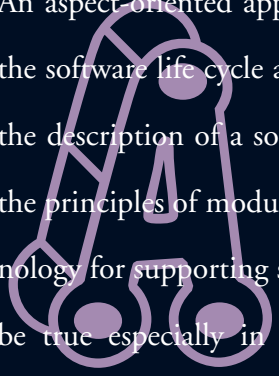
HANDLING CROSSCUTTING CONSTRAINTS IN DOMAIN-SPECIFIC MODELING

Uniting AOP with model-integrated computing.

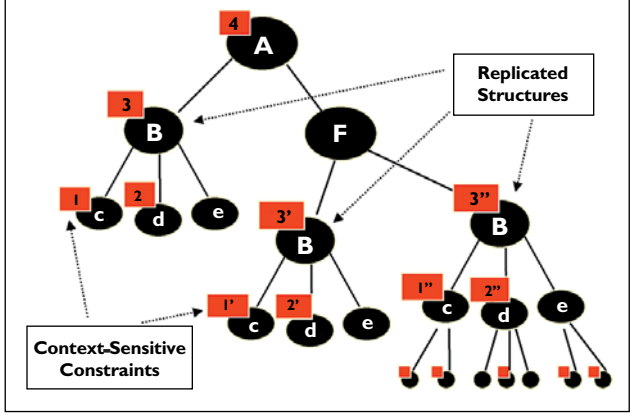
Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck



An aspect-oriented approach can be beneficial at different stages of the software life cycle and at various levels of abstraction. Whenever the description of a software artifact exhibits crosscutting structure, the principles of modularity espoused by AOP offer a powerful technology for supporting separation of concerns. We have found this to be true especially in the area of domain-specific modeling [3].

In domain-specific modeling, a design engineer describes a system by constructing a model using the terminology and concepts from a specific domain. Analysis can then be performed on the model, or the model can be synthesized into an implementation. At the Institute for Software Integrated Systems (ISIS) at Vanderbilt University (see www.isis.vanderbilt.edu/) we implement this approach using a core tool—the Generic Model Editor (GME). The GME is a

Figure 1. Illustration of the difficulty in managing constraints.



modeling environment that can be configured and adapted from metalevel paradigm specifications [8]. In using the GME, a modeler loads a domain, implemented with a metalevel paradigm, into the tool. This provides an environment containing all of the modeling elements and valid relationships that can be constructed in a model. This specific approach to domain-specific modeling has been successfully applied in several different domains, including automotive manufacturing [7], digital signal processing [11], and electrical utilities.

In one particular domain-specific paradigm we created for reconfigurable systems, a modeler may deploy constraints (we use a variant of the Object Constraint Language to specify system properties [12]) to capture application specific rules. In these models, constraints are used to specify properties such as bit precision, timing, and power concerns. Due to the large number of conflicting design criteria in reconfigurable systems, constraints aid in the reduction of the number of design states that must be examined. However, the utility of specifying constraints within the model is often diminished due to the scattering of constraints throughout the model hierarchy. Consequently, constraints represent a type of crosscutting concern.

This article describes the difficulties caused by crosscutting constraints and provides a description of the AO techniques that are being used to ameliorate the problem. Our goal is to encode important issues about the system being modeled in a clean and localized manner. A key feature of this approach is it provides a framework that uses software code generators to create new domain-specific weavers.

Constraints as Aspects

"The crucial choice is, of course, what aspects to study 'in isolation,' how to disentangle the original amorphous knot of obligations, constraints *and goals into a set of 'concerns' that admit a reasonably effective separation."* [2]

The same problems that result from tangled code in programming languages [5] also occur in the tangled constraints of our models [3]. Often, the same constraint is repeatedly applied in many different places in a model, usually with slight node-specific variations. It would be beneficial to describe a common constraint in a modular manner and designate the places and conditions where it is to be applied. With respect to code, a large amount of redundancy can

Figure 2. A tangled resource constraint.

```
constraint K2MULT_RESOURCE()
{

  ((self.children("Forwarder").implementedBy() = self.children("Forwarder").children("forward"))
  implies
    ((self.children("K2_Multiplier").children("K2_Shift_Multiplier").assignedTo() = project().resources("Xilinx_FPGA_1"))
   and
     (self.children("K2_Multiplier").children("K2_Full_Multiply").children("K2_Multiplier").assignedTo() = project().resources("Xilinx_FPGA_1"))
   and
     (self.children("K2_Multiplier").children("K2_Full_Multiply").children("K2_Retrieval_00").assignedTo() = project().resources("Xilinx_FPGA_1"))
    ))
and
  ((self.children("Forwarder").implementedBy() = self.children("Forwarder").children("resource_boundary"))
  implies
   ((self.children("K2_Multiplier").children("K2_Shift_Multiplier").assignedTo() = project().resources("Xilinx_FPGA_2"))
  and
   (self.children("K2_Multiplier").children("K2_Full_Multiply").children("K2_Multiplier").assignedTo() = project().resources("Xilinx_FPGA_2"))
  and
    (self.children("K2_Multiplier").children("K2_Full_Multiply").children("K2_Retrieval_00").assignedTo() = project().resources("Xilinx_FPGA_2"))
  ))

}
```

be removed using AO techniques [6]. We are finding the same applies to our models and constraints.

There are several different kinds of constraints a modeler can apply. Operational constraints express relations between design space alternatives and modes of operation of the system. Composability constraints express compatibility between different alternatives. They can be used to restrict alternatives not compatible with each other. Resource constraints are used to indicate specific hardware resources needed by software modules. Performance constraints are widely used in our models. These constraint expressions indicate the end-to-end latency, throughput, power consumption, and bit precision.

As illustrated in Figure 1, three replicated structures are acted on by context-sensitive constraints. The dominant form of decomposition shown in this figure is concentrated on the functional hierarchy of the system being modeled. Notice that each constraint cuts across this hierarchy. The manner in which a constraint is applied also depends upon the context of the sub-model (for example, constraint "1" may be applied in different ways depending on the context of each model element). However, if it were essential to change the intention of these constraints, it would be necessary to visit each one uniquely and modify it for each context. The dependent nature of each constraint makes change maintenance a daunting task for anything but a simple model.

An example of the complexity that can result from a tangled constraint is evident in Figure 2. This resource constraint describes the effects of two different design alternatives. Our former approach to constraint specification, represented by this figure, required that every possible design alternative be enumerated. The consequence is that constraints become tangled and difficult to understand. Our new approach provides a modular construct for separating such design decisions. Often, what we desire is the ability to express a global system-wide constraint and have it propagated to all relevant nodes in our model.

Figure 3. Strategy and specification aspect examples.

```
strategy ApplyConstraint(constraintName : string, expression : string)
{
  addAtom("ECLConstraint", "Constraint", constraintName).addAttribute("Expression", expression);
}

strategy RemoveConstraint(constraintName : string)
{
  findAtom(constraintName).removeChild();
}

strategy ReplaceConstraint(constraintName : string, expression : string)
{
  RemoveConstraint(constraintName);
  ApplyConstraint(constraintName, expression);
}

strategy PowerStrategy(level : int, power : int)
{

  if (level < 3) then

    <<CComBSTR aConstraint = "power < " + power; >>
    ApplyConstraint("PowerConstraint", aConstraint);
    <<power = power / 10; level++; >>
    modelParts()->forAll(PowerStrategy(level, power));

  endif;

}
```

```
constraint ATR_Power
{
    in Structural models("ProcessingCompound")->select(p | p.name() == "ATR_Top")->PowerStrategy(1, 100);
}
```

Embedded Constraint Language (ECL). The requirements for our new approach necessitate a different type of weaver from those others have constructed in the past (for example, the weaver for AspectJ [5]) because the type of software artifact processed by the weaver differs. Other weavers process source code but our weaver works with the structured textual description of a model. In particular, this new weaver requires the capability of reading a model that has been stored in XML. This weaver also requires the features of an enhanced constraint language.

Our new approach utilizes a constraint language in three different ways:

- Model Constraints: This type of constraint appears as attributes of modeling elements. In this case, constraints are used in the same manner as the former approach (Figure 2 is an example of a model constraint, albeit a tangled one).
- Specification Aspects: A specification aspect is the new modular construct for defining model constraints across the hierarchy. Each specification describes the binding and parameterization of strategies to specific nodes in a model. A specification aspect may be described as a distant relative to a pointcut [5].

Figure 4. Process of using the Constraint Weaver.

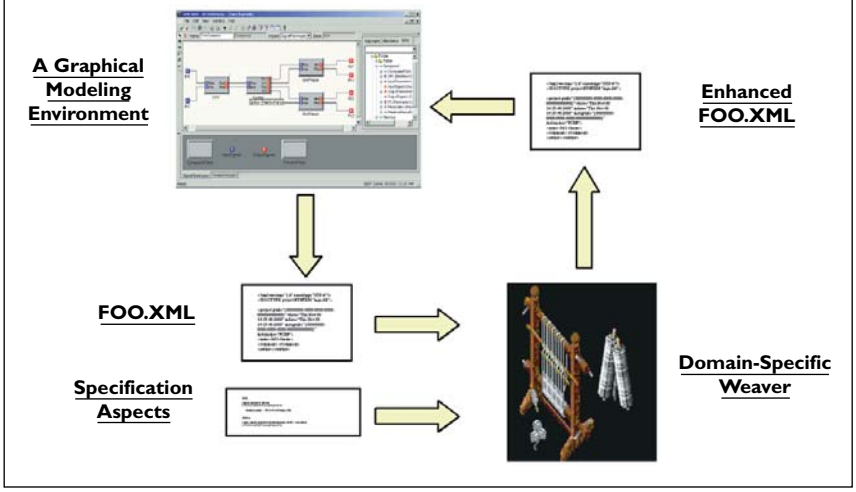
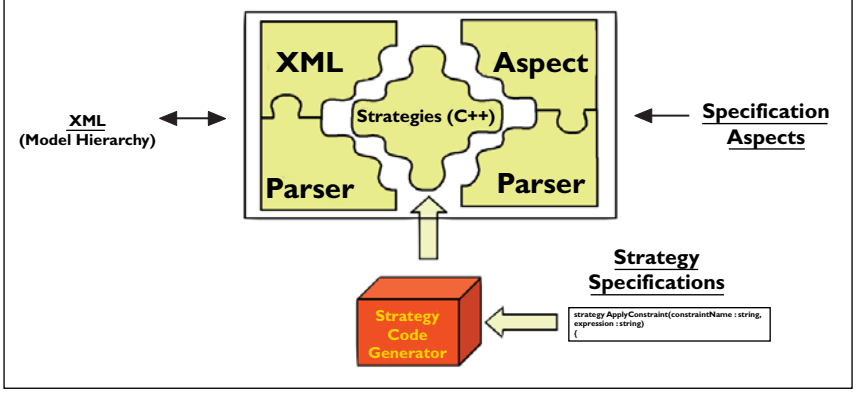


Figure 5. Meta-weaver framework.



- Strategies: A strategy is used to specify elements of computation, constraint propagation, and the application of specific properties to the model nodes. Strategies will be generic in the sense that they are not bound to particular model nodes in their description. Each weaver that supports a specific metalevel GME paradigm will have disparate strategies. The intent of a strategy is to provide a hook the weaver may call in order to process the node-specific constraint application and propagation. Thus, strategies offer numerous ways for instrumenting nodes in the model with constraints.

The three types of constraints listed here differ in purpose and in application, yet each is based on the same underlying constraint language. We call this constraint language the Embedded Constraint Language, which is an extension of the Object Constraint Language [12]. ECL provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (`size`, `forAll`, `exists`, `select`, and so forth). Something unique to ECL and not provided within OCL is a set of reflective operators for navigating the hierarchical structure of a model. These operators can be applied to first-class model objects (for example, a container model or primitive model element) in order to obtain reflective information needed in either a strategy or specification aspect.

Sample Strategies and Specification Aspects. Several sample strategies and specification aspects are specified in Figure 3. The first three strategies at the top of this figure are generic strategies that can be used for constraint application, removal, and replacement. These simple strategies make use of standard functions provided within ECL (for example, `addAtom` and `removeChild`). Because the underlying model hierarchy is stored as an XML file, these standard functions are often implemented as wrappers for the specific calls that are needed to the XML Document Object Model (DOM). The strategy named `ReplaceConstraint` demonstrates that strategies may depend on the capability of other strategies.

The `PowerStrategy` strategy will insert a new ECL model constraint that specifies power properties in an embedded system. There are a few things worth noting about this strategy:

- The strategy language uses ECL in such a way that conditional statements and even recursion are possible.
- It is possible to provide inlined C++ code inside of a strategy (this is indicated by the << .. >> syntax).
- Constraint propagation can be passed along to sub-models by using the ECL functions. In this case, the `modelParts` reflective function returns a collection of all immediate children. The `forAll` standard function then iterates over this collection and invokes `PowerStrategy` on each sub-model (with new values for power and level).
- Although not explicitly shown here, it is possible

to create several different types of `PowerStrategy` by varying the strategy signature. Overloaded strategies can offer various ways of applying the power constraint and propagating it to sub-models.

Notice that strategies are not bound to any particular node in the model. The binding and parameterization of strategies occurs within the specification aspects. An example specification aspect is shown near the bottom of Figure 3. This simple specification aspect will find the node in the model that is of type `ProcessingCompound` and has the name `ATR_Top`. The `PowerStrategy` will then be applied to this specific node using the parameters provided.

Different sets of specification aspects can be weaved into a model. This gives the modeler the capability of constructing "what-if" scenarios. This capability was impossible in our former approach because there was no modular construct for collecting the constraints in a single location. Specification aspects can be much more complicated than shown here. A single specification aspect can cause the weaver to visit many different nodes in the model hierarchy. It is even possible for one global aspect to be diffused across the entire model hierarchy. In the future, we plan to allow wild cards in the naming of models—this will allow even more powerful ECL expressions.

Constraint Weaver: A New Approach

The manner in which our weaver is used is illustrated in Figure 4. The GME can export the contents of a model in the form of an XML document (in this case, the XML DTD is related to the metalevel paradigm from which the model was constructed). In our former approach, the generated XML would be tangled with constraints throughout the document. Using our new approach, it may be quite possible that the exported XML model is void of any constraints. We believe many graphical modeling environments can use this process; it is not necessarily specific to our GME.

The input to the domain-specific weaver consists of the XML representation of the model, as well as a set of specification aspects provided by the modeler. The output of the weaving process is a new description of the model in XML. This enhanced model, though, contains new constraints that have been integrated throughout the model by the weaver.

One way to understand this process is to reconsider the diagram in Figure 1. The XML model that is fed into the weaver will often resemble the hierarchy depicted in this diagram but *without* the constraints (here, provided as the red blocks). The purpose of the specification aspects is to specify the manner in which the constraints are replicated and applied to the context-sensitive model elements. The resultant enhanced model, then, would resemble the diagram in Figure 1 *with* the added model constraints.

The benefits of this approach are numerous. Consider the case of embedded systems where constraints often have contradictory goals (for example, latency and resource usage). In our former approach that did not use AOP, latency and resource requirements would be scattered and mixed throughout the model. As a result, it was quite difficult to isolate the effects of latency or resource constraints on the design. By aspectifying these concerns, the designer may apply specification aspects separately to see how the system is affected in each case. In this way, areas of the system that will have more difficulty meeting a requirement may be given more relaxed constraints, and other parts of the system may be given tighter constraints. In short, it enables the designer to quickly isolate and study the effects of constraints across the entire system. Therefore, the separation of concerns provided by the specification aspects improves the modular understanding of the effect of each constraint. We refer to the plugging/unplugging of various sets of specification aspects into the model as creating "what-if" scenarios. This is somewhat analogous to the ability that AspectJ offers in terms of being able to plug/unplug certain aspects (for example, logging) into a core piece of Java code.

A Meta-Weaver Framework

Each specific GME metamodeling paradigm introduces different types of modeling elements, syntax, and semantics. For example, the metalevel paradigm that we used to create models of the Saturn automobile factory [7] is very different from the paradigm used to create avionics models for Boeing. Therefore, different weavers are needed for different paradigms. This section describes the process in which new

instances of domain-specific weavers are constructed using a meta-weaver framework (see Figure 5).

Strategy Code Generator (StratGen). Strategies are used to aid in the rapid construction of new domain-specific weavers. ECL constraints can succinctly capture portions of a strategy specification. A generative programming approach has been adopted with respect to constructing a weaver [1]. We have developed a code generator capable of translating the strategies into C++ code that is then compiled within the weaver framework. Each domain-specific paradigm can then be considered as being componetized within the weaver.

The C++ code that is generated by StratGen is much more complex than the strategy specification. All of the details of making the appropriate XML DOM calls and the iterations over collections are hidden from the strategy specifier. This allows the construction of a weaver at a higher level of abstraction—a commonly recognized benefit of using domain-specific languages and code generators [1].

XML Parser. The C++ code generated by StratGen is dependent upon several key components. Strategies iterate and manipulate the model, as stored in the DOM. The XML Parser component is responsible for providing wrappers for the methods used to interact with the DOM. XML Parser is also given the task of encapsulating all of the functionality needed to load/save a model using XML. The generated C++ strategies are heavily dependent upon the XML Parser functionality.

Aspect Parser. The Aspect Parser is another piece of the framework. Its purpose is to parse and apply the specification aspects. The application of a specification aspect will result in the invocation of some strategy. It is the task of the Aspect Parser to locate specific nodes in the model hierarchy and invoke specific strategies on those nodes.

An ECL grammar has been created that is used with the PCCTS parser generator [10]. The Aspect Parser uses this grammar, and the associated data structures that represent the parse tree, extensively. In fact, StratGen uses the same grammar during the translation of strategies into C++ code.

Meta-Weaver Instantiation versus Weaver Invocation. A distinction should be made concerning the way these various components are used in the stages of meta-weaver instantiation (the creation of a new domain-specific weaver) versus weaver invocation (executing a weaver on a specific model with a specific set of specification aspects).

While strategies are unique to each instance of a domain-specific weaver, the aspect parser that processes specification aspects is the same for every weaver instance. Another difference between specification aspects and strategies is in the way they are realized. Specifically, ECL constraints applied within strategies are actually used to generate C++ code that is then compiled within the framework to create a new weaver. On the other hand, the ECL constraints used in specification aspects are interpreted, in memory, during the invocation of a weaver. Constraints used in strategies are synthesized during instantiation of the meta-weaver. Constraints used in specification aspects are interpreted during the invocation of a specific weaver.

A Meta-Weaver for Programming Languages. Software development occurs in a polyglot world. Recognizing this truth, we are currently constructing a new type of meta-weaver that works with programming languages rather than domain-specific models. This may be useful to those who want some of the benefits of AOP but use languages other than Java and AspectJ. For example, this new type of meta-weaver would allow the construction of a new weaver that integrates stored procedure code (for example, Oracle PL/SQL) with an aspect language designed for improving the modularity of exception handling. In a sense, each programming and aspect language becomes componentized within the weaver. This new application of a meta-weaver was initially presented in [4].

Our programming language meta-weaver borrows from the previous work of adaptive programming with respect to languages for traversal of object structures [9]. In fact, a key adaptive programming principle—structure shyness—is evident in Figure 5 since there is a distinct separation of behavior (strategy specifications) from structure (the underlying model and specification aspects).

Conclusion

"Even for this let us divided live … That by this separation I may give that due to thee which thou derservest alone."

—William Shakespeare, Sonnet XXXIX

We have found the source of some of our modeling problems was directly related to a lack of support for

separation of concerns with respect to constraints. Constraints may be specified throughout the nodes of a model in order to stipulate design criteria and limit design alternatives. For example, power constraints may be written for all the nodes in a functional hierarchy. However, when the specification changes, each node expressing a power constraint must be visited and updated. Whether the constraints relate to the operation, the composition, or the resources of the system, their scattering throughout various levels of our models has made it difficult to maintain and reason about their effects and purpose.

We have proposed a solution that allows modular specifications of constraints to be propagated throughout a model via a domain-specific weaver. Domain-specific weavers rely on aspect specifications and strategies to carry out their duty. Aspect specifications, similar to pointcuts in AspectJ [5], are used to describe where the constraints will be applied in the model, and strategies describe how a constraint is applied in the context of a particular node in the model. Domain-specific weavers are created as a particular instantiation of a meta-weaver framework. A core component of this framework is a code generator that translates high-level descriptions of strategies into C++ source code.

This approach unites the new area of AOP with model-integrated computing [3]. The preliminary results indicate that simpler, more understandable constraints may be specified and propagated throughout the model hierarchy. This also enables a designer to examine various "what-if" scenarios based on alternative design decisions. Ostensibly, an AOP-based approach to modeling and constraint utilization greatly enhances the maintainability, understandability, and evolvability of domain-specific models. ■

References

1. Czarnecki, K., and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
2. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, 1976.
3. Gray, J., Bapty, T., and Neema, S. Aspectifying constraints in model integrated computing. In *Proceedings of OOPSLA 1999*. (Denver, CO, Nov. 1999).
4. Gray, J. Using software component generators to construct a meta-weaver framework. In *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, (Toronto, Ontario, Canada, May 2001).
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, (Budapest, Hungary, June 2001).
6. Lippert, M., and Lopes, C.V. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the International Conference on Software Engineering* (ICSE 2000), (Limerick, Ireland, June 2000).
7. Long, E., Misra, A., and Sztipanovits, J. Increasing productivity at Saturn. *IEEE Computer* (Aug. 1998).
8. Nordstrom, G., Sztipanovits, J., Karsai, G., and Ledeczi, A. Metamodeling—Rapid design and evolution of domain-specific modeling environments. In *Proceedings of the IEEE ECBS Conference*, (Nashville, TN, Apr. 1999).
9. Ovlinger, J., and Wand, M. A language for specifying recursive traversals of object structures. In *Proceedings of OOPSLA 1999*. (Denver, CO, Nov. 1999).
10. Parr, T.J. *Language Translation Using PCCTS and C++*. Automata Publishing Company, 1993.
11. Sztipanovits, J., Karsai, G., and Bapty, T. Self-adaptive software for signal processing. *Commun. ACM 41*, 5 (May 1998).
12. Warmer, J., and Kleppe, A. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

Jeff Gray (jgray@vuse.vanderbilt.edu) is a graduate student in computer science and a research assistant with ISIS at Vanderbilt University in Nashville, TN.

Ted Bapty (bapty@isis.vanderbilt.edu) is a senior research scientist with ISIS at Vanderbilt University in Nashville, TN.

Sandeep Neema (neemask@isis.vanderbilt.edu) is a research scientist with ISIS at Vanderbilt University in Nashville, TN.

James Tuck (james.m.tuck@vanderbilt.edu) is a staff engineer with ISIS at Vanderbilt University in Nashville, TN.

This work is supported by the DARPA Information Technology Office (DARPA/ITO), under the Program Composition for Embedded Systems (PCES) program, Contract Number: F33615-00-C-1695.