.

[19] B. Meyer. *Reusable software - The base object-oriented component libraries.* Prentice Hall, 1994.

[20] R. T. Monroe and D. Garlan. Style based reuse for software architecture. In *Proceedings of the 1996 International Conference on Software Reuse.* IEEE, April 1996.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

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

[23] R. S. Pressman. *Software Engineering: A Practioneer's Approach.* McGraw-Hill, Inc, European edition, 1994. Adapted by D. Ince.

[24] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[25] M. Shaw. Patterns for software architectures. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design (PLOP'94).* Addison-Wesley, August 1995.

[26] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

A Crane Specification

When specifying the crane, only two crane positions are modelled: the crane is either over the deposit belt or over the feed belt. The crane's state has two variables. The first, *crane_position*, records whether the crane is *over_deposit_belt* or is *over_feed_belt*. The second, *crane_loaded*, records whether the crane is *loaded* or *unloaded*. Initially the crane is *unloaded*.

$$Crane\_Position ::= over\_deposit\_belt \mid over\_feed\_belt$$

$$Component\_Loaded ::= loaded \mid unloaded$$

Crane

$crane\_position : Crane\_Position$
$crane\_loaded : Component\_Loaded$

Init_Crane

$Crane$

$crane\_loaded = unloaded$

The crane has three operations. *Load_Crane_0* picks up a metal blank from the deposit belt. To pick up a metal blank, the crane must be both *over_deposit_belt* and *unloaded*. When the crane has picked up the metal blank, the crane's position is unchanged but the crane is now *loaded*. *Unload_Crane_0* is similar. The crane deposits a metal blank onto the feed belt. The precondition of the unload operation is the crane must be in position, *over_feed_belt*, and the crane must be *loaded*. The crane is not moved after the unload operation but the crane is now *unloaded*.

Load_Crane_0

$\Delta Crane$

$crane\_position = over\_deposit\_belt$
$crane\_loaded = unloaded$
$crane\_position' = crane\_position$
$crane\_loaded' = loaded$

Unload_Crane_0

$\Delta Crane$

$crane\_position = over\_feed\_belt$
$crane\_loaded = loaded$
$crane\_position' = crane\_position$
$crane\_loaded' = unloaded$

The third operation deals with moving the crane between the two possible positions and does not change the crane load state, *crane_loaded*.

Move_Crane_0

$\Delta Crane$
$new\_pos? : Crane\_Position$

$crane\_position' = new\_pos?$
$crane\_loaded' = crane\_loaded$

The independent component specifications do not take into account relationships between different components. For example, the crane cannot pick up a metal blank from the deposit belt if no metal blank is there; however this information is not relevant to the independent specification of either the crane or the deposit belt. The independent specifications, each with their own local state, are combined to give the overall state of the system, the *Cell*. Operations from the independent specifications are promoted to apply to the total state and are adjusted to take into consideration the relationships between components. To simplify the promotion process, sub-states of *Cell* are declared. Equating a sub-state within an operation is equivalent to equating each of the state variables within the sub-state in the operation. In Z, state variables are equated when the operation does not change them. Sub-states are used purely for aesthetics, but highlight which state variables the operation changes. In the case below, only the state variables belonging to the deposit belt and the crane may be changed[8].

$$Cell\_7 \mathrel{\widehat{=}} Cell \setminus (Deposit\_Belt, Crane)$$

The crane picks up metal blanks from the deposit belt. To do this, the crane must perform a *Load_Crane_0* operation and the deposit belt must perform a *Unload_Deposit_Belt_0* operation. The resulting operation over the *Cell* state is called *Load_Crane_1*.

$$\begin{array}{|l}
\hline
\textit{Load_Crane_1} \\
\Delta Cell \\
Unload_Deposit_Belt_0 \\
Load_Crane_0 \\
\hline
\theta Cell_7' = \theta Cell_7 \\
\hline
\end{array}$$

The system is modelled by the overall state and the corresponding operations, but the operations are partial and are only guaranteed to succeed (i.e., have a specified behaviour) when their preconditions are satisfied. The final section of the specification extends the partial operations to total operations. This is achieved by specifying possible error cases for each partial operation as an error schema. Joining the partial operation and the error schema via the schema calculus generates the final operation.
Each operation outputs a value of type *Report*. If the operation is successful, the value is *ok*; however if the operation fails, i.e., satisfies one of the error conditions, the value explains the failure.

$$Report ::= ok \mid wrong\_crane\_position \mid component\_already\_loaded \mid deposit\_belt\_not\_ready \mid \cdots$$

[8]The hiding for *Cell_7* should actually contain the variables being hidden in brackets rather than the state schema to which the variables belong. However, this is messy and the hiding of the state is equivalent. This method of promotion is awkward and an alternate method is proposed in [15].

The total operations take the form:-

$$\text{Total operation} \mathrel{\widehat=} (\text{partial operation} \wedge \mathit{Success}) \vee \text{error operation}$$

The *Success* schema provides the ok report value for the partial operations. The *Error* schema is included in the error schemas for each operation.

$$\mathit{Success} \mathrel{\widehat=} [r! : \mathit{Report} \mid r! = \mathit{ok}] \qquad \mathit{Error} \mathrel{\widehat=} [\Xi \mathit{Cell};\ r! : \mathit{Report}]$$

The crane cannot pick up a metal blank when the crane is over the feed belt, the crane is already loaded, or the deposit belt has not delivered a metal blank.

$$
\begin{array}{|l}
\hline
\mathit{Load\_Crane\_Error} \\
\hline
\mathit{Error} \\
\hline
\mathit{crane\_position} = \mathit{over\_feed\_belt} \Rightarrow r! = \mathit{wrong\_crane\_position} \\
\mathit{crane\_position} = \mathit{over\_deposit\_belt} \Rightarrow \\
\quad (\mathit{crane\_loaded} = \mathit{loaded} \Rightarrow r! = \mathit{component\_already\_loaded} \wedge \\
\quad \mathit{crane\_loaded} = \mathit{unloaded} \Rightarrow \\
\quad\quad (\mathit{deposit\_belt\_loaded} = \mathit{unloaded} \Rightarrow r! = \mathit{deposit\_belt\_not\_ready})) \\
\hline
\end{array}
$$

$$\mathit{Load\_Crane} \mathrel{\widehat=} (\mathit{Load\_Crane\_1} \wedge \mathit{Success}) \vee \mathit{Load\_Crane\_Error}$$

The other total operations involving the crane are *Move_Crane* and *Unload_Crane*.

B Eiffel Short/Flat Crane Interfaces[9]

B.1 ADT-based design

class interface *CRANE*
creation
    init
feature
    init
    is_loaded: *BOOLEAN*
    is_over_dep: *BOOLEAN*
    is_over_feed: *BOOLEAN*
    load
        require is_over_dep **and not** is_loaded
        ensure is_loaded
    move_to_dep
        ensure is_over_dep
    move_to_feed
        ensure is_over_feed
    unload
        require is_over_feed **and** is_loaded
        ensure not is_loaded
end -- class *CRANE*

[9] Most comments have been removed to conserve space.

B.2 Event-based design

To enable inter-object communication the class does not have a specific **creation** routine, but has an initialisation routine that must be called after creation.

class interface *CRANE*
feature
    init(fb: *FEED_BELT*;
        db: *DEPOSIT_BELT*;
        e: *EVENT_MANAGER*)
    ready_to_load
      -- called by the *DEPOSIT_BELT*
      -- crane can now load from waiting belt.
    ready_to_unload
      -- called by the *FEED_BELT*
      -- crane can now unload to the waiting belt.
    stop_ext_over_dep
    stop_ext_over_feed
    stop_horizontal_over_dep
    stop_horizontal_over_feed
    stop_ret_over_dep
    stop_ret_over_feed
end -- class *CRANE*