Transformation as a Design Process and Runtime Architecture for High Integrity Software.

Stephen J. Bespalko
Victor L. Winter

Sandia National Laboratories[1]

Abstract. Designers of mission-critical 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 distinguish. In many cases, large numbers of states arise where the processing is highly dependent on context. Discussed here are successes with representing both the design process and the runtime architecture of such systems as a series of transformations. We believe this approach can have a significant impact on the construction of *high integrity software*. The discussion includes an overview High Assurance Transformation System (HATS), a language independent design tool, which is a syntax derivation tree-based (SDT) transformation system in which transformation sequences are described in a special purpose language. Further, compactness of representation plays a key role, facilitating or impeding specification in terms of transformation sequences in both design and implementation. Thus, we discuss methods for compactly specifying system states and which allow the factorization of complex components into a control module and a semantic processing module. Additionally, we will argue that in the high-consequence realm, there is a need for methods that allow for the explicit representation of ambiguity and uncertainty.

Keywords. Specification and Verification, Component-based Software Engineering, Domain Specific Languages, Software Architectures

1 Overview of High Integrity Software

The purpose of the High Integrity Software Department at Sandia National Laboratories is to develop tools and techniques that will promote the creation of *high integrity software (HIS)*. Software is considered, in this context, to be high integrity when there is *quantifiable* assurance that the software will be:

---

1. *Authors' Address:* Sandia National Laboratories, PO Box 5800, MS 0535, Albuquerque, New Mexico, United States of America, 87185-0535. *Tel:* +1(505) 845-8847. *Fax:* +1(505) 844-9478. *E-mail:* sjbespa@sandia.gov; vlwinte@sandia.gov

1. *reliable* in normal environments,
2. *safe* in abnormal environments, and
3. *secure* in malevolent environments.

As such, we are conducting research programs whose objectives are to explore correctness in the limit. The ultimate goal of the work is to advance the state-of-the-art with regard to *establishing* the correctness of complex software systems [1].

The systems designed and engineered at the National Laboratories require this intense focus on safety (or correctness and robustness), because of the immense consequence of a mistake. The National Laboratories in the United States are responsible for the basic science, engineering and production of nuclear weapons, and other systems deemed in our national interest. Clearly, these weapons must be among the highest consequence devices known to mankind.

It is in response to the challenges of establishing the correctness of increasingly more complicated systems where the consequence of a mistake is also growing, that the HIS program was engendered at the Laboratories. We believe that the development of formal methods for specific problem domains is a promising approach that can make the design and construction of HIS more cost-effective and robust. We perceive demand for tools that establish high integrity in fields as diverse as finance, telecommunications, aviation and medical technology.

Ultimately, the impact of this work will be a reduction in the scale of effort needed for engineering complex systems. The transformation technology will assist the software engineer in breaking the development process down into a series of steps, each small enough that it can be easily (and appropriately) assessed for correctness. The application of transformation to the runtime environment is reducing the amount of coding that is needed to represent the solution by increasing the reuse of well conceived components, which we will refer to as *tools and runtime modules*. Transformation is also providing increasingly more powerful models to visualize the solution.

The balance of the paper will proceed in three parts. Section 2 briefly outlines the work being done under the auspices of the HIS program in formal verification, and prototype tools to test the theory. This section stresses the creation, and quantification of the reliability of software. The software architecture for the tool depends on a sophisticated parser with extensions for backtracking. Section 3 discusses how transformation can be used to establish the safety and security of software, which is predominately a runtime phenomenon. Interestingly, the architecture of the designs discovered to date all depend on the availability of a parser (or rewriting system) of similar capability to that identified for the tools discussed in Section 2. Finally, the last section contains conclusions.

2 Formal Verification through Transformation

Today we are building systems of unprecedented complexity. In addition, technological advances are also dramatically increasing the rate at which complexity is growing. For example, planes are now being designed that are inherently unstable and require com-

plex control functions in order to fly safely. To date, software systems are at the forefront of technology's need/demand for complexity.

It is recognized that partial 'black box' testing, when applied to discontinuous systems, is less effective than when applied to continuous systems [2] [3]. Informally, the reasoning is that as the functionality of a system becomes more discontinuous, inferring the behavior of input elements that are not tested from 'similar' input elements that are tested becomes increasingly problematic. From a sampling theory perspective another way of saying this is that the operational profile of a discontinuous system is (in general) more complex and harder to define than the operational profile of a continuous system.

Given a black box testing environment, it can be argued that as a system becomes more complex it also becomes increasingly difficult to make meaningful inferences concerning portions of its input space that have not been explicitly tested. In turn, this implies that in order to obtain a given level of assurance, a complex system will need to be more extensively tested than a simple system. When high levels of assurance (e.g., less than 10^{-9} failures per operational hour) are required for complex (e.g., highly discontinuous) systems with large input spaces, traditional black box testing paradigms become ineffective [2] [3].

During the design and development phases of a software system there are two general sources of error:

1. the initial set of requirements are incorrect, and
2. a correct set of requirements exist, but they are implemented incorrectly.

Given this partitioning of design and development errors, let us consider black box testing of complex software for which we require a high degree of assurance. As we have already mentioned, demonstrating high assurance of a complex (software) system will require the examination of a very large number of test cases. Realistically, the only way that one could hope to examine large numbers of test cases is through automation. In order to automate testing, a formal (and computable) specification, often called a *test oracle*, is needed to determine (recognize) whether the input/output pairs generated by the testing process are 'correct'[1]. Therefore, in the context of high assurance, when comparing alternative approaches to black box testing we may assume that we are given a correct test oracle. This oracle can also serve as a somewhat non-algorithmic, formal specification which can be used as the basis for formal software development.

Let S_0 denote such a formal specification, S_n denote an implementation of that specification, and $\sqsubseteq$ denote a correctness relation (which is transitive and reflexive). In theory then, if we can show that

$$S_0 \sqsubseteq S_n$$

(automated) black box testing would not be necessary. That is, showing that the implementation S_n is correct with respect to the specification S_0 would, by definition of cor-

---

1. It should be noted that if the test oracle is itself incorrect then the value of oracle-based testing is questionable at best.

rectness, correspond to exhaustive black box testing using the test oracle. Note that we might however, still want to perform testing to **validate** S_0.

Showing that the relation $S_0 \sqsubseteq S_n$ holds is more commonly referred to as *program verification*, *formal verification*, or simply *verification*. In theory, verification works. In practice, however, the calculations needed to directly show that the relation $S_0 \sqsubseteq S_n$ holds are most often overwhelming. Informally, the difficulties encountered here result from the fact that a large part of the verification process is concerned with implementation details and how they interact to solve the desired problem.

Due to the difficulties encountered in directly verifying that a program satisfies a formal specification, a paradigm for obtaining programs from formal specifications is being explored in which the gap between formal specifications and programs is bridged through a sequence of small 'steps' or changes. These steps are traditionally called *transformations*, and their aggregation is called a *transformation sequence*.

Through a transformation sequence one can transform a specification into an implementation via a sequence of (hopefully small) transformations. This process yields a number of intermediate representations of S_0. More specifically, if n transformation steps are performed then we will have the representations: $S_0, S_1, S_2, \ldots, S_n$. Given two representations S_i and S_j in this sequence, it will generally be the case that when $i < j$, S_i will be a representation that 'looks' a little more like the initial specification, S_0, while S_j will be a representation that 'looks' a little more like the final implementation, S_n.

Intuitively, the motivation for having small transformations is that as S_i and S_{i+1} become increasingly similar to one another, $S_0 \sqsubseteq S_n$ should become easier to demonstrate.[1]

And finally, since $\sqsubseteq$ is transitive, we can calculate $S_0 \sqsubseteq S_n$ by showing that

$$\forall i: 0 \leq i < n \rightarrow S_{i-1} \sqsubseteq S_i$$

holds. In this case, we say that the transformation sequence

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$$

is *correctness preserving*.

Under the right circumstances and with careful planning, calculating that a transformation sequence is correctness preserving is significantly easier (to the point of being practical) than a direct calculation of $S_0 \sqsubseteq S_n$. Thus, when handled properly, the approach to program verification offered by transformation can make a substantial contribution towards the construction of high assurance software. For a nontrivial example demonstrating the benefits of a transformation-based approach to software construction see [4].

---

1. Consider the case in the limit where we want to show that $S_i \sqsubseteq S_i$

2.1 An Introduction to HATS

The High Integrity Software program is developing a tool called the High Assurance Transformation System (HATS) to explore the potential of transformation-based software development. Figure 1 gives an overview of HATS.

HATS is an SDT-based transformation system that can be adapted to a problem domain in the following manner:

- specify the tokens of the target language,
- provide a grammar of the target language, and
- specify how target program strings should be formatted (e.g., indentation, carriage returns, etc.).

The results of these activities are represented by shadowed boxes in Figure 1.

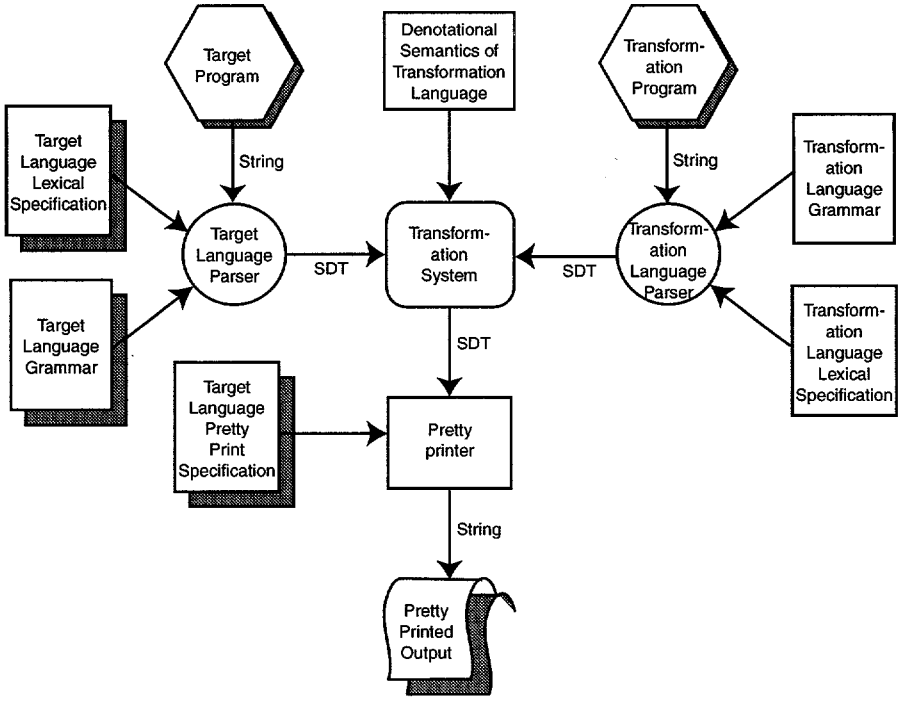

Figure 1 The HATS Architecture

After HATS has been adapted to a problem domain, it can be used as an automatic transformation system by supplying it with (1) a specific target program as input, and (2) a transformation program that describes how the target program should be transformed.

As a first step in the execution of HATS, the transformation program is parsed. During this parse, HATS also checks that all transformations will only produce syntactically valid target programs. After this has been completed, the denotational semantics of the transformation language is used to execute the transformation program.

A transformation program will contain the following sections:

- an input expression that names a specific target program (i.e., a pathname to a file)
- a section where *transform functions* are defined (see *Matching and Control in HATS* on page 7 for a discussion of transform functions)
- a section where transformation sequences are defined (i.e., where transform functions are composed)
- a control section that defines how transformation sequences are applied to the target program

When the execution of the transformation program encounters the 'input target program' expression, a recursive-descent parser is invoked to parse the target program. Because of the desire to (effortlessly) apply to a wide range of target languages, this parser is extended with backtracking capabilities to enable it to resolve local ambiguities that may arise in the parse table. As long as the target grammar is not truly ambiguous, target programs will be able to be parsed. However, it should be noted that in the worst case (e.g., if many parse table locations have multiple entries) the parsing process will be time consuming [1].

2.2 Schemas

Given a target grammar, $\mathcal{G}$, (SDT) schemas are defined as follows:

- Select a nonterminal symbol, *d,* belonging to $\mathcal{G}$. The symbol *d* will be the root of the SDT that HATS is constructing. In this context, we refer to *d* as the *dominating symbol* of the schema.
- Construct a derivation of the form: $d \stackrel{*}{\Rightarrow} \alpha$. Note that α may contain nonterminal symbols.
- The expression $d[:]\alpha'[:]$ is a schema describing an SDT whose root is *d* and whose leaves are α. Here the string '[:]' serves as a begin-end marker. The difference between α and α' is as follows: Let *C* denote an arbitrary nonterminal symbol in α. The distinction between α and α' is that any nonterminal symbol such as, *C*, in α will be represented by a schema-variable of the form $<C>_i$, where *i* is a non-negative integer, in α'. Note that subscripts are used to distinguish instances of schema-variables.

Through recognition of the begin-end markers and by making the assumption that the string $<C>_i$ will not be a legal token in the target language, HATS is able to parse α' to ensure that the derivation $d \stackrel{*}{\Rightarrow} \alpha$ is possible[2]. This assures that schemas are syntactically correct. Furthermore, HATS also requires that tree substitutions made during the transformation process always replace (substitute) trees having the same *dominating symbol.* This results in transformations that, by definition, produce syntactically legal (though not necessarily semantically legal) programs.

---

1. Exponential with respect to the number of productions in the target grammar.

2. Note that since transformations are defined in the transformation program, the target parser is invoked during the parse of the transformation program.

2.3 Matching and Control in HATS

Transformations have traditionally had the form of rewrite rules such as:

$$\mathcal{T} \stackrel{\text{def}}{=} (schema_1 \Rightarrow schema_2)$$

in which the notion of a match is implicit in the transformation, and the notion of control is external to the transformation. For example, to what SDT's within the target program does one attempt to apply $\mathcal{T}$? Historically, this lack of explicit control was not an issue because rewrites were applied manually (e.g., mathematical expressions were simplified by hand). However, in the context of automatic transformation, such rewrite rules lack expressive power. When limited to basic rewrite rules it becomes difficult to express, within a transformation program, a refined application strategy.

We have addressed this problem in HATS by explicitly parameterizing rewrites with respect to the SDT that they are applied to. This is a distinguishing feature of HATS that we believe is extremely powerful. We call these parameterized transformations *transform functions* to distinguish them from the standard unparamterized transformations. Along with the notion of explicit parameterization arises the need for an explicit match operator. In HATS, the explicit match operator is denoted by the symbol: |=| . A basic match is then a boolean-valued expression of the form e_1 |=| e_2 where e_1 and e_2 are schemas, or variables that are or can be bound to schemas. We have also found it useful to introduce a special universal SDT that denotes the empty match. In this paper we denote this SDT by ϕ, and refer to it as the *null tree.* This SDT is unique (and universal) because it cannot match with anything else (including itself). An interesting property of ϕ is that because it cannot participate in a successful match it can be used to terminate a recursive transformation. Abstractly, ϕ is the intersection between the control domain and the SDT value domain.

Using the ideas and notation just described, the transformation given earlier would be expressed as the following transform function:

$$\mathcal{T} \stackrel{\text{def}}{=} (\lambda \textbf{sdt.sdt} \text{ |=| } schema_1 \Rightarrow schema_2)$$

Thus the *pattern* portion, the expression to the left of the ⇒, of the transform function is a boolean expression, whose successful evaluation produces an environment (a generalized substitution list) which is then used to instantiate the *replacement* (i.e., the expression to the right of the ⇒. When viewed from this perspective it becomes natural to consider further extending the pattern expression to include more general boolean expressions containing more general matches. Another extension is to support a Dijkstra-like guarded command construct.

One particularly powerful idea comes from realizing that (1) transform functions produce SDT's as outputs, and (2) match operations bind variables to SDT's. After seeing this connection, it is natural to consider applying transform functions to variables that are bound in match operations and then matching the resulting SDT to a particular schema. In this manner, very refined control can be expressed within a *pattern* expression. From here on out we will refer to patterns that contain this type of control as

control-patterns.

With this capability, a transformation system achieves theorem prover-like characteristics. Transformation sequences can be viewed as focused search strategies, and *control-patterns* can be seen as proving lemmas and providing the transform function with the resulting information.

2.4 Execution of HATS

Since HATS is based on an SDT transformation system, a target program must first be parsed and converted into an SDT before transformation can begin. Because of the desire to (effortlessly) parse programs defined in terms of a wide variety of target languages, the HATS target parser has been extended with a backtracking capability that enable it to resolve local ambiguities which arise in the resulting parsing tables. The traditional methods for resolving parse table ambiguities has been to:

1. modify the parse table manually, or
2. provide the parser with sufficient lookahead capabilities to enable it to automatically resolve the ambiguity.

However, in the context of program transformation, the problem with requiring manually resolving ambiguities is that it is unreasonable to require the user to posses a sufficient level of knowledge of parsing theory to be capable of solving the problem. Additionally, the problem with the automatic resolution which depends on multiple lookaheads, is that the number of lookaheads must be bounded. As such, the parsing algorithm in HATS is extended with a backtracking function that is powerful enough to parse target programs in all cases where the target grammar is not truly ambiguous. In the worse case, such as when many parse table entries have multiple entries, a consequence of the approach chosen for HATS is that the parsing process can be quite time consuming.[1] None-the-less, we have made a relatively weak assumption that, in general, the target grammar will be 'well behaved.' In those instances where the grammar requires an unreasonable amount of time to parse, a parsing expert may need to be consulted to modify the grammar to make it more amenable to parsing.

3 Transformation and the Runtime Environment

In the previous section we outline the technique for utilizing transformation for establishing that:

1. the initial set of requirements are correct, and
2. given a complete and correct specification, that they are implemented correctly.

Software developed with these properties can reasonably expected to be *reliable*, one of the criteria for HIS. We now turn our attention to consideration of how transformation, or more specifically, rewriting technology is crucial for establishing the *safety* and the *security* of a system, the other two criteria for HIS.

---

1. The time to parse a target program will grow exponentially with respect to the number of productions in the target grammar.

We will consider three aspects of how transformation benefits the software engineer when designing HIS, particularly from a real-time, or runtime perspective:

1. The ability to compactly represent an algorithm that involves, among other things, context sensitivity and/or ambiguity, features of many of the systems we have discovered that are considered problematic.
2. Forcing the engineer to predetermine the valid state transitions provides an extremely important invariant for detecting 'abnormal environments' or 'hostile actions.'
3. The factoring of the problem into a set of tools, which are not deployed, and likely useful for more than one problem, and a runtime module, and one or more tables, or statically defined data elements.

The section concludes with comments relating these technical issues to the establishment of safety and security.

3.1 Transformation as a compact representation

The examples below will establish the need for two important features that a runtime transformation capability should possess:

1. The ability to handle more complex grammars than typically handled by parsers available to software engineers.
2. The ability to easily incorporate domain specific properties of the problem.

A parser with these features will likely require small changes for application to most problems, a characteristic identified in the last section as being desirable for promoting a correct adaptation. Further, the characteristic of requiring small changes will also facilitate the introduction of domain specific features, or the creation of custom tools that greatly reduce the scale of the problem. This also will tend to promote the creation of HIS.

3.1.1 Context Sensitivity

Context sensitivity involves the interpretation of data in which the interpretation of one segment of data is dependent upon the interpretation of other 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.

The importance of context sensitive algorithms to researchers is that these have emerged as a source of subtle errors which are difficult to identify and fix. The solution to the problem 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 component of a well-engineered solution. The basic scenario is simple. A data stream includes the following structure:

```
[startVal][count][time][frame1][frame2]...[frame(count)][endVal]
```

Although the grammar is fairly 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 implementation, the parser had the following fragment of BNF:

```
frames     ::= [startVal]frameList [endVal]
frameList  ::= [startVal][count][time]basicFrame  |
               frameList basicFrame
basicFrame ::=
```

The source of errors with this architecture is that the count is basically ignored. Further, the actions for the first trigger are different than for the other triggers. The next fragment attempts to include somewhat more context sensitivity, but still tries to do so with a context free grammar.

```
frames     ::= frameList
frameList  ::= [startVal][1][time]basicFrame  |
               [startVal][2][time]basicFrame basicFrame[endVal]
              ...
basicFrame ::=
```

Although there is now a special case for each of the trigger counts, the actions for processing the frames is replicated

$$7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$$

times. 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 of one of the frames not being processed correctly. The alternative we chose to implement is an extension of the standard context sensitive grammar (where standard here refers to Type 1 in the Chomsky Hierarchy of Grammars [4], 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). 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.

```
frame ::= [time]basicFrame | ...
   {processing action for the frame}
[time]basicFrame ::= [startVal][1][time]basicFrame[endbyte]
[time]basicFrame[startVal][1][time] ::= [startVal][2][time]basicFrame
[time]basicFrame[startVal][2][time] ::= [startVal][3][time]basicFrame
[time]basicFrame[startVal][3][time] ::= [startVal][4][time]basicFrame
[time]basicFrame[startVal][4][time] ::= [startVal][5][time]basicFrame
[time]basicFrame[startVal][5][time] ::= [startVal][6][time]basicFrame
[time]basicFrame[startVal][6][time] ::= [startVal][7][time]basicFra
basicFrame ::=
```

In short, for each production

$$A \rightarrow B$$

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

$$C \rightarrow D$$

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 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.

3.1.2 Ambiguity

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 has 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 have altered the behavior of subsequent processing steps. In this event, assumptions must be made about the lost data so processing can proceed.
3. 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.

More examples are cited in [9]. 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 in the work

by deKleer [6]. Unfortunately, this work is abstract, and the implementations of the work 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).

3.2 Safety Derived from State-driven Design

For the purposes of this discussion, assume the behavior of a complex software system is described by some formal representation of a collection of finite state machines. These representations capture the intended and hence acceptable states the system may be in and transitions governing movement between all the intended states.[1] From the perspective of establishing the safety of a system, one views the departure from this set of states or improper transitions between states as potential hazards that could lead to unacceptable consequences. A subset of the universe of all possible unacceptable states can be described as serious hazards that will lead to compromise of safety. Safety in this context means a lack of possibility for the loss of life, damage to the environment, or other economic loss. Establishing such a state can be precipitated by several events. Those include human errors related to incomplete specifications, improper implementation, and insufficient testing. In addition, dependencies on tools such as compilers can also lead to the creation of a system that does not faithfully adhere to the abstract behavioral models. In addition to the domain of defects related to the human participation in the creation of software, the failures related to the underlying physical hardware contribute a completely disjoint set of possible system failures. That is, random failures of microelectronic devices upon which the software is dependent, can and will cause the execution sequence of software to be violated and hence increasing the likelihood of arriving at an unacceptable state.

A hallmark of an architecture based on transformation (and thus parser-driven) design is the intrinsic ability to detect abnormal situations. It has been proven that the class of parsers known as LR(k) have the properties that they will detect any anomolous input [7], and further they will detect the error as soon as it is possible to do so [8]. This is an extremely elegant manner of achieving a high degree of system safety. The ability to be confident of what the system behavior will be in the presence of unanticipated circumstances is extremely important when the consequence of an erroneous step could be immense. The key to encouraging complex state-driven design is to make available technology that elegantly represents the complete set of valid states, and then operating in a mode where everything else is considered an anomalous situation.

---

1. Keep in mind that the actual system itself is only a replica of what we hoped would be developed based on abstractions of behavior.

3.3 Security from Factoring Components into Tools, Tables, and Runtime Modules

Establishing the security of a system relates to the notion of demonstrating the inability of intentional or malevolent diversion of a system for the purpose of gain by an adversary, or other agent that does not have authorized access to the system. That gain can take forms ranging from financial reward to damage resulting from terrorist activities. The objective of the adversary is to induce an unacceptable state, such creating a sequence of actions that result in an Automated Teller Machine emitting money either from an unauthorized account, or from no account at all. In the case of a weapon, the goal is to be able to assert control of the system, e.g. causing detonation. The successful adversary usually depends on the acquisition of substantial knowledge of both software and hardware. With this knowledge, the adversary will then attempt to identify and exploit a vulnerability present in either the construction of the system (as delineated in section 3.2) or from the systems vulnerability to external environmental stimuli. This might take the form of physical stimuli such as extreme temperatures which could precipitate physical failures, or an attempt to operate the system out of specified limits, e.g. providing digital input that the system is not prepared to handle properly.

A system architecture based on parser technology will have, typically, three types of components. We will call these components tools, tables and runtime modules. (We will ignore the symantic components for the moment). A tool is a non-deployed component that simply transforms one form of information into another. The parser generation technology we have been discussing is a good example of such a component. The table output from a parser generator is another form of component that becomes extremely important in an architecture defined in terms of transformation. The table, though static, embodies a considerable amount of information, and context defining the behavior of the system. Finally, the runtime modules, are codes invariant to the instance, that usually depend on tables and other context to operate correctly. We draw the distinction between a 'normal' library component, that does not depend on much context other than the inputs to operate correctly. Further, the runtime portion of a parser generator has many of the properties we believe are salient: it is invariant to the instance, but bound with a parsing table output from the parser generator (the tool), it is capable of both monitoring and controlling the behavior of a component.

There are several important benefits from decomposing a system into tools, tables and runtime modules: first, the tools can be built once, verified, then reused. Even if a particular domain specific attribute is added to the tool, it will likely be a small transformation (or sequence of transformations). Second, the same holds true for the runtime module related to the tool. Third, the table can be more easily verified with other tools and inspection than a large code component. Finally, the scale of all three components (from our experience) is of a considerably smaller scale than the monotonic code component. This makes the quantification of the high integrity considerably easier, or even possible. Thus, from the standpoint of security, the system designed around transformation, will have fewer vulnerabilities to exploit than a system based on more traditional software engineering approaches. In particular this will be true of systems generated from tools that were generated from domain specific tools. If the tools are properly protected, there is virtually no opportunity for an adversary to gain sufficient knowledge to

identify, let alone, exploit a vulnerability. Further, the likelihood of even finding a significant vulnerability is greatly reduced.

4 Conclusion

We have discussed two aspects of creating high integrity software that greatly benefit from the availability of transformation technology, which in this case is manifest by the requirement for a sophisticated backtracking parser. First, because of the potential for correctly manipulating programs via small changes, an automated non-procedural transformation system can be a valuable tool for constructing high assurance software. Second, modeling the processing of translating data into information as a, perhaps, context-dependent grammar leads to an efficient, compact implementation.

From a practical perspective, the transformation process should begin in the domain language in which a problem is initially expressed. Thus in order for a transformation system to be practical it must be flexible with respect to domain-specific languages. We have argued that transformation applied to specification results in a highly reliable system. We also attempted to briefly demonstrate that transformation technology applied to the runtime environment will result in a safe and secure system. We thus believe that the sophisticated multi-lookahead backtracking parsing technology is central to the task of being in a position to demonstrate the existence of HIS.

5 Acknowledgement

This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

References

[1] Winter, Victor, L., *Software Construction via Abstraction, Synthesis, and Transformation,* Proceedings of the High Integrity Software Conference, IEEE, 1997.

[2] J. Rushby. *Formal Methods and their Role in the Certification of Critical Systems.* Technical Report CSL-95-1, SRI International.

[3] C. M. Holloway. *Why Engineers Should Consider Formal Methods.* Proceedings of the 16th Digital Avionics Systems Conference, October 1997.

[4] S. Stepney. *High Integrity Compilation: A Case Study.* Prentice Hall, 1993.

[5] Cohen, Daniel, I.A., *Introduction to Computer Theory,* Wiley, 1991, 743-754.

[6] DeKleer, Johan, *An Assumption-based Truth Maintenance System,* Artificial Intelligence, **28**(1986) 127-162.

[7] Sippu, S. and Soisalon-Soininen, E., *Parsing Theory, Volume II,* Springer-Verlag, 53

[8] Aho, Alfred V., Sethi, R. and Ulman, J., *Compilers (Principles, Techniques and Tools),* Addison-Wesley, 1985, 215

[9] Bespalko, S. J., Sindt, A., *Context Sensitivity and Ambiguity in Component-based Systems Design,* Proceedings from the Foundations of Component-based Systems Workshop, In conjunction with the ESEC '97, 1997.