Language Support for Identifying Flexible Points in Coercible Simulations

Joseph C. Carnahan
Paul F. Reynolds, Jr.
David C. Brogan

Modeling and Simulation Technology Research Initiative
University of Virginia
151 Engineer's Way
Charlottesville, VA 22903-22904
434-982-2291, 434-924-1039, 434-982-2211
carnahan@virginia.edu, reynolds@virginia.edu, brogan@virginia.edu

Keywords:
COERCE, coercible simulations, flexible points, language constructs, simulation reuse

ABSTRACT: *Simulation developers are forced to make assumptions about how their simulations will be used and possibly revised to support reuse. Even when developers are aware of potential future adaptations for reuse, current programming languages do not support expression of design alternatives reflecting those adaptations. One can use program documentation to describe them, but documentation does not support automatic simulation transformation. Previously we have described COERCE, a semi-automated simulation transformation technology that supports the capture of design alternatives and the subsequent search and exploitation of these alternatives in order to accomplish desired changes in simulation behavior. In this paper, we propose capturing these design alternatives in programming language extensions called flexible points. With metadata about flexible points embedded in simulation code, COERCE-based software tools can preprocess the code, present information about flexible points to the user, and support semi-automatic evaluation of the fitness of different design alternatives for the new requirements. The programming language extensions we describe in this paper would advance our goal of automating simulation coercion to the extent possible. Semi-automated coercion of simulations, in turn, would greatly enhance user experience with simulation reuse.*

1. Introduction

Composing simulations to build new systems is just one example of how reuse is becoming increasingly important to the simulation community. Considering the high cost of building software, users would prefer to adapt and combine existing simulations to solve new problems rather than to develop new simulations from scratch. In many cases, requirements and circumstances change so quickly that developing new simulations for new situations is not realistic. Instead, libraries of reusable simulation components are needed to handle changing phenomena and new streams of information.

1.1 Building Reusable Simulations

Unfortunately, reusable simulations have proven difficult to develop in practice. Parameterizing a simulation for every way that it could be reused is often impractical, both because the number of possibilities is infinite and because adding too many parameters increases complexity and interferes with performance. As a result, developers must make assumptions about how simulations will be used, deciding on an appropriate level of resolution, setting default values for simulation constants, and selecting algorithms to model specific phenomena.

When a simulation is reused, these decisions often must be examined and changed to meet new requirements. For example, a first-principles physics model of a bicyclist may be replaced by an approximation to satisfy a performance requirement [4], or entities in a military simulation may need to be simulated at a different level of resolution in order to interoperate with another simulation [3]. Key decisions and assumptions in these simulations have to change in order for reuse to succeed.

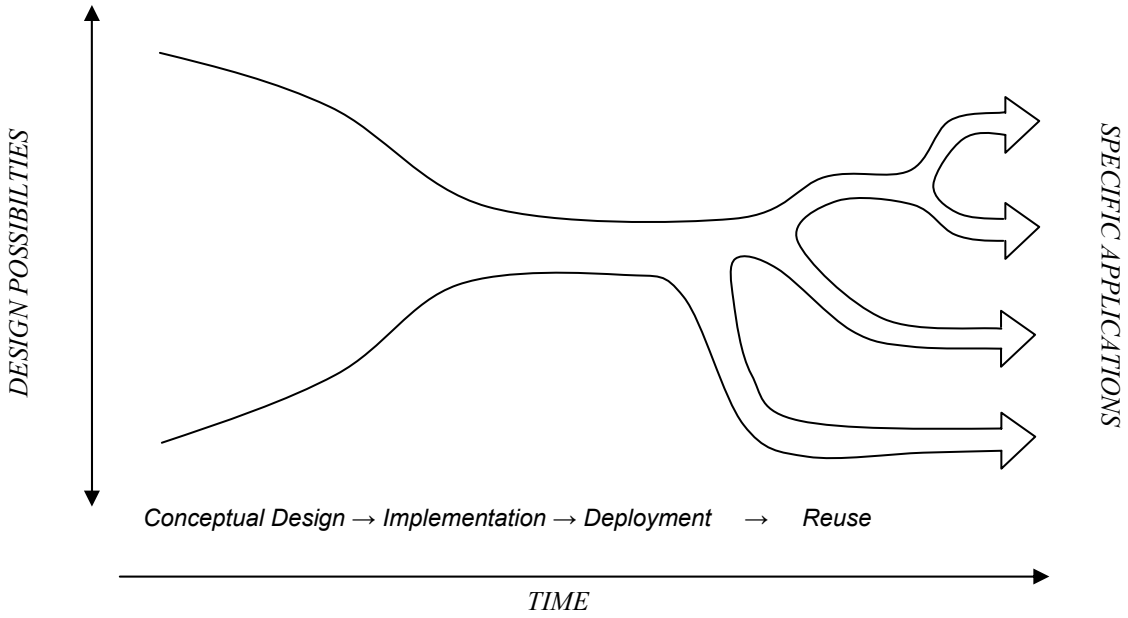

Figure 1: Refinement of a simulation design, with subsequent expansion for reuse

This view of the simulation development process is depicted in Figure 1. As a simulation is designed and implemented, decisions must be made that specify how it can be used and what it can represent. Later, as a simulation evolves and is reused, possible uses that were eliminated during the development of the simulation have to be reopened and explored to meet new requirements.

1.2 Simulation Reuse with Language Support

Automating this kind of exploration should significantly improve the speed and convenience of simulation reuse. This paper discusses a programming language construct designed to support this kind of automation. Our construct makes it possible to capture important assumptions and decisions as they are written into a simulation, as well as alternative choices that could have been made and their effects. These choices can later be automatically extracted and analyzed, making it easier to change them and thus adapt the simulation to meet new requirements.

Consider the following scenario: A company is building a simulation of the game of hockey for a video game to be played on handheld devices. In the process, someone must decide how to model the movement of the hockey puck. The easiest way to implement this is to ignore friction and simply assume the hockey puck slides over the ice with no friction. Of course, a more realistic model would include friction against the ice and air resistance. Based on the marginal benefits of increased simulation fidelity and performance limitations of the device on which the game will be played, the original developer opts for the simple model. However, without input from users, the developer does not know if this compromise will interfere with the experience of playing the game. The developer recognizes that this decision is one that is liable to change in the future, and the developer makes a note of this.

Later, the game is sent to the playtesters. The playtesters complain that the hockey puck behaves strangely: Once the hockey puck is set in motion, it continues to bounce around the rink endlessly, even when no players are touching it. Because this game was developed by a company with a large team of developers, the developer responsible for addressing the playtesters' concerns (called the QA developer) is not the same person who made the decision regarding how to model the hockey puck. So, the QA developer opens up the simulation code in a specialized tool that provides a list of significant decisions that other designers and developers thought might be questionable. The QA developer sees that “hockey puck model” is one of the decisions listed in this display, and that two alternatives are listed: “Friction” and “No Friction.” The "No Friction" alternative has been implemented and is the one that is currently in use. The QA developer looks into the problem further and decides that there are actually two other ways that

the hockey puck could have been simulated, either adding a single frictional term to the model or adding a more complex combination of friction against the ice and drag against the air. Because the playtesters' specific complaint was that the hockey puck was displaying perpetual motion, the QA developer decides that just friction alone will be sufficient to solve the problem. The QA developer adds a note to the model indicating that either friction or both friction and drag could be used, and s/he adds an implementation to the "Friction" alternative and sets that to be the default. The simulation is recompiled with the friction model and returned to the playtesters for more examination.

How could the original developer describe alternatives for the hockey puck model? Similarly, how could the QA developer automatically extract points of interest and add alternatives to each decision? The right language construct could allow both developers to write about these decision points, describe alternatives, and indicate the significance of each alternative. This language construct, called a *flexible point table*, is part of a technology called COERCE, which is outlined in section 2. Section 3 establishes the requirements for this language construct and examines it in the context of other language constructs for flexible software development. In section 4, we describe the details of the flexible point table, and section 5 explores how it benefits semi-automated simulation adaptation and reuse. Finally, we discuss ways to provide additional support for building coercible simulations in section 6 and we summarize our contributions in section 7.

2. Enabling Reuse with COERCE

As just observed, simulation reuse often involves exploring and changing the assumptions and decisions that went into the original development of a simulation. COERCE is a technology that supports reuse by identifying these assumptions and decisions as flexible points and manipulating these flexible points to direct the behavior of the simulation. COERCE has two aspects, coercion and coercibility. Coercion is the study of how to efficiently adapt simulations to new requirements, which is accomplished by selecting flexible points and using a combination of optimization and manual modification on these points to change the behavior of the simulation. Coercibility is the study of how to design and build simulations that can be easily coerced, which is accomplished by identifying flexible points and performing analyses to determine the significance of each flexible point.

2.1 Flexible Points

A *flexible point* is an element of a simulation that can be manipulated to direct the behavior of a simulation in meaningful and effective ways. Flexible points correspond to design decisions in a simulation, decisions that eventually change in order to meet either anticipated or unanticipated new requirements. It is possible to identify flexible points without detailed knowledge about what future requirements will be, although anticipating the general nature of future changes helps determine which flexible points will be most useful.

At a glance, it might appear that every line of a simulation meets this definition of a flexible point. After all, every reachable non-comment line of a program affects its behavior. However, the emphasis is on *directing* the behavior of the simulation, as well as being able to direct it in *meaningful and effective ways*. Picking a random line of code and replacing it with another random line of code is not a meaningful change, nor will it often be effective in meeting new requirements. Instead, flexible points are elements that can be changed either by a user or an optimization program to yield specific effects.

Examples of flexible points include constants that can vary, stochastic elements that can be added or removed, loop convergence criteria that can be tuned, and subroutines that can be replaced. Due to the variety of flexible points, we have begun to establish a taxonomy for classifying them. We expect that these categories can be refined further to provide a complete set of axes on which any flexible point can be plotted. Using these, COERCE-related tools (such as language constructs) can be described in terms of how they apply to different regions of the space of flexible points.

1. Model versus model-implementation.
There is a distinction between flexible points at the level of the model versus flexible points at the level of the simulation code. The decision to allow agents in an artificial society simulation fight with each other provides a conceptual flexible point, while the decision to represent the aggressiveness of an agent with a small range of integers versus a large range of floating-point numbers is an implementation-specific flexible point. Both decisions could be changed to adapt the behavior of the simulation, but switching between different implementations of the same conceptual model generally has different effects than making changes to the conceptual model itself.

2. Narrow versus broad.
Many flexible points, such as the decision to use one random number generator instead of another, can be manipulated by changing a single piece of code (in this case, the call to the random number generator). Other flexible points affect a single object in multiple places

in the code, such as changing the type of a variable and then changing the type of different operations performed on it. Both of these kinds of flexible points could be managed by automatic tools, which either replace contiguous sections of code or select and modify all references to an object. Broader flexible points, which are not as easy to manage automatically, may still be useful to identify. However, because narrow flexible points are easier to manipulate with automatic tools, focusing on narrow flexible points can lead to faster and less labor-intensive coercion of simulations.

3. Ordered versus unordered alternatives.
Another significant distinction in types of flexible points is between those with ordered and unordered alternatives. It is usually not possible to say that one implementation of a function is "greater" than another implementation in the same way that one value for a numerical constant is greater than another. Changing the values of numeric constants often has a more predictable, ordered, effect on a simulation than replacing one function with another.

When working with an ordered flexible point, the user can apply numerical optimization techniques to find the best value to use at that point. Certainly, some simulations exhibit chaotic behavior and phase transitions that make it difficult to predict the results of changes to ordered flexible points. However, with unordered flexible points, it is almost never possible to use numerical methods to select a better alternative. In other words, with unordered flexible points, it is still possible to evaluate each alternative automatically, but it is not possible to look at the results of evaluating a series of alternatives and automatically determine which alternative should be tested next. Also, with ordered flexible points, it is possible to specify alternatives in terms of a range (e.g. "Any value between X and Y could be used here"), whereas unordered flexible points' alternatives must be specified more explicitly. This has a significant effect on how the different types of flexible points can be described with a language construct, as discussed in section 3.1.

4. Independent versus entangled.
One characteristic that raises some of the most difficult issues in simulation coercion is how flexible points affect one another. Adding or removing code at one flexible point may cause code in other flexible points to never be used, or it may break assumptions made in code corresponding to other flexible points. For example, in the hockey puck example from section 1.2, the value of the coefficient of friction is a flexible point, but it is only a meaningful flexible point if the physics model flexible point is given a value that uses friction.

Some dependencies between flexible points can easily be detected, such as when one flexible point changes the value of a variable that is used in another. However, other dependencies are more subtle, such as effects that propagate through one or more intermediate variables. Indirect effects make outcomes harder to predict.

The ability to predict the effects of changing a flexible point is very desirable, because users are primarily interested in flexible points as a means to meet new requirements: It may be intellectually interesting to know what the effects of adding friction to a physical model may be, but the more common question is, "Will adding friction to this model cause the simulation to meet its new requirements?" As a result, it is helpful to distinguish flexible points whose effects on the simulation's behavior can be described without reference to the current values of other flexible points.

2.2 Simulation Coercion

Ordinarily, when a simulation needs to meet a new requirement, its code must be manually edited to exhibit new behavior. With coercion, a combination of automatic transformation and manual modification is used: An expert identifies relevant components as flexible points, and optimization is used to find new values for these flexible points that help the simulation meet its new requirement [12] [14]. When necessary, the simulation code may still be changed to introduce new flexible points or to meet requirements that cannot be met through any changes to existing flexible points. However, even partially automating the process can yield considerable savings in effort, and this approach has been successfully applied in experiments on environmental models of carbon dioxide uptake in forests [7] and physical models of bicyclist movement for animation [5].

2.3 Coercible Simulations

Coercible simulations are the response to the following question: Given the success of simulation coercion, how much easier would it be to coerce simulations if they were designed with flexibility in mind? [14] Coercing a simulation requires selecting flexible points, but many flexible points in simulations could be identified in advance. As discussed in Section 1, developers are aware of making decisions that narrow the range of what their simulations can represent, and many of these decisions become useful flexible points

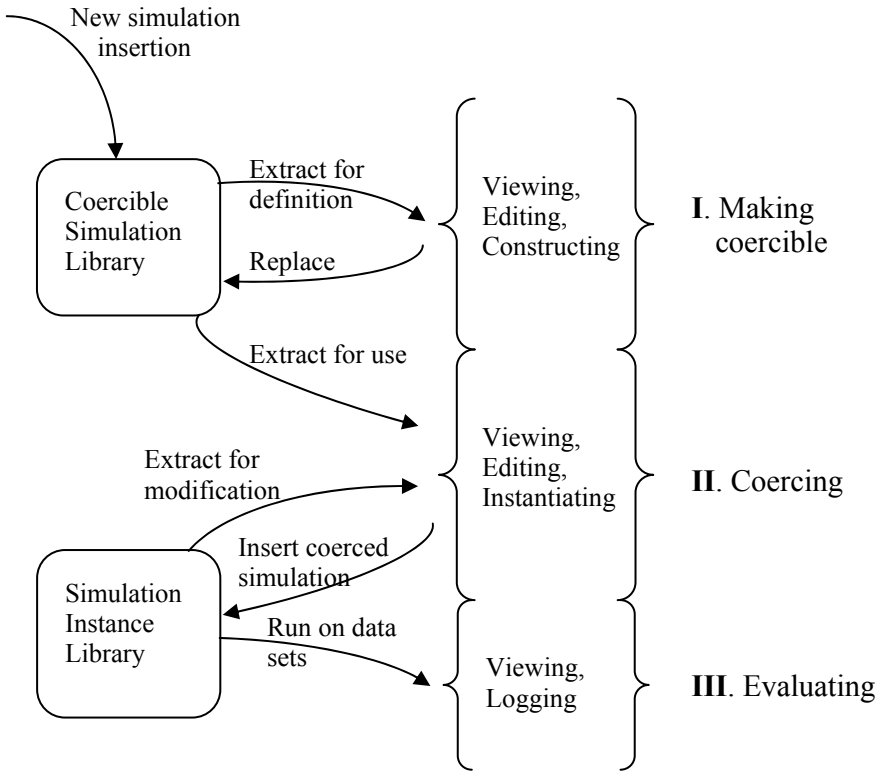

Figure 2: The COERCE Life Cycle

in the future. Therefore, a significant portion of a simulation coercion effort can be saved by capturing information about flexible points as the simulation is constructed. The resulting software with flexible points already identified is a *coercible simulation*.

Figure 2 displays the coercible simulation life cycle. In Phase I, a coercible simulation is created by identifying significant flexible points. In Phase II, a coercible simulation is taken from the library and applied to solve a specific problem. Depending on the application, different flexible points are selected and manipulated to coerce the simulation to fit into its new setting. Visualization tools help the user with selecting flexible points and monitoring the progress of the coercion [6]. Then, in Phase III, the coerced simulation is deployed and evaluated, being coerced again as needed to correct problems and to keep up with changing requirements.

3. Language Support for COERCE

In effect, flexible points are the language of COERCE, the means by which simulation developers communicate information about how their simulations can be reused and how users communicate to one another the ways that they have adapted existing simulations. Simulation developers could describe the flexible points in coercible simulations in the documentation, but formal descriptions would make it possible to automatically extract and change flexible points. Because automation is an important goal of COERCE, we are proposing programming language constructs for embedding information about flexible points in the code itself.

3.1 Language Construct Requirements

In order to design a language construct for describing flexible points, we must identify critical types of information that must be captured. This information includes

1. Which part(s) of the code must be changed to manipulate a flexible point
2. Which alternatives are available for a flexible point
3. What the original intent and implementation of a given flexible point is
4. Effects of changing a flexible point, including
   a. Which simulation variables are affected and in what ways, using sensitivity analyses to determine the magnitude of each effect

b. What the behavioral effects (performance and accuracy) of each alternative are
c. How the effects of different flexible points interact with each other

Of these requirements, certain ones are easier to capture with a language construct than others.

1. Flexible point location.
First, we need to know what code must be changed in order to manipulate each flexible point. It is trivial to indicate the location of a narrow flexible point that affects only one region of code, because the language construct can be inserted into the code at the same point where the change must be made. However, using a single language construct to capture multiple changes to the same source program is more difficult.

2. How a flexible point can be changed.
The form of this information depends on the type of flexible point (see section 2.1). For a numerical flexible point, there may be a range of values that could be used in place of the current one, or an equation might be used to describe what values are valid. For a flexible point that consists of replacing one section of code with another, there may be a list of alternate implementations that could be used.

Intuitively, it may not be possible to represent both kinds of flexible point alternatives with a single language construct. In this paper, we focus on representing flexible points with unordered alternatives. This approach was chosen because previous experiments in simulation coercion have emphasized numerical (ordered) flexible points [7] [5], which can be automatically manipulated without any specialized language constructs. By proposing this construct, we expand the study of automation in simulation coercion to include unordered flexible points.

3. The original intent and value of a flexible point.
This information is ordinarily only available if the user has access to the original implementation of a simulation. However, it is very easy to record this information, and it can prove useful to future coercion efforts if the justification is known for why a flexible point originally had a specific value.

4. Flexible point effects.
This is possibly the most important information to know about a flexible point, although it is also the most difficult to describe quantitatively. This information can be used to work backwards from a new requirement to make a change to a simulation: For instance, given a requirement to make variable X increase, the simulationist needs only to find an appropriate flexible point with an alternative that has the effect of increasing X. There are several aspects of how flexible points can affect a simulation, either by changing important simulation variables, changing behavioral properties such as performance, or by changing the ways that other flexible points affect the simulation.

4a. List of affected variables.
It is important to identify not only the variables that are directly affected by a flexible point, but to also identify which variables depend on the affected variables. In other words, the difficulty comes in identifying the indirect effects of each alternative. In some simulations, changing one flexible point may affect every variable in the simulation, in which case knowing what variables are affected is not nearly as useful as knowing what the amounts of those effects are.

4b. Behavioral characteristics of alternatives.
Many simulation changes are motivated by performance and accuracy concerns, rather than changes in what phenomenon the software is simulating. As such, it is important to identify which alternatives will lead to faster or more accurate simulations. However, as with capturing the effects of flexible points on simulation variables, this information is often hard to determine in advance: Different flexible points may interact with each other, and the behavioral characteristics of a particular alternative often depend on the simulation's current input.

4c. Relationships between flexible points.
As noted above, selecting a specific option at flexible point X may have a completely different effect on simulation outputs or performance characteristics depending on which alternative is selected at flexible point Y. Ideally, we would like to compute these relationships automatically, but the combinatorial number of ways flexible point alternatives could interact makes this computationally infeasible. However, if the developer knows that certain flexible points do not affect one another and indicates this in the program, then the number of combinations that would have to be evaluated drops considerably.

As described here, capturing all of this information about a flexible point with a single language construct is extremely difficult. However, in the following section, we explore one language construct that meets a number of these requirements to provide useful information about a large number of the flexible points that are actually encountered in practice.

Specifically, we are focusing on unordered flexible points that are limited to a single point in the code.

This decision was motivated by our model of simulation development in Section 1: We are interested in capturing design decisions in a simulation that are likely to change in the future. From the programmer's perspective, this often amounts to the decision to write one block of code instead of another. We would like to record this information in a way that does not impose additional cost on running the simulation with its default configuration. However, we would like this information to be encoded in a way that a support tool could apply one or more of the suggested changes to a flexible point and run the simulation automatically.

3.2 Related Language Constructs

Other programming language constructs have been used to capture information about software design decisions that are liable to change. First, many languages contain conditional compilation features, such as the `#ifdef` statement in C and C++ [10] [13]. This enables a developer to include code that may or may not be compiled into the final program depending on the value of a preprocessor variable. Conditional compilation is commonly used for including machine-specific code in a portable program or including a debugging option with a more streamlined version of an application. Because the decision to include a section of code is made at compile time, it imposes no cost on the run time performance of the system. In effect, the flexible point construct that we propose here is an extension of existing conditional compilation mechanisms, with added features to describe the significance of each alternative and to facilitate the presentation of this decision point to the user in terms of how it affects the behavior of the simulation.

Another language structure used to capture information about potentially changing design decisions is software modules. Using the information-hiding principle, each module should be built around one design decision that is liable to change [11]. In practice, this means that modules are built around data structures, because even small changes in how data are represented have the potential to affect arbitrarily large sections of code. However, for coercible simulations, we expect to capture more than just information about data structures, because flexible points can include smaller details such as values for constants and conditional expressions. As a result, the language construct proposed in this paper is complementary to an object-oriented information-hiding design: The decomposition of software into modules protects design decisions based on how they affect data representation, while the inclusion of COERCE flexible points highlights design decisions based on how they impact simulation behavior and reuse.

4. Flexible Point Representation

We propose a language construct called a flexible point table. Our flexible point language construct can be viewed two ways: Visually, a flexible point can be represented with a table of options, with information about each alternative given in each row of the table. In an implementation, a flexible point table can be encoded as an XML document with elements corresponding to fields of the table. This makes it easier to integrate the flexible point description into a larger document and to automatically extract this information when needed.

4.1 Tabular Representation

As shown in Figure 3, a flexible point can be represented as a table of different decisions that could be made at a specific point in a program. Each row of the table corresponds to another alternative, with columns for

- An identifying label
- A summary, describing its implementation and how this alternative affects simulation behavior relative to other alternatives
- An optional implementation, which could be substituted for any of the other alternatives' implementations at this point

Information that is common to all of the alternatives is included at the top of the table, including a label for this flexible point and a list of simulation outputs that are affected by all alternative implementations of this flexible point (note that this list may change as different alternatives are added). In our hockey example, a simulation can use one of several different physics models, from a simple kinematic approximation to a detailed calculation that includes considerations for friction and air resistance. Selecting a different physics model affects simulation outputs about the position and velocity of this particular object in the simulation, which is noted at the top of the table. Depending on the accuracy and performance requirements for the simulation, a user might prefer the simpler and faster model or the more detailed and accurate one.

It is important to note that the *Implementation* field of the table is optional. As a simulation is constructed, a developer is often aware of selecting one design over another but does not normally take the time to implement unused alternatives. This way, a developer is free to acknowledge other options without specifying how they would be implemented, which still provides a

Flexible Point: Physics Model
Affected Outputs: `xPos`, `yPos`, `xVel`, `yVel`

Alternative	*Description*	*Implementation*
No friction	Simple model, very fast but not very accurate. Object moves v/t units per time step.	`xPos += xVel / TIME_STEP;` `yPos += yVel / TIME_STEP;`

Friction	More complicated model, slowing the object by a frictional force at each time step.	`xPos += xVel / TIME_STEP;`
		`yPos += yVel / TIME_STEP;`
		`friction = g * F_COEFFICIENT;`
		`xFric = friction * cos(angle);`
		`yFric = friction * sin(angle);`
		`xVel += xFric / TIME_STEP;`
		`yVel += yFric / TIME_STEP;`

Friction plus air resistance	Most realistic model, but not yet implemented. Needs information on cross-sectional area of object and air pressure, which is not available.	(not implemented)

Figure 3: A Flexible Point Table

benefit to future users without imposing an unreasonable cost on the original simulation developer. Later, as a coercible simulation is coerced and reused, implementations may be filled in and new rows may be added as each flexible point is expanded and new directions are tried.

4.2 XML Representation

Including the flexible point table in the source code of a simulation is not practical for several reasons, such as the width of the table and the issues of how to parse table elements separately from the rest of the simulation code. However, the same flexible point can be described using XML to define elements corresponding to each field of the table [15]. A Document Type Definition for an XML `flexiblePoint` is given in Figure 4.

```
<!ELEMENT flexiblePoint (name,
                         effects*,
                         alternative+)>
<!ELEMENT name (#PCDATA)*>
<!ELEMENT alternative (name,
                      summary,
                      implementation?)>
<!ELEMENT description (#PCDATA)*>
<!ELEMENT implementation (#PCDATA)*>
<!ELEMENT effect (#PCDATA)*>
```

Figure 4: XML DTD for a `flexiblePoint`

5. Utility of the Flexible Point Table

In order to show that this language construct is useful, we must demonstrate that

1. The construct is applicable to a significant set of flexible points.
2. Through COERCE, the construct provides opportunities for automation that will accelerate the process of simulation reuse.
3. The construct is easy to implement and use with existing languages and tools.

5.1 Applicability

The definition of a flexible point that is presented in this paper is deliberately broad, designed to include any element of a simulation that could be targeted as part of a semi-automated coercion process. However, a language construct for building coercible simulations does not need to describe every possible kind of flexible point in order to be beneficial. In particular, the flexible point table can be applied to describe several common kinds of simulation changes. This makes it a useful tool in itself, as well as a stepping-stone towards either a more sophisticated language construct or a toolkit of several constructs that together capture the whole range of flexible points.

First, a flexible point table can be used to replace a named constant with a variable, so that alternate values can be tried. Using the criteria outlined in section 2.1, we see that changing the value of a simulation constant is a particularly useful kind of flexible point:

1. Constants include values in the high-level equations that underlie the model, as well as tuning factors in the model implementation.
2. Constants are narrow in their effect on the code, because changing the value of a named constant does not change how it is referenced elsewhere in the program.
3. Constants are an example of a flexible point with ordered alternatives, making them amenable to optimization.
4. Changing the value of a constant is a relatively simple change to a program, making it easier to understand the effects of the change independent of other changes in the program.

Another kind of flexible point that can be represented by this table is replacing a constant with a call to a function that samples from a stochastic distribution. This allows the user to represent a new element of uncertainty in the model. Likewise, the flexible point table can indicate that a function which samples a stochastic distribution could be replaced by a representative constant, removing uncertainty but reducing the computational cost of assigning a value to the variable.

In general, the flexible point table can be used to replace the body of any function with any other function. The physics-model example used in sections 1.2 and 4.1 is an instance of replacing one subroutine with another. Other examples could include replacing the arrival- or service-time functions in a queuing simulation, changing the constructors that initialize agents' attributes in an agent-based simulation, or altering the model of interactions between units in a strategic military simulation.

5.2 Opportunities for Automation

Given this representation for flexible points in a simulation, it is now possible to automate several elements of the COERCE process. First, the tabular representation of the flexible point can be presented to the user via a "flexible point browser" interface. This could be added as an extension to an integrated development environment so that a user could change or add new implementations using existing source code editing tools.

Ideally, when coercing a simulation, the user would like to know exactly what the effect of each alternative is. This way, the user would need to do nothing more than browse through the flexible points and select the set of alternatives that are already known to yield the desired result. In practice, the complex relationships between flexible points make it difficult to statically document the effects of each alternative. However, another component of the COERCE toolkit is a set of visualization tools [6]. This means that it would be possible to generate visualizations to plot the relationships between sets of flexible points and to link these visualizations with the flexible point browser interface.

Most importantly, once multiple implementations have been specified for a given flexible point, it is possible to automatically test each implementation and evaluate the simulation's output relative to a specified requirement. In other words, each flexible point becomes another possible decision variable to use in the optimization step of simulation coercion [14]. This increases the number of design decisions that can be automatically explored and reduces the amount of manual effort required to coerce a simulation.

5.3 Implementation Technologies

The flexible point table's greatest advantage over other possible flexible point representations is its ease of implementation. Numerous libraries for XML processing already exist (libxml for C, SAX for Java, numerous modules for Perl, etc.). So, only a simple preprocessing step is required to extract all of the `flexiblePoint` XML documents from a larger source file and pass them along to an XML processor for parsing and display. The XML processor then can use an XSLT style sheet to generate an HTML presentation of the flexible point in tabular form [15]. When an alternative is selected, the contents of the corresponding `implementation` element can be extracted and reinserted into the original source file. Then, this modified source file can be compiled without making any changes to the original language's compiler.

6. Future Directions for Coercible Simulations

As noted in Sections 2.1 and 3.1, there is a considerable variety of interesting flexible points in simulations. In addition, the potential amount of information to capture about each flexible point is sizeable. The flexible point table proposed in this paper meets our objective of describing narrow flexible points with ordered and unordered alternatives, but some flexible points unavoidably affect multiple locations in the code. Therefore, we are exploring

additional ways to document flexible points in a coercible simulation.

6.1 Extending the Flexible Point Table

First, it would be possible to extend our construct to link multiple code changes together, so that a single flexible point could be represented by multiple tables. Each table corresponding to a single flexible point would have the same set of rows, and one table would be placed at each point in the code where any of the options requires a change to the code. This approach is the most straightforward to implement, but it raises certain challenges for developers, such as keeping track of all of the different tables spread across the multiple source code files of a simulation.

6.2 Flexible Points as Cross-cutting Aspects

For a different approach, we have observed that a single flexible point that requires code changes in more than one module of a program is a *cross-cutting concern*, which is exactly what aspect-oriented programming (AOP) is designed to capture [8]. AOP is a programming paradigm that extends object-oriented programming by enabling developers to separate concerns that apply to multiple objects in a system (such as security, synchronization, or input-output) from one another. Several different approaches to AOP have been developed, and examples of these approaches can be found in languages such as AspectJ [1], AspectCOOL [2], and Hyper/J [9]. As a result, we are exploring aspect-oriented language constructs to determine if we could leverage AOP to succinctly describe and manipulate flexible points that affect code in multiple modules.

6.3 Using WHEN Statements to Describe Flexible Points without Lexical Information

An alternative for capturing flexible points that transcend multiple points in the simulation code is the `WHEN-DO` construct. The structure of a `WHEN-DO` statement would be similar to a conventional `IF` (conditional) statement:

```
WHEN (Boolean condition is true)
DO
   (block of code to be executed)
DONE
```

However, unlike an `IF` statement, which is bound to a single point in the code, a `WHEN-DO` statement could be declared at the top of a program and apply to the entire scope of the program. If the `WHEN` clause ever becomes true, then normal execution is interrupted and the code contained in the `DO` block is executed.

This behavior is comparable to the behavior of program exceptions with resumption semantics [16]. As such, this construct provides an extremely powerful way to specify interesting program conditions without knowledge or dependence on the location of where those conditions arise in the program. To build a coercible simulation with `WHEN-DO` constructs, the original developer could specify conditions that are associated with important flexible points in `WHEN` blocks. When the simulation is coerced, future users could fill in the corresponding `DO` blocks to change the behavior at flexible points of interest.

We recognize that without considerable attention to implementation issues, the `WHEN-DO` could have excessive run-time costs. However, the fact that we are only interested in capturing simulation flexible points means that we may be able to place certain limitations on the permissible `WHEN` conditions, reducing the cost of testing the conditions and limiting the scope in which tests could ever become true.

7. Conclusion

The driving goal for COERCE continues to be efficient and effective simulation reuse, whether in the context of simulation interoperability, multi-resolution modeling, or data-driven applications. To that end, we will continue to propose new language constructs to facilitate coercible simulation development and develop new tools for automating the manipulation of these flexible points.

7. Acknowledgements

We wish to acknowledge support from the Defense Modeling and Simulation Office, particularly from Sue Numerich and Phil Zimmerman. Additional support was provided by the National Science Foundation (ITR 0426971).

8. References

[1] AspectJ Web Site. `http://www.aspectj.org`. Eclipse Technology. Last visited June 8, 2004.

[2] Avdicaušević, E., M. Mernic, M. Lenic, and V. Zumer: "Experimental Aspect-Oriented Language: AspectCOOL." In Proceedings of the 2002 ACM Symposium on Applied Computing, Madrid, Spain.

[3] Bowers, A, D. Prochnow and J. Roberts: "JTLS-JCATS: Design of a Multi-Resolution Federation for Multi-Level Training." Proceedings of the 2002 Fall Simulation Interoperability Workshop, Orlando, FL.

[4] Brogan, D. and J. Hodgins: "Simulation Level of Detail for Multiagent Control." Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), p. 199-206, 2002.
[5] Carnahan, J., P. Reynolds, and D. Brogan: "An Experiment in Simulation Coercion." Proceedings of the 2003 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), Orlando, FL.
[6] Carnahan, J., P. Reynolds, and D. Brogan: "Visualizing Coercible Simulations." Proceedings of the 2004 Winter Simulation Conference, Washington, DC.
[7] Drewry, D., P. Reynolds, and W. Emanuel: "An Optimization-Based Multi-Resolution Simulation Methodology." Proceedings of the 2002 Winter Simulation Conference, San Diego, CA, pp. 467-475.
[8] Elrad, T., M. Aksits, G. Kiczales, K. Lieberherr, H. Ossher: "Discussing Aspects of AOP." Communications of the ACM, Vol. 44 No. 10, pp. 33-38, October 2001.
[9] Hyper/J Web Site. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm. IBM Research. Last visited June 8, 2004.
[10] Kernighan, B. and D. Ritchie: The C Programming Language, pp. 88-91 Prentice Hall, Upper Saddle River, NJ 1989.
[11] Parnas, D.: "On the Criteria To Be Used in Decomposing Systems Into Modules." Communications of the ACM, v.15 n.12, pp.1053-1058, Dec. 1972.
[12] Reynolds, P.: "Using Spacetime Constraints to Guide Model Interoperability." Proceedings of the 2002 Spring Simulation Interoperability Workshop, Orlando, FL.
[13] Stroustrup, B.: The Design and Evolution of C++, pp. 423-426., Addison-Wesley, Reading, MA, 1994.
[14] Waziruddin, S., D. Brogan and P. Reynolds: "The Process for Coercing Simulations." Proceedings of the 2003 Fall Simulation Interoperability Workshop, Orlando, FL.
[15] XML Web Site. http://www.xml.org. O'Reilly Media. Last visited June 8, 2004.
[16] Yemeni, S. and D. Berry: "A Modular Verifiable Exception-Handling Mechanism." ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7, Issue 2, pp. 214-243, April 1985.

Author Biographies

JOSEPH C. CARNAHAN is a Ph.D. Candidate in Computer Science and a member of MaSTRI at the University of Virginia. Joseph earned his B.S. in Computer Science at the College of William and Mary, and has held the position of Scientist at the Naval Surface Warfare Center, Dahlgren Division.

PAUL F. REYNOLDS, Jr. is a Professor of Computer Science and a member of MaSTRI at the University of Virginia. He has conducted research in modeling and simulation for over 25 years, and has published on a variety of M&S topics including parallel and distributed simulation, multiresolution models and coercible simulations. He has advised numerous industrial and government agencies on matters relating to modeling and simulation. He is a plank holder in the DoD High Level Architecture.

DAVID BROGAN earned his PhD from Georgia Tech and is currently an Assistant Professor of Computer Science and a member of MaSTRI at the University of Virginia. For more than a decade, he has studied simulation, control, and computer graphics for the purpose of creating immersive environments, training simulators, and engineering tools. His research interests extend to artificial intelligence, optimization, and physical simulation.