Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 49–67
www.elsevier.com/locate/entcs

An Improvement of Software Architecture Verification

Zuohua Ding [1]

Center of Math Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, 310018, P.R.China

Jing Liu [2,3]

Shanghai Key Lab of Trustworthy Computing
East China Normal University
Shanghai, 200062, P.R.China

Abstract

Static analysis may cause state space explosion problem. In this paper we explore differential equation model that makes the task of verifying software architecture properties much more efficient. We demonstrate how ordinary differential equations can be used to verify application-specific properties of an architecture description without hitting this problem. An architecture behavior can be modeled by a group of ordinary differential equations containing some control parameters, where the control parameters are used to represent deterministic/nondeterministic choices. Each equation describes the state change. By checking the conditions associated with the control parameters, we can check whether an equation model is feasible. After solving a feasible equation model, based on the solution behavior and the state variable representation, we can analyze properties of the architecture. A WRIGHT architecture description of the Gas Station problem has been used as the example to illustrate our method. All of the equations have been computed with Matlab tool.

Keywords: Static analysis; architecture; ordinary differential equation; Wright.

1 Introduction

Static analysis is an approach to program behavior verification without execution. The approach is particularly useful in identifying program design errors prior to implementation. It has been demonstrated that detecting errors early in the lifecycle greatly reduces the cost of fixing those errors. A number of static analysis techniques

[1] Email:zuohuading@hotmail.com
[2] Email:jliu@sei.ecnu.edu.cn
[3] Corresponding author

doi:10.1016/j.entcs.2009.07.005

have been proposed. They span such approaches as reachability-based analysis techniques, symbolic model checking, flow equations, and dataflow analysis.

These techniques and approaches have been used in several analysis tools such as *Flow equation* is used in INCA [5], *data flow analysis* is used in FLAVERS [11], *Reachability Analysis* is used in SPIN [14], *Symbolic model checking* is used in SMV [4] and SMC [21].

In general all existing approaches appear to be very sensitive to the size of the program being analyzed in terms of the use of concurrency constructs and the number of asynchronous processes. Particularly, reachability analysis may cause state space explosion problem since it has to exhaustively explore all the reachable state space to detect concurrency errors. Although many techniques have been proposed to combat this explosion, such as state space reductions, compositional techniques, abstraction, the state explosion problem still is the main technical obstacle to transition from research to practice.

Current concurrent systems are described as discrete event system models which are suited to model system concurrency. However, the discrete will lead to state explosion problem since model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [3]. Hence, to thoroughly solve the state explosion problem, one solution is that the discrete event system model should be continunized to continuous system model, such that the systems can be described with analytic expressions. Therefore, instead of counting states, we can analyze the solutions of the analytic expressions.

Petri net seems a good candidate that bridges discrete event systems and continuous systems. On one hand, Petri nets have been used extensively as tools for the modeling, analysis and synthesis of discrete event systems. Petri nets offer advantages over finite automata, particularly when the issues of model complexity and concurrency of processes are of concern. On the other hand, a continuous system can be approximated by a Petri net [20] and a Petri net model is used as discrete event representation of the continuous variable system by Lunze et al. [17].

However, Petri nets also suffer from the state explosion problem while doing reachability analysis[18] even through there are some net reduction methods. One way to tackle that problem is to use some kind of relaxation by removing the integrality constraints. This relaxation leads to a continuous-time formalism: Continuous Petri Net (CPN) by David and Alla[6][7]. A continuous Petri net, in fact, is an approximation of the timed (discrete) Petri net. The semantics of a continuous Petri net is defined by a set of ordinary differential equations (ODEs), where one equation describes the continuous changes over time on the marking value of a given place. Different firing styles in the CPN can lead to different semantics of CPN. In this paper, we consider a modified VCPNs in which the instantaneous firing speeds depend on the markings such that the markings are continuous without points of discontinuity.

Based on CPN, a concurrent system can be modeled by a group of ordinary differential equations containing some control parameters, where the control param-

eters are used to represent deterministic/nondeterministic choices. Each equation describes the state change, which indicates that the state can be reached to some extent when the program is in execution. By checking the conditions associated with the control parameters, we can check whether an equation model is feasible. After solving a feasible equation model, based on the solution behavior and the state variable representation, we can analyze properties of the architecture. All equation groups will be solved by Matlab.

The primary goal of our work is to investigate the applicability of our skill for verifying application-specific properties of architectures. We investigate one example architecture, a WRIGHT description of the gas station problem, and illustrate the kinds of properties that can be verified and the kinds of errors that can be found early in the lifecycle. Since our architecture is described with CSP, we first translate CSP to Petri net and then build a differential equation model in order to analyze the state space.

This paper is organized as the following. Section 2 specifies a Gas-station with Wright. Section 3 builds Petri net model from Wright specification. Section 4 builds differential equation model based on Petri net. In Section 5, we show how to compute state measures. Section 6 is property analysis based on the solutions. In Section 7, we demonstrate how our method can be used to check the properties of Gas-station. The last section, Section 8, is the conclusion and discussion of the paper.

2 Specification of The Gas-Station Using Wright

According to [1], the Architectural Specification Language, Wright, was developed to provide a formal basis for specifying both the structure and behavior of architectural descriptions. Wright is built around the basic architectural abstractions of components, connectors, and configurations. Each component is defined by a component type description which consists of an interface and a computation. An interface further contains a number of ports. Each port represents an interaction with the environment that the component may participate in, while the computation describes the internal behavior of the component. Connectors in Wright define patterns of interaction between components. Each connector consists of the glue and a set of roles. Roles indicate the constraints on the components that will participate in the interaction. Glue is actually the counterpart of the computation in components. Wright uses CSP [13] to formalize the system behaviors.

The Gas-Station problem has been widely studied for property analysis, specially deadlock analysis. Generally, the automated gas station consists of a set of operators, a set of pumps and a set of customers. The scenario is as follows: customers continuously arrive at the gas station requesting a certain amount of gas from one of the pumps chosen at random. Each customer must go to an available cashier and pay for the gas before being allowed to pump it. If all the cashiers are serving customers, a customer who needs to pay must wait for the next available

cashier. After the gas is paid for, the cashier will activate the customer's pump. If all the pumps are being used, the customer has to wait until a pump is free. In our study case, we consider a simplified instance of the Gas-Station problem, which consists of two customers, one cashier and one gas pump. A Wight specification is illustrated in the following.

```
Component Casher
    Port Customer1 = accept?x→ Customer1
    Port Customer2 = accept?x→ Customer2
    Port ToPump = activate!x → ToPump
    Computation = customer1.accept?x→ ToPump.activate!x →
                  Computation □ Customer2.accept?x→
                  ToPump.activate!x → Computation
Component Pump
    Port Hose1 = take?x→ Hose1
    Port Gas1 = pump!x → Gas1
    Port Hose2 = take?x→ Hose2
    Port Gas2 = pump!x → Gas2
    Port FromCasher = get_order?x → FromCasher
    Computation = FromCasher.get_order?x →
                  ((Hose1.take?x→ Gas1.pump!x → computation )
                  □(Hose2.take?x→ Gas2.pump!x →
                  computation))
Component Customer
    Port Pay = pay!x → Pay
    Port Take = take!x → Take
    Port Gas = pump!x → Gas
    Computation = Pay.pay!x → Take.take!x →
                  Gas.pump?x→Computation
Component Customer_Casher
    Role Givemoney = pay!x → Givemoney
    Role Getmoney = accept?x → Getmoney
    Glue = Givemoney.pay?x→ Getmoney.accept!x Glue
Component Customer_Pump_Hose
    Role Gethose = take!x → Gethose
    Role Givehose = take?x → Givehose
    Glue = Gethose.take?x→ Givehose.take!x Glue
Component Customer_Pump_Gas
    Role Getgas = pump?x → Getgas
    Role Givegas = pump!x → Givegas
    Glue = Givegas.pump?x→ Getgas.pump!x Glue
Component Casher_Pump
    Role Tell = activate!x → Tell
    Role Ack = get_order?x → Ack
    Glue = Tell.activate?x→ Ack.get_order!x Glue

Instances
    customer1: Customer
    customer2: Customer
    casher: Casher
    pump: Pump
    (omitted)


Attachments

    (omitted)
```

We created four component instances and seven connector instances under the Instances section. The cashier module has a total of three communication ports (Customer1, Customer2 and ToPump). Two are inputs (Customer1 and Customer2, depicted by the CSP "?" operator) and the remaining one is output communication (ToPump, depicted by the CSP "!" operator). The internal action of this cashier module is specified as the Computation of the cashier com-

ponent, which nondeterministically (with the CSP □ operator) selects one out of two possible execution paths: rendezvous with the input communication from port Customer1 then the output communication to port ToPump (which is the "$Customer1.accept?x \rightarrow ToPump.activate!x$" branch), or rendezvous with the input communication from port Customer2 then the output communication to port ToPump (which is the "$Customer2.accept?x \rightarrow ToPump : activate!x$" branch). All other component and connector specifications can be interpreted similarly.

As in [19], we will check two specifications of the architecture. The first one is a critical race, in which one customer pays for gas and the second customer then pays and takes the pump before the first customer gets gas. The second specification is that no customer receives gas without paying for it.

3 Building a Petri Net Model of the Gas Station

In this section, we will explain how to map the Wright specification of a system into a Petri-net model. We need some general concepts which can be found in [16].

Hierarchy and modules: The basic Petri-net model does not support hierarchy and composition explicitly. As in most ADL specifications, a whole system is normally composed of components and connectors, which are most suitably described as submodules of a whole system. To support this, we introduce the module concept into the Petri-net model. We can map each component or connector into a small Petri-net module (sub-graph). With all the modules, together with the IO-nets concept below, we can model the whole system.

IO-nets, IO-places and IO-transitions: The concept of IO-nets comes from the idea of modularity in software design. An IO-net is a combination of a Petri-net and an interface. The interface specifies the way that an IO-net (module) interacts with its environments. It can represent a set of asynchronous communication channels. The interfaces of IO-nets are represented using IO-places together with the related arcs as illustrated in Figure 1. IO-transitions are those transitions that have outgoing arcs to, or incoming arcs from, IO-places.

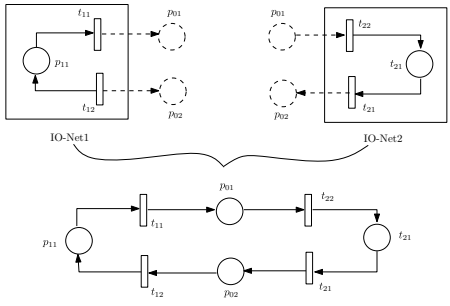

Fig. 1. Example of IO net.

Internal transitions: All the Petri-net transitions that have no outgoing arcs to, or incoming arcs from, IO-places are defined as internal transitions.

The basic idea of the mapping algorithm is to map each component instance to a submodule of a Petri-net model and each atomic connector instance to an IO-place.

The atomic connector here refers to the connectors whose Glue part only contains the actions defined in the Role part. If there are internal computation steps other than simply carrying data between Role parts, which make the connectors non-atomic, we can map the computation steps of these connectors into sub-modules in the same way as components. The basic unit of a CSP behavior specification is an event which represents an action in Wright. Thus, each event in Wright can be mapped into a transition in a Petri-net. Actions carrying data can be mapped to IO-transitions. Output actions (e!x) contain outgoing arcs to IO-places while input actions (e?x) contain incoming arcs from IO-places. Actions not carrying data can be mapped into an internal transition. Once transitions have been fixed, places can be added between transitions to connect the whole Petri-net together based on the specifications in the Computation part. Normally, each Port of the component instance can be mapped into an aggregation of transitions which contains at least one IO-transition. In most cases, the Port contains no internal transition, so the Port can be mapped into an IO-transition because such Ports normally contain only one input/output action. For detailed rules, we refer to [16].

This mapping is actually a conversion of CSP operators into Petri-net transition structures. A set of conversion rules have been defined to cover the most common CSP operations [22]. Under these rules, the properties of the architecture, such as deadlock, will be reserved.

In this way, we get Petri nets such that each transition has at most two input arcs and at most two output arcs. We have the following definitions.

Definition 3.1 *A Place/Transition Chain is a net such that transitions are connected by a head place that has one output arc and no input arc, an end place that has one input arc and no output arc, and places that have one input arc and one output arc. If the head place and the end place are overlapping, then the chain is called Place/Transition Cycle.*

Definition 3.2 *A place/transition cycle is called Process Cycle, if every transition in the cycle is 1) a transition that has one input arc and one output arc; this transition is called an Internal Transition of the process, 2) a transition that has one input arc and two output arcs; this transition is called an Output Transition of the process, here one output arc is to construct the cycle and the other is for the output of the cycle, 3) a transition that has two input arcs and one output arc; this transition is called an Input Transition of the process, here one input arc is to construct the cycle and the other is for the input of the cycle, 4) a transition that has two input arcs and two output arcs; this transition is called an Input-Output Transition of the process, here one input arc and one output arc are used to construct the cycle and the other two are used for the input and output of the cycle, respectively.*

Thus, each component contains one or many process cycles depending on whether the component contains no or some select controls.

The complete Petri-net model of the Gas-Station problem is shown in Figure 2.

While building a Petri net, we may obtain some "valued-oriented" constructs such as parameters, variables that define the dynamic state of a program. We use

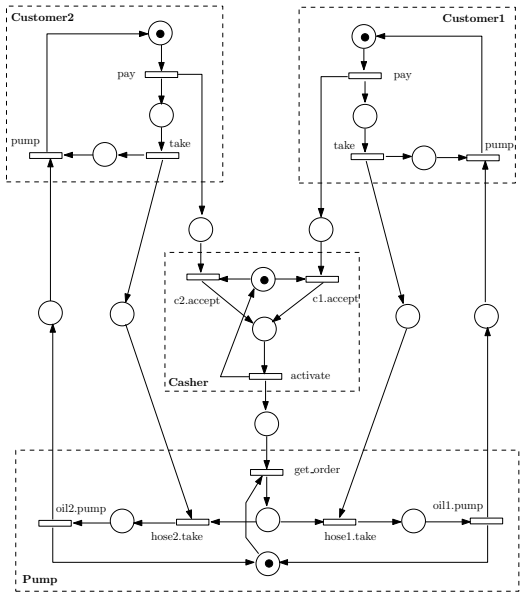

Fig. 2. Gas station net.

special variables, called *state variables*, to record them at each place such that when the system is executed from one place to another place, *state variable*'s values will be changed. In this paper, these variables are extracted from control structures such as *if/else, switch*, etc. In the Gas-Station example, we may have two variable, x_1 and x_2, that represent who pays the money and who picks the hose, respectively. Thus each variable has the domain {customer1, customer2}. We have the following definition.

Definition 3.3 *(System State) A system state is defined as a group of variables whose values have changed.*

Thus, each place corresponds to a state.

4 Building A Model Using Ordinary Differential Equation

4.1 Continuous Petri Net

The Petri net obtained in the last section is discrete Petri net, in which the number of marks in the places are integers. A transition is enabled if each input place of the transition is marked with a token. An enabled transition fires by removing a token from each input place and adding a token to each output place.

Now check the following example to find out how the data is processed. As shown in Figure 3(a), a process cycle has places $p_1, p_2, \ldots$ and has an input place p_i at transition t_1. We assume that place p_1 has a token, meaning that the process is visiting this place, and p_i has 3 tokens, meaning that there are three data in the buffer.

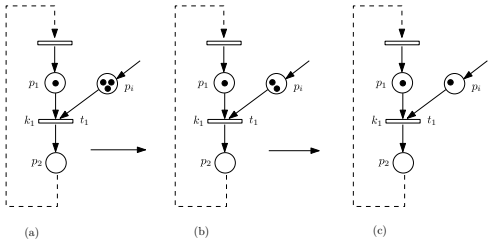

Fig. 3. Marking changes in discrete Petri net.

The process has to visit place p_1 for 3 times to move away all 3 tokens in the place p_i. In other words, 3×1 tokens will be moved from place p_1 as shown in Figure 3(b)(c). Thus the tokens in the input place p_i can be regarded as an impact factor while tokens are moved from the process place p_1. If the number of data in the buffer is big, and a program has many such buffers, then we will get large number of reacheable markings which could limit the use of discrete Petri nets.

Now we assume that the marking is moving as a continuous flow, then the marking moving rate can be regarded as the product of $m_1(t) \times m_i(t)$, where $m_1(t)$ and $m_i(t)$ are the markings of p_1 and p_i at time t, respectively. This can be pictured in Fig. 4.

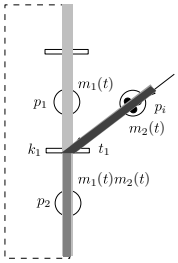

Fig. 4. Continuous flow in Petri net.

Based on this idea, we propose a new continuous Petri net model. In this model, the instantaneous firing speed of a transition is proportional to the product of the markings of the input places.

Definition 4.1 *A Continuous Petri Net is a tuple* $CPN =< P, T, A_{pre}, A_{post}, v >$, *where*

(i) $P = \{p_1, p_2, ..., p_n\}$ *is a finite nonempty set of places,*

(ii) $T = \{t_1, t_2, ..., t_m\}$ *is a finite nonempty set of transitions,*

(iii) $A_{pre} = \{p \to t\}$ *is a set of directed arcs which connect places with transitions,* $A_{post} = \{t \to p\}$ *is a set of directed arcs which connect transitions to places,*

(iv) $v : T \to (0, \infty)$ *is a mapping to assign a firing rate to each transition.*

Definition 4.2 *Let* $I = [0, \infty)$ *be the time interval and let* $m_i : I \to [0, \infty), i = 1, 2, \ldots, n$ *be a set of mappings that associated with place* p_i. *A marking of a Continuous Petri Net* $CPN =< P, T, A_{pre}, A_{post}, v >$ *is a mapping*

$$m : I \to [0, \infty)^n,\ m(\tau) = (m_1(\tau), m_2(\tau), \ldots, m_n(\tau)).$$

Definition 4.3 *A marked CPN is a 2-tuple* (N, M_0) *where*

- N is a CPN,

- $M_0 = (m_1(0), m_2(0), \ldots, m_n(0))$ *is its initial marking, where* $m_i(0)$ *takes value 1 or 0.*

A place holding initial marking 1 is called *start place*. The marking of a place can be used to measure how often this place has been visited. We have the definition:

Definition 4.4 *(State Measure) Given any time moment* $t \in [0, \infty)$*, the state can be reached to some degree. This degree is called State Measure, denoted as* $m(t)$*. State measures take nonnegative real numbers as their values.*

Later, we will prove that the states of each process cycle take values from [0,1]. For a state s, if $m(t) = 1$, then we say that the program is completely in the state s, or simply in the state s. If $m(t) = 0$, then we say that the program is not in the state s.

All the m_i defined above are the state measures. So, if new marking is moved into a place, we say that the state is increasing; if some marking is moved out from a place, we say that the state is decreasing. The change rate of state measure can be calculated as the following.

Let p_1 and p_2 be the input places of a transition t and their markings are $m_1(\tau)$ and $m_2(\tau)$, respectively. Let v be the firing rate associated with t, then the mark moving rate from each place is defined as the product $v * m_1(\tau) * m_2(\tau)$, where $*$ represents the regular multiplication. This expression contains the enabling information: if one of m_1 and m_2 is zero, then the firing rate is 0, meaning the transition is not enabled. Our definition magnify the states, which is useful when we study the state trend. Our definition is to make the state marking differential, thus the state change is continuous without points of discontinuity.

Gilbert and Heiner [12] have successfully used the similar continuous Petri net model to study biochemical systems to explore possible observable behaviors, where the firing rates of all the atomic actions is the product of the concentrations of the involved substances. Here concentrations are continuous functions, which are the state measure functions in our paper.

Definition 4.5 *A stationary state of a marked CPN is a state where all transitions are firing.*

4.2 Building A Differential Equation Model

The net marking (state measure) change depends on the program structures and the firing rates. Based on the semantics defined in the above section, the marking at each place can be represented by a differential equation. We consider choice structure here. For other cases, please see [9][8] for details.

1) Deterministic choice as shown in Figure 5.

Let m be a place that can move the marking to t_2 or t_3 deterministically based on some conditions. We assign a variable k, called *control parameter*, to m to assist building equations. k is associated with some condition. If condition is true, then

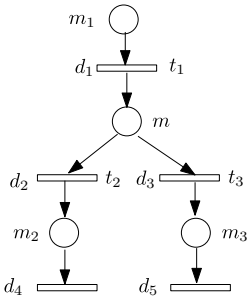

Fig. 5. Deterministic choice.

k takes on the value 1, otherwise k is 0. Assume that the condition is true from m to t_2, then the input of transition t_1 is km, while the input of t_3 is $(1-k)m$. The equation model for this net is

$$\begin{cases} m' = d_1 m_1 - k d_2 m - (1-k) d_3 m, \\ m_2' = km - d_2 m_2, \\ m_3' = (1-k)m - d_3 m_3. \end{cases}$$

2) Nondeterministic choice as shown in Figure 6. Let m be a place that can move a marking to t_1 or t_2 nondeterministically. We also assign a *control parameter* k to m to assist building equations. But this time no condition is associated with the control parameter. k takes value 0 or 1. In this picture, the input of transition t_1 of the first process is $m_1 km$, while the input of t_3 of the second process is $m_3(1-k)m$.

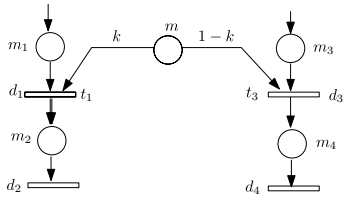

Fig. 6. Nondeterministic choice.

The corresponding differential equations are:

$$\begin{cases} m_1' = ()_1 - d_1 m_1 km, \\ m_2' = d_1 m_1 km - d_2 m_2, \\ m_3' = ()_2 - d_3 m_3 (1-k)m, \\ m_4' = d_3 m_3 (1-k)m - d_4 m_4. \end{cases}$$

Here we use () to represent some other state measures. If more than two processes are involved in sharing m, say 3 processes p_1, p_2 and p_3, then we will have two parameters: k_1 and k_2. Let t_1, t_3 and t_5 be the input transitions that need resource. The inputs to the transitions are: $()_1 * k_1 * m$, $()_2 * (1-k_1) * k_2 * m$, and $()_3 * (1-k_1) * (1-k_2) * m$, respectively.

All the parameters in the system form a vector, $(k_1, k_2, \ldots, k_n)$, called *control*

vector. By selecting the values of $k_1, k_2, \ldots, k_n$, we can show the different aspect while the system is executing.

5 Computing Program States

The differential equation model contains some nonlinear ordinary differential equations, and thus it is hard to give analytic expressions to the solutions. Nevertheless, we may compute the solution trend and estimate the solution range.

Proposition 5.1 *For any place/transition cycle, if it has at most one start place, then the state measures of the cycle are all converging to numbers in [0,1] no matter how the firing rates are chosen. Particularly, 1) If the cycle does not contain start place, then all state measures are 0. 2) If the cycle contains one start place, and all the inputs to the cycle have positive state measures, then the state measures of the cycle converge to the numbers in (0, 1).*

We assume that each process has one start place.

Proposition 5.2 *When a system reaches stationary state, the measures of each process are all approaching to numbers in [0,1] no matter how the firing rates are chosen.*

Proof. Since each process is one or several place/transition cycles containing a start place. From Proposition 5.1, the state measures of the process will approach to the numbers in [0,1].

Note: We did not count the input places and the output places. Their state measures may exceed 1, unless they are also in some place/transition cycle.

Proposition 5.3 *Given a process cycle, if it does not have inputs, then all the state measures converge to numbers in (0, 1).*

6 Property Analysis

The ordinary equation model we obtained based on the Petri net may be infeasible, i.e. the system is not able to take on certain states whose state measures can be solved in the equation model. In other words, some states can not be visited in the execution. Thus, in order to study the program properties, we first need to determine that the obtained equation model is feasible.

When a state is visited, some state variable's value has been changed, or state variable's domain has been changed. If some state variable's domain at the state is empty, then this state may not be visited and the equation model is *infeasible*. If no state has empty state variable domain, then the equation model is *feasible*. Basically the variable domain is updated by the conditions from control parameters. For example, let x be a state variable and we also use $\mathscr{D}_x$ to denote the domain of x. We call $(x, \mathscr{D}_x)$ the *representation* of variable x. Let $cond[x]$ be a condition for variable x, then $cond[x] \wedge \mathscr{D}_x \subset \mathscr{D}_x$. We also define

$$cond[x] \wedge (x, \mathscr{D}_x) = (x, cond[x] \wedge \mathscr{D}_x).$$

If $\mathscr{D}_x = \emptyset$, then $(x, \mathscr{D}_x) = \perp$.

Let k be a control parameter. A condition *cond* is associated with k if $k = 1$, denoted as $(k = 1) \doteq cond$. Assume that k_1 and k_2 are two parameters such that $(k_1 = 1) \doteq cond_1[x]$ and $(k_2 = 1) \doteq cond_2[x]$, and $cond_1[x] \wedge cond_2[x] = \emptyset$. Then

$$\begin{aligned} & cond_1[x] \wedge cond_2[x] \wedge (x, \mathscr{D}_x) \\ =\ & cond_1[x] \wedge (x, cond_2[x] \wedge \mathscr{D}_x) \\ =\ & (x, cond_1[x] \wedge cond_2[x] \wedge \mathscr{D}_x) \\ =\ & (x, \emptyset) = \perp. \end{aligned}$$

Hence, the state with k_1 or the state with k_2 will not be visited. The equation model is infeasible.

Generally, for the system, we have a vector of variable representation: $(\dots, (x, \mathscr{D}_x), \dots)$. For this vector, we have rule

$$cond[x] \wedge (\dots, (x, \mathscr{D}_x), \dots) = (\dots, cond[x] \wedge (x, \mathscr{D}_x), \dots).$$

Properties of the system will be implied from this vector. Note that in the case that condition has input or output, we will use marco to separate input/output from the condition. So the above definitions are still true.

For feasible equation models, we have the following definition:

Definition 6.1 *(Stable and Unstable System) For any group of control parameter values, if all states of a system are converging to numbers in [0,1] no matter how the firing rates are chosen, then the system is* stable. *For some group of control parameter values, if there exists one group of firing rates such that there is at least one state measure converging to a number greater than 1, then the system is* unstable.

In the case that the system is stable, we have proved the following result for concurrent systems which consists of a set of processes that communicate with one another via message passing[10].

Theorem 6.2 *A program has a deadlock iff there exists a group of control parameter values such that every state measure of the program either converges to 1 (including identically to 1) or converges to 0 (including identically to 0) no matter how the firing rates are chosen.*

In the case that the system is unstable, we have the following result.

Theorem 6.3 *Given an unstable system, there is at least one input / output place that does not belong to any place / transition cycle.*

Proof. Assume that the system is unstable. Then there is a state whose state measure is greater than 1. Let the state be m, $m(t) > 1$. If every place is in some cycle, then from Proposition 5.1, we know that m is in [0,1]. Hence m is not in any cycle. We say that m is not in any process cycles since all state measures in the process cycle are in [0, 1]. So m must be an input/output place. This completes the proof.

This result implies that with some control parameter values, there must exist one place/transition chain that is attached to a process cycle. Hence, we can infer

that in this situation the system has chance to hit some synchronization problems or nondeterministic behavior, such as Race Condition.

Informally, an execution of a program contains a race if the result of some computational step depends upon the scheduling of the individual threads of execution. For example, in Figure 7, processes P_1 and P_2 send messages to P_3.

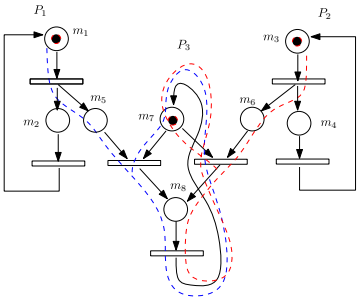

Fig. 7. Race example.

We assume that the firing rates all equal 1. The equation model is

$$\begin{cases} m_1' = m_2 - m_1, \\ m_2' = m_1 - m_2, \\ m_3' = m_4 - m_3, \\ m_4' = m_3 - m_4, \\ m_5' = m_1 - km_5m_7, \\ m_6' = m_3 - (1-k)m_6m_7, \\ m_7' = m_8 - km_5m_7 - (1-k)m_6m_7, \\ m_8' = km_5m_7 + (1-k)m_6m_7 - m_8. \end{cases}$$

If $k = 1$, then process P_3 takes the message from P_1 and if $k = 0$, then process P_3 takes the message from P_2. Let $cond =$ "P_3 gets message from P_1" and $\neg cond =$ "P_3 gets message from P_2". Then $(k = 1) \doteq cond$ and $(k = 0) \doteq \neg cond$. In both cases, three processes are all running. But with the same inputs from P_1 and P_2, we may get different outputs. This will be reflected in the following equation models.

If $k = 1$, we get the equation model

$$\begin{cases} m_1' = m_2 - m_1, \\ m_2' = m_1 - m_2, \\ m_3' = m_4 - m_3, \\ m_4' = m_3 - m_4, \\ m_5' = m_1 - m_5m_7, \\ m_6' = m_3, \\ m_7' = m_8 - m_5m_7, \\ m_8' = m_5m_7 - m_8. \end{cases}$$

By the simple calculation, m_5 will be accumulated to a number greater than 1. Let x be the state variable that describes the situation in which process P_3 will get message. So the variable representation is $(x, \{P_1, P_2\})$. This representation will be updated to $(x, \{P_1, P_2\}) \wedge cond = (x, \{P_1\})$. If $k = 0$, then we get the equation model

$$\begin{cases} m_1' = m_2 - m_1, \\ m_2' = m_1 - m_2, \\ m_3' = m_4 - m_3, \\ m_4' = m_3 - m_4, \\ m_5' = m_1, \\ m_6' = m_3 - m_6m_7, \\ m_7' = m_8 - m_6m_7, \\ m_8' = m_6m_7 - m_8. \end{cases}$$

By calculation, m_6 will be accumulated to a number greater than 1, and the representation will be updated to $(x, \{P_1, P_2\}) \wedge \neg cond = (x, \{P_2\})$.

Hence, in the above two models we get two different outputs for the same messages from P_1 and P_2.

7 Analyzing the Gas-Station With Differential Equations

Based on the net of Figure 2, we label each place with a state measure as shown in Figure 8.

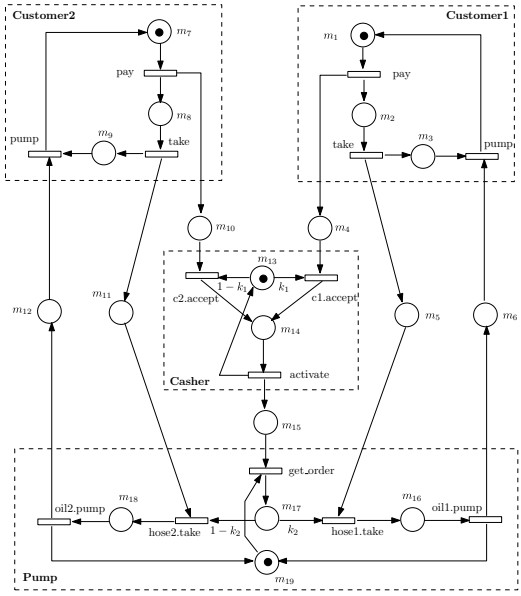

Fig. 8. Gas station net with marking and control parameters.

There are 19 places, and therefore, we will have 19 equations. In the component Casher, since the token in m_{13} can be assigned to *c1.accept* or *c2.accept* nondeterministically, we need a control parameter for this place, which is k_1 in the figure. If $k_1 = 1$, then casher gets money from customer1 first, otherwise customer2 first. Also in component Pump, both hose1.take and hose2.take have the same chance to get the token from m_{17}, we need another control parameter k_2 here for m_{17}. If $k_2 = 1$, then customer1 will take the hose first, otherwise customer2 takes the hose. Without loss of generality, we may assume that these two customers use the same speed to pay the money, to pick the hose and pump the gas. Thus we may assume the firing rates all equal 1. We may use two state variables x_1 and x_2 to represent from whom the casher accepts the money and who picks the hose, respectively. Their domains are:

$$\mathscr{D}_{x_1} = \{\mathsf{customer1}, \mathsf{customer2}\}, \mathscr{D}_{x_2} = \{\mathsf{customer1}, \mathsf{customer2}\}.$$

Vector (x_1, x_2) will be associated with each state. Let

$cond_1[x_1]$ = casher accepts money from customer1,
$\neg cond_1[x_1]$ = casher accepts money from customer2,
$cond_2[x_2]$ = customer1 pick the hose,
$\neg cond_2[x_2]$ = customer2 pick the hose.

Two parameters k_1 and k_2 are associated with these conditions by

$$(k_1 = 1) \doteq (cond_1[x_1]), \quad (k_1 = 0) \doteq (\neg cond_1[x_1])$$
$$(k_2 = 1) \doteq (cond_2[x_2]), \quad (k_2 = 0) \doteq (\neg cond_2[x_2]).$$

The differential equation model of Gas-station is in the following:

$$\begin{cases} m_1' = m_6 m_3 - m_1 \\ m_2' = m_1 - m_2 \\ m_3' = m_2 - m_3 m_6 \\ m_4' = m_1 - k_1 m_4 m_{13} \\ m_5' = m_2 - k_2 m_5 m_{17} \\ m_6' = m_{16} - m_6 m_3 \\ m_7' = m_9 m_{12} - m_7 \\ m_8' = m_7 - m_8 \\ m_9' = m_8 - m_9 m_{12} \\ m_{10}' = m_7 - (1-k_1) m_{10} m_{13} \\ m_{11}' = m_8 - (1-k_2) m_{11} m_{17} \\ m_{12}' = m_{18} - m_{12} m_9 \\ m_{13}' = m_{14} - k_1 m_{13} m_4 - (1-k_1) m_{13} m_{10} \\ m_{14}' = k_1 m_{13} m_4 + (1-k_1) m_{13} m_{10} - m_{14} \\ m_{15}' = m_{14} - m_{15} m_{19} \\ m_{16}' = k_2 m_5 m_{17} - m_{16} \\ m_{17}' = m_{15} m_{19} - k_2 m_5 m_{17} - (1-k_2) m_{11} m_{17} \\ m_{18}' = (1-k_2) m_{11} m_{17} - m_{18} \\ m_{19}' = m_{16} + m_{18} - m_{19} m_{15} \end{cases}$$

The initial values for the equation model are $m_1(0) = m_7(0) = m_{13}(0) = m_{19}(0) = 1$, $m_2(0) = m_3(0) = m_4(0) = m_5(0) = m_6(0) = m_8(0) = m_9(0) = m_{10}(0) = m_{11}(0) = m_{12}(0) = m_{14}(0) = m_{15}(0) = m_{16}(0) = m_{17}(0) = m_{18}(0) = 0$.

Based on the values assigned to k_1 and k_2, we totally have 4 cases:

1) Case 1. $(k_1, k_2) = (1, 1)$. The solutions of the model are plotted in Fig. 9(a). Since all the state measures are in [0,1], in this case, the system is stable. State variable domains can be calculated as following. For the process Casher, we have

$$\begin{aligned} &((x_1, \mathscr{D}_{x_1}), (x_2, \mathscr{D}_{x_2})) \wedge cond_1[x_1] \\ =\ &((x_1, cond_1[x_1] \wedge \mathscr{D}_{x_1}), (x_2, \mathscr{D}_{x_2})) \\ =\ &((x_1, \{\mathsf{customer1}\}), (x_2, \mathscr{D}_{x_2})). \end{aligned}$$

For the process Pump, we have

$$\begin{aligned} &((x_1, \{\mathsf{customer1}\}), (x_2, \mathscr{D}_{x_2})) \wedge cond_2[x_2] \\ =\ &((x_1, \{\mathsf{customer1}\}), (x_2, cond_2[x_2] \wedge \mathscr{D}_{x_2})) \\ =\ &((x_1, \{\mathsf{customer1}\}), (x_2, \{\mathsf{customer1}\})). \end{aligned}$$

Thus for the whole system, we have the variable representation:

$$((x_1, \{\mathsf{customer1}\}), (x_2, \{\mathsf{customer1}\})).$$

We can interpret this expression as that customer1 pays the money and will get the gas. From the solution curves, we can also find that the state measures

$$m_7 \to 0,\ m_8 \to 0,\ m_{12} \to 0,\ m_{18} \to 0.$$

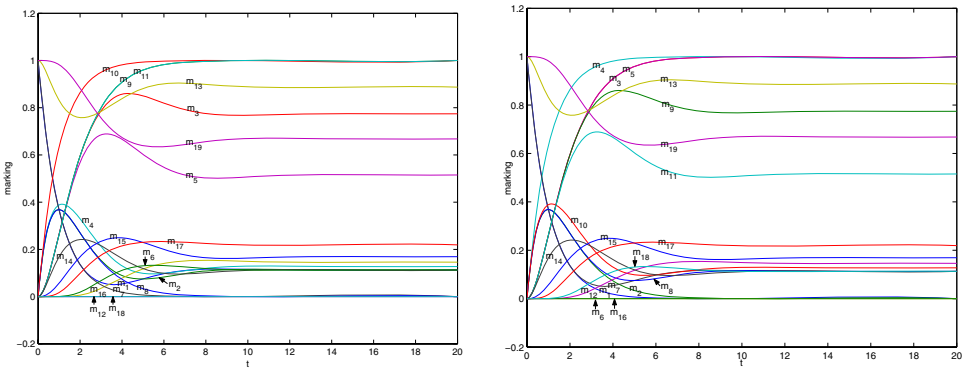

Fig. 9. (a). Case1: $(k_1, k_2) = (1, 1)$. (b). Case 2: $(k_1, k_2) = (0, 0)$.

$m_8 \to 0$ means that the state m_8 is eventually not being visited, in other words, Customer 2 does not pay the money. Similarly, $m_{18} \to 0$ means that the state m_8 is eventually not being visited, in other words, Customer 2 does not have chance to pick the hose and thus does not get the gas.

2) Case 2. $(k_1, k_2) = (0, 0)$. The solutions of the model are plotted in Figure 9(b). The system is stable. This case is for customer2 as that for Customer1 in case 1. Customer2 pays the money and will get the gas. Customer1 does not pay the money and will not get the gas.

Hence, from Case 1 we imply that if a customer pays, then he/she will get the gas; from Case 2 we imply that if a customer does not pay, then he/she will not get the gas. Thus, we may conclude that the second specification "no free gas" is satisfied.

3) Case 3: $(k_1, k_2) = (1, 0)$. The solutions of the model are plotted in Figure 10(a). Form the curves, we find that state measure m_{10} approaches to 2. Thus the system is unstable. Variable domains can be calculated as following. For the process Casher, we have

$$\begin{aligned} &((x_1, \mathscr{D}_{x_1}), (x_2, \mathscr{D}_{x_2})) \wedge cond_1[x_1] \\ =&((x_1, cond_1[x_1] \wedge \mathscr{D}_{x_1}), (x_2, \mathscr{D}_{x_2})) \\ =&((x_1, \{\mathsf{customer1}\}), (x_2, \mathscr{D}_{x_2})). \end{aligned}$$

For the process Pump, we have

$$\begin{aligned} &((x_1, \{\mathsf{customer1}\}), (x_2, \mathscr{D}_{x_2})) \wedge \neg cond_2[x_2] \\ =&((x_1, \{\mathsf{customer1}\}), (x_2, \neg cond_2[x_2] \wedge \mathscr{D}_{x_2})) \\ =&((x_1, \{\mathsf{customer1}\}), (x_2, \{\mathsf{customer2}\})). \end{aligned}$$

Thus for the whole system, we have the variable representation:

$$((x_1, \{\mathsf{customer1}\}), (x_2, \{\mathsf{customer2}\})).$$

We can interpret this expression as that customer1 pays the money and customer2 gets the gas.

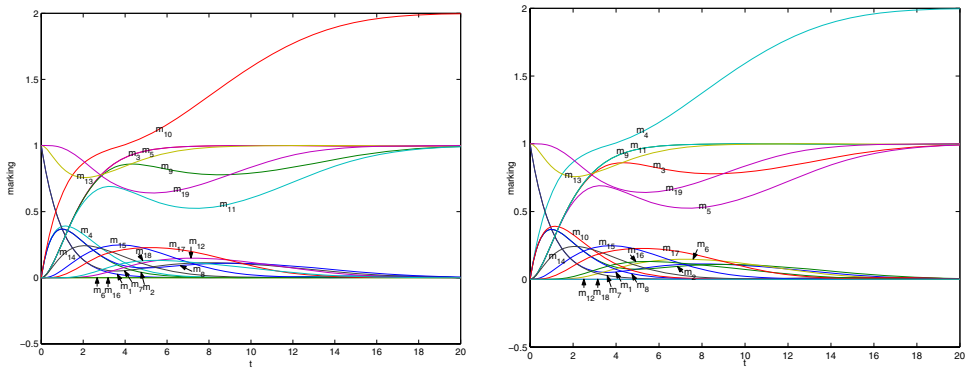

Fig. 10. (a). Case1: $(k_1, k_2) = (1, 0)$. (b). Case 2: $(k_1, k_2) = (0, 1)$.

4) Case 4: $(k_1, k_2) = (0, 1)$. The solutions of the model are plotted in Fig. 10(b). The system is unstable as well. This is the same as case case 3) except the positions of Customer1 and Customer 2 are changed.

Hence, from Case 3 and Case 4, we may conclude that the first specification is satisfied.

8 Discussion and Conclusion

The most closed work is by Naumovich et al. in [19], where they used two concurrency analysis tools, INCA and FLAVERS, to verify the same specifications of Gas station architecture as in our paper. *Flow equation* technique has been used in INCA to check the consistency of the system. Inequalities are solved using standard integer linear programming packages. However, integer linear programming problems are generally NP-hard, and the standard techniques involved are potentially exponential. FLAVERS is data flow analysis based tool. By approximating the execution model of a program, properties can be efficiently checked using a polynomial algorithm. However the conclusion thus obtained is usually either complete or sound but not both.

Generally speaking, the existing static analysis techniques can not complete avoid hitting state explosion problem even some state reduction techniques have been used such as in SPIN. The reason is that we have to search all reachable states to determine if the properties can be satisfied. With our method, we only need to solve 2^k different differential equation groups, where k is the number of control parameters. Given a group of values for these ks, we will obtain an equation group. Hence, these equation groups are independent and can be solved in parallel. So the complexity is determined only by one equation group. By analyzing the solutions, we may determine if the architecture can satisfy the requirement. The more larger the system, the more powerful this method will express. This is why we can avoid the state explosion problem when we do statically analysis.

The existing static analysis tools can automate the checking of properties, but it is still up to the system architect to formulate those properties. With our method, it

is very easy to write equation group and compute them with Matlab. If the numbers of the equations is huge, we do require a tool to analyze the discrete-valued solutions, not just to check the curves from Matlab.

We make no claim that our technique can replace the existing analysis tools. In the existing tools, properties are expressed by Temporal Logic. To check the property of interest, we can design some check statements and insert them to the input language. In this way, we can immediately know if the specified property is satisfied. The disadvantage is that the tools may not suitable for large systems. In our case, we can easily find the abnormal behavior, and then based on the state variable representation to analyze if the properties of the architecture are satisfied. The disadvantage is that we need to interpret the variable representation. Also, we have not proved yet that the state variable representation can describe all the properties of the architecture.

By combining our technique with the existing skills, we may also improve the checking speed. For example, use our equation method as the first step to find abnormal behavior which corresponds to a group of control parameter values, and then based on the interpretation from these values to design some statements to be inserted to the input languages for the model checking. Since in the first step we have narrowed the analysis scope, we may quickly check the property without hitting explosion problem.

As one might concern, if the system is very large, the equation model will be very big. And sometime, even a single equation group may contain huge number of equations, for example to model a system with 10^{20} states that has been used for Symbolic Model Checking by Burch et al.[2]. Matlab may not have enough power to do such computing. A solution is to solve the equation group in parallel. Currently, we are developing a computing algorithm based on the work by Intievergelt[15]. A large differential equation group can be separated into several small equation groups that can be computed in parallel. Eventually, we will develop a tool to support the analysis from building equation model to parallel computing, and to the solution analysis.

As the next step study, we will check the efficiency of our method applied to the systems that consist of large number of behaviorally similar processes, e.g. in the gas station example there exist lots of customer processes, pump processes and casher processes, and find out how our model complement to those discrete event based models, e.g. [23], that are used to handle the large number of processes.

Acknowledgement

We are grateful to the anonymous reviewers for their detailed comments and suggestions that improved the paper. This work is partially supported by the National Key Research program of Dependable Software Theory under Grant No. 90718014, National High Tech Research 863 Program of China under Grant No.2006AA01Z165; the National Natural Science Foundation of China under Grant No.60673114; 60603037; International Cooperation Program of Shanghai under

Grant No.08510700300.

References

[1] Allen R. and Garlan D. 1997. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology* 6(3), pp.213-249.

[2] Burch J. R., Clarke E. M., and Long D. E. 1991. Representing circuts more efficiently in symbolic model checking, In *Proceedings of the 28th Design Automation Conference*, IEEE Computer Society Press, Los Alamltos, Calif., pp.403-407.

[3] Clarke E.M., Grumberg O., Peled D. 1999. Model Checking, MIT Press.

[4] Clarke E., McMillan K., Campos S. and Hartonas-Garmhausen V. 1996. Symbolic Model Checking, *Proceedings of 8th Computer Aided Verification Conference*, Springer, Berlin.

[5] Corbett J.C. and Avrunin G.S. 1995. Using integer programming to verify general safety and liveness properties, *Formal Methods in System Desin*, 6, pp.97-123.

[6] David R. and Alla H. 1987. Continuous Petri nets. *8th European Workshop on Application and Theory of Petri nets*, Zaragoza, Spain, pp.275-294.

[7] David R. and Alla H. 1990. Autonomous and timed continuous Petri nets, *11th Int. Conf. On Application and Theory of Petri nets*, Paris, France, pp.367-381.

[8] Ding Z. and Xiao L., Hu J. 2008. Performance analysis of service composition using ordinary differential equations, *Proceedings of FTDCS08*, IEEE Computer Society Press, Kunming, China, October 21-23.

[9] Ding Z. and Zhang K. 2008. Performance analysis of concurrent programs using ordinary differential equations, *COMPSAC08*, IEEE Computer Society Press, Turku, Finland, July 28-August 1.

[10] Ding, Z., Zhang K., and Kandel A. Detecting program deadlocks with ordinary differential equations. *Journal of IEEE Transactions on Software Engineering*, submitted.

[11] Dwyer M.B. and Clarke L.A. 1994. Data flow analysis for verifying properties of concurrent programs, *Proc. Second Symp. Foundations of Software Enginemng*, pp. 62-75.

[12] Gilbert D. and Heiner M. 2006. From Petri nets to differential equations-an integrative approach for biochemical network analysis, *Lecture Notes in Theoretical Computer Science*, vol.4024, pp.181-200.

[13] Hoare C. A. R. 1978. Communicating sequential processes. *Communication of ACM* 21(8), pp.666-677.

[14] Holzmann G.J. 1980. Basic Spin Manual, http://cm.bell-labs.com/netlib/spin/ whatispin.html, 1980.

[15] Intievergelt J. 1964. Parallel methods for integrating ordinary differential equations, *Communications of the ACM*, 7(12), pp.731-733.

[16] Juan E., Tsai J.J.P. and Murata T. 1998. Compositional verification of concurrent systems using Petri-nets-based condensation rules. *ACM Transactions on Programming Languages and Systems* 20(3), pp.917979.

[17] Lunze J., Nixdorf B., and Richter H. 1997. Hybrid modelling of continuous-variable systems with application to supervisory control. In *Proceedings of the European Control Conference 97*, Brussels, Belgium.

[18] Molloy M. K. 1985. Fast bounds for stochastic Petri nets. *International Workshop on Timed Petri Nets*, Torino, July, pp.244-249.

[19] Naumovich G., Avrunin G. and Clarke L. 1997. Applying static analysis to software architectures, *ACM SIGSOFT Notes* 22(6), pp.77-93.

[20] Peleties P. and DeCarlo R. 1994. Analysis of hybrid systems using symbolic dynamics and Petri nets. *Automatica* 30(9), pp. 1421-1427.

[21] Sistla A.P., Miliades L. and Gyuris V. 1997. SMC: A symmetry based model checker for verification of liveness properties, *Proceedings of 9th Computer Aided Verification Conference*, Haifa, Israel.

[22] Tsai J.P. and Xu K. 1999. An empirical evaluation of deadlock detection in software architecture specifications, *Annals of Software Engineering* 7, pp.95-126.

[23] Wang X., Kwiatkowska M. 2007. Compositional state space reduction using untangled actions, *Electronic Notes in Theoretical Computer Science*, 175(3), pp. 27-46.