15 Things System Dynamics can Learn from Software Development

Nathaniel Osgood
Yuan Tian
Department of Computer Science
University of Saskatchewan

Abstract

While System Dynamics involves a diverse set of issues outside the scope of software development, one of the key outcomes of the System Dynamics process is software artifacts, in the form of simulation models. Many of the principles and concerns of software development more generally apply by extension to the development of System Dynamics models. In recent decades, the software development field has benefited greatly from a series of process-based and technologies to make the production of software artifacts more reliable, more timely, and more predictable. Within this paper, we argue that System Dynamics projects can benefit from adoption of processes and techniques in software development, and discuss some of the adaptations required to take best advantage of such approaches.

1. Introduction

As a form of computational modeling, System Dynamics is tied up with the creation of computational artifacts. While such artifacts are built with a specificity of purpose that distinguishes them from most software development, as computational artifacts, the process of building, developing confidence in, reasoning about, and interacting with System Dynamics models has important parallels to the process of building, developing confidence in, reasoning about, and interacting with other types of software. As someone who teaches multiple courses in both software engineering and System Dynamics, the author has long been struck by underlying similarities between the methods, and the prospects for each discipline to learn from the other.

Beginning with the seminal work of Abdel Hamid and Madnick [1], there has been an increasingly rich literature applying System Dynamics insights to improve the timeliness [2, 7] and quality [6] of software development process. A recent book by Madachy [2] testifies as to the continuing promise of this approach. While accessible and feature-full software packages have provided a key enabler for System Dynamics modeling, there have been very few attempts to explicitly apply lessons learned in software engineering within the System Dynamics modeling process.

This paper seeks to set out an initial survey in which principles of Software Engineering (including the process of software development) could be applied to the field of System Dynamics. The paper's scope is purposefully broad, sacrificing detailed discussion about any one method in favor of a wide look at possible enablers. This paper also does not seek to be comprehensive, but rather focuses on a small handful of principles and innovations – typically focusing on the process side of System Dynamics – that seem likely to offer the greatest leverage within the System Dynamics field. For size reason, we defer to later papers certain relevant topics that merit extensive examination on their own, such as the use of ideas from software development methodology, and requirements elicitation and engineering.

2. Process Related Innovations

This section focuses on principles and innovations focusing on the process of System Dynamics modeling, rather than on the technical elements of building up a model.

2.1. Debugging Processes

In software development, much time is spent in the process of "debugging"– the process of tracking down the underlying cause of observed system faults. The debugging process is a meticulous one, and contains elements of both science and art. It has been estimated that developers vary in their debugging productivity by an order of magnitude. These variations are not so much reflective of variations in raw intelligence or technical support for debugging, but of the fact that some developers make use of very powerful debugging strategies. Within recent decades, the software development has been enriched by articulation of powerful debugging strategies in the literature [8, 9] as well as powerful tools to support debugging [8, 10].

System Dynamics tools already offer one of the most powerful enablers for debugging, due to their capacity to record the full state of a model over time, and the ability – offered by some – to literally step backwards over time (such as in Vensim's gaming mode). Despite such technical support, the process of debugging System Dynamics models remains challenging. One reason is the difficulty of distinguishing "bugs" from unexpected but legitimate emergent model behavior. Another significant reason is the predominance of feedback loops within models –complicating the process of distinguishing the symptom of a problem from its cause. Yet another overarching issue is the complicated nature of many models, with many "moving parts" that could be of issue. Based on observations over the years we posit that the variability in debugging proficiency in System Dynamics – partly as a result of these challenges – may approach that in other areas of software engineering. Many experienced System Dynamicists have developed strong debugging skills, permitting them to quickly home in on problems they encounter. By contrast, many others appear to have difficulty knowing where to start. For this reason, we believe that it is important to borrow from the software development experience, and to articulate principles and semi-formal debugging methodologies to assist System Dynamics in developing their debugging skills.

2.2. Build Mechanisms

In Software Development, a "build" process is a key step in transitioning from work on the specification of a software system to work observing or interaction with execution of that system. At its most traditional level, a build involves the "compilation" of a system's specification into a form that can be executed. However, modern development environments use the build step for many additional purposes – for triggering automated tests, for stylistic and risk checking of the system's specification, for updating developers as to the current status of the project, for deployment of the system to remote servers and to developers' machines, and for initialization of test databases. Current packages for System Dynamics modeling do support "Build" functionality – occurring just prior to model execution – but expose limited (if any) options for customizing or extending it. Modular support for "pluggable" extensions could be valuable for both individual and team developers, and could help build up a marketplace of 3rd party plug-ins to enhance the quality and transparency of the System Dynamics modeling process. For example, pug-ins could be made for stylistic checking of models, more sophisticated unit checking mechanisms, publishing findings of the projects, and testing. Such "hooks" into the development process could further provide support for 3rd party tools such as SILVER [3]. Modeling software support for annotations and other mechanisms for supporting 3rd-party defined metadata could further extend the richness of build mechanisms that could be supported.

The sections below discuss below two particular processes – automated testing and smoke testing – in greater detail.

2.2.1 Continuous Integration and Smoke Testing

Software development traditionally suffered from a “big bang” phenomenon, in which new pieces of a system – which depended loosely on one another – were developed separately, and then joined together later in the process. This approach tended to lead to an “explosion” of integration problems in the later phases of the project – often at a point and to an extent that was not anticipated in the schedule. Microsoft’s teams in the 1990s [11] popularized an approach of instead perform frequent, bite-size integration of new pieces of this system. Importantly, this was combined with a “smoke test” that confirmed after each such step the basic operation of the system by running system-wide tests of essential functionality. Individuals who “broke the build” by preventing integration could be easily identified and would be notified, and would generally be responsible for rectifying the situation. Because of the poor modularity associated with current System Dynamics models (see Section 3.3), the “big bang” phenomenon has not typically been as pronounced in System Dynamics, and there is appreciation for the need to start simple, and expand a model incrementally. However, the use of “smoke tests” could be fruitfully adopted from the software development area, so as to provide rapid checks to confirm that overall system behavior stays within certain recognized limits (e.g. that stocks representing physical resources do not go negative).

2.2.2 Automated Testing

Since at least the 1980s (and probably earlier), testing in software development has taken place in a combination of modes: Automated and Manual. In addition to its critical use in Smoke Testing (see section 2.2.1), automated testing is also widely used for a variety of other types of tests, notably including regression tests. Such tests – designed to prevent a “Regression” in the software – both check to for resurfacing of old defects and to make sure that previously implemented features continue to operate properly even as new features are added to a piece of software. Frequently there are many – hundreds or thousands – of such tests run on a nightly or semi-daily basis. Additional focuses of automated testing include random testing of algorithms where quick (if not definitive) checks on results are possible, and of User Interface elements (helping to raise confidence that the user interface works consistently and properly in invoking the remainder of the system).

We believe that similar sorts of testing could play a role in the System Dynamics area. While System Dynamics models often produce unexpected and surprising results, we contend that something similar to regression testing could find a place in System Dynamics modeling. While we leave thorough discussion of possible test criteria – and discussions of implementation strategies – to another paper, obvious criteria would include checking that variables (including input parameters) are within some clear range (e.g. non-negative, or between 0 and 1), confirming conservation properties (that the sum of a set of stocks is close a particular value), validating history properties (e.g. that a given quantity never declines over time), ordering properties (e.g. one value is always above another).

We particularly note the applicability of regression tests for confirming whether any mistakes resurface (whether due to similar flawed lines of thinking, or because flawed structure is resurrected in a later stage of modeling). Certain mistakes are common in System Dynamics modeling – for example, mistaking a rate for a time constant, etc. Many – but not all – such mistakes could be characterized in tests immediately after their discovery, inspected for on a regular basis.

2.3. Customizable Style Checking

In both System Dynamics and software development, build tools routinely check the apparent semantic consistency of the software artifacts that they process. Like their more general software development counterparts, System Dynamics build tools routinely find errors both in the phrasing of models ("Syntax errors" such as a misspelled function name for a forgotten "+" in an expression) and in the semantics of models (e.g. cases in which a subscript is applied to a variable that does not support it). Such checks help avoid errors during execution of a model or program, and are a key productivity enhancer. However, such routine error messages do not address cases where a model is potentially correct, but where risky practices are being employed.

Since at least the 1970s (which saw the advent of the "lint" program for the C programming language), there have been a growing set of tools provided for checking software programs. Such tools could be readily applied in the System Dynamics areas. For example, simple heuristics could be used to spot many common errors (e.g. discovering the addition or subtraction of a stock & flow, or similar variable names). More sophisticated (but readily realizable) analysis could be used to, for example, recognize cases where a divisor could plausibly be zero, when a formula that expects a value between 0 and 1 is passed a value that could fall outside of that range, or cases where there could be strong loss of precision in calculations due to the underlying mathematics floating-point arithmetic. Additional examples of where a “boilerplate” style checker could be helpful would be for spotting variable names that are very similar (and thus risk confusion), a quotient where the divisor could apparently be zero, unit inconsistencies, and cases where subscripts on the left and right side of an equation do not vary in a typical way, and cases where there are many formulas that could be phrased more succinctly (and with reduced risk) in a single formula.

The past 10 years have witnessed the growing use of customizable style checkers in development. Such checkers can enforce user-specified rules, and are frequently used to check team-dictated stylistic conventions. For example, such style checks could confirm consistency in the capitalization or punctuation of variables. It is our conjecture that customizable style checkers can offer considerable promise for enforcing project- or team- specific conventions in the System Dynamics world as well.

2.4. Peer Reviews

One of the foremost discoveries in the software quality assurance area within the past several decades has been the efficacy – and cost-effectiveness – of performing peer reviews. While software development projects routinely devote major efforts to automated and manual testing of software, peer reviews – review of technical artifacts by peers – have been demonstrated to be even more effective in maintaining software quality. Peer reviews vary along many dimensions, including the whether they take place during a pre-scheduled meeting, whether artifacts to be reviewed are circulated prior to the meeting, the level of preparation required, whether there are formal roles assigned to different parties, the presence of follow-up meetings, etc. Types of reviews of particular relevance for System Dynamics modeling include the following:

- **Pair programming**. Pair Programming (popularized as a part of the Extreme Programming Agile methodology [12]) involves two or more individuals working at software development side-by-side. Despite the name, this approach can be applied at any phase of the modeling process, from requirements recording to architecture and design to implementation to test design and execution and debugging. Experience has demonstrated that pair programming can significant improve the quality of reasoning exercised during software development.
- **Peer Deskchecks**. Peer deskchecks are an informal, unscheduled sort of review that can take place at ad-hoc points, when a creator seeks feedback on their work.

- **Formal Inspections**. Recognized as a software development industry Best Practice [13], formal inspections have been shown to find a larger fraction of defects [14], and to do so with less investment of human time per defect found [14].

While these approaches are applied overwhelmingly for the verification of software engineering systems, we posit that they could offer strong benefits both for technical correctness ("did I build the model right") and model representation ("Did I build the right model" when applied to System Dynamics modeling). While such approaches enjoy a measure of informal use in current modeling practice, we believe that the value offered could be much greater if they were institutionalized in a manner comparable to what is seen in contemporary software development organizations. At the experimental level, we see the quantitative evaluation of such methods as an important line of investigation.

3. Technical Innovations

In addition to the process-related innovations, there are a number of technical innovations that can shape how System Dynamics models are specified.

3.1. Language Support for Annotations and Metadata

The languages used for specifying programs by many – but not all – programming languages are statically defined. While languages such as LISP have long allowed for extending the syntax of the language, most procedural and object-oriented languages have long adhered to a fixed grammar, in which a pre-defined set of constructs can be used. While such languages are computational universal – and are thus capable of expressing the same computations specified by any program in more flexible languages such as LISP – sometimes expressiveness and programs suffered as a result. In more recent years, languages such as Java and C# have relaxed the strictures of language definition, permitting user "annotations" that allow the user to mark up elements of a program with user-specified information. Thus a software team might define annotations to indicate the operating system platform or position in a distributed computational system (client/server) to which particular pieces of code belong, or even the unit of measure associated with the results of a construct (such as a function or method). Alternatively, annotations might be used to describe more process-oriented metadata (here, used to mean data about the program), such as the organization or individual responsible for creating a given construct, the time at which it was created, etc. When joined with language parsing and *Reflection* mechanisms – which allow a program to access information about its own structure – such annotations help support the creation of custom or 3rd party tools to run over the specification ("source code") of a program, and identify or extract information of interest (e.g. reporting all elements created older than a certain date, or created by a team within a given time period). It can also allow for custom reasoning – for example, performing semantic analysis such as that conducted by unit checkers.

The use of annotations bears consideration within the System Dynamics context. Tools that are currently "hard coded" into System Dynamics packages – such as Unit Checkers – might be more flexibly and easily built by riding atop of annotations. Currently, much metadata – the source of a parameter estimate, the party responsible for deriving an expression or specifying a table function, the degree of confidence about or estimated standard error associated with a given piece of data, notes on alternative formulations that have been explored for some structure – is either not specified at all, or is specified in external documents or in comments.

While maintaining metadata in external documents or comments is better than neglecting it entirely, it does not lend itself to automated processing. Such processing could, for example, allow for automated reasoning (e.g. unit checking), or identification of components of the program in response to custom queries – for example, identifying pieces of data with specified confidence intervals (or with confidence intervals of above a certain size), those lacking a clear attribution or provenance, or those using "stale" data (data timestamped as having a source prior to a specified date).

3.2. Naming Conventions

Both System Dynamicists and Software Developers have long needed to provide names to quantities in their models and programs. For System Dynamicists, such names apply to variables in the model (stocks, flows, auxiliaries, table functions, etc.) as well as to custom graphs, output filenames, and other constructs that articulate with a model. For Software Developers, primary naming concerns relate to program constructs – such as variables, functions, modules, classes, and types – but also to components such as database tables and columns, resources, filenames, and filenames of various sorts.

For many years, such naming was treated as purely a personal matter, and one of limited concern to the software development enterprise or the software engineering process. In the late 1970s, the pioneering work of Simonyi [15] helped draw attention to the potential for naming conventions for communicating software developer intention, thereby enhancing reasoning and elevating the quality of the software artifacts produced. The ensuing years have growing recognition of the benefits of choosing names carefully [16, 17], and a proliferation of formalized and semi-formalized naming and stylistic conventions [18]. While some approaches have focused on the importance of clarity in the meaning of the name of a construct, others [19] have sought to explicitly encode metadata within the chosen name – for example, using the name to encode information about how it is to be used (e.g. as an index, as a map, as a storage array), and the associated formal universe of values ("type") from which it is drawn. Concurrently, recent advances in language design have witnessed a relaxing of constraints on variable name selection (e.g. as newer languages put aside earlier imposed limits on variable name length), thereby providing developers with enhanced flexibility in selecting names.

By contrast, System Dynamics practice has seen comparatively little in the way of formal naming guidelines or conventions. While guides to good modeling practice do give some guidelines for selection of clear names, there has been little formal attempts to encoding metadata in names, or even to define basic lexicographic guidelines.

We would argue that clear capitalization, consistent abbreviation, and word separation conventions within System Dynamics models would aid in the sharing, transparency and adoption of models. In the absence of the sophisticated analysis tools that allow some metadata (e.g. types) to be provided to software developers "live" while editing code, we also believe the benefits of incorporating metadata into naming guidelines could be particularly valuable. For example, variables could be named in such a fashion to indicate the associated subscripts. To minimize the risk of unit errors, some variables might additionally incorporate an indication of the units associated with the variable.

3.3. Modularity

One of the key principles that has emerged in the software engineering area is the importance of modularity in software. The capacity to subdivide a program (or model) into pieces affords a number of benefits: A divide-and-conquer strategy in which different developers are responsible for developing or modifying different pieces of a program in parallel, the capacity to build up a program piece-by-piece, the cognitive and the capacity to substitute one piece for another, and the capacity to "mix and match" different systems out of a universe of pieces, and the ability to reduce the number of changes required when a given conceptual subcomponent must be modified. While constructs such as the "molecules" of Hines [20, 21] have sought to articulate modular subcomponents of System Dynamics models, and some tools have emerged to piece together a program out of pieces, most System Dynamics modules remain stubbornly monolithic. While different conceptual pieces (e.g. a 1st order delay formulation, or an aging chain) might be used in different models, often this is accomplished simply by cutting and pasting from the original. A change in that original structure (e.g. to fix a mistake or capture information with added fidelity) would then need to be replicated across many different particular models.

Modularity in System Dynamics models could be enhanced permitting different pieces of a program to be composed by different modules (represented, for example, in different files), with the whole model. A given module might be used in several different models. Modifications to that module would only need to be made at one location, and would then be automatically exploited by other models.

3.4. Encapsulation and Abstraction by Specification

A concept related to but distinct from modularity is that of encapsulation. Encapsulation is concerned with taking a modular system further by hiding the details of the modular subcomponents, and allowing outside components to reason about and interact with that modular component through a well-defined interface. Encapsulation helps ensure that the other pieces of the system that depend on a module do so through a small set of well-defined pieces – thereby minimizing the cognitive load of those creating the interfacing components, and allowing for modification of the internal elements of the module of concern with confidence that their structure is not relied upon by the external modules. *Specifications* of the behavior of a module – focusing on *what* it does, rather than *how* it does it – provide the "well defined" understanding between the module and external modules that use that functionality. As with a legal contract, such a specification allows both parties to be clear about what is expected. The creator of a module is clear about what is being promised to outside parties – and about what is not being promised (and can therefore be modified). The user of a module has a clear specification of what is being promised, and need not be concerned about the internal details of how the module works.

Following investments in modularity, System Dynamics modules would be well suited to abstraction by specification. Such specifications could state the variable names and characteristics (e.g. units) that are provided as output of or input into a module, and what behavior is must be provided for the inputs (e.g. that a variable must take on a value between 0 and 1 inclusive, or that another variable must be non-negative. Given these conditions, the specification would further specify the properties guaranteed for the outputs – including invariants that specify properties that hold at any one time (e.g. the output is strictly positive) and history properties that specify any guaranteed behavior over time (e.g. the output is non decreasing).

Encapsulation via specification provides the well-defined guarantees that permit different parties to work on different pieces of a system with confidence that they will “play well together” even as the system evolves.

3.5. Interface-Based Development

A further extension to the principle of encapsulation and specification is to distinguish the specification itself from a particular module that implements that specification. Modules would depend not on each other’s implementation, but instead could use any module that implemented some specification. A given module might then implement one or more specifications – and could be used in any place that one of those specifications was called for. This approach is termed “interfaced based programming”, and allows for ready substitution of one module for another – as long as they the second one obeys the required interface. Interface based development also allows for components of a program to be developed and to have some validation attempted before the pieces on which they depend (via interfaces) are developed – something made possible because the interface is well defined. Interface-based development permits use of mocking (see Section 3.6).

We propose that System Dynamics packages using interface-based development for modules would offer a significant enhancement of the System Dynamics development process. Among other considerations, just as interface-based development has allowed for emergence of a 3rd party marketplace of components in interface-based languages such as Java, C#, it could also allow interchangeable System Dynamics modules (components).

3.6. Mocking

Given the presence of an interface-based module (component) system, the opportunity is raised for a software development technique offering great power in system development and debugging – mocking. Mocking takes advantage of the ability to interchange any implementation of an interface. Specifically, it deliberately substitutes a greatly simplified – but cleverly built – “mock” where a full-fledged component (matching the same interface) would otherwise be expected. Such a mock for A can allow for testing the client code before the full component A is developed. It can also allow for more localized testing of one or more components C of a program (or model) by using purposely simple (mocked) versions of other components used by components C. Debugging can also be enhanced by substituting in more and more mocks where full components would otherwise be used, thereby increasingly localizing a problem.

A modularized, interface-based System Dynamics program could benefit from the use of mocking to speed development, testing, and debugging.

3.7. Plug-In Architectures for System Dynamics Modeling Packages

An important enabling technical advance in software development concerns the creation of modular, "pluggable" integrated development environments. While the early decades of software development were marked by the use of simpler text editors for use in editing programs, the 1980s and 1990s witnessed the rise of integrated development environments that would support not just editing text documents, but also the source code browsing, build, debugging or testing processes, profiling, database access, and integration with version control systems (see Section 3.8). Building on the vision of GNU EMACS [4] but expanding beyond its text-only interface, a variety of graphical "integrated development environments" were developed to provide a rich platform for developers. While early environments (such as those associated with the Microsoft Visual X series [22]) supported a wide range of functionality, and increasingly provided interfaces for 3rd party extensions (e.g. for additional build tools, UML or data modelers, etc.). The rise of the Eclipse project [5] in the 2000s marked the emergence of a prominent open-source system offering deep extensibility via a "plug-in" architecture that was core to the project. Eclipse helped both to stimulate and benefitted a massive outpouring of 3rd party plug-ins, handling tasks from profiling to visual modeling to database integration to UI design to version control access. Such tools ease and enrich the software development process, and the vibrant 3rd party market coupled with the ability to "mix and match" different plug-ins for a given project aided in significant enriching the Java development space.

While the user base of System Dynamics modelers is nowhere near as big as that for Software Developers, we posit that System Dynamics modeling platforms could benefit from the adoption of plug-in architectures. Among other factors, this could allow 3rd party developers to more readily support tools in all of the major platforms, and further enrich the tools available to modelers, and reduce pressure on the primary package developers to add support for non-core functionality (e.g. tools for calibration and sensitivity analysis, model and data visualization, software artifact provenance and versioning systems, etc.)

3.8. Version Control and Documentation Systems

For decades, Software Developers have placed source code, design documents, test scripts and code, database schemas, requirements specifications, graphical resources, estimation documents, and other elements of a software project, within repositories. These repositories – such as Subversion, RCCS, SCCS, Visual Sourcesafe, etc. – are known variously as “Version Control” systems, “Source Code Control” systems. However, the use of such systems have evolved over the years, and such names do not fully do justice to the breadth of functionality and versatility of such systems, or to the absolutely central role that such system have played in modern software development.

Such repositories do support saving away successive version of software artifacts, but also operate as key connective processes linking different developers with information on the history of contributions to the project, help understand the relationship between two developer’s changes, integrate with the continuous integration and build processes (see Section 2.1), allow for retrieval of historic documents, support status updates to enable “change awareness” across the project, and increasingly serve as a rich set repository of information for process visualization and decision making.

While some System Dynamicists routinely use such version control mechanisms, they offer potential for much wider and richer use. For example, version control systems could support cross-linking various artifacts produced analyzing run output, to the assumptions underlying such runs, and the outputs of such runs, all associated with the corresponding model version. Just as in Software Development, “change awareness” can add important insight and inform decision making in the System Dynamics process, and version control systems can provide such awareness. As in software development, such systems can also serve as sources of data for managerial insight, via visualization, and through use of data mining and business intelligence tools. The recent SILVER [3](and accompanying SILVERVIZ[23]) system offers an important contribution in this regard, but suffers from poor support for 3rd party tools in many of today’s software platforms.

3.9. Assertions

One of the most powerful technically-involved innovations to assist software development in the past several decades is the use of assumption-checking predicates known as assertions. The key idea behind assertions is that software developers routinely make use of assumptions when building software, and that the correctness of such assumptions is essential to the correct operation of that software. A close corollary to that premise is the observation that it is desirable to have such assumptions documented and, if there is a defect in a software developer’s assumptions, it is best that it be recognized as soon as possible in the testing of the software system. Assertions are a technique for systematically, explicitly and operationally documenting a software developer’s assumptions, such that they are automatically checked throughout the testing, development, and debugging processes, and laid bare for anyone seeking to modify the program specification. Because many software projects disable assertions once the software artifacts are provided to clients, assertions will sometimes verify quite high-level program properties – for example, the fact that a tuned version of an algorithm yields the same results as a straightforward but computationally expensive version of that algorithm, or that two different ways of computing some quantity yield the same results. Experienced software developers will often routinely pepper their code with such "sanity checks", reflexively creating an assertion to document assumptions that they consider while designing the program. Such assertions are responsible for uncovering a massive number of defects – both defects that reflect logical reasoning errors and those that reflect communication lapses or other misunderstanding between developers. While individually humble and modest in their implications, collectively the assertions for a software project commonly represent one of the pillars on which the quality processes for that project rest.

The fundamental motivations for, ease of creating, and capacity to implement assertions apply as much to System Dynamics as they do to other areas of software development. Like all areas of Software Development, building System Dynamics routinely involves reliance upon assumptions. Some of these assumptions – concerning the fashion in which external processes work in the real world – are not typically feasible to test from assertions. However, others assumptions can be readily and valuably tested. For example, one could confirm that a stock is non-negative, that parameter representing a fractional quantity is between 0 and 1, that a discount rate is non-negative, that a set of stocks total up to some known value, or that several computed fractions for each of the different sub-pieces of some whole total up to 1. With somewhat more mechanism, one could confirm history properties, such the fact that a given stock is non-decreasing, or that it never declines beyond its initial value. Just as Software Developers will sometimes check assumptions with more involved computations (e.g. comparing two versions of an algorithm), System Dynamicists might even try checking the divergence of results between two different implementations of some model structure (one a simplified proposed representation structure, another more complex structure purely used for the sake of testing). If the two implementations prove to be consistently sufficiently close in their results, the degree of confidence in the adequacy of the simplified representation would be enhanced.

As for Software Development more generally, such assertions are, on an individual level, of quite modest benefit. However, by serving as documentation for assumptions, they collectively help to significantly reduce the risk that model modifications will inadvertently run afoul of assumptions – and, worse, that violations of some of the hundreds of relative assumptions in a model might go unnoticed. Moreover, there are some cases which raise significant risk of the chance of violation of such assumption. Examples would be sensitivity analyses and calibration processes. Such processes routinely vary one or more sets of parameters over ranges that may inadvertently violate assumptions – such as by drawing a stock of a physical quantity negative, or by depressing the values of auxiliary values outside some range. An assertion failure during such exercises could signal that the parameter variation has gone outside the assumptions of the model, but allow for continuing on to examine other simulations with acceptable assumptions.

Existing System Dynamics packages do offer some manner of implementing assertions – for example, by having conditional statements that force an arithmetic expression (e.g. cause a division by 0 error) in the event that the tested condition is violated. As in regular software development, such assertions could be disabled in the event that the code is used by clients, or when faster simulation is sought (e.g. for a demonstration). However, System Dynamics packages would benefit from explicit support for assertions – for example, allowing logging of custom error messages upon assertion failure, and providing built-in options for enabling or disabling assertions.

3.10. Reflection and Metaprogramming

While early computing approaches tended to blur the distinction between program encoding and the data on which they operated [24], for many subsequent decades, the dominant computing languages enforced a strict and narrowly defined separation of these two domains. Within this methodology, programs operated on data, but not on their own representation. While this imposed constraints, this approached simplified reasoning about many aspects of program structure. For many years, the construction of programs that reasoned about – and even modified – their own structure was a black art, and was relegated to experts in small subsets of the computing field.

There are compelling reasons for giving programs access to their representation and to information about their behavior, and to “metadata” regarding the data circulating within them – a technique that broadly goes under the contemporary label of “reflection”. Such access allows programs to reason about their own structure, eases the creation of debugging and testing instrumentation. For example, the use of reflection permits general test suites such as JUnit [25] or NUnit [26] to identify testing methods within the programs to be tested. Reflection also permits debuggers to access information on the structure of the program being debugged, on the data fields and methods associated with different elements of program structure (e.g. within classes or associated with functions or structures). Access to metadata can also allow a program to implement just one or a small number of general mechanisms where many specialized mechanisms would instead be required – for example, implementing a single routine to persist objects to a database (where that routine uses metadata to identify the relevant data in each object), rather than using a set of specialized methods, with one such method for each object. The capacity to *change* a program's representation (also supported by some implementations of reflection) can allow for performance enhancement (e.g. user's query could trigger creation of specialized search code custom-tuned, or empirical observation of program use could allow for the run-time specialization of pieces of a program to certain highly utilized cases), for modification of program semantics (e.g. "hooks" to allow for incorporation of logging functionality, or debugging messages or reports), and for application of techniques such as genetic programming [27] that permit evolving program structure.

Within a System Dynamics context, the prospects for access to metadata and information on model structure offers some significant opportunities. As for general purpose programs, the creation of 3rd party debugging and testing tools can be considerably facilitated by support for reflection. While the provision of existing APIs do allow for support for such tools, opening access to reflective capabilities within the model logic might aid this process further, and support the creation of 3rd party tools for common tasks such as calibration and sensitivity analysis. The creation of evolving System Dynamics models has been explored in some previous contributions, but could be greatly facilitated through the use of reflection. While Aspect oriented approaches might provide a better avenue for instrumenting program functionality (e.g. for logging or persistence), thorough support for reflection techniques could also give access to such functionality.

3.11. Aspect-Oriented Approaches

Languages popular for software development have long been effective in supporting the definition of centralized points of logic for program functionality. In classic procedural languages (such as C, PASCAL and FORTRAN), functions (or "subroutines") are responsible for much functionality. In modern object-oriented languages, classes group together state and functionality associated with certain types of entities described within a program. While such abstractions support the succinct description of many types of computational needs, there are certain types of needs for which they offer a poor fit. One of the most notable is “cross-cutting” functionality -- common services (such as persistence, logging, reporting, object pooling, transactioning, security concerns, etc.) that are needed across many modules of entire program.

While cross-cutting functionality can be achieved by traditional mechanisms – for example, by scattering function calls to invoke such services throughout a program – such distribution of control is fragmented, offers limited transparency, is brittle to change, and time consuming to place. Frequently underlying logic by which the calls are placed is not documented or captured in a single location; if it is, it is typically only the result of commenting rather than explicit language support, and can be easily overlooked. As a result, the system can easily fall prey to omission of such service requests where they are needed, or placement of requests for services where they are not required. When the service interface evolves (perhaps requiring an extra parameter, or just one call and not two), the program must be changed at many locations. Through oversight, a developer might neglect to update certain of the uses, potentially yielding a defect. Finally, the placement of such service requests throughout a program can be very time consuming. The time can care required for such placement less of an issue when the code involved is for long-term use – e.g. for transactioning, object pooling, and security concerns. However, such placement of calls is sometimes used to address ephemeral needs – for example, to report on certain components of program operation or for logging during debugging. Placement of the full set of calls required across can entail substantial effort; rushed deployment of such calls may yield omissions or erroneous placemen, thereby impairing the accuracy of the reporting and of the insights gained during debugging.

To allow for succinct, expressive specification (and, implicitly, documentation) of cross-cutting functionality, researchers in the 1990s and 2000s introduced the notion of “Aspect Oriented” programming. Aspect Oriented programming provides a centralized means of describing and deploying cross-cutting functionality across a program. In its most popular form [28], aspect oriented programming permits software developers to specify the patterns (“pointcuts”) specifying locations at which specified bits of code should be woven into a program, and the specific code (“advice”) to weave in at those points. Pointcuts provide a succinct way of specifying the rules for location of such cross-cutting functionality – rules that would otherwise often be left implicit within the structure of a program.

There are a number of cross-cutting concerns that apply within design and use of System Dynamics models. During debugging, having regular updates on the value of model variables (or expressions involving such variables) can be very helpful. Aspects would provide a succinct way of indicating the variables on which to report, the frequency of such reporting, and any processing required prior to reporting (e.g. computing a function of several of the variables, such as totalling them up). Aspects would also provide a clean way of implementing custom persistence mechanisms (e.g. saving certain data to a database). In addition, aspects could be used to test certain invariants across a program -- thereby providing a concise mechanism to specify cross-model assertions (see Section 3.9). Aspects might also merit consideration as a powerful and more transparent way to specify subscripts across an entire section of a program from one central place, thereby reducing the effort that is required when adding or deleting a subscript. Coupled with Reflection mechanisms (see Section 3.10), Aspects could potentially further provide a convenient way to implement model analysis algorithms, such as loop gain analysis, eigenvalue elasticity computations, etc.

Most current implementations of aspect oriented programming are geared towards incorporation in object-oriented languages, and an implementation in System Dynamics packages would need to be adopted dramatically to accommodate the declarative nature of existing System Dynamics languages. This effort may draw important insights from other attempts to apply aspect orientation to functional and declarative languages, such as from the PolyAML experience [29].

Acknowledgements

Nathaniel Osgood gratefully acknowledges the generous support of the National Engineering and Science Research Council (NSERC) Discovery Grant program for this work.

References

[1] T. A. Hamid and S. E. Madnick. 1991. Software project dynamics: an integrated approach. Prentice-Hall, Inc., Upper Saddle River, NJ.

[2] R. J. Madachy. 2007. Software Process Dynamics. Wiley-IEEE Press, Los Alamitos, CA.

[3] N. Osgood. 2009. Silver: Software in support of the system dynamics modeling process. The 27th International Conference of the System Dynamics Society.

[4] GNU Emacs. http://www.gnu.org/software/emacs/.

[5] Eclipse. http://www.eclipse.org/.

[6] H. Rahmandad and D. M. Weiss. 2009. Dynamics of concurrent software development. System Dynamics Review 25(3): 224-249.

[7] D. M. Raffo. 1996. Modeling software processes quantitatively and assessing the impact of potential process changes on process performance, Ph.D. Dissertation. Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA.

[8] A. Zeller. 2005. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[9] R. Metzger. 2004. Debugging by thinking: A multidisciplinary approach. Burlington, MA: Elsevier Digital Press.

[10] B. Lewis. 2003. Debugging backwards in time, In 5th Workshop on Automated and Algorithmic Debugging (AADEBUG), Ghent, Belgium.

[11] M. A. Cusumano. 1998. Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People. The Free Press, New York, NY.

[12] K. Beck. 1999. Extreme Programming Explained: Embrace Change. Addison Wesley, Reading, MA.

[13] D. A. Wheeler, B. Brykczynski, and R. N. Meeson (editors). 1996. Software Inspection: An Industry Best Practice for Defect Detection and Removal. IEEE Computer Society Press Los Alamitos, CA, USA.

[14] K. E. Wiegers. 2001. Peer-Reviews in Software: A Practical Guide. Boston, Addison-Wesley.

[15] C. Simonyi. 1977. Meta-Programming: A Software Production Method. PhD thesis, Stanford University.

[16] S. McConnell. 2004. Code complete, 2nd ed. Microsoft Press, Redmond, WA.

[17] R. C. Martin. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

[18] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur and P. Thompson. 2000. The Elements of Java Style. New York, NY: Cambridge, University Press.

[19] Hungarian Notation. http://msdn.microsoft.com/en-us/library/aa260976(VS.60).aspx.

[20] J. H. Hines. 1996. Molecules of Structure. System Dynamics Group, Sloan School of Management, MIT.

[21] R. Eberlein and J. Hines. 1996. Molecules for modelers. Proceedings of the International System Dynamics Society. Cambridge: System Dynamics Society.

[22] Z. Naboulsi and S. Ford. 2011. Coding Faster: Getting More Productive with Microsoft Visual Studio. Microsoft Press.

[23] Y. Xue, N. Osgood and C. Gutwin. 2011. SILVERVIZ: Extending SILVER for coordination in distributed collaborative modeling. In The 29th International Conference of the System Dynamics Society, Washington, DC. 108.

[24] H. Abelson, M. Halfant, J. Katzenelson and G. J. Sussman. 1988. The Lisp Experience. Annual Review of Computer Science, 3, pp. 167-195.

[25] JUnit. http://www.junit.org/.

[26] NUnit. http://www.nunit.org/.

[27] J. R. Koza. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). A Bradford Book, 1st edition.

[28] R. Laddad. 2009. AspectJ in Action: Enterprise AOP with Spring Applications. Manning Publications, 2nd edition.

[29] D. S. Dantas , D. Walker , G. Washburn , S. Weirich. 2005. PolyAML: a polymorphic aspect-oriented functional programming language. Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, Tallinn, Estonia.