

Using Algebra-Algorithmic and Term Rewriting Tools for Developing Efficient Parallel Programs

Anatoliy Doroshenko¹, Kostiantyn Zhreb¹ and Olena Yatsenko¹

¹ Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
doroshenkoanatoliy2@gmail.com, zhreb@gmail.com, oayat@ukr.net

Abstract. An approach to program design and synthesis using algebra-algorithmic specifications and rewriting rules techniques is proposed. An algebra-algorithmic toolkit based on the approach allows building syntactically correct and easy-to-understand algorithm specifications. The term rewriting system supplements the algebra-algorithmic toolkit with facilities for transformation of the sequential and parallel algorithms, enabling their improvement.

Keywords. Algebra of algorithms, code generation, formalized design of programs, parallel computation, term rewriting

Key terms. FormalMethod, HighPerformanceComputing, ConcurrentComputation, Integration

1 Introduction

Nowadays uniprocessor systems are almost fully forced out by multiprocessor ones, as the latter allow getting the considerable increase of productivity of programs. Thus, the need of program parallelization arises [10]. There are libraries, such as pthreads, OpenMP, TBB and others [1], allowing developers to write parallel programs. Using these libraries a programmer manually divides code into independent sections, describes data exchange and synchronization between them. However, such method has substantial defects, in particular, related to committing of errors into program code and a time required for parallelization and debugging. Therefore, the parallelization process has to be automatized as much as possible, and in an ideal, should be carried out fully automatically, without participation of a programmer.

This paper continues our research on automation of process of designing and development of efficient parallel programs, started in [2], [9], [10], [11]. Our approach is based on usage of Integrated toolkit for Designing and Synthesis of programs (IDS) [2], [19]. The process of algorithm designing in IDS consists in the composition of reusable algorithmic components (language operations, basic operators and predicates), represented in Systems of Algorithmic Algebras (SAA) [2], [9], [19]. We used IDS for generation of sequential and parallel programs in Java and C++ on the basis

of high-level algorithm specifications (schemes). To automate the transformations of algorithms and programs we use term rewriting system Termware [8], [11]. The novelty of this paper is 1) adjusting IDS to generate parallel code in Cilk++ language, which is an extension to the C and C++ programming languages, designed for multi-threaded parallel computing [7] and 2) closer integration between IDS and Termware systems. The approach is illustrated on a recursive sorting algorithm (quick sort).

The problem of automated synthesis of program code from specifications has been studied extensively and many approaches have been proposed [13], [14]. Important aspects of program synthesis include 1) format of inputs (specifications), 2) methods for supporting concrete subject domains and 3) techniques for implementing transformation from specifications to output program code (these aspects roughly correspond to 3 dimensions of program synthesis discussed in [14]). For input specification, a popular option is using domain-specific languages (DSLs) [4], [17] that allow capturing requirements of subject domain. Other options include graphical modeling languages [5], [17], formal specification languages [16], ontologies [6] and algebraic specifications [3]. Using such formalisms enables analysis and verification of specifications and generated code. There are also approaches that provide specification not of program or algorithm, but of problem to be solved, in form of functional and non-functional constraints [18], examples of input/output pairs [15], or natural language descriptions [14].

Another crucial aspect of program synthesis is specialization for subject domain. Some approaches are restricted to a single domain, such as statistical data analysis [12] or mobile application development [17]; others provide facilities for changing domain-specific parts, by using ontological descriptions [6], grammars [16], or by providing generic framework that is complemented by domain-specific tools [18].

Finally, an important aspect is transformation from input specification into source code in a target language. A transformation algorithm can be hand-coded [12], but it reduces flexibility of system. Therefore, transformation is often described in a declarative form, such as rewriting rules [16], visualized graph transformations [17], code templates [6]. More complex approaches require searching the space of possible programs [18], possibly using genetic programming or machine learning approaches [14]. In [4], partial synthesis is proposed: generic parts of application are generated, and then completed with specific details manually.

In comparison, our approach uses algebraic specifications, based on Glushkov algebra of algorithms [2], but they can be represented in three equivalent forms: algebraic (formal language), natural-linguistic and graphical, therefore simplifying understanding of specifications and facilitating achievement of demanded program quality. Another advantage of IDS is a method of interactive design of syntactically correct algorithm specifications [2], [19], which eliminates syntax errors during construction of algorithm schemes. Specialization for subject domain is done by describing basic operators and predicates from this domain. Our approach uses code templates to specify implementations for operators and predicates; program transformations, such as from sequential to parallel algorithm, are implemented as rewriting rules. Such separation simplifies changing subject domain or transformations.

2 Formalized Design of Programs in IDS and Termware

The developed IDS toolkit is based on System of Algorithmic Algebras (SAA), which are used for formalized representation of algorithmic knowledge in a selected subject domain [2], [9], [19]. SAA is the two-based algebra $SAA = \langle \{U, B\}; \Omega \rangle$, where U is a set of logical conditions (predicates) and B is a set of operators, defined on an informational set; $\Omega = \Omega_1 \cup \Omega_2$ is the signature of operations consisting of the systems Ω_1 and Ω_2 of logical operations and operators respectively (these will be considered below). Operator representations of algorithms in SAA are called regular schemes. The algorithmic language SAA/1 [2] is based on mentioned algebra and is used to describe algorithms in a natural language form. The algorithms, represented in SAA/1, are called SAA schemes.

Operators and predicates can be basic or compound. The basic operator (predicate) is an operator (predicate), which is considered in SAA schemes as primary atomic abstraction. Compound operators are built from elementary ones by means of operations of sequential and parallel execution operators, branching and loops, and synchronizer `WAIT 'condition'` that delays the computation until the value of the `condition` is true (see also Table 1 in next section).

The advantage of using SAA schemes is the ability to describe algorithms in an easy-to-understand form facilitating achievement of demanded quality of programs. The IDS is intended for the interactive designing of schemes of algorithms in SAA and generating programs in target programming languages (Java, C++, Cilk++). In IDS algorithms are designed as syntactically correct programs ensuring the syntactical regularity of schemes. IDS integrates three forms of design-time representation of algorithms: regular schemes, SAA schemes (textual representation of SAA formulae) and flow graphs. For integration with Termware, in this paper IDS was also adjusted on generation of programs in Termware language.

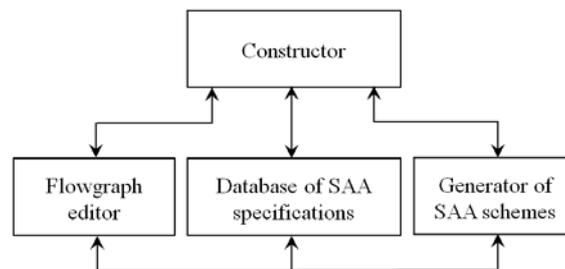


Fig. 1. Architecture of the IDS toolkit

The IDS toolkit consists of the following components (Fig. 1): constructor, intended for dialogue designing of syntactically correct sequential and concurrent algorithm schemes and generation of programs; flow graph editor; generator of SAA schemes on the basis of higher level schemes, called hyper-schemes [19]; and data-

base, containing the description of SAA operations, basic operators and predicates in three mentioned forms, and also their program implementations.

The constructor is intended to unfold designing of algorithm schemes by superposition of SAA language constructs, which a user chooses from a list of reusable components for construction of algorithms. The design process is represented by a tree of an algorithm [2], [19]. On each step of the design process the constructor allows the user to select only those operations, the insertion of which into the algorithm tree does not break the syntactical correctness of the scheme. The tree of algorithm constructing is then used for automatic generation of the text of SAA scheme, flow graph and the program code in a target programming language.

Example 1. We illustrate the use of SAA on Quicksort algorithm, which is given below in the form of SAA scheme. The identifiers of basic operators in the SAA scheme are written with double quotes and basic predicates are written with single quotes. Notice that identifiers can contain any text explaining the meaning of operator or predicate. It is not interpreted: it has to match exactly the specification in the database (however, since constructs are not entered manually, but selected from a list, the misspellings are prevented). The comments and implementations of compound operators and predicates in SAA schemes begin with a string of “=” characters.

```
SCHEME QUICKSORT_SEQUENTIAL =====

"main(n)"
===== Locals (
  "Declare an array (a) of type (int) and size (n)";
  "Declare a variable (i) of type (int)";
  "Declare a variable (end) of type (int)";
  "Fill the array (a) of size (n) with random
  values";
  "end := a + n";
  "qsort(a, end)";

"qsort(begin, end)"
===== IF NOT('begin = end')
  "Reduce (end) by (1)";
  "Reorder array (a) with range (begin) and (end)
  so that elements less than pivot (end) come
  before it and greater ones come after it; save
  pivot position to variable (middle)";
  "qsort(begin, middle)";
  "Increase (middle) by (1)";
  "Increase (end) by (1)";
  "qsort(middle, end)"
  END IF

END OF SCHEME QUICKSORT_SEQUENTIAL
```

To automate the transformation (e.g. parallelization) of programs we augment capabilities of IDS with rewriting rules technique [8], [11]. At the first step we construct high-level algebraic models of algorithms based on SAA in IDS (see also [2], [9], [19]). After high-level program model is created, we use parallelizing transformations to implement a parallel version of the program on a given platform (multicore in this paper). Transformations are represented as rewriting rules and therefore can be applied in automated manner. The declarative nature of rewriting technique simplifies adding new transformations. Also transformations are separated from language definitions (unlike approach used in [16]), therefore simplifying addition of new transformations or new languages.

We use the rewriting rules system Termware [8], [11]. Termware is used to describe transformations of *terms*, i.e. expressions in a form $f(t_1, \dots, t_n)$. Transformations are described as Termware *rules*, i.e. expressions of form `source [condition]-> destination [action]`. Here `source` is a source term (a pattern for match), `condition` is a condition of rule application, `destination` is a transformed term, `action` is additional action that is performed when rule fires. Each of 4 components can contain variables (denoted as `$var`), so that rules are more generally applicable. Components `condition` and `action` are optional. They can execute any procedural code, in particular use the additional data on the program.

3 Generation of Terms and Programs and Experimental Results

IDS system performs generation of programming code on the basis of an algorithm tree, received as a result of designing an algorithm in the IDS Constructor (see Section 2), and also code templates – implementations of basic operators and predicates in a target language (Java, C++, Cilk++), that are stored in IDS database. In the process of generation, IDS translates SAA operations into corresponding operators of programming language. Compound operators can be represented as subroutines (methods). IDS database contains various code patterns for generation of parallel programs, namely using WinAPI threads, Message Passing Interface (MPI), and Cilk++ operations [7]. For implementation of parallel version of our illustrative example (Quicksort algorithm), we used Cilk++ as it facilitates programming of recursive parallel programs [7]. Cilk++ is a general-purpose programming language, based on C/C++ and designed for multithreaded parallel computing.

Table 1 gives a list of main SAA operations and templates of their implementation in Termware and Cilk++, which are stored in the IDS database. The implementations contain placeholders like `^condition1^`, `^operator1^` etc., which are replaced with program code during the program generation.

For the purpose of transformation of some algorithm, IDS performs the generation of a corresponding term and developer specifies a set of rules for transformation. Then Termware carries out the actual transformation, the result of which can further be used for code generation in a programming language.

Table 1. The main SAA operations and templates of their implementation in Termware and Cilk++ languages

Text of SAA operation	Termware implementation	Cilk++ implementation
"operator1"; "operator2"	then (^operator1^, ^operator2^)	^operator1^; ^operator2^
IF 'condition' THEN "operator1" ELSE "operator2" END IF	IF (^condition1^, ^operator1^, ELSE (^operator2^))	if (^condition1^){ ^operator1^ } else {^operator2^}
FOR '(var) from (begin) to (end)' LOOP "operator1" END OF LOOP	FOR (%1, %2, %3, ^operator1^)	for (%1, %2, %3) { ^operator1^ }
("operator1" PARALLEL "opera- tor2")	Parallel(^operator1^, ^operator2^)	cilk_spawn ^operator1^; ^operator2^
WAIT 'condition'	WAIT (^condition1^)	cilk_sync;

Example 2. We will parallelize the sequential Quicksort algorithm (see Example 1), using IDS and Termware. For the parallelization, function `qsort` has to be transformed, so we generated the term for this function:

```
qsort(Params(begin, end),
      IF (NOT(Equal(begin, end)),
        then (Dec(end, 1),
              then (Partition(a, begin, end, end),
                    then (CALL(qsort(begin, middle)),
                          then (Inc(middle, 1),
                                then (Inc(end, 1),
                                      CALL (qsort(middle, end))))))))))
```

Then the operation of parallel execution of operations has to be added to this term. This is done by applying the following two Termware rules:

1. `then(CALL($x), then ($y, $z)) ->`
`Parallel (CALL($x), then($y, $z))`
2. `then($x1, Parallel($x2, $x3)) ->`
`then($x1, then(Parallel($x2, $x3),`
`WAIT(AllThreadsCompleted(n)))`

The first rule replaces the operation of sequential execution of operators with parallel execution. The second rule adds a synchronizer `WAIT(AllThreadsCompleted(n))`

pleted(n), which delays the computation until all threads complete their work. The result of the transformation is given below.

```

qsort(Params(begin, end),
IF(NOT(Equal(begin, end)),
  then (Dec(end, 1),
  then (Partition(a, begin, end, end),
  then (Parallel(
    CALL (qsort(begin, middle)),
    then (Inc(middle, 1),
    then (Inc(end, 1),
    CALL (qsort(middle, end))))),
    WAIT(AllThreadsCompleted(n))))))

```

Thus, as a result of parallelization, the first operator (thread) of `Parallel` operation executes the operator `qsort(begin, middle)`, and the second one calls two `Inc` operators and `qsort(middle, end)`. Operation `WAIT(AllThreadsCompleted(n))` performs the synchronization of threads. The threads are created recursively; their quantity is specified as an input parameter of function `main`. Notice that these transformations are only valid if two `qsort` calls are independent. The system doesn't check this property: it has to be asserted by a developer.

The resulting parallel algorithm scheme Quicksort was used for generation of code in Cilk++ using IDS system. The parallel program was executed on Intel Core 2 Quad CPU, 2.51 GHz, Windows XP machine. Fig. 2 shows the program execution time in seconds. The speedup at execution of program with usage of 2, 3 and 4 processors was 2; 2.9 and 3.8 accordingly, which shows that the program has a good degree of parallelism and is scalable.

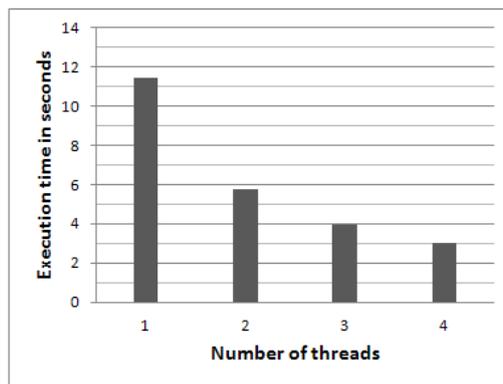


Fig. 2. The execution time of parallel Quicksort program on a quad-core processor; the size of input array is $5 \cdot 10^7$ elements

4 Conclusion

We have described our approach of constructing efficient parallel programs using high-level algebra-algorithmic specifications and rewriting rules technique. Algebra-algorithmic toolkit IDS and rewriting rules engine Termware are combined to enable formal, yet easy-to-understand algorithm specifications and automate program synthesis and parallelization process. The combined development toolkit can be retargeted to various subject domains and implementation languages, as exemplified by Cilk++. The developed system could be further extended with automated code analysis facilities based on rewriting technique.

References

1. Akhter, S., Roberts, J.: Multi-Core Programming. Intel Press, Hillsboro (2006)
2. Andon, F. I., Doroshenko, A. Y., Tseytlin, G. O., Yatsenko, O. A.: Algebra-Algorithmic Models and Methods of Parallel Programming. Akademperiodika, Kyiv (2007) (in Russian)
3. Apel, S. et al.: An Algebraic Foundation for Automatic Feature-Based Program Synthesis. *Science of Computer Programming*. 75(11), 1022–1047 (2010)
4. Bagheri, H., Sullivan, K.: Pol: Specification-Driven Synthesis of Architectural Code Frameworks for Platform-Based Applications. In: Proc. 11th Int. Conf on Generative Programming and Component Engineering, pp. 93–102, ACM, New York (2012)
5. Batory, D.: Program Refactoring, Program Synthesis, and Model-Driven Development. In: Proc. 16th Int. Conf. on Compiler Construction. LNCS 4420, pp. 156–171 Springer-Verlag, Berlin Heidelberg (2007)
6. Bures, T. et al.: The Role of Ontologies in Schema-Based Program Synthesis. In: Proc. Workshop on Ontologies as Software Engineering Artifacts, Vancouver (2004)
7. Cilk Home Page, <http://cilkplus.org/>
8. Doroshenko A., Shevchenko R.: A Rewriting Framework for Rule-Based Programming Dynamic Applications, *Fundamenta Informaticae*, 72(1–3), 95–108 (2006)
9. Doroshenko, A., Tseytlin, G., Yatsenko, O., Zachariya, L.: A Theory of Clones and Formalized Design of Programs. In: Proc. Int. Workshop on Concurrency, Specification and Programming (CS&P'2006), pp. 328–339, Wandlitz, Germany (2006)
10. Doroshenko, A. Y., Zhreb, K. A., Yatsenko, Ye. A.: On Complexity and Coordination of Computation in Multithreaded Programs. *Problems in Programming*, 2, 41–55 (2007) (in Russian)
11. Doroshenko, A., Zhreb, K.: Parallelizing Legacy Fortran Programs Using Rewriting Rules Technique and Algebraic Program Models. In: Ermolayev, V. et al. (eds.) *ICT in Education, Research, and Industrial Applications*. CCIS 347, pp. 39–59. Springer Verlag, Berlin Heidelberg (2013)
12. Fischer, B., Schumann, J.: AutoBayes: a System for Generating Data Analysis Programs from Statistical Models. *J. Funct. Program.* 13(3), 483–508 (2003)
13. Flener, P.: Achievements and Prospects of Program Synthesis. In: Kakas, A. C., Sadri, F. (eds.) *Computational Logic: Logic Programming and Beyond*. LNCS 2407, pp. 310–346, Springer Verlag, London (2002)

14. Gulwani, S.: Dimensions in Program Synthesis. In: 12th Int. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, pp. 13–24. ACM, New York (2010)
15. Kitzelmann, E.: Inductive Programming: a Survey of Program Synthesis Techniques. Approaches and Applications of Inductive Programming, LNCS 5812, pp. 50–73. Springer Verlag, Berlin Heidelberg (2010)
16. Leonard, E. I., Heitmeyer, C. L.: Automatic Program Generation from Formal Specifications using APTS. In: Automatic Program Development. A Tribute to Robert Paige, pp. 93–113. Springer Science, Dordrecht (2008)
17. Mannadiar, R., Vangheluwe, H.: Modular Synthesis of Mobile Device Applications from Domain-Specific Models. In: Proc. 7th Int. Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 21–28. ACM, New York (2010)
18. Srivastava, S., Gulwani, S., Foster, J. S.: From Program Verification to Program Synthesis. In: Proc. 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 313–326. ACM, New York (2010)
19. Yatsenko, O.: On Parameter-Driven Generation of Algorithm Schemes. In: Proc. Int. Workshop on Concurrency: Specification and Programming (CS&P'2012), pp. 428–438, Berlin, Germany (2012)