Property Specification Language and Mining* Specifications

(CS615 Term Paper)

Tejaswi N (04329016),
KReSIT, IIT Bombay

Instructor: Prof. Supratik Chakraborty

November 28, 2004

Abstract

Formal methods increase a system's reliability to a very large extent. It has still not found wide spread acceptance due to its complex and hard-to-implement nature. The necessity of charting thorough and correct specifications of systems is one major deterrent in adopting formal methods. In this report we study two ideas that attempt to reduce this complexity involved in coding system specifications. Property Specification Language (PSL) is a language used in formal methods to specify logic designs. PSL adds syntactic sugar and regular expressions to logic representation systems to make them more intuitive and easy to use. The second part of the report covers the novel concept of "mining" specifications from large software applications. A machine learning based approach to discover formal specifications in the interaction of software systems with certain APIs is studied.

1 Introduction

Specifications are an integral part of the formal verification of systems. For any verification technique to proceed, thorough formal specifications need to be laid down. The problem that people mostly face during this stage of the verifications process is the complexity involved in designing and formalizing non-trivial specifications. Intuitively we see that there are two aspects to this problem. The first is the difficulty that is inherent in the representation of specifications. In the first part of the report, we study techniques that help in formalizing specifications easily. Specifications like temporal logic statements become very complex for reasonably large systems. These logical specifications are more like atomic elements which need to be strung together precisely to form non-trivial specifications. High level languages like PSL/Sugar provide constructs like regular expressions that help write specifications easily. The second aspect of the specifications complexity problem is its design itself. Semi-automatic ways of coming up with specifications so that a human expert can later partially correct them or design additional specifications on top of them will make formal verification more widely accepted. In the second half of this report we study a machine learning driven approach that attempts to *mine* specifications from software systems. A properly tested code base is used to train the system so that similar (extended) code chunks can be tested for conformance with the learnt specifications.

2 PSL/Sugar

PSL has its roots in Sugar, a simple concise and expressive language developed by IBM. The Sugar language was submitted to the Accelera EDA standards organization, who selected Sugar as the basis for an IEEE international standard and renamed it to PSL and is now commonly referred to as PSL/Sugar. Sugar is a formal specification language for hardware. A hardware specification written in Sugar can be used by a formal verifications tool, such as a model checker. The Sugar specification can automatically be compiled into one of the standard standard temporal logics: LTL or CTL. These temporal logics,

*Mining used as verb (not as gerund).

while designed to have efficient model checking algorithms have the disadvantage that many specification elements are very difficult to code. Sugar was designed to solve this problem without compromising the integrity or efficiency of the temporal logics. Sugar adds the power of Regular Expressions and an extensive set of operators that provide *syntactic sugar* to help construct more intuitive and concise Sugar formulas instead of long and cumbersome temporal logic formulas. It is to be noted that these extra features do *not* give additional expressive power to Sugar.

Sugar specifications are made up of four layers: Boolean, Temporal, Verification and Modeling. The boolean layer comprises the most basic elements of a hardware design: the boolean values. A high is treated as boolean *true* and a low is treated as boolean *false*. Temporal layer has the temporal properties which describe the relationship between boolean expressions over time. For example: **always**$((\alpha) \rightarrow$ **next**$(\beta))$ is a temporal property expressing that whenever signal α is asserted, then $(\rightarrow)$ in the next cycle signal (β) is asserted. Verification layer consists of directives which describe how the temporal properties should be used by verification tools. For example: **assert**(α) tells a verification tool that the property (α) should always hold. Verification layer allows *verification* units, or groups of Sugar statements to be constructed. Modeling layer provides means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. It also is used to give names to entities from the temporal layer. In this report we will study some introductory aspects of the temporal layer. For a full definition of the language, formal semantics, and more examples, refer to the complete language definition at [3].

2.1 Temporal Layer

The temporal layer represents relationships between boolean variables over time. Every Sugar property starts reasoning at the start of a sequence and from there *only* temporal operators can move it forward. This is best understood by looking at some of the operators that are provided by Sugar in this layer.

2.1.1 Some Temporal Operators

always: The basic temporal property of having a boolean expression hold all the time is captured by the **always** operator. Example: If a boolean expression α holds on some initial clock cycle, **always**(α) means that α holds at every clock cycle.

never: As per normal intuition, the **never** operator indicates conditions that never hold. **never**(α) means that α never holds.

next: The **next** operator is used to specify conditions that hold in the immediately following clock cycle. If **next**(α) holds in a clock cycle, then α holds in the next clock cycle. Sugar also allows **next[n]** operations. This means that if **next[5]**(α) holds in this clock cycle, α holds after 5 clock cycles.

until, **until_**, **until!** and **until_!**: (α)**until**(β) holds on a clock cycle iff α holds until β holds. β might immediately follow α, or might come after an indefinite number of clock cycles. After β holds, α need not hold anymore. **until_** operator insists that α hold in the state in which β holds. The plain **until** and **until_** operators do not mandate that β should happen. These are mandated by the **until!** and **until_!** operators which are otherwise similar. The use of exclamation is referred to as *strong* sense of an operator.

eventually!: The **next**(α) operator insisted that α hold in the next cycle. **eventually!**(α) insists that α hold after some arbitrary number of cycles. The strong sense here means that α has to hold after sometime.

before, **before_**, **before!** and **before_!**: (α)**before**(β) holds in a cycle iff in the future α holds in some cycle earlier than where β holds. In order to specify that α can happen before or in the same cycle as β, **before_** is used. As expected, both these are used in a weak sense, ie. it is not mandated that α is not required to occur. The strict sense operators need to be used to enforce the occurrence of α before β.

next_event and **next_event!**: If **next_event**$(\alpha)(Condition)$ holds, then in the future, whenever the α holds next, *Condition* should hold. In the weak sense, α is eventually not mandated to hold; but in the strong sense, α has to hold eventually and whenever that happens, in the same state, *Condition* is also supposed to hold. If **next_event**(α)**[n]**$(Condition)$ holds in this state, then, in the future, the state in which nth occurrence of α happens, *Condition* must hold.

abort: This is not a temporal operator in itself, but helps write better temporal expressions. (T)**abort**(α) means that α aborts any check for assertions that the temporal formula T might have

enforced. In other words, if T contains say a **next** or an **eventually**, the verifier looks for their operand to hold in the future. If α is seen before that operand, the check for that operand is aborted.

2.1.2 Sugar Extended Regular Expressions (SEREs)

Sugar provides a way to extend the conventional notion of regular expressions to what are called Sugar Extended Regular Expressions (SEREs). SEREs provide an alternate way to reason about sequences of expressions which are more concise and easy to understand. The basic SEREs are built using a series of expressions, separated by semicolons. Semicolons are integral to SEREs. For example: the SERE $\{\alpha; \overline{\beta}; \gamma\}$ describes a sequence in which α is asserted in the first cycle, it is asserted that β is not true in the next cycle, followed by an assertion for γ. Note that SEREs are enclosed in curly parenthesis as opposed to normal boolean expressions which are enclosed in regular parenthesis. Formally:

Every boolean expression is a SERE. If α and β are SEREs, then $\{\alpha\}$, $\alpha;\beta$, $\alpha \sim \beta$, $\alpha||\beta$, $\alpha\&\&\beta$ and $\alpha[*]$ are SEREs representing grouping, concatenation, overlapping concatenation (last state of α coincides with the first state of β), disjunction, conjunction and indefinite repetition respectively.

Two SEREs can be linked to form Sugar formulas of the linear fragment:

Strong suffix implication $\{\alpha\} \mapsto \{\beta\}!$

Weak suffix implication $\{\alpha\} \mapsto \{\beta\}$

Strong suffix implications are liveliness formulas, stressing on some end condition always holding true. They indicate the a sequence of states in which α holds must be followed by a sequence of states in which β holds. Weak suffix implications, on the other hand, are safety formulas indicating that a sequence of states in which α holds need not be followed by a sequence of states that contradict β. Weak formulas, like in temporal formulas do not mandate that β eventually hold.

These implications are used to define other complex operators like **within!**$(\alpha, \beta)\{\gamma\}$ which indicate that γ must occur after α is asserted, and before β is asserted. Using temporal operators and SEREs, it is defined as:

$$\textbf{within!}(\alpha, \beta)\{\gamma\} == \alpha \mapsto \{\gamma \;\&\&\; \{\overline{\beta}[*]\};\ \overline{\beta}[*],\ \beta\}!$$

2.1.3 Examples

Here we see a few case studies where some temporal operators and SEREs are used to express temporal logical conditions.

1. $\{[*]; \alpha; \beta\} \mapsto \{\gamma[*]; \delta\}!$ - This means that every sequence of two states with α holding first followed by β, need to be followed by a set of sequences in which γ holds an arbitrary number of times followed by D.

2. **always** $(\alpha \rightarrow$ **next** $\beta)$ - This means that in all if this property holds true in some state, then from that state, in the future, if α holds in any state, β must hold in the next state.

3. **always** $(\alpha \rightarrow$ **next** $(\beta$ **before** $\gamma))$ - If this condition holds true in a state, then from that state, in the future, if α holds in any state, after that, β must be seen after α before γ.

2.2 Applications and Further Research

Verification of Sugar properties is primarily intended to be by model checking (for infinite paths) and simulation (for finite paths). Model checkers and Simulators are the primary consumers of Sugar-written specifications. Sugar specifications are also used in automatic theorem proving.

In spite of all the comfort and ease Sugar provides while charting specifications, design experts who are novices in formal theory find it considerably hard to translate natural language to Sugar statements. This area of using techniques from natural language processing and formal methods is a new area of research which can be explored.

3 Mining Specifications

Formal Verification methods explore all possible paths of execution of a system and check for conformance to predefined specifications.System designers are reluctant to adopt formal methods mostly due to the complexity involved in charting formal specifications that verification methods use. This is especially

true in software systems because the penalty for failure, though prohibitive, is not fatal. Semi-automated methods of designing specifications is an alternative that is promising under such circumstances. In this report we study a machine learning approach to discover formal specifications of the protocols that code must obey when interacting with a third party entity like an Application Program Interface (API) or an Abstract Data Type (ADT). This method is due to [1].

More precisely, this approach attempts to discover some temporal relationships and data dependencies that a program follows while interacting with such APIs and ADTs. These interactions from a thoroughly tested code base are observed and recorded; they are later used to infer more general rules about how these interactions should proceed. The underlying assumption is that the tested code base is correct. The problem is then reduces to that pf probabilistic learning from execution traces of the tested code. We further see how the problem of extracting specifications from execution traces reduces to learning regular languages, for which off-the-shelf learners are available. The learnt rules are summarized as state machines which can be used by verification tools to identify bugs in future interactions that are coded into the system.

3.1 Problem Definition

At a very high level, the specifications mining problem is to construct an automaton that extracts the set $C \subseteq I$, where C is the correct set of traces from all set of traces I, given a training set of traces T. [1] observes that this problem is undecidable as there are no restrictions on C or T. For this problem to be solvable, C needs to be recursively enumerable. Additionally, as finite state automaton learners are available, and verifications tools require finite state specifications, C needs to be a regular language as well. Having C regular makes it very strict because most execution traces are not regular. Therefore, traces need to be converted to regular interaction scenarios. Interaction scenarios are made regular by allowing them to manipulate no more than k data objects. The second problem is that as there are no restrictions on the training set T; any correct set of scenarios C needs to be learnt from *any* training set of scenarios T. Under such conditions correct specifications cannot be chosen. [1] remedies this by restricting T to contain only elements from C.

The formal problem definition: Let I be the set of all interaction scenarios with an API that manipulate no more than k data objects. Let $C \subseteq I$ be the regular set of all such correct scenarios. Let $T = c_0, c_1, \ldots$ be an infinite sequence of elements from C in which each element of C occurs at least once. For $n > 0$ examine the first n elements of T and produce a finite state automata A_n, such that the sequence of finite-state automaton $A_0, A_1, \ldots$ has the following property: for some $N \geq 0$, A_N generates exactly the scenarios in C and $A_n = A_N$ for all $n \geq N$. We say that the sequence $A_0, A_1, \ldots$ identifies C in the limiting sense.

It has been proven that learning regular languages in the limiting sense is undecidable [2]. The learner's dilemma is that any finite sequence of examples from the infinite language could also be from a finite subset of the infinite language. The learner has no way of choosing one over the other. In this case, C can be an infinite regular language and thus, cannot be learnt. To remedy this, the learner is provided with a set of examples generated according to a probability distribution. Now, the task of the learner is to learn a close approximation of this distribution. $\overline{P}$ is an $\epsilon - good$ approximation of distribution P if $D(P, \overline{P}) \leq \epsilon$, for some distance function D. Here, we choose probability distributions that generated by Probabilistic Finite State Automata (PFSA). A PFSA is similar to an NFA, but has a probability associated with each transition. It generates a string with some probability that take into account probabilities associated with all the transitions needed to generate the strings. We now reduce the specifications mining problem to that of learning probabilistic finite automata.

The final formal problem definition: Let I be the set of all interaction scenarios with an API that manipulate no more than k data objects. Let M be the *correct* target PFSA and P^M be the distribution over I that M generates. As M is the *correct* PFSA, it generates correct traces with high probability and buggy traces with low probability. Given a confidence parameter $\delta > 0$ and an approximation parameter $\epsilon > 0$, find with probability at least $1 - \delta$, a PFSA $\overline{M}$ such that its distribution $P^{\overline{M}}$ is an $\epsilon - good$ approximation of P^M for some reasonable distance metric D.

3.2 Mechanics

The three stages involved in building the specifications automata are:

1. **Tracing and Flow Dependence annotation**: Interaction elements are extracted out of the training traces in this stage. These interactions have a name (say a function name) and a set of attributes (the function's parameters). These are then annotated with flow dependency tags to indicate operational order. To do this, initially, a human expert divides the attributes into those that define underlying objects, and those that use underlying objects. For example, a *socket.bind* operator defines an object and a *socket.read* operator uses an object. Given such a list of attributes which define and use objects, deducing the flow dependence between interactions is equivalent to solving the Reaching Definitions Problem. A dynamic programming algorithm coupled with Tarjan's union find algorithm is used to tag each interaction so that flow dependency with respect to definitions and usage of objects is available later.

2. **Scenario Extraction**: Scenarios are extracted out of interactions in this stage. Scenarios are flow dependent interactions. A human expert chooses types of scenarios that are to be extracted. These types are identified by a seed interaction element on which the scenarios are extracted on. A graph is constructed out of the flow dependencies of interactions. This graph, coupled with the seed element is used to construct an entire scenario by traversing the graph starting at the seed. The size of the scenario (maximum distance between the seed and the farthest node traversed) is restrained by some external constant. Flow dependent interactions around the seed are scenarios. These interactions that makeup scenarios are given logical names from some alphabet so that each scenario might be treated as an ordered string. A minor glitch is that: given a seed, logically similar, but physically different scenarios might be selected by the graph traversal algorithm. This might happen if the flow dependency matrix is sparse. To eliminate this logical redundancy, a standardization step is performed on the extracted scenario strings. In principle, this standardization step compares $dependency - preserving$ permutations of scenario strings to extract logically equivalent scenarios. At the end of this stage a set of scenario strings pertaining to the training traces are available for the next stage.

3. **Automaton Learning**: The training scenario strings are given to an off-the-shelf PFSA learner [4]. This learner generates a PFSA that accepts a superset of the training strings. Spurious transitions in the generated PFSA are trimmed by using some cut off on the probability on transitions. Finally, we have the NFA that models the correct specifications as per the training traces and can be validated by a human expert, and used later during the verification stage.

3.3 Verification

The NFA generated by the automaton learner is a specification and arbitrary traces are verified for conformance with this specification. All possible interaction seeds are extracted from a test trace, and scenarios are constructed out of these seeds. These scenario strings are tested against the generated NFA to see whether the scenario conforms to the specification. If all the scenarios generated by all the seeds of the test trace conform to the NFA specification, then the test trace is said to satisfy the specification. Additionally, the training set of the learner can be expanded by adding validated test strings to it. This expands the "coverage" of the specification.

3.4 Applications and Further Research

As indicated in section 3.3, the specifications discovered by this learning approach are used primarily to aid verification tools. Human experts might also gain some valuable insight by these discovered specifications: this can happen when the system is coded in a way it was not designed for, or in a way contrary to its original purpose; both of which a human expert can assess by checking the discovered specifications against requirements and expert intuition. It was seen in section 3.2 that a human expert needs to intervene a few times to set up a few basic parameters. Further research in this area can automate this part by exploring generic interactions between objects and ways to deduce relationships between them. Also, instead of off-the-shelf learners, dedicated learners can be used, and this might help in bringing some domain knowledge into the learning process.

4 Conclusion

Discovery of specifications and their concise representation (PSL/Sugar) help in wide spread adaptation of formal verification techniques like model checking. These in turn, increase the reliability of systems to a significantly higher extent; even making them inevitable in some cases.

References

[1] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on the Prinicples of Programming Languages (POPL '02)*, pages 4–16, 2002.

[2] Mark E. Gold. Language identification to the limit. *Information and Control*, 1967.

[3] PSL/Sugar. Literature. http://www.haifa.il.ibm.com/projects/verification/sugar/literature.html.

[4] Anand V. Raman and Jon D. Patrick. The sk-strings method for inferring probabilistic finite state automata. In *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on Machine Learning (ICML97)*, 1997.