Formal Parsing Analysis of Context-Free Grammar using Left Most Derivations

Khalid A. Buragga
College of Computer Sciences and I. T.
King Faisal University
Hofuf, Saudi Arabia
Email: kburagga@kfu.edu.sa

Nazir Ahmad Zafar
Department of Computer Science
King Faisal University
Hofuf, Saudi Arabia
Email: nazafar@kfu.edu.sa

***Abstract*—Formal approaches are useful to verify the properties of software and hardware systems. Formal verification of a software system targets the source program where semantics of a language has more meanings than its syntax. Therefore, program verification does not give guarantee the generated executable code is correct as described in the source program. This is because the compiler may lead to an incorrect target program due to bugs in the compiler itself. It means verification of compiler is important than verification of a source program to be compiled. In this paper, context-free grammar is linked with Z notation to be useful in the verification of a part of compiler. At first, we have defined grammar, then, language derivation procedure is described using the left most derivations. In the next, verification of a given language is described by recursive procedures. The ambiguity of a language is checked as a part of the parsing analysis. The formal specification is analyzed and validated using Z/Eves tool. Formal proofs of the models are presented using powerful techniques, that is, reduction and rewriting of the Z/Eves.**

Keywords-parsing analysis; context-free language; formal specification; Z notation; verification.

I. INTRODUCTION

Formal methods are mathematical-based approaches used for specifying, proving and verifying properties of software and hardware systems [1]. The process of formal verification means applying the mathematical techniques to verify the properties ensuring correctness of a system. Formal verification of software systems targets the source program where the semantics of the programming language gives a precise meaning to the programs to be analyzed. On the other hand, program verification does not give guarantee that the generated executable code is correct as described by the semantics of the source program. This is because the compiler may lead to an incorrect target program because of bugs in the compiler and it can invalidate the guarantees obtained by formal methods to source program. It means the verification of compiler is more important than verification of source program to be compiled.

The design, construction and exploitation of a fully verifying compiler will remain a challenge of twenty first century in the area of computer science. The main functionality of compiler is to translate a source code understandable by programmers to an executable machine code correctly and efficiently. Although compiler is a mature area of research but it needs further investigation, as mentioned above, because bugs in the compiler can lead to an incorrect machine code generated from a correct source program. That is why design and construction of a bug free compiler is an open area of research. Further, as executable code generated by the compiler is tested and if bugs are detected it might be due to the source program or compiler itself. This has led to verification of compiler that proves automatically that a source program is correct before allowing it to be run.

In this paper, parsing analyzing of language is presented using Z notation by left most derivations, which will be useful in our ongoing project on verification of compiler. Another objective of this research is linking context-free grammar with formal techniques to be useful in development of automated computerized systems. Currently, it is not possible to develop a complete software system using a single formal technique and hence integration of approaches is required. Although integration of approaches is a well-researched area [2][3][4][5][6][7][8], but there does not exist much work on formalization of context-free languages. Dong et al. [9][10] have described the integration of Object Z and timed automata. Constable has proposed a constructive formalization of some important concepts of automata using Nuprl [11][12]. A relationship is investigated between Petri-nets and Z in [13]. An integration of B and UML is presented in [14][15]. W. Wechler has introduced some algebraic structures in fuzzy automata [16]. A treatment of fuzzy automata and fuzzy language theory is discussed in [17]. Some important concepts of algebraic theory and automata are given in [18].

In [19], preliminary results of this research were presented by linking context-free grammar and Z notation. In this paper, first, formal definition of context-free grammar is given. Then a derivation procedure is described by replacing non-terminal with a string of terminal and non-terminals. The derivation procedure is extended to a sequence of derivations to derive a string form a given string using production rules of the context-free grammar. Then parsing analysis is described for word by left most derivations resulting a parsing tree. The parsing analysis for a language is specified by introducing recursion using derivations used in generation of a word. Next, ambiguity of a word is checked by specifying if there exists more than two left most derivation trees for a given words. The same concept is formalized for the language to check if it is ambiguous. The

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

formal specification is analyzed and validated using Z Eves tool set. The major objectives of this research are:

- Identifying and proposing an integration of context-free grammar and formal methods to be useful in verification of the compiler
- Providing a syntactic and semantic relationship between Z and context-free grammar
- Proposing and developing an approach for supporting an automated tools development

Rest of the paper is organized as: in Section 2, an introduction to formal methods is given. In Section 3, an overview of context-free grammar and its applications is provided. Formal construction of models of context-free grammar is given in Section 4. Formal analysis for validating the models is presented in Section 5. Finally, conclusion and future work are discussed in Section 6.

II. FORMAL METHODS

Formal methods are approaches based on mathematical techniques and notations used for describing and analyzing properties of software and hardware systems. These formal techniques are based on discrete mathematics such as sets, logic, relations, functions, graphs, automata theory and higher order logic. Formal methods may be classified in terms of property and model-oriented methods [20].

Property oriented methods are used to describe software in terms of properties, constraints or invariants that must be true. Model-oriented methods are used to construct a model of a system [21]. Formal methods are being applied successfully to improve quality by means of describing and specifying software systems in a well-precise and structured manner. Although there are various tools, techniques and notations of formal methods but at the current stage of their development, it needs an integration of formal techniques and traditional approaches for the complete design, description and construction of a system.

Z notation is a popular specification language in formal methods used at an abstract level. The Z is a model-oriented approach based on set theory and first order predicate logic [22]. Usually, it is used for specifying behavior of sequential programs of systems by abstract data types. In this paper, Z is selected to be linked with context-free language because of a natural relationship which exists between both of these approaches. The Z is based upon set theory including standard set operators, for example, union, intersection, comprehensions, Cartesian products and power sets. On the other hand, the logic of Z is formulated using first order predicate calculus. The Z is used in our research because it allows organizing a system into its smaller components known as schemas. The schema defines a way in which the state of a system can be described and modified. A promising aspect of Z is its mathematical refinement that is a verifiable stepwise transformation of an abstract specification into an executable code. Once formal specifications in Z are written, it can be refined into implemented system by a process of series of stepwise mathematical refinements.

III. APPLICATIONS OF CONTEXT-FREE GRAMMAR

The context-free grammar (CFG) is important in design and description of a programming language and its compiler. Initially, formalism of CFG was developed by Chomsky who described linguistics in a grammatical form and converted into mathematical models providing a precise and simple mechanism of description of languages. The context-free grammars allow a simple and an efficient way of parsing the algorithms. Using the grammar, it can be determined whether a particular pattern can be generated and the way of generation is also determined.

Inclusion of empty string is always required for completeness of a language. All context-free grammars cannot generate the empty string. If a grammar generates the empty string then it is needed to include some rules generating the empty string. Every context-free grammar without null production has an equivalent grammar in Chomsky Normal Form (CNF). Here by equivalence we mean that both the grammars generate the same language. The CNF grammar is important both in theoretical and practical point of view, it can be constructed from a given context-free grammar. By using CNF, it can be decided for a given string if it can be accepted in polynomial time algorithm. Context-free grammars contain both the decidable and un-decidable problems. Deciding for a grammar that it accepts the language of all the strings is an example of un-decidable problem which can be proved by reduction by linking it with the Turing machine. Deciding whether two context-free grammars describe the same language is another example of the un-decidability.

On the other hand, context-free languages have their own limitations. Some of the operators which are well-defined in many other models of automata theory do not behave well in case of the context-free grammar. For example, the intersection of two context-free languages, in general, is not context-free. Similarly, the complement of a context-free language is not context-free one. However, union, concatenation and Kleene star operators produce context-free languages when applied to it.

Context-free grammar can be applied to many areas of diversity, for example, robotics, speech recognition, software engineering, and software maintenance [23]. The applications of CFG in the area of pattern recognition increase the accuracy of patterns to be recognized. This is because it can provide a higher level of abstraction by defining the semantics of patterns as compared to its other counterparts of specification, for example, strings and regular expressions. This semantic analysis can be used to reduce the false identification of the patterns [24]. Further, the applications of pattern matching can be observed everywhere from language processing to networks.

In automatic speech recognition system, the spoken words can be generated by a context-free grammar using dynamic programming algorithms. As an example of application of CFG in the area software engineering, the components in a source code are recognized and re-generated using context-free grammar [25]. As the output of parsing are larger and less-ambiguous and have meaning of

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

the structures in a sentence, therefore, for question answering and interactive voice response systems, the use of context-free grammar can highly be effective and useful in such kind of systems and applications [26][27].

IV. FORMAL ANALYSIS OF CONTEXT-FREE GRAMMAR

Context-free grammar is a 4-tuple (V, ∑, R, S) where:

- V is a finite set of non-terminal called variables representing different types of clauses in a sentence.
- The ∑ is a finite set of terminals and final contents of a string or sentence are based on it.
- The third one R is the start variable used to represent the whole string or a sentence.
- The last one S is a relation consisting of set of all the productions or rules of the grammar.

Every production is of the form: S → t, where S is a non-terminal consisting of a single character or symbol and t is a string which may contain only terminals or non-terminals or combination of both. Further, t might me an empty string. The notations, S → t, are called productions or rules which are applied one after other producing a parse tree. The tree ends with terminals called leaves and each internal node is a non-terminal which produces one or more further nodes. The left hand side of a production rule of a context-free grammar is always a single non-terminal. Because all rules only have non-terminals on the left hand side and it can easily be replaced with the string on the right hand side of this rule.

Further the context in which the symbol occurs is therefore not important and hence the grammar is called context-free grammar. It is to be noted that context-free grammar is always recognized by finite state machines having a single infinite taps. For keeping track of nested units, the current parsing state is pushed at the start of the unit and it is recovered at the end.

In this section, formal analysis of CFG is presented using Z notation. We start with the definition of context-free grammar which is a 4-tuple as defined above. R in the tuple is a relation from V to $(V \cup \sum)^*$ such that $\exists w \in (V \cup \sum)^*$, $S \in V$ and $(S, w) \in R$. The symbol * represents to any combination of characters of V and ∑.

In the specification of CFG, we define the sets of non-terminal by V and terminal by Sigma. The set of terminals and non-terminals together denoted by *vandt* and alphabets of the grammar are of type seq X. The sequence of elements of X, seq X, denotes the set of all sequences containing terminals and non-terminals. The notation for rules is defined by the relation between V and seq X. The production rules are defined by the relation denoted by rules. Further there exists exactly one rule, (s0, w) ∈ rules where s0 is the start non-terminal and w is string s of type seq X. With these definitions, a formal definition of context-free grammar is given in terms of a schema *CFG*. The variables are given in first part and constraints are defined in the second part of schema. The V, X and *Sigma* are defined as sets at an abstract level of specification.

$[X]; V == X;$

$Sigma == X$

CFG

$variables: \mathbb{F}\ V$
$terminals: \mathbb{F}\ Sigma$
$vandt: \mathbb{F}\ X$
$rules: V \leftrightarrow \text{seq}\ X$
$s0: V$

$variables \cap terminals = \{\}$
$vandt = variables \cup terminals$
$\text{dom}\ rules \subseteq variables$
$\forall s: \text{seq}\ X \mid s \in \text{ran}\ rules \bullet \text{ran}\ s \subseteq vandt$
$s0 \in variables$
$\exists w: \text{seq}\ X \mid w \in \text{ran}\ rules \bullet (s0, w) \in rules$

Invariants:

- The terminals and non-terminals are disjoint sets.
- The entire set of alphabets is union of terminals and non-terminals.
- The domain of *rules* relation is a subset of variables.
- The set of elements in the range of *rules* relation are defined based on members of *alphabets*.
- The variable *s0* must be an element of *variables*.
- There exists at least one rule which contains start variable on the left hand side of it.

A. Producing Left Most Derivations

In this section, we describe the formal left derivations procedure using the production rules. The substitution can be performed recursively to derive new string from a given string of terminal and non-terminal. First, we specify the process of generating a string using a single production by the schema *LeftDerivation* given below. In the specification, s1 and s2 are two strings of type seqX. We say s1 yields s2 if ∃ a∈V and b, s3, s4∈ seq X such that $s1 = s3 \frown \langle a \rangle \frown s4$ and $s2 = s3 \frown b \frown s4$. It is to be noted that *a* is an element in set of variables, the ranges of sequences b, s3, s4 are subsets of *vandt*, (a, b) is a production rule.

LeftDerivation

CFG
$drives: \text{seq}\ X \leftrightarrow \text{seq}\ X$

$\forall s1, s2: \text{seq}\ X \mid \text{ran}\ s1 \subseteq vandt \land \text{ran}\ s2 \subseteq vandt$
$\bullet (s1, s2) \in drives \Rightarrow (\exists a: V; b: \text{seq}\ X; s3, s4: \text{seq}\ X$
$\mid a \in variables \land \text{ran}\ b \subseteq vandt$
$\land (a, b) \in rules \land \text{ran}\ s3 \subseteq terminals$
$\land \text{ran}\ s4 \subseteq vandt \bullet s1 = s3 \frown \langle a \rangle \frown s4 \land s2 = s3 \frown b \frown s4)$

Now, we describe a sequence of left derivations using the approach of single left derivation defined above. The derivation procedure is described below and is denoted by the schema *LeftDerivations* which is an extension of schema

LeftDerivation. It describes the generation from a string of terminals or non-terminals to another string of the same type. In the specification, two strings are considered denoted by s1 and s2 as in the schema which uses the *LeftDerivation* recursively by introducing a sequence s3 of sequences representing an order of the derivations.

LeftDerivations

$$
\begin{array}{l}
\textit{LeftDerivation} \\
\textit{drivess:}\ \mathrm{seq}\ X \leftrightarrow \mathrm{seq}\ X \\
\hline
\forall s1, s2: \mathrm{seq}\ X \mid \mathrm{ran}\ s1 \subseteq vandt \land \mathrm{ran}\ s2 \subseteq vandt \\
\quad \bullet\ (s1, s2) \in drivess \\
\quad \Rightarrow (\exists s3: \mathrm{seq}\ (\mathrm{seq}\ X) \\
\qquad \mid 1 \leqslant \#\, s3 \land (\forall ss: \mathrm{seq}\ X \mid ss \in \mathrm{ran}\ s3 \bullet \mathrm{ran}\ ss \subseteq vandt) \\
\qquad \bullet\ (s1, s3\ 1) \in drives \\
\qquad\quad \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, s3 \bullet (s3\ (i - 1), s3\ i) \in drives) \\
\qquad\quad \land (s3\ (\#\, s3), s2) \in drives)
\end{array}
$$

B. Verification of Language Generated From CFG

In this section, verification of a language generated from a context-free grammar is done. First, verifying procedure of a word is defined then it is extended to the whole language. For this purpose, the formal procedure is described in the schema *WordLeftDerivation* given below. The schema *LeftDerivations* and a *word* are given as input to the schema and it is checked if the *word* can be generated from the CFG using the procedure *WordLeftDerivation* defined above. The symbol, ?, is used to represent that word is an input variable. In the predicate part of the schema, first, it is checked that all alphabets of the word must be from the set of terminal of CFG. Secondly, it is verified that in the derivation of the word the first production used contains the start variable (non-terminal) on the left hand side of the production.

WordLeftDerivation

$$
\begin{array}{l}
\textit{LeftDerivations} \\
\textit{word?:}\ \mathrm{seq}\ \textit{Sigma} \\
\hline
\mathrm{ran}\ word? \subseteq terminals \\
(\langle s0 \rangle, word?) \in drivess
\end{array}
$$

To verify a language, the approach of word verification is used. In the schema *LanguageLeftDerivation* described below, the schema *LeftDerivations* and *language* are given as input and is checked if the *language* can be generated from the CFG by using universal quantifier.

LanguageLeftDerivation

$$
\begin{array}{l}
\textit{LeftDerivations} \\
\textit{language?:}\ \mathbb{P}\ (\mathrm{seq}\ \textit{Sigma}) \\
\hline
\forall w: \mathrm{seq}\ \textit{Sigma} \mid w \in language? \bullet \mathrm{ran}\ w \subseteq terminals \\
\forall w: \mathrm{seq}\ \textit{Sigma} \mid w \in language? \bullet (\langle s0 \rangle, w) \in drivess
\end{array}
$$

C. Checking Ambiguity of Language

In this section, ambiguity of a context-free language is verified. The language is ambiguous if there is a word for which there exists at least two parsing trees based on leftmost derivations. First, verification procedure for a word is defined if it is ambiguously generated using the schema *AmbiguousWord* given below. The schema takes *LeftDerivations* and a *word* as input and checks if the *word* has more than one left most derivations.

AmbiguousWord

$$
\begin{array}{l}
\textit{LeftDerivations} \\
\textit{word?:}\ \mathrm{seq}\ \textit{Sigma} \\
\hline
\mathrm{ran}\ word? \subseteq terminals \\
(\langle s0 \rangle, word?) \in drivess \Rightarrow (\exists s3, s4: \mathrm{seq}\ (\mathrm{seq}\ X) \mid s3 \neq s4 \\
\qquad \land 1 \leqslant \#\, s3 \land 1 \leqslant \#\, s4 \land \mathrm{ran}\ s3 \subseteq \mathrm{ran}\ rules \\
\qquad \land \mathrm{ran}\ s4 \subseteq \mathrm{ran}\ rules \bullet (\langle s0 \rangle, s3\ 1) \in drives \\
\qquad \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, s3 \bullet (s3\ (i - 1), s3\ i) \in drives) \\
\qquad \land (s3\ (\#\, s3), word?) \in drives \land ((\langle s0 \rangle, s4\ 1) \in drives \\
\qquad\quad \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, s4 \bullet (s4\ (i - 1), s4\ i) \in drives) \\
\qquad\quad \land (s4\ (\#\, s4), word?) \in drives))
\end{array}
$$

To verify if the language is ambiguous, the verification procedure of a word is reused. In the schema *AmbiguousLanguage* described below, it is checked if there exists any word having more than one derivations by using the universal quantifier. If this is the case the given language is ambiguous.

AmbiguousLanguage

$$
\begin{array}{l}
\textit{LeftDerivations} \\
\textit{language?:}\ \mathbb{F}\ (\mathrm{seq}\ \textit{Sigma}) \\
\hline
\forall w: \mathrm{seq}\ \textit{Sigma} \mid w \in language? \bullet \mathrm{ran}\ w \subseteq terminals \\
\exists w: \mathrm{seq}\ \textit{Sigma} \mid w \in language? \bullet (\langle s0 \rangle, w) \in drivess \\
\quad \Rightarrow (\exists s3, s4: \mathrm{seq}\ (\mathrm{seq}\ X) \mid s3 \neq s4 \land 1 \leqslant \#\, s3 \land 1 \leqslant \#\, s4 \\
\quad \land \mathrm{ran}\ s3 \subseteq \mathrm{ran}\ rules \land \mathrm{ran}\ s4 \subseteq \mathrm{ran}\ rules \bullet (\langle s0 \rangle, s3\ 1) \in drives \\
\quad \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, s3 \bullet (s3\ (i - 1), s3\ i) \in drives) \\
\quad \land (s3\ (\#\, s3), w) \in drives \land ((\langle s0 \rangle, s4\ 1) \in drives \\
\quad \land (\forall i: \mathbb{N} \mid i \in 2\ ..\ \#\, s4 \bullet (s4\ (i - 1), s4\ i) \in drives) \\
\quad \land (s4\ (\#\, s4), w) \in drives))
\end{array}
$$

V. Model Analysis

There does not exist any computer tool which may guarantee about complete correctness of a computer model. Therefore, even the specification is written using any of the formal languages it may contain potential hazardous or errors. It means an art of writing a formal specification never assures that the developed system is consistent, correct and complete. On the other hand, if the specification is checked and analyzed with the computer tool support it certainly increases the confidence over the system to be developed by

identifying the potential errors, if exist, in syntax and semantics of the formal description. The Z/Eves is one of the most powerful tools which can be used for analyzing the formal specification written by Z notation. A snapshot of the formal specification using Z/Eves tool is presented in Figure 1. The first column on left most of the figure shows a status of the syntax checking and the second one presents the status of proof correctness. The symbol 'Y' shows that specification is correct syntactically and proof is correct while the symbol 'N' stands that errors are identified. In schemas, it is checked that specification is correct in syntax and has a correct proof.

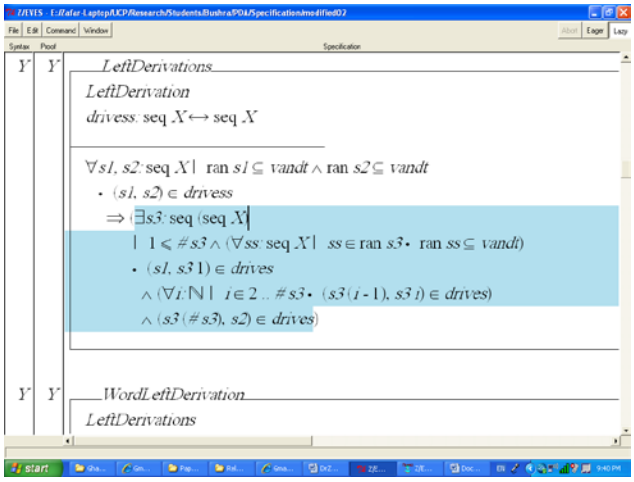

Figure 1. Snapshot of the Model Analysis.

The results of the formal specification are presented in the Table 1. The schema name represents the name of the schemas described for specification. These schemas are analyzed by using the model exploration techniques provided in the Z/Eves tool. The symbol "Y" in column 2 indicates that all the schemas are well written and proved automatically. Similarly, domain checking, reduction and proof by reduction are represented in column 3, 4 and 5, respectively. The symbol "Y*" describes that the schemas are proved by performing reduction on the predicates to make the specification meaningful.

TABLE I. RESULTS OF MODEL ANALYSIS

Schema Name	Syntax Type Check	Domain Check	Reduction	Proof
CFG	Y	Y	Y	Y
LeftDerivation	Y	Y	Y*	Y
LeftDerivations	Y	Y	Y*	Y
WordLefDerivation	Y	Y	Y	Y
LanguageLefDerivation	Y	Y	Y	Y
AmbiguousWord	Y	Y	Y*	Y
AmbiguousLanguage	Y	Y	Y*	Y

VI. CONCLUSION AND FUTURE WORK

An efficient and correct translation from a programming language to machine language is an open issue in the area of computer science and this task is usually done by compilers. Errors in the compiler can lead to incorrect machine code from a source program even the source is correct and verified. Therefore, design and construction of correct compiler is more important than verifying the source programs. If the compiler is formally verified it gives guarantee that the executable code generated behaves exactly as described in the source program. In this paper, formal procedure of identification and analysis of ambiguities is done which is a real challenge in parser development. We know it is an un-decidable problem but this exercise is useful for applying it to a simple compiler for academic purpose, which can be extended to formally verify the compiler.

Both regular expressions and context-free grammars are widely used in construction of the compiler. Regular expressions are not powerful enough and are used to identify token from the source program while syntax is checked by the context-free grammar. The design of a complier can be benefited by transforming context-free grammar to Z specification because Z notation being abstract in nature and having computer tool support enhances reliability and correctness providing a context in which important properties of the system can be formally analyzed and verified. Further, formal specification helped us to make it possible describing precise, unambiguous and easier to understand the resultant model.

An approach is developed by linking context-free grammar with Z notation defining a relationship between fundamentals of these techniques. It is observed that a natural relationship exists between these approaches. This linkage will be useful in verification of compiler in addition to many other applications. At first, we have described the structures of CFG using Z then formal description of derivation process from a sequence of terminals and non-terminals is presented. Further, a procedure of derivations is described by identifying the productions to be used in this process. Then formal models are defined to check the generation of the words and language from the context-free grammar. Finally, ambiguity of the language is verified by using the left most derivations. Formal proofs of the relationship are presented under certain assumptions. The specification is verified and validated using Z/Eves tool.

An extensive survey of existing work was done and explored before initiating this research. Some interesting work [28][29][30][31][32][33] [34][35][36] was found but our work and approach are different because of conceptual and abstract level integration of Z and CFG. Few of the benefits of Z are listed as follows. Every object is assigned a unique type providing useful programming practice. Several type checking tools exist to support the specification. The Z/Eves is a powerful tool to prove and analyze the specification used in this research. The rich mathematical notations made it possible to reason about behavior of a specified system more rigorous and effectively.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

Formalization of some other concepts, useful in compiler verification, is under progress and will appear soon in our future work. Further, we have taken some assumptions for simplicity of construction. In future work, a more generic formal integration will be proposed after relaxing such assumptions.

REFERENCES

[1] C. J. Burgess, "The Role of Formal Methods in Software Engineering Education and Industry," Technical Report, University of Bristol, UK, 1995.

[2] H. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "State/Event-Based Software Model Checking," Integrated Formal Methods, Springer, vol. 2999, pp. 128-147, 2004.

[3] O. Hasan and S. Tahar, "Verification of Probabilistic Properties in the HOL Theorem Prover," Integrated Formal Methods, Springer, vol. 4591, pp. 333-352, 2007.

[4] F. Gervais, M. Frappier, and R. Laleau, "Synthesizing B Specifications from EB3 Attribute Definitions," Integrated Formal Methods, Springer, vol. 3771, pp. 207-226, 2005.

[5] K. Araki, A. Galloway, and K. Taguchi, "Integrated Formal Methods," Proceedings of the 1st International Conference on Integrated Formal Methods, Springer 1999.

[6] B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of Cadence SPW Fixed-Point Arithmetic in HOL," Integrated Formal Methods, Springer, vol. 2335, pp. 185-204, 2002.

[7] J. Derrick and G. Smith, "Structural Refinement of Object-Z/CSP Specifications," Integrated Formal Methods, Springer, vol. 1945, pp. 194-213, 2000.

[8] T. B. Raymond, "Integrating Formal Methods by Unifying Abstractions," Springer, vol. 2999, pp. 441-460, 2004.

[9] J. S. Dong, R. Duke, and P. Hao, "Integrating Object-Z with Timed Automata," pp 488-497, 2005.

[10] J. S. Dong, et al., "Timed Patterns: TCOZ to Timed Automata," The 6th International Conference on Formal Engineering Methods, pp 483-498, 2004.

[11] R. L. Constable, P. B. Jackson, P. Naumov, and J. Uribe, "Formalizing Automata II: Decidable Properties," Technical Report, Cornell University, 1997.

[12] R. L. Constable, P. B. Jackson, P. Naumov, and J. Uribe, "Constructively Formalizing Automata Theory," Foundations of Computing Series, MIT Press, 2000.

[13] M. Heiner and M. Heisel, "Modeling Safety Critical Systems with Z and Petri nets," International Conference on Computer Safety, Reliability and Security, Springer, pp. 361–374, 1999.

[14] H. Leading and J. Souquieres, "Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B," Asia-Pacific Software Engineering Conference, pp. 495-504, 2002.

[15] H. Leading and J. Souquieres, "Integration of UML Views using B Notation," Proceedings of Workshop on Integration and Transformation of UML Models, 2002.

[16] W. Wechler, "The Concept of Fuzziness in Automata and Language Theory," Akademic-Verlag, Berlin, 1978.

[17] N. M. John and S. M. Davender, "Fuzzy Automata and Languages: Theory and Applications," Chapman & HALL, CRC, 2002.

[18] M. Ito, "Algebraic Theory of Automata and Languages," World Scientific Publishing Co., 2004.

[19] N. A. Zafar, S. A. Khan, and B. Kamran, "Formal Procedure of Deriving Language from Context-Free Grammar," International Conference on Intelligence and Information Technology, vol. 1, pp. 533-536, 2010.

[20] M. Brendan and J. S. Dong, "Blending Object-Z and Timed CSP: An Introduction to TCOZ," Proceedings of 20th International Conference on Software Engineering, pp. 95, IEEE Computer Society, 1998.

[21] J. M. Spivey, "The Z Notation: A Reference Manual," Englewood Cliffs, NJ, Printice-Hall, 1989.

[22] J. M. Wing, "A Specifier, Introduction to Formal Methods," IEEE Computer, vol. 23 (9), pp. 8-24, 1990.

[23] J. A. Anderson, "Automata Theory with Modern Applications," Cambridge University Press, 2006.

[24] H. C. Young, J. Moscola, and J. W. Lockwood, "Context-Free Grammar based Token Tagger in Reconfigurable Devices," Proceedings of International Conference of Data Engineering (ICDE/SeNS), pp. 78, 2005.

[25] M. v. d. Brand, A. Sellink, and C. Verhoef, "Generation of Components for Software Renovation Factories from Context-Free Grammars," Conference on Reverse Engineering, pp. 144-153, 2001.

[26] M. Balakrishna, D. Moldovan, and E. K. Cave, "Automatic Creation and Tuning of Context-Free Grammars for Interactive Voice Response Systems," Proceedings of IEEE NLP-KE '05, pp. 158 – 163, 2005.

[27] L. Pedersen and H. Reza, "A Formal Specification of a Programming Language: Design of Pit," Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 111-118, 2008.

[28] D. P. Tuan, "Computing with Words in Formal Methods," Technical Report, University of Canberra, Australia, 2000.

[29] S. A. Vilkomir and J. .P. Bowen, "Formalization of Software Testing Criterion," South Bank University, London, 2001.

[30] A. Hall, "Correctness by Construction: Integrating Formality into a Commercial Development Process," Praxis Critical Systems Limited, Springer, vol. 2391, pp. 139-157, 2002.

[31] B. A. L. Gwandu and D. J. Creasey, "Importance of Formal Specification in the Design of Hardware Systems," School of Electron & Electr. Eng., Birmingham University, 1994.

[32] D. K. Kaynar and N. Lynchn, "The Theory of Timed I/O Automata," Morgan & Claypool Publishers, 2006.

[33] D. Jackson, I. Schechter, and I. Shlyakhter, "Alcoa: The Alloy Constraint Analyzer," Proceedings of The 22nd International Conference of Software Engineering (ICSE'2000), pp. 730-733, 2000.

[34] D. Aspinall and L. Beringer, "Optimisation Validation," Electronic Notes in Theoretical Computer Science, vol. 176, pp. 37–59, 2007.

[35] S. Briaisa and U. Nestmannb, "A Formal Semantics for Protocol Narrations," Theoretical Computer Science, vol. 389, pp. 484–511, 2007.

[36] L. Freitas, J. Woodcock, and Y. Zhang, "Verifying the CICS File Control API with Z/Eves: An Experiment in the Verified Software Repository," Science of Computer Programming, vol. 74, pp. 197-218, 2009.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6