Proceedings of the Tenth International Workshop of Software Specification and Design. 5-7 November 2000. San Diega, CA. Pages 195 – 199.

Design Guidance Through the Controlled Application of Constraints

Aaron G. Cass and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
{acass, ljo}@cs.umass.edu

Abstract

We seek to facilitate development of high quality software designs and architectures by using rigorous process definitions to guide application of the complex structure of relations and constraints that define well-formedness. We identify various types of constraints and demonstrate specific instances of these types. We endorse the value of maintaining the integrity of these constraints by reacting to their violation with diagnostics and remedies. The sheer number and diversity of these constraints, however, indicates the desirability of a mechanism for controlling the scope and effect of their enforcement. Thus, we propose to use proactive process specifications to control the enforcement of, and reaction to, the various constraints. This will result in a process driven system that supports designers and architects by guiding them through orderly development and rework processes, disciplined by the application of constraint enforcement at the right times and in the right ways. This work entails research in defining both types and instances of constraints with programmable enforcement strategies, in embedding such constraints in processes, and in appropriate process definition formalisms.

1. Introduction

Software artifacts such as designs and architectures are complex structures of diverse types of software entities. These various entities can be related to each other by means of relationships between set of those entities. We can define a constraint as a boolean expression, expressed in terms of these relationships and boolean operators, which we can require to evaluate to true. In defining a well-formed software artifact, by definition, to be one for which all constraints evaluate to true (alternatively we say that the artifact is consistent with respect to these constraints), these constraints then essentially define what quality means for that artifact. As complex software artifacts are generally quite large, it is clearly desirable to support the incremental checking of these constraints in order to detect inconsistencies as they arise, and constantly improve quality. This can be done by augmenting the basic notion of the constraint with further semantics that specify when and how the constraint is to be checked, and what action is to be taken in case the constraint is found not to hold. Thus, clearly, constraints can be used to signal developers when they need to remedy inconsistencies.

There seems to be ample evidence, however, that allowing a large collection of constraints to send signals in an unrestricted, undisciplined way can cause problems that undermine their effectiveness. In order to address this problem, we see the need for a countervailing capability for disciplining the way in which constraints are evaluated and enforced. It is our position that an outer proactive process layer can be highly effective in doing this. In our work we expect to demonstrate that the effective integration of a high level, proactive specification for the process of design or architecting, with a comprehensive suite of lower level constraints can be an excellent aid to design and architecting. Such a combination can be used to provide developers with automated guidance on how to proceed forward with design, and with other sorts of reasoning support.

In this paper, we present issues for the design of a constraint specification model and discuss ways in which the web of these constraints might be used to support automated guidance, well-formedness enforcement, and reasoning. We also suggest how incorporating these constraints within a larger procedural structure can improve effectiveness in doing all of these things.

2. High-Level Introduction to the Model

We aim to provide aid to software designers in the complex task of design. A designer must deal with a large number of diverse artifacts and ensure that together they give

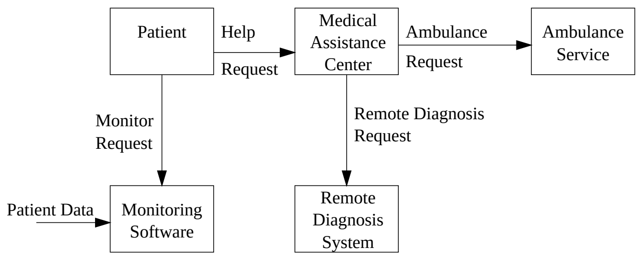

Figure 1. Example Data Flow Diagram

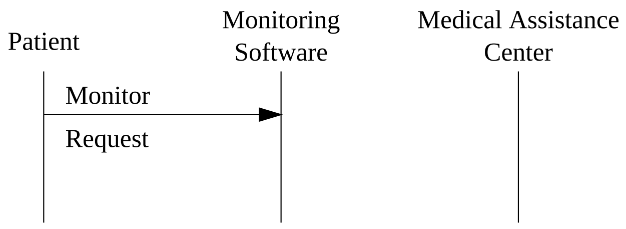

Figure 2. Example Message Sequence Chart

a clear picture of the system under design. These artifacts certainly include requirements specifications and architecture descriptions but might also include related test cases, maintenance plans, and user documentation, and therefore the complexity of the entire system can be overwhelming.

2.1. An Example

We propose an approach to dealing with the complexity of keeping all artifacts mutually consistent. We will demonstrate it by way of a simplified example based on the IWSSD-10 Case Study, a remote medical assistance system. For clarity, we focus only on one pair of diagrams. Figure 1 shows a partially complete Data Flow Diagram(DFD) giving a subset of the requirements of the system, and Figure 2 shows the beginnings of one of the many Message Sequence Charts(MSC) for a scenario a designer might model to refine the design of the system. The two figures represent the state of the artifacts at a point shortly after the designer has modified the DFD to include Monitoring Software, when previously the designer had only been concerned with requests coming directly from a human patient. The MSC and DFD show that we have modeled the patient's request, but we have not yet modeled the interaction between the Monitoring Software and the Medical Assistance Center in either diagram. While this example only deals with two artifacts, we will show than even this small system of diagrams can present a substantial burden, if not appropriately handled.

We wish to examine the kinds of consistency checking issues that are raised when the designer adds a data flow edge in the DFD that allows flow of Help Requests from the Monitoring Software to the Medical Assistance Center. This addition of edges to the diagrams is part of an overall design process in which construction of the DFD is in parallel with construction of the various MSCs for different scenarios, and in parallel with the construction of many other artifact types. We believe it is very useful to think of construction of a DFD and construction of an MSC as steps in this process, potentially decomposed into substeps.

2.2. A Model and Its Application

As an example, consider the enforcement of the constraint that all edges in each MSC correspond to edges in the DFD, and those edges are consistent with respect to data flow types and direction. Clearly, checking this condition every time that we add an edge to either diagram will be too eager – immediately after an edge is added, there is not yet any corresponding edge in other diagrams and the condition will fail. Systems which continually check constraints and immediately indicate failure (such as Argo [10]) would likely overwhelm the designer with many consistency warnings, especially in the case where the designer is trying to keep a very large set of artifacts mutually consistent. We propose to use a less-eager mechanism. We propose to implement constraints as tuples, $C = \langle C_p, C_c, C_w, C_r\rangle$, where C_p is a progress measure, or specification of when to check C_c, which is the constraint condition that we wish to be true. Similarly, C_w is a specification of when to execute C_r, which is a specification of what to do in response to condition failure.

The progress measure is a formula, in some logic, that is true when it is appropriate to check condition C_c. C_p can be specified in terms of artifact state, but also in terms of the history of those states or the actions that arrive at that state. In the example, C_p can be defined, perhaps using a state machine model, so as to limit the checking of the condition to only after we have added two edges, one in the DFD and one in an MSC, each between the same two components in the corresponding diagrams. This specification would limit the checking of the condition, but perhaps in an unsafe way – the designer can possibly add an edge in one diagram and never add a corresponding edge in another, thus never triggering the constraint condition. Our solution to this problem is to add to the progress specification an indication that if the designer gets to the end of the current design phase, as specified by the name of a process step, that the constraint should be checked. So, the full progress specification would be if the state machine accepts OR the current design phase finishes.

C_p specifies when to check the condition, while C_c specifies the condition. This condition will also be a formula, in some logic, perhaps different from the logic used for the progress measure. For this example, a first-order formula over the attributes of the artifacts will work for the constraint condition C_c:

$$\begin{aligned}\forall M \in MSC, \forall e = (\alpha, \beta) \in Edges(M) :\\ \exists x = (\alpha, \beta) \in Edges(DFD) :\\ x.direction = e.direction\\ \wedge x.annotation.type = e.annotation.type\end{aligned}$$

This formula uses the values of the artifact attributes Edges, direction, and annotation to determine if the diagrams are consistent with one another. If it succeeds and returns true, we need do nothing. However, if it fails, we must make some reaction. A straight forward approach, used by some design environments, is to simply warn the designer about the inconsistency in the design. We propose to allow more flexibility through the use of a response specification, consisting of C_w, a specification similar to C_p that indicates when to execute the response specified by C_r.

3. Model Specification Issues

A key part of the future direction of this work is to determine useful formalisms for the specification of the four elements of the constraint tuple. In this section, we explore some of the issues with those formalisms.

3.1. Progress Specifications

Both C_p and C_w need to specify some measure of progress. We see three main elements that might be used in such a specification, namely the state of the artifacts, the events that arrive at a state of the artifacts, and the phase of design currently being performed.

Conceivably, a progress measure can be specified in terms of attribute values using the Object Constraint Language(OCL) [1] or another first-order formalism. However, it seems that history or temporal relationships between actions will be valuable. So, a progress specification in terms of actions on the artifacts is being explored, as outlined above. We might also consider a temporal logic such as LTL [9] or GIL [7].

Progress might also be specified in terms of the process step which we are currently performing. In fact, Little-JIL [3] has pre- and post-requisites designed to be used to specify that some task needs to be performed before or after a step is performed. This task could be activating or deactivating the checking of a constraint.

3.2. Constraint Condition Specification

To motivate some of the issues involved in choosing a formalism for constraint conditions, in this section we give examples of relations between artifacts and constraints on those relations. For this discussion, we focus on the set of relationships within and between architectures and requirements specifications because requirements are a driver for architectures [6]. The following is a partial list of the sorts of relationships that are typically used to specify well-formedness in software products. We denote by A_i an element of an architecture description and by R_i an element of a requirements specification.

- A is part of a component mentioned in R
- A provides the functionality required by R
- A takes as input the data item defined by R
- A outputs the data item defined by R
- A is connected to all A_i such that A_i relates to R
- A implements an interface defined in R
- A_1 uses A_2
- A is composed of A_1, A_2, ..., A_n

A constraint condition is a boolean expression which we require to be true. Therefore, each of these relationships could serve as conditions (the relationship exists or it doesn't). Such a suite of conditions is valuable, but does not seem to us to be sufficient to provide the support that developers need. For example, well-formedness cannot realistically be defined solely in terms of single relationships but rather on the combination of multiple relationships like the following:

- There is no cyclic inheritance. This is a statement about the set of relationships defined by the $inheritsfrom$ relationship, namely that the graph representing the relationship is acyclic.

Also, as the Object Constraint Language (OCL) [1] allows, we would like to be able to express constraints about properties of artifacts. These properties can be constrained by constant values or they can be constrained by the values of other properties, of the same artifact or others.

This is clearly not an exhaustive list. In particular, these are structural relationship between requirements and architectures and behavioral assumptions(that might cause architectural mismatch [8]) are not included.

Clearly, the space of all potentially interesting constraints contains some that are not easily expressible in languages for which automated checking will be efficient. Some of the constraints mentioned above can be represented easily in OCL through the use of constraints between properties of objects, while others can be checked by type-checking algorithms used in compilers. However, in order to specify all of the kinds of constraints outlined above, something more powerful seems to be required. An important part of the work will be incrementally developing a constraint language that effectively supports the specification of constraints that are both useful and automatable. We will take an experimental approach to doing this.

3.3. Failure Response Specification

As stated above, a simple response to a constraint violation is to inform the designer of the problem. This will not always be very useful and we propose to allow a specification of precisely what to do in these cases. Because design is a process, presumably formulated in a process programming language such as Little-JIL [3], we can take advantage of the process language's features for this specification. For example, if the process formalism has an exception handling mechanism, then C_r can indicate which exception to throw. In Little-JIL's scoped exception handling mechanism, this would then cause a response by a handler, possibly in some outer scope. The handler is a potentially complex process step and might involve the re-instantiation of some previously executed activity. It might also be profitable to specify a local handling of the constraint failure by specifying in C_r a process step to instantiate instead of an exception to throw.

4. Classification of Constraints

The sheer number of constraints that restrict the components of an architecture can be overwhelming. Because of this, it seems profitable to determine ways to classify the constraints as a basis for understanding them better, and also as the basis for devising disciplines that can be effective in better managing the complexity inherent in enforcing them.

4.1. Meta-constraints Versus Instance Constraints

Constraints might be applied to all instances of a class or to a single instance. Those applied to all instances might be profitably viewed as constraints on meta-level entities. They are constraining the meta-entities from which the individual entities are instantiated. It might also, then, be profitable to view some constraints as meta-constraints: constraints on constraints. The meta-constraints are essentially types of constraints: a type restricts the instances of the type in much the same way that a constraint restricts the instances. In our work, we are attempting to outline what types of constraints will be useful, and for what purposes.

4.2. Process Constraints Versus Artifact Constraints

Another distinction between constraints that we have found useful is between process constraints and artifact constraints. The examples above are artifact-constraints: constraints on the legal structuring of the relationships between artifacts. However, because our goal is to provide automated support for designers, we want to help the designer to decide what actions to perform next. Some of the information about what to do next is captured in a standard process definition. The process definition represents a set of constraints on the sequencing relationships between process tasks. We shall call these constraints process constraints. An important goal of our work is to develop approaches to using the information provided by artifact constraints to derive process constraints. This is based on the hypothesis that the way artifacts need to be constrained is related to how those artifacts should be built.

In the next section, we present some of our ideas about how the web of constraints might be used to support the design of complex software systems.

5. Using Constraints

5.1. Well-Formedness Checking

Constraints have already been proposed for well-formedness checking [1, 10]. Constraints are used to specify the composition rules for a design or architecture and those constraints are checked to ensure the rules are followed. Argo's critics [10] are checked continuously and pessimistically – they assume that every inconsistency represents an oversight of the designer instead of assuming that the artifact is simply in a temporarily inconsistent state. Argo does allow users to turn off the evaluation of constraints, but obliges users to do this on their own initiative, and to also decide when enforcement is to be reactivated. We believe that we can offer users substantial relief from these obligations by providing process guidance.

5.2. Design Guidance

In a minimally-constrained software development process, there are, at any point, many choices of what next to do. For example, dependencies might state that we must have the code before we can test it, but they would not say that we have to wait for the code before we can begin developing test cases. In order to allow the expert developer the latitude to make intelligent decisions, it is desirable to keep the process definition minimally-constrained, only specifying those process constraints that are independent of the artifact constraints and absolutely required to produce appropriate, complete, and correct artifacts.

In an architecting or designing process, there may be very few of these process constraints – perhaps we need the specification of a module before we can begin design of it or perhaps we need specifications of data types before we can design inter-component connections to communicate those data types between components. Therefore, a design process will be relatively unconstrained and will provide much freedom to choose which aspects of the architecture to work on next. However, this freedom can be overwhelming, especially to the novice designer.

Therefore, a key part of helping people do design is to help them make these choices. This is essentially a prioritizing problem. Others have suggested that risk should be a key factor in deciding which parts of a software project to explore next [2] and processes and tools have been proposed for helping determine where risks are.

Our position is that the artifact constraints are a key to determining where risks are. It has been proposed that the linkages between requirements and architecture be used to determine if the requirements have been covered by the architecture and also to determine the effects of changes (or what elements contribute to a bug). These linkages have been defined relatively vaguely, but if they could be made more precise, they could possibly be used for helping novices with their designs by providing the basis from which to infer process constraints (or at least priorities). The web of constraints between artifacts could be used to help determine where the risks are and therefore where to focus effort.

6. Status and Future Work

Our investigation of artifact constraints is an outgrowth of our long-running research program in process engineering and our desire to develop approaches to aid software designers. This research program has seen the development of Little-JIL [3], a process programming language with both proactive and reactive control mechanisms. Via its proactive control specification, Little-JIL allows the specification of a class of process constraints, while the reactive control mechanisms include an exception management facility that seems like a natural fit for specification of constraint responses. We plan to experiment to determine what classes of constraints are and are not easily captured using these two mechanisms.

We have developed a prototype interpreter for Little-JIL [3, 4] and we have begun to use it to define processes and to define environments based on those process definitions [11]. Our plan is to further explore the approaches to unobtrusively, yet helpfully, driving a design environment with a process. We believe that inferring process constraints from artifact constraints in a design environment will allow us to present priorities upon which designers, especially novices, can base decisions about how to proceed.

Lastly, our work on constraint modeling is informed by previous work outlining a meta-model for constraints [5]. In this paper, we have presented further development of a classification of the types of constraints with which we wish to deal. We have also presented some of the issues that we see must be addressed as the constraint model develops.

Our plan is to further develop the constraint model and experiment to determine its usefulness for design guidance. We will also experiment with our process programming technology to determine its applicability to this problem domain.

Acknowledgements

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032 and by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, or the U.S. Government.

References

[1] *Object Constraint Language Specification*, Sept. 1997. version 1.1.

[2] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[3] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr., and A. Wise. Little-JIL/Juliette: A process definition language and interpreter. In *Proc. of the* 22^{nd} *Int. Conf. on Soft. Eng.*, June 2000. Limerick, Ireland.

[4] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. Technical Report 99-65, U. of Massachusetts, Dept. of Comp. Sci., Nov. 1999.

[5] L. Clarke, E. K. McCall, A. Naithrakashyap, L. J. Osterweil, K. Ramamritham, J. Shanmugasundaram, S. M. Sutton, Jr., and P. Tarr. Internal papers of Inconsistency Management Working Group. University of Massachusetts, 1997.

[6] P. C. Clements. Understanding architectural influences and decisions in large-system projects. Apr. 1995. Seattle, WA.

[7] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifiying concurrent systems. *ACM Trans. on Soft. Eng. and Methodology*, Apr. 1994.

[8] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[9] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[10] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Software architecture critics in Argo. pages 141–144, Jan. 1998. San Francisco, CA.

[11] T. Sliski. An architecture for flexible, evolvable process-driven environments. Master's thesis, University of Massachusetts, 2000.