Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 421926, 19 pages
http://dx.doi.org/10.1155/2013/421926

Research Article

Approximate Bisimulation and Optimization of Software Programs Based on Symbolic-Numeric Computation

Hui Deng[1] and Jinzhao Wu[2]

[1] *School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China*
[2] *Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, Nanning, Guangxi 530006, China*

Correspondence should be addressed to Jinzhao Wu; jinzhaowu205@gmail.com

Received 9 April 2013; Revised 6 August 2013; Accepted 7 August 2013

Academic Editor: Yang Xu

To achieve behavior and structure optimization for a type of software program whose data exchange processes are represented by nonlinear polynomial systems, this paper establishes a novel formal description called a nonlinear polynomial transition system to represent the behavior and structure of the software program. Then, the notion of bisimulation for software programs is proposed based on the equivalence relation of corresponding nonlinear polynomial systems in their nonlinear polynomial transition systems. However, the exact equivalence is too strict in application. To enhance the flexibility of the relation among the different software systems, the notion of approximate bisimulation within a controllable error range and the calculation algorithm of approximate bisimulation based on symbolic-numeric computation are given. In this calculation, an approximate relation is represented as a MAX function that is resolved with the full filled method. At the same time, the actual error is calculable. An example on a multithreading program indicates that the approximate bisimulation relation is feasible and effective in behavior and structure optimization.

1. Introduction

With the development of computer software technology and the increase in user requirements for the function of software programs, the structure and behavior of software programs have been growing more and more complex. This change will make the design and property verification processes increasingly difficult. However, if we can achieve a method that can optimize the behavior and structure, then this accomplishment will most likely make a significant difference.

In the process of achieving this method, establishing a formal description is a prerequisite of researching the behavior and structure of the software program. The previous formal descriptions of the software programs are discrete concurrent systems, which are composed of abstract behaviors, discrete states, and transitions between states [1–5]. These descriptions are good at calculating and reasoning against programs that are based on ergodic states. Nevertheless, they cannot successfully express the data exchange processes of the software program. The representations of data exchange processes are diverse for example, they can be described by a polynomial algebraic system, a semialgebraic system, a differential algebraic system, and a differential semialgebraic system [6–9]. Among these systems, research on a particular type of software program whose data exchange processes can be described by polynomial algebraic systems can form a basis for researching other types of programs. Thus, we will study this type of software program first. For a polynomial algebraic system, its representation can also be divided into several different types, for example, a homogeneous linear polynomial system, a nonhomogeneous linear polynomial system, or a nonlinear polynomial system, among others. The software programs that have different behavior and structure correspond to different types of polynomial systems and optimization methods. We have already studied the descriptions of one type of software program, whose data exchange processes can be described by linear polynomial systems, and one type of high-level datapath system, whose data exchange

processes can be expressed by nonlinear polynomial systems, and these are referred to as a linear polynomial transition system and a polynomial transition system, respectively [10, 11]. These descriptions are a basis for proposing an appropriate description of the type of software programs that are researched in this paper, whose data exchange processes can be expressed by nonlinear polynomial systems.

In addition to the above prerequisite, to define and determine the behavior equivalence relation for programs is another precondition to obtain behavior and structure optimization. With equivalence, the equivalent nondeterministic branches are removed, and the complex branch is replaced with a simpler branch. Two software programs are behavior equivalent representations that have the same function. To evaluate equivalence, bisimulation is the best behavior equivalence relation that we can utilize today [12, 13]. If two or more software programs are bisimular, then the solutions of the polynomial systems of their corresponding branches in their transition systems are the same. Based on the nature of bisimulation, researchers have already used it to optimize the design and verification of complex systems, such as hybrid systems and dynamical systems [14, 15]. We have also chosen it to process software programs that are represented by linear polynomial transition systems. For the type of software program researched in this paper, this type of equivalence theory will be established.

Exact equivalence can achieve behavior and structure optimization, but this type of equivalence is too strict to be a good solution for programs in which there are errors in the applications. Take a square with side length 1, for example; the length of its diagonal line is $\sqrt{2}$. Because $\sqrt{2}$ is an irrational number, strictly speaking, we can only make the length of the diagonal line approximately be equal to $\sqrt{2}$. Therefore, there exists an error. For this type of application, exact equivalence is not practical. To strengthen the flexibility of the equivalence relation for different systems, an approximate equivalence within a controllable error range is a good solution. Thus far, approximate bisimulation has been established and applied to optimize the task of property verification for hybrid systems and dynamical systems [16–18]. For software programs that are represented by linear polynomial transition systems, the corresponding approximate relation has been given. Unfortunately, this calculation of an approximate equivalence relation cannot be applied to process the types of software programs that are researched in this paper.

For the calculation of approximate equivalence, symbolic-numeric computation can provide us with technical support [19, 20]. The fact that two software programs are approximate means that the solutions of the polynomial systems of the corresponding branches in their transition systems are approximately the same. It also means that the solutions of one system can make another system approximately be equal to zero. This relationship can be expressed as a constrained global optimization, which is called the MAX function in this paper. In the existing literature, there are many different methods that can be used to solve this constrained global optimization. Among these methods, the full filled method proposed by Ge [21–23] is one of the most widely used. Based on this author's studies, researchers perform some related research and made some improvements [24, 25]. We will use the full filled method in the literature [26] because it has fewer unknown parameters. With this method, the value of the exact error caused by an approximation can be computed.

In the next section, we provide a new formal description, called a nonlinear polynomial transition system, which is used to describe the behavior and structure of the type of software program whose data exchange processes are represented by nonlinear polynomial systems. In Section 3, we give the notion and algorithm of bisimulation after analyzing the condition of equivalence for these software programs. Based on this relation, the notion and the calculation of the algorithm for approximate bisimulation are described in detail in Section 4. The approximate relation is expressed as a constrained global optimization called the MAX function, which is solved by the full filled method. In Section 5, an example of a multithreading program shows that our approach obtains three useful results, which are listed as follows. The last section presents our conclusions.

(i) The data exchange processes of software programs are described.

(ii) The equivalence and approximate relations are determined for software programs based on symbolic-numeric computation.

(iii) The behavior and structure optimization for the software program are achieved.

2. Nonlinear Polynomial Transition System

The definition of a nonlinear polynomial transition system is introduced in this section. This definition is used to describe a type of software program whose data exchange processes can be represented by nonlinear polynomial systems.

This nonlinear polynomial transition system is composed of states and transitions, which provide us with a supportive framework to research the behavior and the structure of software programs. A transition that is expressed by a nonlinear polynomial system shows a part of the data exchange process. A function is a transition. The statements between two functions can also be considered to be transitions.

First, we must establish a transform rule between a part of the data exchange process and a nonlinear polynomial system, which is established as follows.

Algorithm 1. The transform rule between a part of the data exchange process of a software program and a nonlinear polynomial system.

Input: A part of the data exchange process within a software program.

Output: A nonlinear polynomial system.

Step 1. If there are many assignment statements that assign values to the same variable and that variable does not appear on the right side of the assignment statements, then remove the assignment statements except for the last one.

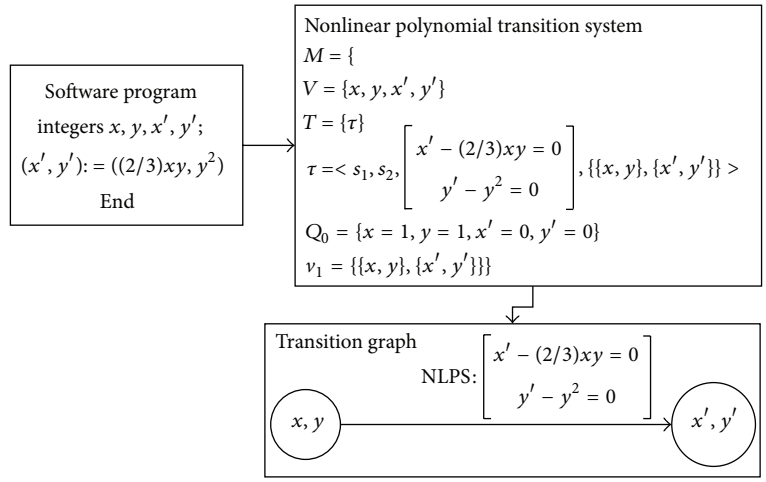

Figure 1: Transform process.

Step 2. Change the symbols of the variables on the left side of the assignment statements by adding intermediate variables after *Step 1*.

Step 3. If a variable on the right side of the assignment statements is the same as a variable on the left side of the assignment statements before *Step 2*, then change the symbol of the variable on the right side of the assignment statements with the symbol of the variable on the left side of the last assignment statement after *Step 2*.

Step 4. Add the assignment statements into NLPS after being processed by *Steps 1*, *2*, and *3*.

Step 5. For the conditional statements, if there are variables that were assigned before them, then first change the symbol of these variables with their symbols on the left side of their last assignment statements after *Step 2*, respectively; then, add these conditional statements into NLPS. Otherwise, add the conditional statements into NLPS directly.

In NLPS, the variables are temporary variables if they can be eliminated. The remainders are global variables, which play an important role in judging the equivalence relations of the software programs. Based on this transform rule, the notion of a nonlinear polynomial transition system for software programs can be given as follows.

Definition 2. A nonlinear polynomial transition system is a tuple $\langle V, S, T, Q_0, v_1\rangle$ that includes

(i) a set V of variables, including temporary variables and global variables;

(ii) a set S of states, which are the values of the variables in V;

(iii) a set T of transitions; each of the transitions is a tuple $\langle s_1, s_2, \text{NLPS}, v_0\rangle$:

(a) s_1 and s_2 are the pre- and postglobal states of a transition;

(b) the transition relation NLPS is a nonlinear polynomial system;

(c) v_0 represents the global variables of this transition;

(iv) Q_0 is the initial state of this software program;

(v) a set v_1 of global variables; the form of v_1 can be defined as follows:

$$\begin{aligned} v_1 &= \{v_{\text{in}}, v_{\text{out}}\} \\ &= \{\{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } n}\}, \\ &\quad \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } n}\}\}, \end{aligned} \tag{1}$$

where v_{in} is a set of global input variables and v_{out} is a set of global output variables. The layer number of the nonlinear polynomial transition system is n. If we consider every nonlinear polynomial transition system to be a tree, then the layer number is considered to be the depth of the tree. Every bracket represents the set of global input or output variables of one branch in this layer.

Corresponding to the form of v_1, we can give a definition of a set of global states s:

$$\begin{aligned} s &= \{s_1, s_2\} \\ &= \{\{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } n}\}, \\ &\quad \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } n}\}\}, \end{aligned} \tag{2}$$

where s_1 is a set of global input states and s_2 is a set of global output states.

To describe the transform process between a software program and a nonlinear polynomial transition system clearly, an example is given in Figure 1. The value of Q_0 is given in advance.

With this transform process, we discover that the behavior and structure of a software program are successfully expressed by a nonlinear polynomial transition system, which can also be represented by a transition graph. Compared with the description of a software program language, the forms of a transition system and transition graph are more direct. Moreover, based on the transform process, the behavior equivalence relation between two data exchange processes of software programs can be described by Definition 3.

Definition 3. The data exchange processes $\langle s_{11}, s_{12}, \mathrm{NLPS}_1, v_{01}\rangle$ and $\langle s_{21}, s_{22}, \mathrm{NLPS}_2, v_{02}\rangle$ of a software program are equivalent; if NLPS_1 and NLPS_2 have solutions in $\mathbf{Q}$, v_{01} and v_{02} are the same, then there is

$$\begin{aligned}&\text{(i) } \forall x_1 \xrightarrow{\mathrm{NLPS}_1} x_2, \exists y_1 = x_1, y_1 \xrightarrow{\mathrm{NLPS}_2} y_2, \text{ and } x_2 = y_2,\\ &\text{(ii) } \forall y_1 \xrightarrow{\mathrm{NLPS}_2} y_2, \exists x_1 = y_1, x_1 \xrightarrow{\mathrm{NLPS}_1} x_2, \text{ and } y_2 = x_2,\end{aligned}$$

where $x_1 \times x_2 \in s_{11} \times s_{12}$, $y_1 \times y_2 \in s_{21} \times s_{22}$, the variables that correspond to $x_1$and $y_1$are the same, and the variables that correspond to x_2 and y_2 are the same. $\mathbf{Q}$ is a rational number field.

Within this definition, stating that the solutions of two nonlinear polynomial systems are the same implies that their corresponding data exchange processes in a software program are behavior equivalent.

After determining the relation for two data exchange processes, how to establish the definition of behavior equivalence for the whole software program must also be researched because there could be many layers and many branches in the same layer in a software program.

3. Bisimulation for Nonlinear Polynomial Transition Systems

In process algebra, equivalence verification is one of the most important parts and bisimulation is the finest equivalence semantic that can provide the most exact equivalence relation for systems. With equivalence determination, the extra transitions, called nondeterminism, can be removed and a complex branch can be replaced with a simple branch.

3.1. The Definition of Bisimulation. Let there be two nonlinear polynomial transition systems:

$$A : \langle V_1, S_1, T_1, Q_{01}, v_{11}\rangle, \qquad B : \langle V_2, S_2, T_2, Q_{02}, v_{12}\rangle. \tag{3}$$

The global state sets of them are

$$\begin{aligned}s_A &= \{s_{1A}, s_{2A}\}\\ &= \{\{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } a}\},\\ &\quad \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } a}\}\},\\ s_B &= \{s_{1B}, s_{2B}\}\\ &= \{\{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } b}\},\\ &\quad \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } b}\}\},\end{aligned} \tag{4}$$

where the symbol s_{ijmn} represents the nth branch of the mth layer in s_{ij}.

The sets of nonlinear polynomial systems of A and B can also be given as follows:

$$\begin{aligned}\mathrm{NLPS}_1 &= \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } a}\},\\ \mathrm{NLPS}_2 &= \{\underbrace{\{\},\ldots,\{\}}_{\text{layer: 1}},\ldots,\underbrace{\{\},\ldots,\{\}}_{\text{layer: } b}\},\end{aligned} \tag{5}$$

where the symbol NPLS_{imn} represents the nth branch of the mth layer in NLPS_i.

Definition 4. A binary relation $\mathbf{R} \subseteq s_A \times s_B$ is called bisimulation; if $v_{11}$and v_{12} are exactly the same, NLPS_{1Alm} and NLPS_{2Blg} have solutions in $\mathbf{Q}$, and $a = b$, then, for all $(x_1 \times y_1) \in \mathbf{R}$, there are

$$\begin{aligned}&\text{(i) } \forall x_1 \xrightarrow{\mathrm{NLPS}_{1Alm}} x_2, \exists y_1 = x_1, y_1 \xrightarrow{\mathrm{NLPS}_{2Blg}} y_2, x_2 \mathbf{R} y_2, \text{ and } x_2 = y_2,\\ &\text{(ii) } \forall y_1 \xrightarrow{\mathrm{NLPS}_{2Blg}} y_2, \exists x_1 = y_1, x_1 \xrightarrow{\mathrm{NLPS}_{1Alm}} x_2, x_2 \mathbf{R} y_2, \text{ and } x_2 = y_2.\end{aligned}$$

Then, A and B are bisimular, with the notation $A =_{BE} B$.

Where $x_1 \in s_{1Alm}$, $x_2 \in s_{2Alm}$, $y_1 \in s_{1Blg}$, $y_2 \in s_{2Blg}$, the variables that correspond to x_1 and y_1 are the same, and the variables that correspond to x_2 and y_2 are the same.

With this definition, we can determine whether there is a bisimulation relation or not between two software programs. If two software programs are bisimular, then their transition systems are bisimular. This fact also means that if there exists a transition for one, there exists the same transition for another and vice versa. This statement represents that the solutions are the same for the polynomial systems of the corresponding branches in the nonlinear polynomial transition system for the software program as well. Therefore, the bisimulation relation between software programs has the following nature.

Theorem 5. *The bisimulation relation for nonlinear polynomial transition systems exists.*

(i) *Reflexivity: $A =_{BE} A$, if there is a nonlinear polynomial transition system A.*

(ii) *Symmetry: $B =_{BE} A$, if $A =_{BE} B$.*

(iii) *Transitivity: $A =_{BE} C$, if $A =_{BE} B$ and $B =_{BE} C$.*

Proof. (i) For reflexivity: the behavior and structure of A are the same as itself; thus, $A =_{BE} A$.

(ii) For symmetry: if $A =_{BE} B$, then the corresponding branches of A and B are equivalent, which implies that

the corresponding branches of B and A are equivalent; thus, $B =_{BE} A$.

(iii) For transitivity: if $A =_{BE} B$, then the corresponding branches of A and B are equivalent.

If $B =_{BE} C$, then the corresponding branches of B and C are equivalent. Therefore, the corresponding branches of A and C are equivalent, which can be expressed with the notation $A =_{BE} C$.

Thus, the bisimulation relation between the software programs is an equivalence relation. □

3.2. The Calculation of Bisimulation. After obtaining the definition of behavior equivalence for a software program, we must establish a method for determining whether the solutions of two nonlinear polynomial systems are the same or not. For the type of software programs studied in this paper, Ritt-Wu's method [27] can be used to determine this solution relation.

Theorem 6. *If C_A and C_B are the same, then the solutions of A and B are the same, which means that A and B are equivalent.*

Proof. Because C_A and C_B are the same, then the initials I_{Ai} and I_{Bi} are the same, which correspond to C_A and C_B, respectively. At the same time,

$$s_A = \bigcup_{i=1}^{e} \text{Zero}\left(C_{Ai}/I_{Ai}\right), \qquad s_B = \bigcup_{i=1}^{e} \text{Zero}\left(C_{Bi}/I_{Bi}\right). \quad (6)$$

The irreducible characteristic set of a polynomial system is unique; thus, there is

$$s_A = \bigcup_{i=1}^{e} \text{Zero}\left(C_{Ai}/I_{Ai}\right) = s_B = \bigcup_{i=1}^{e} \text{Zero}\left(C_{Bi}/I_{Bi}\right). \quad (7)$$

Then, A and B are equivalent where A and B are nonlinear polynomial systems. $C_A = [C_{A1}, \ldots, C_{Ae}]$ and $C_B = [C_{B1}, \ldots, C_{Be}]$ are the irreducible characteristic sets of A and B, respectively. Here, s_A and s_B are the solutions of A and B, respectively.

Therefore, a unique condition that is used to determine the solution relation of the nonlinear polynomial systems is obtained.

This determination in the equivalence relation is based on symbolic computation. From analysis, we discover that numerical computation can also be used in this process, because if the solutions of the two polynomial systems are the same, then the solutions of one system can make another system be equal to zero. In other words, if we change one system into a special formula, the solutions of another system can approximately make the maximum value of this formula be equal to zero. This process can be considered to be a constrained global optimization. Because there are approximate processes while we are addressing this constrained global optimization problem, we will therefore implement this process in the next section. □

4. Approximate Bisimulation for Nonlinear Polynomial Transition Systems

Exact equivalence requires that the solutions of two polynomial systems must be identical, which appears to be too restrictive and not practical for some software programs. An appropriate approximation within a controllable error range can be a good solution for addressing this issue. By approximation, different software programs might be changed into approximately the same program. Then, the simpler programs can be used to approximately replace the complex programs. Therefore, behavior and structure approximate optimization can be achieved.

4.1. The Definition of Approximate Bisimulation. Suppose that there are two nonlinear polynomial transition systems. The descriptions of the variables of these two systems are the same as the descriptions in Section 3.1.

Definition 7. The data exchange processes $\langle s_{11}, s_{12}, \text{NLPS}_1, v_{01}\rangle$ and $\langle s_{21}, s_{22}, \text{NLPS}_2, v_{02}\rangle$ of a software program are approximate; if NLPS_1 and NLPS_2 have solutions in $\mathbf{Q}$, v_{01} and v_{02} are the same, and $\varepsilon \le \delta$, then there is

(i) $\forall x_1 \xrightarrow{\text{NLPS}_1} x_2, \exists y_1 \xrightarrow{\text{NLPS}_2} y_2$ and $\exists x_1 \xrightarrow{\text{NLPS}_2}_{\varepsilon} x_2$, then, $y_1 =_{\varepsilon} x_1 \cup y_1$, $y_2 =_{\varepsilon} x_2 \cup y_2$.

(ii) $\forall y_1 \xrightarrow{\text{NLPS}_2} y_2, \exists x_1 \xrightarrow{\text{NLPS}_1} x_2$ and $\exists y_1 \xrightarrow{\text{NLPS}_1}_{\varepsilon} y_2$, then, $x_1 =_{\varepsilon} x_1 \cup y_1$, $x_2 =_{\varepsilon} x_2 \cup y_2$,

where $x_1 \times x_2 \in s_{11} \times s_{12}$, $y_1 \times y_2 \in s_{21} \times s_{22}$, the variables that correspond to x_1 and y_1 are the same, and the variables that correspond to x_2 and y_2 are the same. Here,

$$m =_{\varepsilon} m \cup n \quad (8)$$

represents that we place n into m to form a new m with an error ε. Then,

$$m \xrightarrow{\text{NLPS}}_{\varepsilon} n \quad (9)$$

represents that m and n are the approximate solutions of NLPS with an error ε. Here, ε is the exact error and δ is the precision.

This definition shows that two data exchange processes are approximate if the solutions of their corresponding nonlinear polynomial systems are approximate. Based on this definition, the approximate relation for the entire software program can be given as follows.

Definition 8. A binary relation $\mathbf{R}_{\varepsilon} \subseteq s_A \times s_B$ is called an approximate bisimulation; if v_{11} and v_{12} are the same, NLPS_{1Alm} and NLPS_{2Blg} have solutions in $\mathbf{Q}$, $a = b$, and $\varepsilon \le \delta$, then, for all $(x_1 \times y_1) \in \mathbf{R}_{\varepsilon}$, there are

(i) $\forall x_1 \xrightarrow{\text{NLPS}_{1Alm}} x_2$, $\exists y_1 \xrightarrow{\text{NLPS}_{2Blg}} y_2$, and $\exists x_1 \xrightarrow{\text{NLPS}_{2Blg}}_{\varepsilon_{l(mg)}} x_2$; then, $y_1 =_{\varepsilon_{l(mg)}} x_1 \cup y_1$, $y_2 =_{\varepsilon_{l(mg)}} x_2 \cup y_2$, and $x_2 \mathbf{R}_{\varepsilon} y_2$.

(ii) $\forall y_1 \xrightarrow{\text{NLPS}_{2Blg}} y_2$, $\exists x_1 \xrightarrow{\text{NLPS}_{1Alm}} x_2$, and $\exists y_1 \xrightarrow{\text{NLPS}_{1Alm}}{}_{\varepsilon_{l(mg)}} y_2$; then, $x_1 =_{\varepsilon_{l(mg)}} x_1 \cup y_1$, $x_2 =_{\varepsilon_{l(mg)}} x_2 \cup y_2$, and $x_2 \mathbf{R}_\varepsilon y_2$.

Then, A and B are approximately bisimular, with the notation $A =_{AB} B$.

Where $x_1 \in s_{1Alm}$, $x_2 \in s_{2Alm}$, $y_1 \in s_{1Blg}$, $y_2 \in s_{2Blg}$, the variables that correspond to x_1 and y_1 are the sam, and the variables that correspond to x_2 and y_2 are the same.

Here, $\varepsilon_{l(mg)}$ represents the error between NPLS_{1Alm} and NPLS_{2Blg}.

For two different nonlinear polynomial systems, the error between them can be calculated by the function norm. For the nonlinear polynomial transition system researched in this paper, there are always many branches and layers; thus, a measurement rule for the exact error must be given.

Step 1. For different branches in the same layer, compute the errors of every two of them first. Then, choose the largest one as the error in that layer.

Step 2. For the whole nonlinear polynomial transition system, compute the error of every layer with Step 1; then, the total error of the system is found by taking the sum.

With the previous definition, software programs that are approximately bisimular are those for which the solutions of the polynomial systems of the corresponding branches in their nonlinear polynomial transition systems are approximately the same. In other words, if we change a nonlinear polynomial system into a special formula, the solutions of another system can make the maximum value of this formula approximately be equal to zero. This process can be considered to be a constrained global optimization and its description is given as follows.

Given two nonlinear polynomial systems, as follows:

$$\begin{aligned} p_1 = 0, \quad p_2 = 0; \\ f_1 = 0, \quad f_2 = 0, \quad f_3 = 0. \end{aligned} \tag{10}$$

Then, change them into two constrained global optimizations, called MAX functions, as follows

$$\begin{aligned} &\max \quad \sqrt{p_1^2 + p_2^2} \\ &\text{s.t.} \quad f_1 = 0 \\ &\qquad f_2 = 0 \\ &\qquad f_3 = 0; \\ &\max \quad \sqrt{f_1^2 + f_2^2 + f_3^2} \\ &\text{s.t.} \quad p_1 = 0 \\ &\qquad p_2 = 0, \end{aligned} \tag{11}$$

which can also be expressed as follows:

$$\begin{aligned} &\sqrt{\max\left(p_1^2 + p_2^2\right)} \\ &\text{s.t.} \quad f_1 = 0 \\ &\qquad f_2 = 0 \\ &\qquad f_3 = 0; \\ &\sqrt{\max\left(f_1^2 + f_2^2 + f_3^2\right)} \\ &\text{s.t.} \quad p_1 = 0 \\ &\qquad p_2 = 0. \end{aligned} \tag{12}$$

The results of these two constrained global optimizations fall into three categories.

(i) If the two maximum values are equal to zero, then the two nonlinear polynomial systems are equivalent. Thus, the determination of bisimulation between two nonlinear polynomial transition systems in Section 3.2 can be solved based on numerical computation.

(ii) If the two maximum values are not more than the precision, then two nonlinear polynomial systems are approximate. Therefore, the determination of approximate bisimulation between two nonlinear polynomial transition systems is achieved.

(iii) If one of the two maximum values is more than the precision, then two nonlinear polynomial systems are not approximate.

The maximum value can be achieved by the full filled method. In this process, the maximum number of iterations and the step length of variables must be given in advance.

4.2. The Calculation of Approximate Bisimulation. For a software program, we hope to optimize its behavior and structure through approximately optimizing its formal description; this approach is called a nonlinear polynomial transition system with approximate bisimulation.

When the variables of the corresponding branches in polynomial transition system are the same, the framework of approximate bisimulation of a software program contains four parts:

(i) change the software program into a nonlinear polynomial transition system;

(ii) group the corresponding branches of the transition system;

(iii) determine the relations of corresponding branches by symbolic-numeric computation;

(iv) achieve the behavior and structure optimization by reducing the equivalent or approximate branches for the given program.

The detailed computation algorithm of approximate bisimulation is shown as follows, which is described in Figure 2.

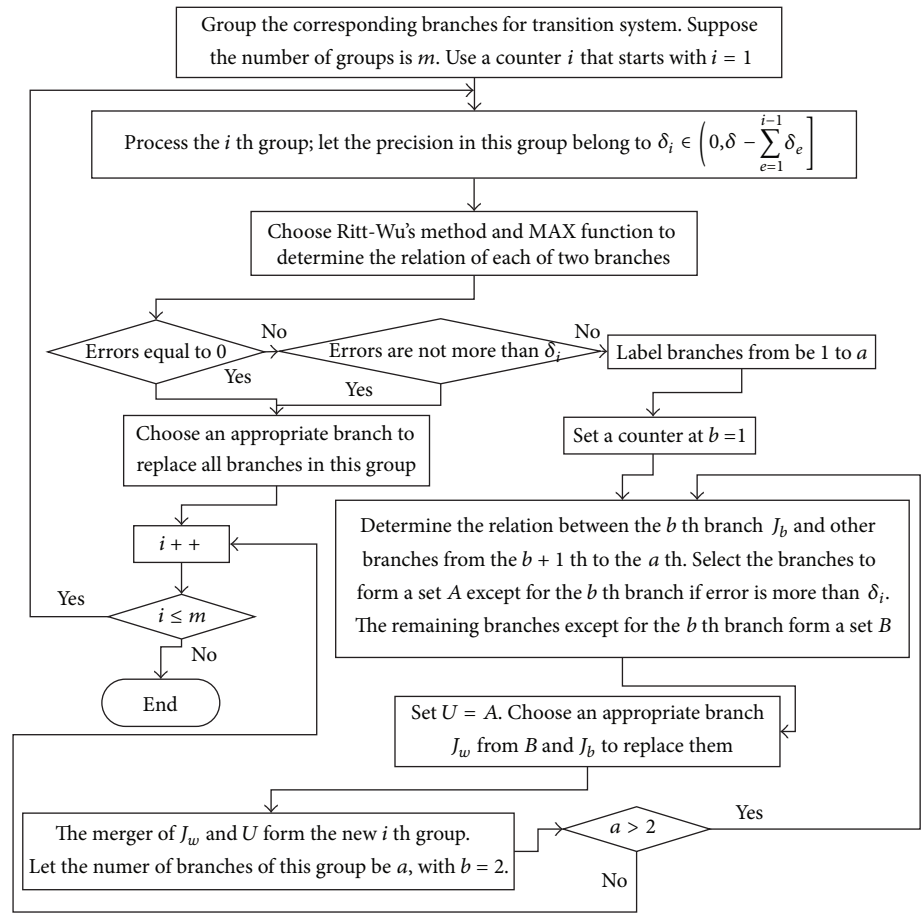

Figure 2: Approximate bisimulation for the nondeterministic branches of a software program.

Algorithm 9. The calculation algorithm of approximate bisimulation for the nondeterministic branches of a software program based on symbolic-numeric computation.

Input: A software program expressed by a nonlinear polynomial transition system.

Output: An optimized software program for the given software program.

Step 1. Change the software program into a nonlinear polynomial transition system. Let δ be the precision. Group the corresponding branches with bisimulation relations. Suppose that the number of groups is m. We are given a counter i that starts with $i = 1$.

Step 2. Process the ith group. Let the precision δ_i in this group belong to the following:

$$\left(0, \delta - \sum_{e=1}^{i-1} \delta_e\right]. \tag{13}$$

Step 3. Choose Ritt-Wu's method and the MAX function to determine the relation of each of the two branches in this group.

Step 3.1. If irreducible characteristic sets are the same and the errors are equal to zero, then these branches are equivalent. Choose the simplest branch to replace them and go to *Step 5*. Otherwise, go to *Step 3.2*.

Step 3.2. If the errors are not more than δ_i, then these branches are approximate. Then, choose an appropriate branch to replace them and go to *Step 5*. Otherwise, go to *Step 3.3*.

Step 3.3. Let the number of branches be a, and label them from 1 to a Initiate a counter b at $b = 1$ and go to *Step 4*.

Step 4. Determine the relation between the bth branch and the other branches from the $(b + 1)$th to the ath branch. Let the symbol of the nonlinear polynomial system that corresponds to the jth branch be J_j. Let

$$\begin{aligned} \varepsilon_{b+1} &= \left\| J_b - J_{b+1} \right\|, \\ \varepsilon_{b+2} = \left\| J_b - J_{b+2} \right\|, \ldots, &\qquad \varepsilon_a = \left\| J_b - J_a \right\|. \end{aligned} \tag{14}$$

Select J_j except for J_b to form a set A when the errors are larger than δ_i. The remainder except for J_b forms a set B; go to *Step 4.1*.

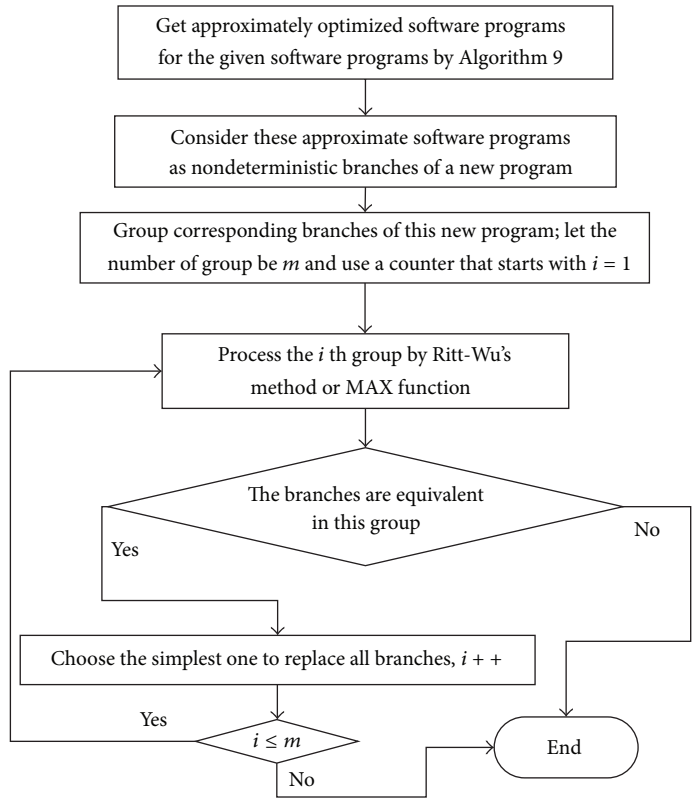

Figure 3: Approximate bisimulation for software programs.

Step 4.1. Let $U = A$. Choose an appropriate element from B and J_b to replace them. Let the symbol of this element be J_w.

Step 4.2. The merger of U and J_w forms a new group, which is to be considered as the new ith group. Let the number of branches of this new group be a, with $b = 2$. If $a > 2$, then go to *Step 4*; otherwise, go to *Step 5*.

Step 5. Increment i with $i++$; if $i \leq m$, then go to *Step 2*; otherwise, end. A new polynomial transition system is achieved, which also means that a new optimized software program is got.

This algorithm gives a method to optimize the behavior and structure of a software program with approximate bisimulation.

If there are several different software programs, then we must know how to implement the determination of an approximate bisimulation for them.

The framework of approximate bisimulation of different software programs contains five parts:

(i) optimize software programs by Algorithm 9 respectively;

(ii) consider the optimized programs to be the nondeterministic branches of a new program;

(iii) group the corresponding branches of the new program;

(iv) determine the relations of corresponding branches by symbolic-numeric computation;

(v) get the behavior and structure optimizations for the given programs by equivalence.

The detailed algorithm is also given as follows and is described by Figure 3.

Algorithm 10. The calculation algorithm of approximate bisimulation for software programs based on symbolic-numeric computation.

Input: Different software programs that are represented by nonlinear polynomial transition systems.

Output: Optimized software programs for the given software programs.

Step 1. Get the approximately optimized software programs for the given software programs by Algorithm 9. Then, consider these approximately optimized programs as the nondeterministic branches of a new software program.

Step 2. Group the corresponding branches of this new program. Let the number of groups be m and use a counter i that starts with $i = 1$.

Step 3. If there are equivalent branches in the ith group as determined by Ritt-Wu's method or the MAX function, then go to *Step 4*. Otherwise, go to *Step 6*.

Step 4. Choose the simplest one of them to replace them.

Step 5. Increment i with $i++$; if $i \leq m$, then go to *Step 3*. Otherwise, go to *Step 6*.

Step 6. End. Achieve the last optimized software programs of the given programs.

5. Example Verification and Results

In this section, a multithreading program is used to demonstrate the efficiency of approximate bisimulation in the behavior and structure optimization of software programs.

5.1. Example Verification. The multithreading program is represented in Algorithm 1.

Let $\delta = 0.1$. The maximum number of iterations is $NL = 10$ and the step length of variables is $d = 0.5$ in the full filled method. The recommended values of the variables are in the range from −1 to 1.

Step 1. With the substitution rules and transform rules, change the multithreading program into a nonlinear polynomial transition system, as shown in Figure 4.

Step 2. Because the variables of the corresponding branches are the same, there is a need to determine whether there is an equivalence or approximate relation or not of the corresponding polynomial systems in the multithreading program, which is accomplished as follows:

$$\{\mathrm{NLPS}_1, \mathrm{NLPS}_2\}; \quad \{\mathrm{NLPS}_3, \mathrm{NLPS}_4, \mathrm{NLPS}_5\}; \quad \{\mathrm{NLPS}_6, \mathrm{NLPS}_7, \mathrm{NLPS}_8, \mathrm{NLPS}_9\}. \tag{15}$$

Step 2.1. For $\{\mathrm{NLPS}_1, \mathrm{NLPS}_2\}$: Ritt-Wu's method is applied to verify the relations. Then, we achieve the irreducible characteristic sets as follows:

$$\begin{aligned} C_1 &= \left\{m_2^2 - x_1, 2m_1m_2 + x_2\right\}; \\ C_2 &= \left\{m_2^2 - x_1, 2m_1m_2 + x_2\right\}. \end{aligned} \tag{16}$$

The irreducible characteristic sets are equivalent. Moreover, the behavior of NLPS_2 is simpler; thus, the branch that corresponds to NLPS_2 can be used to replace and merge the two branches that correspond to NLPS_2 and NLPS_1.

Step 2.2. For $\{\mathrm{NLPS}_3, \mathrm{NLPS}_4, \mathrm{NLPS}_5\}$.

First, Ritt-Wu's method is applied to determine the relation for $\{\mathrm{NLPS}_3, \mathrm{NLPS}_5\}$. Then, we can obtain the irreducible characteristic sets:

$$\begin{aligned} C_3 &= [C_{31}, C_{32}] \\ &= \left[\left\{y_1^2 + 3x_1 + x_2, -2y_1y_2 + x_2\right\}, \{x_1, x_2, y_1\}\right]; \\ C_5 &= [C_{51}, C_{52}] \\ &= \left[\left\{y_1^2 + 3x_1 + x_2, -2y_1y_2 + x_2\right\}, \{x_1, x_2, y_1\}\right]. \end{aligned} \tag{17}$$

Table 1: Results.

Number of iterations: K	Initial states	Solutions	Minimum
$K = 1$	(1, 1, 1, 1)	(0.06, 0.82, 1, 0.82)	$-9.08e-005$
$K = 2$	(0.04, 0.86, 1, 0.86)	(0.05, 0.86, 1, 0.86)	$-9.17e-005$
$K = 3$	(0.05, 0.85, 1, 0.85)	(0.05, 0.85, 1, 0.85)	$-9.19e-005$
$K = 4$	(0.04, 0.84, 1, 0.85)	(0.05, 0.84, 1, 0.85)	$-8.97e-005$
$K = 5$	(0.04, 0.85, 1, 0.84)	(0.05, 0.85, 1, 0.85)	$-9.13e-005$

Table 2: Results.

Number of iterations: K	Initial states	Solutions	Minimum
$K = 1$	(1, 1, 1, 1)	(0.05, 0.84, 1, 0.84)	$-1.00e-004$

Therefore, they are equivalent. The behavior of NLPS_5 is simpler; thus, the branch that corresponds to NLPS_5 is used to replace and merge the two branches that correspond to NLPS_5 and NLPS_3.

For NLPS_4 and NLPS_5, the MAX function is chosen to determine the relation for them.

Step 2.2.1. With the full filled method, calculate

$$\begin{aligned} \min \quad & -\left(3x_1 - 0.99y_1^2 + y_2y_1 - 0.01y_2x_1\right)^2 \\ & - \left(x_2 - y_2y_1 + 0.01y_2x_1\right)^2 \\ \text{s.t.} \quad & 3x_1 - y_1^2 + x_2 = 0; \\ & x_2 - y_2y_1 = 0. \end{aligned} \tag{18}$$

The results are showed in Table 1.

In this process, the solutions of the objective function are the same when $K = 3$ and $K = 5$. The solutions will be duplicated from $K = 5$; thus, the trend in the objective function will become smooth. Then, the result of

$$\max\left(\left(3x_1 - 0.99y_1^2 + y_2y_1 - 0.01y_2x_1\right)^2 + \left(x_2 - y_2y_1 + 0.01y_2x_1\right)^2\right)^{1/2} \tag{19}$$

is 0.0096.

Step 2.2.2. With the full filled method, calculate

$$\begin{aligned} \min \quad & -\left(x_1 - y_1^2 + x_2\right)^2 - \left(x_2 - y_2y_1\right)^2 \\ \text{s.t.} \quad & 3x_1 - 0.99y_1^2 + y_2y_1 - 0.01y_2x_1 = 0; \\ & x_2 - y_2y_1 + 0.01y_2x_1 = 0. \end{aligned} \tag{20}$$

The results are showed in Table 2.

In this process, the solutions will be duplicated from $K = 1$. Thus, the trend in the objective function becomes smooth. The result of

$$\max \sqrt{\left(x_1 - {y_1}^2 + x_2\right)^2 + \left(x_2 - y_2y_1\right)^2} \tag{21}$$

is 0.01.

```
#include “main.h”
WINMULTITHREAD Branch2Thread5(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
 double y1 = tharg->il, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((y1∗y1 - (0.01∗y2 + 1)∗z1) > -APPZERO &&
 (y1∗y1 - (0.01∗y2 + 1)∗z1) < APPZERO) &&
 ((2∗y1∗y2 - z2 - 0.01) > -APPZERO &&
 (2∗y1∗y2 - z2 - 0.01) < APPZERO)) {
 printf(“one solution of HLPS9 is z1 = %f, z2 = %f\n”, z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch2Thread4(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
 double y1 = tharg->i1, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((y1∗y1 - z1) > -APPZERO &&
 (y1∗y1 - z1) < APPZERO) &&
 ((2∗y1∗y2 - z2) > -APPZERO &&
 (2∗y1∗y2 - z2) < APPZERO)) {
 printf(“one solution of NLPS8 is z1 = %f, z2 = %f\n”, z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch1Thread3(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
 double y1 = tharg->i1, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((-(y1∗y2 + 1)∗(y1∗y2 + 1) + 3∗z1) > -APPZERO &&
 (-(y1∗y2 + 1)∗(y1∗y2 + 1) + 3∗z1) < APPZERO) &&
 ((-z1 + z2∗(y1∗y2 + 1)) > -APPZERO &&
 (-z1 + z2∗(y1∗y2 + 1)) < APPZERO)) {
 printf(“one solution of NLPS6 is z1 = %f, z2 = %f\n”, z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch1Thread4(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
```

Algorithm 1: Continued.

```
 double y1 = tharg->i1, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((-z1 + (y1∗y2 + 1)∗z2) > -APPZERO &&
 (-z1 + (y1∗y2 + 1)∗z2) < APPZERO) &&
 ((-(y1∗y2 + 1)∗(y1∗y2 + 1) + 2∗z1 + z2∗(y1∗y2 + 1)) >
 -APPZERO &&
 (-(y1∗y2 + 1)∗(y1∗y2 + 1) + 2∗z1 + z2∗(y1∗y2 + 1)) <
 APPZERO)) {
 printf("one solution of NLPS7 is z1 = %f, z2 = %f\n", z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch1Thread2(LPVOID pParam)
{HANDLE hThread13 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg12 = {0};
 double x1 = tharg->i1, x2 = tharg->i2;
 double y1 = MINIMUMVALUE, y2 = MINIMUMVALUE;
 do {
 do {
 if (((3∗x1 - y1∗y1 + y2∗y1) > -APPZERO &&
 (3∗x1 - y1∗y1 + y2∗y1) < APPZERO) &&
 ((x2 - y2∗y1) > -APPZERO &&
 (x2 - y2∗y1) < APPZERO)) {
 printf("one solution of NLPS3 is y1 = %f, y2 = %f\n", y1, y2);
 tharg12.i1 = y1;
 tharg12.i2 = y2;
 hThread13 = CreateThread(NULL, 0, Branch1Thread3,
                                    (LPVOID)&tharg12, 0, NULL);
 WaitForSingleObject(hThread13, INFINITE);
 CloseHandle(hThread13);
 }
 y2 += GRANULARITY;
 }while (y2 <= MAXIMUMVALUE);
 y1 += GRANULARITY;
 y2 = MINIMUMVALUE;
 } while (y1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch2Thread3(LPVOID pParam)
{HANDLE hThread25 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg23 = {0};
 double x1 = tharg->i1, x2 = tharg->i2;
 double y1 = MINIMUMVALUE, y2 = MINIMUMVALUE;
 do {
 do {
 if (((3∗x1 - y1∗y1 + x2) > -APPZERO &&
 (3∗x1 - y1∗y1 + x2) < APPZERO) &&
 ((x2 - y2∗y1) > -APPZERO &&
 (x2 - y2∗y1) < APPZERO)) {
 printf("one solution of NLPS5 is y1 = %f, y2 = %f\n", y1, y2);
 tharg23.i1 = y1;
```

Algorithm 1: Continued.

```
 tharg23.i2 = y2;
 hThread25 = CreateThread(NULL, 0, Branch2Thread5,
                          (LPVOID)&tharg23, 0, NULL);
 WaitForSingleObject(hThread25, INFINITE);
 CloseHandle(hThread25);
 }
 y2 += GRANULARITY;
 }while (y2 <= MAXIMUMVALUE);
 y1 += GRANULARITY;
 y2 = MINIMUMVALUE;
 } while (y1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch2Thread2(LPVOID pParam)
{HANDLE hThread24 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg22 = {0};
 double x1 = tharg->i1, x2 = tharg->i2;
 double y1 = MINIMUMVALUE, y2 = MINIMUMVALUE;
 do {
 do {
 if (((3∗x1 - 0.99∗y1∗y1 + y2∗y1 - 0.01∗y2∗x1) >
 -APPZERO &&
 (3∗x1 - 0.99∗y1∗y1 + y2∗y1 - 0.01∗y2∗x1) <
 APPZERO) &&
 ((x2 - y2∗y1 + 0.01∗y2∗x1) > -APPZERO &&
 (x2 - y2∗y1 + 0.01∗y2∗x1) < APPZERO)) {
 printf("one solution of NLPS4 is y1 = %f, y2 = %f\n", y1, y2);
 tharg22.i1 = x1;
 tharg22.i2 = x2;
 hThread24 = CreateThread(NULL, 0, Branch2Thread4,
                          (LPVOID)&tharg22, 0, NULL);
 WaitForSingleObject(hThread24, INFINITE);
 CloseHandle(hThread24);
 }
 y2 += GRANULARITY;
 }while (y2 <= MAXIMUMVALUE);
 y1 += GRANULARITY;
 y2 = MINIMUMVALUE;
 } while (y1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch2Thread1(LPVOID pParam)
{HANDLE hThread22 = NULL, hThread23 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg21 = {0};
 double m1 = tharg->i1, m2 = tharg->i2;
 double x1 = MINIMUMVALUE, x2 = MINIMUMVALUE;
 do {
 do {
 if (((m2∗m2 - x1) > -APPZERO &&
 (m2∗m2 - x1) < APPZERO) &&
 ((2∗m1∗m2 + x2) > -APPZERO &&
 (2∗m1∗m2 + x2) < APPZERO)) {
 printf("one solution of NLPS2 is x1 = %f, x2 = %f\n", x1, x2);
 tharg21.i1 = x1;
 tharg21.i2 = x2;
 hThread22 = CreateThread(NULL, 0, Branch2Thread2,
                          (LPVOID)&tharg21, 0, NULL);
 hThread23 = CreateThread(NULL, 0, Branch2Thread3,
```

Algorithm 1: Continued.

```
                                    (LPVOID)&tharg21, 0, NULL);
 WaitForSingleObject(hThread22, INFINITE);
 WaitForSingleObject(hThread23, INFINITE);
 CloseHandle(hThread22);
 CloseHandle(hThread23);
 }
 x2 += GRANULARITY;
 }while (x2 <= MAXIMUMVALUE);
x1 += GRANULARITY;
 x2 = MINIMUMVALUE;
 } while (x1 <= MAXIMUMVALUE);
return 0;
}
WINMULTITHREAD Branch1Thread1(LPVOID pParam)
{HANDLE hThread12 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg11 = {0};
 double m1 = tharg->i1, m2 = tharg->i2;
 double x1 = MINIMUMVALUE, x2 = MINIMUMVALUE;
 do {
 do {
 if (((2*m1*m2 + m2*m2 - x1 + x2) > -APPZERO &&
 (2*m1*m2 + m2*m2 - x1 + x2) < APPZERO) &&
 ((2*m1*m2 - m2*m2 + x1 + x2) > -APPZERO &&
 (2*m1*m2 - m2*m2 + x1 + x2) < APPZERO)) {
 printf("one solution of NLPS1 is x1 = %f, x2 = %f\n", x1, x2);
 tharg11.i1 = x1;
 tharg11.i2 = x2;
 hThread12 = CreateThread(NULL, 0, Branch1Thread2,
                                    (LPVOID)&tharg11, 0, NULL);
 WaitForSingleObject(hThread12, INFINITE);
 CloseHandle(hThread12);
 }
 x2 += GRANULARITY;
 }while (x2 <= MAXIMUMVALUE);
 x1 += GRANULARITY;
 x2 = MINIMUMVALUE;
 } while (x1 <= MAXIMUMVALUE);
return 0;
}
int main()
{HANDLE hThread1 = NULL, hThread2 = NULL;
 LARGE_INTEGER litc, litStart, litEnd;
 ThreadArg iTharg = {0};
 printf("Please enter the value of input m1 = ");
 scanf("%lf", &iTharg.i1);
 printf("Please enter the value of input m2 = ");
 scanf("%lf", &iTharg.i2);
 QueryPerformanceFrequency(&litc);
 QueryPerformanceCounter(&litStart);
 hThread1 = CreateThread(NULL, 0, Branch1Thread1,
                                  (LPVOID)&iTharg, 0, NULL);
 hThread2 = CreateThread(NULL, 0, Branch2Thread1,
                                   (LPVOID)&iTharg, 0, NULL);
 WaitForSingleObject(hThread1, INFINITE);
 WaitForSingleObject(hThread2, INFINITE);
 QueryPerformanceCounter(&litEnd);
 printf("Execution time is %fs\n", (double)
         (litEnd.QuadPart - litStart.QuadPart)/litc.QuadPart);
 return 0;
}
```

Algorithm 1: Multithreading program.

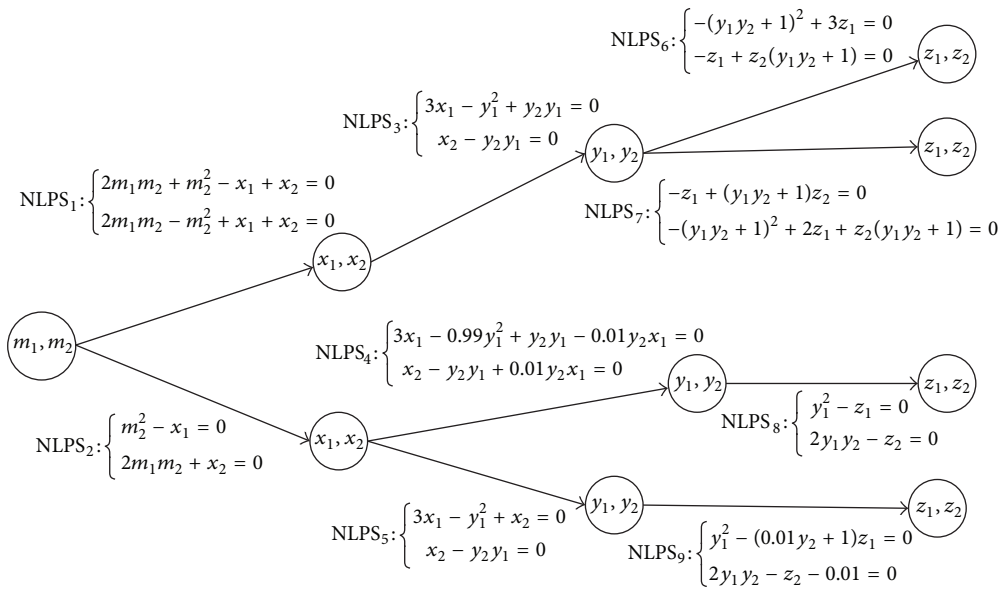

Figure 4: Transition graph.

Table 3: Results.

Number of iterations: K	Initial state	Solution	Minimum
$K = 1$	(1, 1, 1, 1)	(0.91, 0.56, 0.82, 1)	$-1.21e-004$
$K = 2$	(0.9, 0.54, 0.83, 0.98)	(0.91, 0.54, 0.82, 0.98)	$-1.20e-004$

The above two maxima are less than the precision; therefore, NLPS_4 and NLPS_5 are approximate. The branch that corresponds to NLPS_5 can replace and merge the branches that correspond to NLPS_3, NLPS_4, and NLPS_5.

Step 2.3. For $\{\mathrm{NLPS}_6, \mathrm{NLPS}_7, \mathrm{NLPS}_8, \mathrm{NLPS}_9\}$.

First, the Ritt-Wu's method is applied to verify the relation for NLPS_6 and NLPS_7. Then, we obtain the irreducible characteristic sets as follows:

$$\begin{aligned} C_6 &= [C_{61}, C_{62}] \\ &= \left[\left\{y_1^2 y_2^2 + 2y_1y_2 + 1 - 3z_1, y_1y_2 - 3z_2 + 1\right\}, \right. \\ &\qquad \left. \{y_1y_2 + 1, z_1\}\right]; \\ C_7 &= [C_{71}, C_{72}] \\ &= \left[\left\{y_1^2 y_2^2 + 2y_1y_2 + 1 - 3z_1, y_1y_2 - 3z_2 + 1\right\}, \right. \\ &\qquad \left. \{y_1y_2 + 1, z_1\}\right]. \end{aligned} \tag{22}$$

Therefore, NLPS_6 and NLPS_7 are equivalent. Because NLPS_6 is simpler, the branch that corresponds to NLPS_6 is used to replace and merge the two branches that correspond to NLPS_7 and NLPS_6.

For NLPS_8 and NLPS_9, the MAX function is used to determine the relation for them.

Step 2.3.1. With the full filled method, calculate

$$\begin{aligned} \min \quad & -\left(y_1^2 - z_1\right)^2 - \left(2y_1y_2 - z_2\right)^2 \\ \text{s.t.} \quad & y_1^2 - \left(0.01y_1 + 1\right)z_1 = 0; \\ & 2y_1y_2 - z_2 - 0.01 = 0. \end{aligned} \tag{23}$$

The results are showed in Table 3.

The results will be duplicated from $K = 2$. At this point, the trend in the objective function becomes smooth. Thus, the result of

$$\max \sqrt{\left(y_1^2 - z_1\right)^2 + \left(2y_1y_2 - z_2\right)^2} \tag{24}$$

is 0.011.

Step 2.3.2. With the full filled method, calculate

$$\begin{aligned} \min \quad & -\left(y_1^2 - \left(0.01y_1 + 1\right)z_1\right)^2 - \left(2y_1y_2 - z_2 - 0.01\right)^2 \\ \text{s.t.} \quad & y_1^2 - z_1 = 0; \\ & 2y_1y_2 - z_2 = 0. \end{aligned} \tag{25}$$

The results are showed in Table 4.

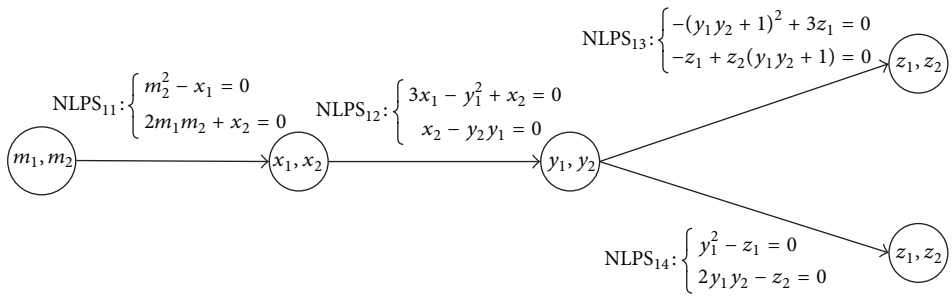

Figure 5: Approximate transition graph.

Table 4: Results.

Number of iterations: K	Initial state	Solution	Minimum
$K = 1$	(1, 1, 1, 1)	(0.91, 0.55, 0.82, 1)	$-1.21e-004$
$K = 2$	(0.92, 0.55, 0.84, 0.99)	(0.92, 0.54, 0.84, 0.99)	$-1.21e-004$

Table 5: Results.

Number of iterations: K	Initial state	Solution	Minimum
$K = 1$	(1, 1, 1, 1)	(0.5, 1, 0.25, 1)	−3.81

There are duplicate minimums when $K = 1$ and $K = 2$. Thus, the trend in the objective function becomes smooth. Then, the result of

$$\max \sqrt{\left(y_1^2 - \left(0.01y_1 + 1\right)z_1\right)^2 + \left(2y_1y_2 - z_2 - 0.01\right)^2} \tag{26}$$

is 0.011.

The precision of this layer is $0.7 - 0.01 \approx 0.69$. The achieved two maxima are less than this precision; thus, NLPS_8 and NLPS_9 are approximate. The branch that corresponds to NLPS_8 is used to replace and merge the two branches that correspond to NLPS_8 and NLPS_9.

Step 2.3.3. For NLPS_8 and NLPS_6, calculate

$$\begin{aligned} \min \quad & -\left(-\left(y_1y_2 + 1\right)^2 + 3z_1\right)^2 - \left(z_2\left(y_1y_2 + 1\right) - z_1\right)^2 \\ \text{s.t.} \quad & y_1^2 - z_1 = 0; \\ & 2y_1y_2 - z_2 = 0. \end{aligned} \tag{27}$$

The results are showed in Table 5.

Therefore, the first maximum of

$$\sqrt{\left(-\left(y_1y_2 + 1\right)^2 + 3z_1\right)^2 + \left(z_2\left(y_1y_2 + 1\right) - z_1\right)^2} \tag{28}$$

is 1.95, which is more than the precision in this layer; thus, NLPS_8 and NLPS_6 are not approximate.

Step 2.4. By analysis, the transition graph in Figure 4 is turned into a new transition graph, which is shown in Figure 5, and the optimized multithreading program for the given multithreading program in Algorithm 1 also can be expressed in Algorithm 2.

5.2. Results. By the example verification in this section, many advantages of our method are achieved.

(i) Achieve a formal description: software programs are represented by nonlinear polynomial transition systems and transition graphs that contain states and transitions, such as in Figures 4 and 5. With these descriptions, all of the data exchange processes, the behavior, and the structure of the software program are described more directly.

(ii) With Ritt-Wu's method and MAX function, the equivalence and approximate relations are determined for the software program based on symbolic-numeric computation.

  (a) With Ritt-Wu's method and MAX function, the equivalence and approximate relations are expressed by the "standard forms" and "constrained global optimizations" without calculating the exact values of the states of the software programs. The traditional method must know the exact values of the inputs and outputs to determine the relations for the different software programs. Unfortunately, the exact values of the variables of a software program always exist in a range so that the traditional method will cost too much computing time. Thus, our method is more flexible and effective.

  (b) Based on these relations, the behavior and structure optimization of software programs are achieved and are compared, whereby Figure 4 is compared with Figure 5 and Algorithm 1 is compared with Algorithm 2. For example, the number of branches in Algorithm 1 is 9 and the number of branches in Algorithm 2 is 4.

```
#include "main.h"
WINMULTITHREAD Branch1Thread3(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
 double y1 = tharg->i1, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((-(y1*y2 + 1)*(y1*y2 + 1) + 3*z1) > -APPZERO &&
 (-(y1*y2 + 1)*(y1*y2 + 1) + 3*z1) < APPZERO) &&
 ((-z1 + z2*(y1*y2 + 1)) > -APPZERO &&
 (-z1 + z2*(y1*y2 + 1)) < APPZERO)) {
 printf("one solution of NLPS13 is z1 = %f, z2 = %f\n", z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
 return 0;
}
WINMULTITHREAD Branch1Thread4(LPVOID pParam)
{PThreadArg tharg = (PThreadArg)pParam;
 double y1 = tharg->i1, y2 = tharg->i2;
 double z1 = MINIMUMVALUE, z2 = MINIMUMVALUE;
 do {
 do {
 if (((y1*y1 - z1) > -APPZERO &&
 (y1*y1 - z1) < APPZERO) &&
 ((2*y1*y2 - z2) > -APPZERO &&
 (2*y1*y2 - z2) < APPZERO)) {
 printf("one solution of NLPS14 is z1 = %f, z2 = %f\n", z1, z2);
 }
 z2 += GRANULARITY;
 }while (z2 <= MAXIMUMVALUE);
 z1 += GRANULARITY;
 z2 = MINIMUMVALUE;
 } while (z1 <= MAXIMUMVALUE);
 return 0;
}
WINMULTITHREAD Branch1Thread2(LPVOID pParam)
{HANDLE hThread13 = NULL, hThread14 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg targ12 = {0};
 double x1 = tharg->i1, x2 = tharg->i2;
 double y1 = MINIMUMVALUE, y2 = MINIMUMVALUE;
 do {
 do {
 if (((3*x1 - y1*y1 + x2) > -APPZERO &&
 (3*x1 - y1*y1 + x2) < APPZERO) &&
 ((x2 - y2*y1) > -APPZERO &&
 (x2 - y2*y1) < APPZERO)) {
 printf("one solution of NLPS12 is y1 = %f, y2 = %f\n", y1, y2);
 targ12.i1 = y1;
 targ12.i2 = y2;
 hThread13 = CreateThread(NULL, 0, Branch1Thread3,
                    (LPVOID)&targ12, 0, NULL);
 hThread14 = CreateThread(NULL, 0, Branch1Thread4,
  (LPVOID)&targ12, 0, NULL);
 WaitForSingleObject(hThread13, INFINITE);
 WaitForSingleObject(hThread14, INFINITE);
```

Algorithm 2: Continued.

```
 CloseHandle(hThread13);
 CloseHandle(hThread14);
 }
 y2 += GRANULARITY;
 }while (y2 <= MAXIMUMVALUE);
 y1 += GRANULARITY;
 y2 = MINIMUMVALUE;
 } while (y1 <= MAXIMUMVALUE);
 return 0;
}
WINMULTITHREAD Branch1Thread1(LPVOID pParam)
{HANDLE hThread12 = NULL;
 PThreadArg tharg = (PThreadArg)pParam;
 ThreadArg tharg11 = {0};
 double m1 = tharg->i1, m2 = tharg->i2;
 double x1 = MINIMUMVALUE, x2 = MINIMUMVALUE;
 do {
 do {
 if (((m2∗m2 - x1) > -APPZERO &&
 (m2∗m2 - x1) < APPZERO) &&
 ((2∗m1∗m2 + x2) > -APPZERO &&
 (2∗m1∗m2 + x2) < APPZERO)) {
 printf("one solution of NLPS11 is x1 = %f, x2 = %f\n", x1, x2);
 tharg11.i1 = x1;
 tharg11.i2 = x2;
 hThread12 = CreateThread(NULL, 0, Branch1Thread2,
                 (LPVOID)&tharg11, 0, NULL);
 WaitForSingleObject(hThread12, INFINITE);
 CloseHandle(hThread12);
 }
 x2 += GRANULARITY;
 }while (x2 <= MAXIMUMVALUE);
 x1 += GRANULARITY;
 x2 = MINIMUMVALUE;
 } while (x1 <= MAXIMUMVALUE);
 return 0;
}
int main()
{LARGE_INTEGER litc, litStart, litEnd;
 HANDLE hThread1 = NULL;
 ThreadArg iTharg = {0};
 printf("Please enter the value of input m1 = ");
 scanf("%lf", &iTharg.i1);
 printf("Please enter the value of input m2 = ");
 scanf("%lf", &iTharg.i2);
 QueryPerformanceFrequency(&litc);
 QueryPerformanceCounter(&litStart);
 hThread1 = CreateThread(NULL, 0, Branch1Thread1,
                 (LPVOID)&iTharg, 0, NULL);
 WaitForSingleObject(hThread1, INFINITE);
 QueryPerformanceCounter(&litEnd);
 printf("Execution time is %fs\n", (double)
    (litEnd.QuadPart - litStart.QuadPart)/litc.QuadPart);
 return 0;
}
```

Algorithm 2: Approximate multithreading program.

Table 6: Results.

Nonlinear polynomial systems	Solutions	Computing costs
$NLPS_1$	(0.0001, −0.0002)	6.789s
$NLPS_2$	(0.0001, −0.0002)	4.878s
$\{NLPS_1, NLPS_2, NLPS_3, NLPS_4, NLPS_5\}$	{(0.0001, −0.0002), (0.0001, −0.0002), {(−0.01, 0.02), (0.01, −0.02)}, {(−0.01005, 0.0199), (0.01005, −0.0199)}, {(−0.01, 0.02), (0.01, −0.02)}}	12.041s
$\{NLPS_{11}, NLPS_{12}\}$	{(0.0001, −0.0002), {(−0.01, 0.02), (0.01, −0.02)}}	9.678s

(c) Because the states and branches of the software programs are reduced with these relations, the number of feasible paths for the software program is decreased; therefore, the "state explosion" problem can be alleviated in the property verification and the occupancy factor of having a deadlock becomes small.

(iii) The value of the behavior and the structure optimization of the software program is divided into two parts, as follows.

(a) Behavior optimization: for example, for $\{NLPS_1, NLPS_2\}$ in Figure 4, $NLPS_1$ is replaced by $NLPS_2$ in Figure 5. If we allow the inputs of the software programs that correspond to $NLPS_1$ and $NLPS_2$ to be equal to 0.0, and let the step length of variable in this process be equal to 0.00005, then the computing costs are 6.789 s and 4.878 s when calculating the same outputs, respectively. Therefore, the simpler the behavior is, the smaller the computing cost. The solutions of the variables and the computing costs of the software programs are given in Table 6.

(b) Structure optimization: for example, the set $\{NLPS_1, NLPS_2, NLPS_3, NLPS_4, NLPS_5\}$ in the given nonlinear polynomial transition system is reduced to two branches $\{NLPS_{11}, NLPS_{12}\}$ in the approximate system. If we let the initial inputs of $NLPS_1$ and $NLPS_{11}$ be equal to 0.01 in the programs that correspond to $\{NLPS_1, NLPS_2, NLPS_3, NLPS_4, NLPS_5\}$ and $\{NLPS_{11}, NLPS_{12}\}$ and let the step length of variable in this process be equal to 0.00005, then the computing costs are 12.041 s and 9.678 s when calculating the approximate outputs of the software programs, respectively. Thus, the fewer the number of branches that are present, the lower the computing cost. The solutions of the variables and the computing costs of the software programs are given in Table 6.

(iv) The approximate bisimulation in this paper can also be used to process the type of software programs whose data exchange processes are expressed by linear polynomial systems [11].

(a) In the calculation algorithms for approximate bisimulation for linear polynomial transition systems, the entire process is a linear computation that is based on the properties of the matrix in paper [11]. However, the whole process in determining the approximate bisimulation relation of the software programs is a nonlinear computation in this paper because the MAX function is applied to compute the value of the errors caused by the approximation. Therefore, the computing process in [11] requires less time and is more effective than the calculation proposed in this paper for linear polynomial systems.

(b) Unfortunately, the method in [11] cannot be applied to process the nonlinear polynomial system researched in this paper because the property of the matrix cannot describe the solutions' relations of the nonlinear polynomial systems.

6. Conclusions

In this paper, we propose a nonlinear polynomial transition system to describe the type of software program whose data exchange processes are expressed by nonlinear polynomial systems. Then, a novel calculation of an approximate bisimulation is provided to achieve the behavior and the structure optimization for the software programs, based on symbolic-numeric computation. The advantages of our approach are verified and listed in Section 5.

Except for those advantages that are achieved by our approach in this paper, on the one hand, how to improve the efficiency of symbolic-numeric computation is a substantial challenge. This goal will be researched in our future work. On the other hand, the numbers of the states and branches of the software program are reduced due to the behavior and the structure optimization. How to apply these circumstances to optimize the property verification of the software program based on the state, such as safety verification, is also a major direction for our research in the future.

Acknowledgments

This work was supported by the Natural Science Foundation of Guangxi under Grant nos. 2011GXNSFA018154 and 2012GXNSFGA060003, the Science and Technology Foundation of Guangxi under Grant no. 10169-1, and Guangxi Scientific Research Project no. 201012MS274.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[2] R. van Glabbeek and U. Goltz, "Equivalence notions for concurrent systems and refinement of actions," in *Mathematical Foundations of Computer Science*, vol. 379 of *Lecture Notes in Computer Science*, pp. 237–248, Springer, Berlin, Germany, 1989.

[3] M. Nielsen, "Models for concurrency," in *Mathematical Foundations of Computer Science*, vol. 520 of *Lecture Notes in Computer Science*, pp. 43–46, 1991.

[4] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 278–292, 1996.

[5] R. Milner and D. Sangiorgi, "Barbed bisimulation," in *Automata, Languages and Programming*, vol. 623 of *Lecture Notes in Computer Science*, pp. 685–695, 1992.

[6] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*, Springer, New York, NY, USA, 1995.

[7] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using groebner bases," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 318–329, 2004.

[8] Y. H. Chen, B. C. Xia, L. Yang, N. J. Zhan, and C. C. Zhou, "Discovering non-linear ranking functions by solving semi-algebraic systems," in *Proceedings of the Theoretical Aspects of Computing (ICTAC '07)*, vol. 4711 of *Lecture Notes in Computer Science*, pp. 34–49, 2007.

[9] V. P. Mysore, *Algorithmic algebraic model checking: hybrid automata and systems bilogy [Ph.D. thesis]*, New York University, New York, NY, USA, 2006.

[10] H. Deng, J. Z. Wu, and N. Zhou, "Approximate equivalence and optimization for high-level datapath," *Journal of Information and Computational Science*, vol. 8, no. 16, pp. 4131–4142, 2011.

[11] H. Deng and J. Z. Wu, "On approximate bisimulation equivalence for linear semi-algebraic transition systems," *Journal of Jilin University*, vol. 43, no. 4, pp. 1052–1058, 2013.

[12] R. J. Van Glabbeek, "The linear time-branching time spectrum," in *Proceedings of the Theories of Concurrency: Unification and Extension (CONCUR '90)*, vol. 458 of *Lecture Notes in Computer Science*, pp. 278–297, 1990.

[13] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Information and Computation*, vol. 94, no. 1, pp. 1–28, 1991.

[14] G. J. Pappas, "Bisimilar linear systems," *Automatica*, vol. 39, no. 12, pp. 2035–2047, 2003.

[15] A. J. van der Schaft, "Equivalence of dynamical systems by bisimulation," *Institute of Electrical and Electronics Engineers*, vol. 49, no. 12, pp. 2160–2172, 2004.

[16] A. Girard and G. J. Pappas, "Approximate bisimulations for nonlinear dynamical systems," in *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference (CDC-ECC '05)*, pp. 684–689, December 2005.

[17] A. A. Julius, "Approximate abstraction of stochastic hybrid automata," in *Hybrid Systems: Computation and Control*, vol. 3927 of *Lecture Notes in Computer Science*, pp. 318–332, Springer, Berlin, Germany, 2006.

[18] A. Girard and G. J. Pappas, "Approximate bisimulation relations for constrained linear systems," *Automatica*, vol. 43, no. 8, pp. 1307–1317, 2007.

[19] H. J. Stetter, *Numerical Polynomial Algebra*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pa, USA, 2004.

[20] D. Bates, C. Peterson, and A. J. Sommese, "A numerical-symbolic algorithm for computing the multiplicity of a component of an algebraic set," *Journal of Complexity*, vol. 22, no. 4, pp. 475–489, 2006.

[21] R. P. Ge, "A filled function method for finding a global minimizer of a function of several variables," *Mathematical Programming*, vol. 46, no. 2, pp. 191–204, 1990.

[22] R. P. Ge and Y. F. Qin, "A class of filled functions for finding global minimizers of a function of several variables," *Journal of Optimization Theory and Applications*, vol. 54, no. 2, pp. 241–252, 1987.

[23] R. P. Ge and Y. F. Qin, "The globally convexized filled functions for global optimization," *Applied Mathematics and Computation*, vol. 35, no. 2, pp. 131–158, 1990.

[24] X. Liu, "Finding global minima with a computable filled function," *Journal of Global Optimization*, vol. 19, no. 2, pp. 151–161, 2001.

[25] L.-S. Zhang, C.-K. Ng, D. Li, and W.-W. Tian, "A new filled function method for global optimization," *Journal of Global Optimization*, vol. 28, no. 1, pp. 17–43, 2004.

[26] Y. L. Zhang, *The filled function method for solving constrained global optimization [Ph.D. thesis]*, Tianjin University, 2007.

[27] D. M. Wang, *Selected Lectures in Symbolic Computation*, Tsinghua University Press, Beijing, China, 2003.