Generation Method for Correct Parallel Programs Based on Equivalent Transformation

Hidekatsu Koike
Faculty of Social Infomation, Sapporo Gakuin University, Ebetsu, Hokkaido, 069-8555, Japan

Kiyoshi Akama
Information Initiative Center, Hokkaido University, Sapporo, Hokkaido, 060-0811, Japan

Katsunori Miura
Information Processing Center, Kitami Institute of Technology, Kitami, 090-8507, Japan

ABSTRACT

In the equivalent transformation computation model (ETCM), a set of correct programs is mathematically defined by a formal specification, and a correct program can be generated by selecting a program that is efficient with respect to a desired run-time environment from the set. In the ETCM, a program is a set of rewriting procedures which preserve the meaning of a formal specification—and are thus called equivalent transformation (ET) rules. In the model, computation is regarded as the rewriting of a state that is represented by a set of clauses, and is repeated until a set of answers is obvious, e.g., a set of ground unit clauses is typically obtained. The computation is intrinsically nondeterministic, and hence parallelism is also inherent. We have proposed a sufficient condition for correctness of parallel programs based on the ETCM. In this paper, we present a parallel program generation method and show how the correctness of the programs generated can be guaranteed.

Keywords: Parallel Computing, Program Synthesis, Program Correctness, Rule Generation, Equivalent Transformation

1. INTRODUCTION

In this paper, we propose a method for generating parallel programs based on the equivalent transformation computation model (ETCM) [1] and additional algorithms that guarantee correct parallel computation [2]. In the proposed method, a generated parallel program is correct with respect to a given formal specification. In the ETCM, computation can be regarded as a sequence of equivalent transformations, while a program comprises a set of equivalent transformation (ET) rules and descriptions for control of their application. We can use a variety of computational procedures with correctness theorem by using ET as a basis for computation. A declarative description P mathematically determines its meaning $\mathcal{M}(P)$, which is intuitively an extension of the declarative semantics of logic programs. The formal definition can be found in several papers related to the ETCM, for example [1, 3]. A rewriting rule is an ET rule with respect to a declarative description P iff the rule rewrites P into P' and $\mathcal{M}(P) = \mathcal{M}(P')$. We can use various ET rules for computation in order to improve program efficiency since ET rules can represent more detailed procedures than clauses. We designed our programming language with the rule syntax so that it can represent a variety of ET rules and can execute efficiently by adopting specified head patterns, one-sided matching, and applicability conditions [4, 5]. A program is always correct if it consists only of ET rules since the correctness of each rule is completely independent of that of other rules. This independence characteristic allows us to check the correctness of each newly generated ET rule in isolation and then accumulate them one by one to form a complete program. In addition, the independence allows us to freely combine ET rules that seem to run the fastest for a given problem. Given a specification in the form of a declarative description, a hand written set of rewriting rules may be correct if they are carefully constructed to preserve the semantic meaning of the specification. We can check the correctness of each rule with rigorous theory [3, 6, 7, 8] and can even generate a wide class of ET rules, including non-trivial ones, automatically [9].

We previously proposed a method [5] in which a sequential program is first generated using a program generation method we developed [9], and then obtaining a parallel program by introducing a set of abstracted parallel procedures, represented by rewriting rules, into the sequential program. The correctness of all the computation steps for parallelism can be verified if each of the rewriting rules follows the theories proposed in [2]. The parallel program can execute computations in which the degree of parallelism may vary according to the run time situation, various procedures run in parallel with shared variables, and the response time of each of the procedures may differ and be unpredictable. We use nonogram puzzles as an example of such situations.

In this paper, we propose a method for generation of correct parallel programs from given specifications, and look at the generation procedures and the relationship among the procedures in more detailed than in previous work. We also generalize parallel program generation by regarding parallel program generation as the generation of ET rules from specification, and discuss the correctness of the programs generated. The remainder of this paper is organized as follows: Section 2 recounts the underlying concept of the ETCM in an intuitive way. Section 3 gives an overview of our parallel program generation method and introduces the rewriting rules used for parallelism. Section 4 describes the program generation procedures. Section 5 discusses the correctness of the proposed program generation method. Section 6 compares our approach with existing approaches. Section 7 concludes this paper.

2. THE ETCM

Formal Specification as a Declarative Description

In the ETCM, a declarative description is not a program and has no procedural semantics; thus, the order of the atoms and the clauses used is not significant. A declarative description only determines its meaning. A simple example of a declarative description, $P = D \cup Q$, is illustrated in Figures 1 and 2. P consists of clause sets D and Q. D declaratively defines the rules of nonogram; Q presents a specific nonogram puzzle to be solved and is changed by rewriting rules at run-time, while D remains unchanged during the computation process. P implicitly determines the answer to Q (a more detailed explanation can be found in [5]). A filled cell, ■, is represented by 1, while a blank cell, $\times$, is represented by 0.

ET as Computation

In the ETCM, an answer is obtained by repeatedly rewriting Q until the answer is obvious, e.g., Q is transformed into Q' (shown in Figure 3) after being rewritten more than one time. Figure 4a corresponds to Q—while Figure 4b corresponds to Q'.

D ={

```
(pat () *pl)<--(allZero *pl).
(pat (*n | *ns) (0 | *pls))
    <--(check_c *ns *pl),(pat (*n | *ns) *pls).
(pat (*n|*ns) (1 | *pls))
    <--(sub *n 1 *n2),(seq1 *n2 *pls *rest),
       (start0 *rest), (pat *ns *rest).
(seq1 *n (1|*pls) *pls2)
   <--(> *n 0), (:= *n2 (- *n 1)),
      (seq1 *n2 *pls, *pls2).
(seq1 0 *pls *pls2)<--(= *pls *pls2).
(start0 ())<--.
(start0 (0|*rest))<--.
(allZero ())<--.
(allZero (0 | *rest))<--(allZero *rest).
(check_c *ns *pl)<--
    (len *ns *nl), (len *pl *pll),
    (listSum *ns *sum), (sub *nl 1 *nl2),
    (add *sum *nl2 *ml), (>= *pll *ml).}
```

Figure 1: The knowledge part of P.

In the ETCM, computation is a sequence of rewritings; and rewriting procedures are specified by a set of rewriting rules (the syntax of them and procedural meaning are described in [5]). Computation is correct iff $\mathcal{M}(D \cup Q) = \mathcal{M}(D \cup Q')$.

3. PARALLEL PROGRAM GENERATION

Overview

Figure 5 shows the relationship among the constructs of our proposed program generation method. In the figure, $D \cup Q$ is a formal specification that mathematically determines the set of ET rules that can be used. A program consists of a partial set of ET rules, each of which is selected from the set determined by $D \cup Q$ and utilized in the program. There are three programs in the figure. Each program is given a set of runtime libraries and application programming interfaces (APIs) that are specialized in its runtime environment. Programs are abstractly written using a set of ET rules, and then adapted to various runtime environments by incorporating with a suitable set of runtime libraries and APIs. Although our parallelism model is master/slave, we can represent more than one process by a program as long as each process can know its role at runtime. Thus, a program consists of client rules, server rules, and library rules. Figure 6 depicts the structure of such a program. It is described in more detail in a later section.

Q ={

```
(ans (*11 *12 *13 *14 *15 *16
      *21 *22 *23 *24 *25 *26
      *31 *32 *33 *34 *35 *36
      *41 *42 *43 *44 *45 *46
      *51 *52 *53 *54 *55 *56
      *61 *62 *63 *64 *65 *66))<--
    (pat (1 3)  (*11 *12 *13 *14 *15 *16)),
    (pat   (5)  (*21 *22 *23 *24 *25 *26)),
    (pat   (3)  (*31 *32 *33 *34 *35 *36)),
    (pat   (4)  (*41 *42 *43 *44 *45 *46)),
    (pat (2 2)  (*51 *52 *53 *54 *55 *56)),
    (pat   (1)  (*61 *62 *63 *64 *65 *66)),
    (pat (1 3)  (*11 *21 *31 *41 *51 *61)),
    (pat   (5)  (*12 *22 *32 *42 *52 *62)),
    (pat   (3)  (*13 *23 *33 *43 *53 *63)),
    (pat   (5)  (*14 *24 *34 *44 *54 *64)),
    (pat (2 1)  (*15 *25 *35 *45 *55 *65)),
    (pat   (1)  (*16 *26 *36 *46 *56 *66)).}
```

Figure 2: The query part of P.

Q' ={

```
(ans (0 1 0 1 1 1
      1 1 1 1 1 0
      0 1 1 1 0 0
      1 1 1 1 0 0
      1 1 0 1 1 0
      1 0 0 0 0 0))<--.}
```

Figure 3: Result obtained by computation.

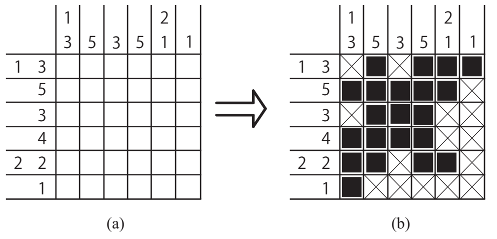

Figure 4: Example of a nonogram problem and its solution.

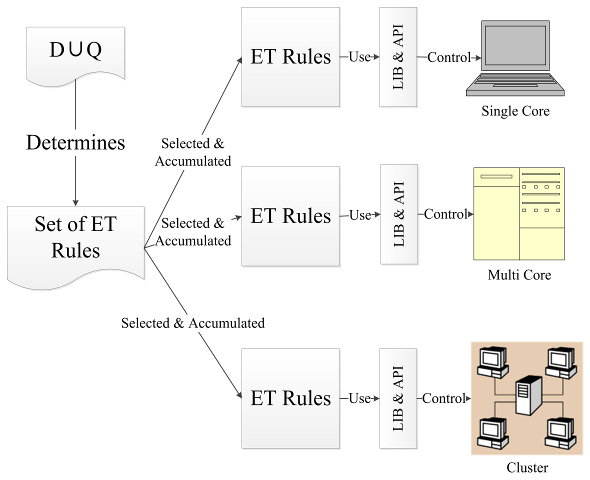

Figure 5: Relationship among the constructs of our proposed method.

Client Rules

Requesting rules

The applicability condition for the atoms that are to be processed in parallel and the procedures representing how to process the atoms are generalized. The applicability conditions are as follows:

- An atom that has not been rewritten yet is always applicable.
- An atom that has been rewritten already is applicable iff its shared variable is substituted by processing other atoms.

The first condition and its corresponding procedure are represented by the following rule:

```
(pat *n *cs)==>(pat *n ? *cs).
```

The rule rewrites any two-argument `pat` atom into a three-argument `pat` atom whose second and third arguments represent the previous and current states of cells, respectively. The new atom is applicable since the third argument is more specialized than the second argument, which means that the atom has a new state and then is need to be checked whether a new substitution is obtained from the atom. The second condition and its corresponding procedure are represented by the following rule:

```
(pat *n *pcs *cs),
 {(specialized *pcs *cs),
  (sendReq *co (pat *n *cs))}
 ==>{(copy *cs *npcs)},(pat *n *npcs *cs *co).
```

The rule is applicable to a three-argument `pat` atom whose third argument is more specialized than its second argument and a request to process two-argument `pat` atom is successfully sent to the server. When the rule is applied, it copies the current state of the cells represented by the variable `*cs` to `*npcs` and replaces the original atom with a four-argument `pat` atom that has `*npcs` and `*co` (an object for communication [5] with the server) in addition to the original arguments. The built-in rule (B-rule) `specialized/2` checks whether the second argument is more specialized than the first argument. The B-rule `sendReq/2` encapsulates a procedure that sends a request to the server for parallel computation [5].

Receiving rules

Receipt of the result of a computation from a server is abstracted by the following rule:

```
(pat *n *pcs *cs *co),
 {(terminated *co)}
 ==>{(copy *cs *cs2), (getResult *co *cs),
     (makePCS *pcs *cs2 *cs *npcs)},
    (pat *n *npcs *cs).
```

The rule is applicable to a four-argument `pat` atom iff the computation being carried out by the server referred to by the communication object `*co` has ended. When applied, the rule copies the current state `*cs` to `*cs2`; retrieves the computation result from the server and unifies the result and `*cs`; makes a variable `*npcs` representing the previous state of the cells according to `*pcs`, `*cs`, and `*cs2`; and replaces the original atom with a new three-argument `pat` atom. The B-rule `terminated/1` checks whether the computation being done by the server specified by its argument has ended. The B-rule `getResult/2` gets the result from the server and unifies it with the second argument. The B-rule `makePCS/4` makes the new previous state the second argument of the new three-argument `pat` atom according to the substitution made from the server and/or the outside of the atom while the server process runs.

Server Rules

A server runs in parallel in a process that processes a requested computation from a client. The process is a conceptual parallel process and can be replaced with a thread at the implementation stage for example. A server receives an atom via the `sendReq/2` call by a client, and then makes a clause from the atom. For example, if `(pat (1 3) (*1 *2 *3 *4 *5 *6))` is received, the following clause set, Q_s, is made:
Q_s ={

```
  (pat (1 3) (*1 *2 *3 *4 *5 *6))
    <--(pat (1 3) (*1 *2 *3 *4 *5 *6)).}
```

By repeated application of ET rules [1] to Q_s, ${Q_s}'$ is obtained as follows.
${Q_s}'$ ={

```
  (pat (1 3) (1 0 1 1 1 0))<--.
  (pat (1 3) (1 0 0 1 1 1))<--.
  (pat (1 3) (0 1 0 1 1 1))<--.}
```

${Q_s}'$ contains three answers. The server sends a result made from ${Q_s}'$. The way in which a result is made from a set of clauses depends on the characteristics of the problem to be solved; so that the way must be appropriately specified with respect to the problem. In this case, the result is made by determining common values. For example, the result `(? ? ? 1 1 ?)` is made from ${Q_s}'$ since the fourth and fifth elements of the second arguments in the `pat/2` atoms are all 1. The result is sent via a communication object, i.e., the result is set to the container in a communication object and then the termination flag is set to *true* to notify the client that the server process has terminated.

[1]An example of the ET rules used can be found in [5]

Library Rules

Implementation of the rules for `terminated/1`, `getResult/2`, and `sendReq/2` depends on the desired runtime environment. The client rules and the server rules encapsulate parallel procedures by using the rules.

Specification-Dependent Rules

Efficient parallel procedures depend on a problem's specification. In the above example, the `pat` atom can be processed in parallel. Consequently, the requesting rules and the receiving rules must be tailored according to the atom pattern.

4. PROGRAM GENERATION PROCEDURE

This section details the procedure used in our proposed parallel program generation. The procedure consists of the following steps: specification writing, generation of ET rules, preparation of specification-dependent rules for parallelism, and preparation of library rules. Each of the steps comprising the procedure is described below. A parallel program is generated by combining all the rules generated in the overall procedure.

Specification Writing

In our proposed method, a specification is a declarative description that is represented as the union of two sets of definite clauses, $D \cup Q$. D represents background knowledge and Q represents problems of interest. Q also represents an initial computation state while D never changes at runtime.

Generation of ET Rules

Given a specification P $(= D \cup Q)$, its declarative meaning $\mathcal{M}(P)$ is mathematically determined and thus a set of ET rules is determined. If an ET rule rewrites Q into Q', then $\mathcal{M}(D \cup Q) = \mathcal{M}(D \cup Q')$; thus P also determines a set of ET rules with respect to P. To generate programs in our proposed method is to select ET rules that efficiently transform Q into a form in which the answer to P is obvious (e.g., a set of ground unit clauses), from the set. The correctness of handwritten ET rules is verified by proving that the rules always preserve declarative meaning [3, 6, 7, 8]. We have also developed an automatic rule generation technique [9].

Preparation of Specification-Dependent Rules for Parallelism

In this step, the atoms to be processed in parallel are determined and the requesting rules for the atoms are prepared. Although the nonogram example discussed above had only a `pat` atom as the atom processed in parallel, more than one different atom pattern can be processed in parallel in our proposed method. A specification-dependent rule for parallelism can be determined by determining an atom pattern since, except for their head atom pattern, the rules have a common structure.

Preparation of Library Rules

Library rules are used by specification-dependent rules for parallelism. The library rules control parallel processing functions via APIs. Although the library rules depend on their runtime environments, they are independent of specifications; thus we can assume that they are sufficiently tested and correct.

5. PROGRAM CORRECTNESS

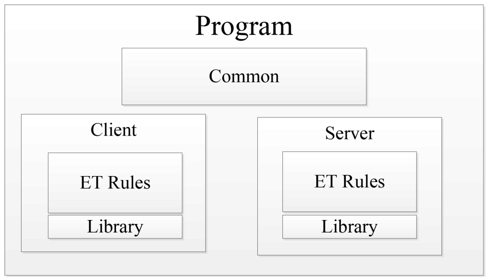

Figure 6: General structure of our generated parallel programs.

Figure 6 depicts the general structure of the parallel program generated by our proposed method. It comprises a common part, a client part, and a server part. The client part consists of ET rules generated from specification and the library rules for clients. The server part consists of ET rules generated from specification and the library rules for servers. The common part is independent of a specific problem specification. Thus, we can assume that the common part is correct.

Correctness of the Client Part

ET rules in the client part are generated from specification using our rule generation methods [3, 6, 7, 8, 9]. Thus the rules always yield correct computation. Requesting rules and receiving rules in the client part follow correct parallel procedures [2]. Consequently, the client part is correct.

Correctness of the Server Part

Just as the client part, ET rules in the server part are generated from specification using our rule generation methods. Therefore, the rules are guaranteed correct. Result construction process is described in the server part. The process generates a result from the server computation result, which is correct. Its correctness is proved by verifying that its procedure is equivalent to the correct parallel procedures[2].

6. COMPARISON

Various methods for mathematically verifying systems have been proposed. The approaches generally try to prove the correctness of target systems by verifying all possible computation states. However, these approaches are encumbered by a major scalability problem called the state space explosion problem. The consequent solutions for the problem often introduce some approximations that may cause false warning or missing errors. In contrast, in our approach, correctness of an entire system can be strictly ensured by proving that each rewriting rule is correct. Thus, the cost of proving correctness can be represented by $O(n)$, where n is the number of writing rules; which means that our approach proves correctness at a lower cost than other approaches. In our approach, we assume that a given specification is correct, a specification is completely declarative, and a program is a set of ET rules with respect to the specification. Also, in our approach, a programmer can implicitly exploit parallelism merely by writing a declarative specification since specification and program are strictly differentiated and mathematically related to each other. Parallel procedures are generated automatically from the specification. Parallel execution of logic programs [10], by contrast, identifies a specification with a program so that a programmer may have to write elaborate control code in order to ensure efficiency. Constraint Handling Rules (CHRs) support a very fine-grained form of parallelism [11] can be viewed as a proper subclass of ETRs [12]. However CHR theory does not have a specification of a program from the point of view of the ETCM. Correctness of computation in the CHR theory is based on the logical equivalence of computation states and confluence [13] while correctness of computation in the ETCM is based on meaning preserving transformation. In the ETCM, any kind of procedures can be used as long as they preserve the meaning of a declarative description. Such procedures include extra-logical operators and procedures represented by multi-body rules, which CHR theory does not allow [12].

7. CONCLUSION

In this paper, we proposed a method for generating correct parallel programs from given formal specifications. In our proposed method, abstracted parallel procedures, represented by ET rules, are accumulated to generate a parallel program. A program comprising a set of ET rules is guaranteed correct as each ET rule is individually correct. This paper distinguished abstract parallel procedural representation from low-level, environment-dependent representation of a program by using rewriting rules. As a result, we can adapt parallel programs generated using our proposed method to various runtime environments by giving them a suitable set of runtime libraries and APIs.

Acknowledgements
This study is partly supported by the Collaborative Research Program 2012, Information Initiative Center, Hokkaido University, Sapporo, Japan.

8. REFERENCES

[1] K. Akama, T. Simizu, and E. Miyamoto, "Solving problems by equivalent transformation of declarative programs," **Journal of the Japanese Society for Artificial Intelligence**, vol. 13, pp. 944–952, 1998.

[2] K. Akama, E. Nantajeewarawat, and H. Ogasawara, "Generation of correct parallel programs based on specializer generation transformations," in **Proceedings of the 7th international conference on intelligent technologies (InTech'06)**, 2006, pp. 90–99.

[3] K. Akama, E. Nantajeewarawat, and H. Koike, "A class of rewriting rules and reverse transformation for rule-based equivalent transformation," **Electronic Notes in Theoretical Computer Science**, vol. 59 (4), pp. 1–16, 2001.

[4] H. Koike, K. Akama, and H. Mabuchi, "A programming language interpreter system based on equivalent transformation," in **2005 IEEE 9th International Conference on Intelligent Engineering Systems (INES 2005)**, 2005, pp. 283–288.

[5] H. Koike and K. Akama, "Generation of correct parallel programs guided by rewriting rules," in **Proceedings of The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications**, vol. 1, 2011, pp. 12–18.

[6] K. Akama, E. Nantajeewarawat, and H. Koike, "Componentwise program construction: Requirements and solutions," in **WSEAS Transactions on Information Science and Applications**, vol. 3, 2006, pp. 1214–1221.

[7] K. Akama, E. Nantajeewarawat, and H. Koike, "Program generation in the equivalent transformation computation model using the squeeze method," in **Perspectives of Systems Informatics PSI2006**, ser. Lecture Notes in Computer Science, vol. 4378. Springer-Verlag, 2007, pp. 41–54.

[8] K. Akama and E. Nantajeewarawat, "Formalization of the equivalent transformation computation model," **Journal of Advanced Computational Intelligence and Intelligent Informatics**, vol. 10, no. 3, pp. 245–259, 2006.

[9] H. Koike, K. Akama, and E. Boyd, "Program synthesis by generating equivalent transformation rules," in **Proceedings of the Second International Conference on Intelligent Technologies (InTech'01)**, 2001, pp. 250–259.

[10] G. Gupta, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: A survey," **ACM Transactions on Programming Languages and Systems**, vol. 23, p. 2001, 1995.

[11] T. Frühwirth, "Parallelizing union-find in Constraint Handling Rules using confluence," in **Proc. of ICLPf05. LNCS**, vol. 3668. Springer-Verlag, 2005, pp. 113–127.

[12] Y. Shigeta, K. Akama, H. Mabuchi, and H. Koike, "Converting constraint handling rules to equivalent transformation rules," **Journal of Advanced Computational Intelligence and Intelligent Informatics**, vol. 10, pp. 339–348, 2006.

[13] E. S. Lam and M. Sulzmann, "Concurrent goal-based execution of Constraint Handling Rules," **Theory and Practice of Logic Programming TPLP**, vol. 11, pp. 841–879, 2009.