Context Sensitivity and Ambiguity in Component-based Systems Design

Stephen J. Bespalko, Principal Investigator
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-0977 USA
Tel: (505) 845-8847
Fax: (505) 844-2057
Email: sjbespa@sandia.gov

Alexander Sindt, Technical Intern
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-0977 USA
Email: scapino@mit.edu

Abstract

Designers of component-based, real-time systems need to guarantee the correctness of software and its output. Complexity of a system, and thus the propensity for error, is best characterized by the number of states a component can encounter. In many cases, large numbers of states arise where the processing is highly dependent on context. In these cases, states are often missed, leading to errors. The following are proposals for compactly specifying system states which allow the factoring of complex components into a control module and a semantic processing module. Further, the need for methods that allow for the explicit representation of ambiguity and uncertainty in the design of components is discussed. Presented herein are examples of real-world problems which are highly context-sensitive or are inherently ambiguous.

Keywords: Context Sensitivity, Ambiguity, Component-based Software, Real-time Systems, Formal methods

Workshop Goals: Share research; publicize the importance of ambiguity and context sensitivity in the design of real-time systems; find new solutions to real-world problems.

Acknowledgement and Disclaimer: This work was supported by the United States Department of Energy and was performed at Sandia National Laboratories under Contract DE-AC04-94AL85000.

The opinions expressed in this document are those of the authors, and do not necessarily reflect the opinions or positions of their employers, other individuals, or other organizations. Mention of commercial products does not constitute endorsement by any person or organization.

1 Background

Stephen Bespalko has been involved in the design and implementation of large-scale, real-time systems for over twenty years. For the past five years Mr. Bespalko has been involved in primary research into the problems confronting the implementors of large-scale systems at Sandia National Laboratories. His current research is in formal methods applied to satellite ground station components. Prior to joining Sandia National Laboratories, he spent 8 years as part of the R&D staff at Xerox Corporation. While at Xerox, Mr. Bespalko successfully applied formal language theory to the design of numerous high-tech components including a digital scanner, document pagination algorithms, graphics algorithms, fax devices, and communications protocols. Mr. Bespalko holds several US and International Patents resulting from his work at Xerox.

Alexander Sindt is a Junior at the Massachusetts Institute of Technology where he is double-majoring in Computer Science and Philosophy. He has spent the last two summers working as a Student Intern at Sandia National Laboratories in the areas of GPS applications and Formal Languages.

2 Position

Complex systems designed to control real-world events are becoming commonplace, yet no feasible method exists for confirming that such systems behave as desired. These complex systems are typically several hundred thousand lines of code, and are occasionally in excess of one million lines. We define mission surety as a key aspect that a complex system must exhibit; i.e., the user can be sure that an event has actually occurred as reported, and software reacts to that event as expected. Current development methods attempt to demonstrate mission surety through extensive testing with a wide range of simulations. Oftentimes these efforts result in surprises (such as determining that the tests are incomplete so that an incorrect portion of the code is not exposed until late in the development cycle), which in turn leads to higher costs and missed schedules. There are two principal strategies for achieving mission surety: first, develop (or obtain) reusable modules that can be included in systems - thus simplifying the development process, and second, automate as many of the complex development tasks as possible. Although there are numerous alternatives for most functions common to large complex systems such as display managers, network infrastructure subsystems, and database managers, other tasks are not well supported in Commercial Off-The-Shelf (COTS) software or other reusable forms. These functions include splitting bulk data into pieces, validating the sequence and structure of the data, extracting meaning from data, detecting runtime errors and, finally, automatically performing error recovery. Unfortunately, these are complex and error-prone operations that easily compromise mission surety. Automating the development and implementation of these functions is the key to generating more capable systems at a lower cost with greater mission surety.

Unlike the strategy of achieving a high degree of correctness and completeness through component-based development, or code reuse, the ability to model a large set of system states appears to require a different strategy. It is suggested that many 'real-world' applications require the automatic generation of custom components to handle the inherent complexity of the problem.

The position taken here is the result of an investigation into the reasons why certain software components are considered to be problematical. That is, after numerous development cycles, the software is still perceived as error prone. All of the modules we investigated provide the similar function of transforming data from one structure into another. In each case, the developers attempted to apply various compiler generating tools to the problem at hand, but failed because the tools were simply overwhelmed. The developers then attempted to hand-build components that are vastly more complicated (in term of the numbers of states the system can enter) than modern compilers. Our conclusion is that the software can be improved though the factoring of the software into a control (or state modeling) module, and a semantic processing module. The former can be generated automatically if the interface between the two components can be specified with a formal language. This, in and of itself, is not a particularly new suggestion - the functions provided by the problematical modules are not that different than the first stages of a modern compiler. No competent computer scientist would attempt to develop a compiler without the tools to automatically generate the lexical (see [1]) and syntactic (see [2]) processing modules.

Thus, our goal is to raise awareness that there are classes of problems that are not well suited to the formal language tools available to the software engineering community. The basic problem we have seen relates to the tendency for the tools to handle small problems well, but not problems of interest to the engineer working on state-of-the-art problems. The second issue we wish to raise is that there are problems too complicated to be approached with a finite-state solution. We will illuminate this by discussing two applications requiring models more complicated than typically associated with systems development tools. The first is the context-sensitive grammar, and the second is directed backtracking, sometimes referred to as the assumption-based truth maintenance system. The former is an important and powerful tool for compactly representing large numbers of system states in situations where the complexity arises from the specific context in which the system is operating; the latter is appropriate when the complexity arises from multiple branches in the solution algorithm.

2.1 Context Sensitive Situations

Context sensitive situations involve the interpretation of data in which the interpretation of one segment of data is dependent upon the interpretation of another segment. In other words, the meaning of one piece of data can potentially change the meaning of a piece of data elsewhere in the system. This is often the case with data transmission, where signals to change the interpretation mode are encountered regularly, or in situation-reactive systems, where interpretation of data from one set of sensors or instruments affects the interpretation of another set of data. Context sensitive situations are a source of subtle programming errors which are difficult to identify and fix. The solution we are exploring is to define a set of formal languages powerful enough to handle the common context sensitive situations identified in problematic components.

The following example outlines the problem, and shows why concise specification is an extremely important attribute of a well-engineered solution. The basic scenario is simple. A data stream includes the following structure:

$\texttt{startVal count time frame}_1\ \texttt{frame}_2\texttt{...}\ \ \texttt{frame}_{\text{count}}\ \texttt{endVal}$

Although the grammar is simple, implementing the parser for the data stream with a context-free (or worse yet, with a hand-built) parser has led to several unintentional errors. In the most straightforward formal specification, the grammar had the following fragment of BNF:

```
frames::=   startVal frameList endVal
frameList::=startVal count time frame  |
       frameList frame
```

The source of errors with this scheme is that the count is ignored; there is no declaration of the legal values for count. Further, the actions for the first frame are different than for the second and subsequent frames. The next fragment attempts to include more context sensitivity, but still tries to do so with a context free grammar.

```
frames::=    frameList
frameList::=startVal 1 time frame  |
            startVal 2 time frame frame endVal
                    ...
```

Although there is now a special case for each possible value of count, the actions for processing the frames are replicated

$$\text{maxCount} + \text{maxCount} - 1 + \ldots + 3 + 2 + 1 = \frac{(\text{maxCount} + 1)\text{maxCount}}{2} \tag{1}$$

times, where maxCount is the largest value count can assume. From a linguistic point of view, the grammar accomplished its mission. However from the point of view of the engineered solution, there was a high degree of likelihood one of the frames would not be processed correctly.

The alternative we propose is an extension of the standard context sensitive grammar (where standard here refers to Chomsky level 2, which has limitation that the length of the left side of a production be less than or equal to the length of the right side of the production, see [3]). The following grammar is compact enough for human verification and also explicitly allows the designer to specify the semantic processing for the frame in exactly one location.

```
frameList ::= completeFrame |
          frameList completeFrame
completeFrame ::=  time frame
```

{processing action for the frame}

```
time frame::= startVal 1 time frame endVal
time frame startVal 1 time ::= startVal 2 time frame
time frame startVal 2 time ::= startVal 3 time frame
...

time frame startVal maxCount-1 time ::= startVal maxCount time frame
```

In short, for each production

$$A \rightarrow B \tag{2}$$

where the length of A is n and the length of B is m, and $m < n$, there must also exist a second production

$$C \rightarrow D \tag{3}$$

such that

1. the length of $D = x$, where $x \geq n - m + 1$
2. the length of $C = 1$
3. D corresponds to the first x symbols of A.

Even though this is a small generalization of the standard (simplest) context sensitive grammar, the impact on the architecture of the software component studied was enormous: rather than a huge number of 'special cases', the component can now be specified with a few dozen compact and precise rules. This example, along with our interactions with the engineers that built the original implementation of the module, leads to several observations:

1. The formal specification must match the application closely for it to be useful to the application designer.
2. Most software engineers avoid generating (or even admit that they know about) formal specification methods because they are so difficult to use.
3. Therefore, there exists a need for simpler formal specification generation tools.

2.2 Ambiguous Situations

The following are examples where the very nature of the problem appears to be ambiguous from the point of view of the engineered problem solution:

1. High-tech devices such as a photocopying machine have one set of states associated with each 'normal' mode of operation, and a completely separate set of states for failure modes. Examples of failure modes included paper exhaustion, paper jams, mechanical part failure, incorrect paper in all of the paper trays, and power interruption during operation.
2. In certain high-consequence operations involving data transfer, data processing must continue during and after periods of data loss. In a context sensitive situation, the data lost might alter the behavior of subsequent processing steps. In this event, assumptions must be made about the lost data so processing can proceed.
3. Designs are being considered for locomotives that manage peak energy demands independently from average energy demand. Although accurate systems can be imagined for controlling the locomotive energy, assumptions must be made concerning the actual performance of the system. Reacting to discrepancies between the actual and predicted performance of the

locomotive currently involves extremely complicated heuristics. One of the more common concerns of railroad operators is the loss of efficiency due to changes in air pressure and the friction loss between the tracks and the rails as the wheels wear. In the former, it is very difficult to determine what the actual performance of the locomotive is until it is measured, while in the latter the possibility does exist for a trend to be measured.

4. The performance of tools to translate between different data forms could be greatly improved if they were less ad-hoc and convoluted. A project that one of the authors has worked on involves attempting to match spatial data between systems operating at different scales and resolution. Even the location of the same feature is difficult to identify, given that it may have different representations and actually appear to be situated in different locations in absolute space.

5. A huge majority of the traffic lights in the US are capable of operating on a network and acting in an adaptive mode. As of December 1996, officials at the California Transportation Department (CALTRANS) estimated that, nation-wide, only about 5% of overall traffic lights were operating in any kind of adaptive mode.

6. Cases have been found in the work here at Sandia National Laboratories where the answer depends on the order in which the data is received, yet the data order cannot be known ahead of time.

There has been a considerable amount of research put into developing systems capable of dealing with ambiguous situations (at least from the stand-point of handling inconsistent or uncertain data). In general these are referred to as either a non-monotonic logic system, assumption based truth maintenance system, or a directed backtracking grammar. The foundation of this work is covered by de Kleer [6]. Unfortunately, this work is abstract, and the implementations are too inefficient for deployment in high consequence applications. Further, none of the Lisp (or AI implementations, in general) have any method of generating domain specific representations of the ambiguity. Given the number of examples we have identified where there is some form of ambiguity or inconsistency in the information flowing into applications, we conclude there is a need for:

1. formal computation models based on some form of non-monotonic logic

2. research into the structure of formal languages for specifying inconsistent and ambiguous data, and

3. better tools for generating formal models based on (1) and (2).

3 Comparison

The current software engineering methodologies, such as the Object-Oriented methods espoused by Shlaer & Mellor [4], and Rumbaugh [5], are limited by the underlying processing model, in each case a finite state machine. Further these methods require the finite state machine to be constructed manually (that is, all of the states and transitions must be manually enumerated). Although adequate for simple real-time applications, such as consumer electronics products, these methods are inappropriate for large-scale systems, such as satellite ground stations or mission critical electronics and control software. These methods are also heavily oriented to a 'data-flow' analysis where the actual state architecture is relegated to exception handling.

There have been other attempts at raising the awareness of the software engineering community to the value of the formal specification in component design. The paper by Bentley [7] is noteworthy. He takes the position that small languages are useful for specification of a large number of 'small' problems, which is true. In Bentley's context, ADA would be considered a language, the PIC graphics language is considered a small language. A parser for the former would consist of several thousands of states and the latter perhaps a few hundred states - both approachable with current technology. Unfortunately, our work shows that there are other classes of problems: the examples outlined above are at least an order of magnitude larger than an ADA compiler. Thus these problems are neither small nor are they simple. There must be a completely new level of technology developed if these problems are to be solved with the same level of confidence that compiler developers have grown to expect.

The now classic paper by Strachey [8] is the earliest paper found promoting the construction of systems to perform complex tasks through the application of symbol processing. The philosophical approach was later adopted by Xerox [9] in the TIP (Terminal Interface Package), which used the symbolic (or formal) specification to generate parsing modules for transforming concrete user actions into higher level abstractions meaningful to a system designer. The TIP package defined the concept of a Trigger as a change in the state of the device of interest, in this case usually the keyboard. Although this formalism was ideal for providing a highly effective means for each application designer to allow various key combinations to have a 'local context', the model did not provide a method for handing more complex situations (such as longer sequences of key strokes, or even the power to represent the context sensitive problem discussed above).

The explicitly state-driven formal specification appears to show the most promise for providing designers of large-scale systems the opportunity to design systems that have a high degree of mission surety. We intend to extend the state-of-the art with regard to modeling formalisms that are capable of dealing with context sensitive situations and algorithms which must deal with ambiguous situations. We define ambiguous here to mean situations where the data must be inherently inconsistent by nature of the methods used to acquire it.

The work we are doing is intended to provide new methods of developing components by providing new computational models aimed at the types of applications outlined above. The technology we are starting from has achieved some notoriety from interesting demonstrations in the Artificial Intelligence community, but has not achieved wide acceptance due to the inefficiency of the current Lisp implementations. There is also a lack of adequate formal specification languages to form the basis of the interface between the application and the backtracking technology.

References

[1] Aho, Sethi, Ullman, "Compilers: Principals, Techniques, and Tools," Addison Wesley, 1985, 257-266

[2] Aho, Sethi, Ullman, "Compilers: Principals, Techniques, and Tools," Addison Wesley, 1985, 105-113

[3] John E. Hopcroft, Ullman, J., "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley Publishing Company, 1979, 223-227.

[4] Sally Shlaer, Mellor, Stephen, J., "Object LifeCycles, Modeling the World in States," Yourdon Press, 1992.

[5] ______,"The Rational Approach to Software Development," Rational Corporation, 1997

[6] Johan De Kleer, "An Assumption-Based TMS," Elsevier Science Publishers, Artificial Intelligence, volume 28, (1986) 127-162.

[7] Jon Bentley, "Programming Pearls: Little Languages," CACM, 1986, 29, 711-721

[8] C. Strachey, " A general purpose macrogenerator," The British Computer Society, The Computer Journal, Volume 8, Number 3 225-241.

[9] ______, "The Viewpoint Programmer's Manual," Xerox Corporation, Document 610E00190.

4 Biographies

Stephen Bespalko has a degree in Mathematics and Physics from Binghamton University (formerly the State University of New York at Binghamton) and an MBA from New York University. He began his career as an applied mathematician at the US Headquarters of Xerox Corporation in Rochester, New York where he was responsible for constructing financial forecasting tools. Mr. Bespalko subsequently was transferred to the Xerox International Headquarters where he was an analyst on the staff of the Executive Vice President for Strategy and Policy. After spending two years as the manager of computer aided publishing for a Xerox-owned publishing company in Lexington, Massachusetts, he transferred to the Xerox R&D staff in Palo Alto, California and subsequently to San Diego, California. Mr. Bespalko has won numerous awards and other recognition for his work in applying mathematical techniques to real-world problems. His current research interests are in the area of formal languages, in particular those involving uncertainty and ambiguity.

Alexander Sindt is a Junior at the Massachusetts Institute of Technology where he is double-majoring in Computer Science and Philosophy. Mr. Sindt is a National Merit and Presidential Scholar. He has spent three summers at Sandia National Laboratories; in 1995 Mr. Sindt was chosen as a participant in the US Department of Energy Honors Program for Science and Engineering Students. The top science student from each state is selected to participate in the program and Mr. Sindt was selected from his home state of Alaska. For the last two summers he has been a technical intern, most recently working on projects involving spatial data and formal languages.