Towards Rewriting Semantics of Software Architecture Specification

Yujian Fu
Department of Computer Science
Alabama A&M University
yujian.fu@aamu.edu
Zhijiang Dong
Department of Computer Science
Middle Tennessee State University
zdong@mtsu.edu
Phil Bording
Department of Computer Science
Alabama A&M University
phil.bording@aamu.edu
Xudong He
School of Computer Science
Florida International University
hex@cis.fiu.edu

1. Introduction

During the past decade, architectural design has emerged as an important subfield of software engineering. This is because a good architecture can help ensure that a system will satisfy user requirements. Consequently, a new discipline emerged, which concerns formal notations for representing and analyzing architectural designs using Architecture Description Language (ADL) [25]. These notations provide both a conceptual framework and a concrete syntax for characterizing software architectures [25]. Combine with software architecture models that are not considered as ADLs, we call them software architecture specifications.

Software architecture specifications (i.e. software architecture model, software architecture description languages (ADLs) (such as Rapide [24], Wright [1] and XM-ADL [23]), etc.) allow software designers to focus on high level aspects of an application by abstraction of the details of the subsystems and components. It is precise and accurate to use formal methods to describe the abstraction that makes software architecture specifications are suitable for verification using model checking techniques. Software specifications are, in a way, domain-specific languages for aspects such as coordination and distribution. Software Architecture Model (SAM) is a formal approach based on two formal languages - Petri nets

and temporal logic for distributed concurrent software systems. Further, SAM has been used to interpret semantics of Unified Modeling Language (UML) diagrams [9, 10].
The theory of rewriting logic (RWL) has proved to be a unifying formal framework to describe concurrency formalisms [26, 27]. It is useful to specify concurrent behavior of various types of systems. In the verification aspect, the issue of using rewriting logic in the model checking group is how to convert a logic notations based on Boolean results to a state based labeled transition system. As a high performance and reflective language, the Maude specification [6] was developed to support rewriting logic based model checking on concurrent systems. Maude is a high performance declarative programming language that is based on rewriting and equational theory. Equational theory is used to describe the properties and rewriting logic is used to describe the state transition of systems.
Major concurrency formalisms have been successfully translated in rewrite theories [27]. In addition, rewriting logic has been used by several authors in the verification of architectural notations such as architectural description languages and object-oriented design formalisms. However, there are few works in the formal description of software architecture specifications using rewriting logic.
Current research on software architectures has a focus on how to express and verify functional and nonfunctional aspects statically and dynamically. It is highly desirable to precisely describe the semantics of software architecture specification. Some ADLs do not provide formal semantics [28] while many do [1, 24]. One category provides benefits for easy to understand and use, but hardly to be reasoned and analyzed in the architecture level. The other category, in contrast, has benefits of precise reasoning and verification. Analysis software architecture specification using rewriting logic aims at formally reasoning distributed concurrent systems at the architecture level.
This helps to reduce errors that are introduced to implementation during development process [17, 18].
Recent work has been done [13, 9, 12, 8] to verify the system model in both the design and implementation level with an integrated framework, which combines the formal verification (model checking) with implementation verification technique (runtime checking). However, there is no work on the logic based semantics analysis of software architecture specification – SAM. This chapter is to present a systematic translation algorithm as well as a validation approach towards rewriting logic semantics of software architecture model.
Related Works. Several related works are investigated in the software architecture specification and rewriting logic semantics. The paper in [7] presents a framework for describing global optimizations by rewrite rules with Computation Tree Logic (CTL) formulae as side conditions, which allows the generation of correct optimizations, but cannot be used for verification of (possibly incorrect) optimizations. The correctness is established in an imperative language without procedures. The work in [15] proposes a method for deploying optimizing code generation while correct translation between input program and code. They focus on code selection and instruction scheduling for SIMD machines.
Xie et al. [31] presented an approach to transforming software specification syntax to the model checking programming language. In [31], static analysis is used to validate the syntax, semantics and property translations, on which software testing strategy is adopted to validate the translation. Preserving equivalence conditions was not checked in that work. The work in [22] describes a translation from textual transition system to Petri nets. Using

bisimulation, they validated the translation by proof of the translated model are equivalent with the source model based on the step by step comparison. Although it is a stepwise proof between translated model and the source model, the soundness and completeness cannot be established because it is based on the semi-formal explanation for the comparison by the natural language.
Several works have been done for the analysis of SAM specification either using theorem proving [21] or model checking technique [20]. All these methods are based on the either partial order reduction [20] or encoded symbolic states [21]. This chapter presents for the first time an approach to convert Software Architecture Model (SAM) [30] to rewriting logic based semantics. This work has an interesting property of actually executing the SAM semantics to do the simulations, that is, rewriting a topology, and the verification, since the transformation from SAM to rewriting logic makes rewriting logic an alternative executable semantics of SAM.
The remainder of this paper is organized as follows. In Section 2, we review SAM with predicate transition nets for high-level design and rewriting logic with the tool support – Maude's syntax. A translation algorithm is presented in Section 3. Then coffee machine case study was presented in Section 4. After that, a validation approach is demonstrated in Section 5. Finally, we draw conclusions and describe future work in Section 6.

2. Preliminaries

2.1 SAM – Software Architecture Model

SAM (Software Architecture Model) [30] is hierarchically defined as follows. A set of compositions $C = \{C1, C2, \ldots, Ck\}$ represents different design levels or subsystems. A set of component C_{mi} and connectors C_{ni} are specified within each composition Ci as well as a set of composition constraints C_{si} , e.g. $Ci = \{C_{mi}, C_{ni}, C_{si}\}$. In addition, each component or connector is composed of two elements, a behavioral model and a property specification, e.g. $Cij = (Sij, Bij)$. Each behavioral model is described by a Petri net, while a property specification by a temporal logical formula. The atomic proposition used in the first order temporal logic formula is the ports of each component or connector. Thus each behavioral model can be connected with its property specification. A component C_{mi} or a connector C_{ni} can be refined to a low level composition Cl by a mapping relation h, e.g. $h(C_{mi})$ or $h(C_{mi}) = Cl$. SAM is suitable to describe large scale systems' description. However, there is no high level behavior definition of components and connectors. In our work, we only consider the flattened version of SAM specification in which each component/connector cannot be further refined.

2.1.1 Predicate Transition Nets Predicate

Transition (PrT) net [14] is a high level Petri net. A Predicate/Transition net is a 9-tuple(P, T, F, $\sum$, Eq, φ, L,G, $M0$), where:

1. P is a finite set of places, T is a finite set of transitions ($P \cap T = \Phi$, $P \cup T \neq \Phi$), and F is a set of arcs or flow relations between each pair of P and T, e.g. $F \subseteq (P \times T) \cup (T \times P)$. The tuple (P, T, F) forms a basic Petri net structure.
2. $\sum =< St, Op >$ consists of some sorts (St) of constants together with set of operations (Op) and relations on the sorts.
3. Eq defines the meanings and properties of operations in OP.

4. $\varphi : P \to St$ is a relation associated each place p in P with a subset of sorts.
5. L is a labelling function on places, transitions, arcs, and variables. Given a place $p \in P$ or a transition $t \in T$, $L(p)$ returns the name of place p, $L(t)$ returns the name of transition t. Given an arc $f \in F$, the labelling function of f , L(f), is a set of labels associated with the arc f , which are tuples of constants (*CONs*) and variables (X), which is best described by L(f, *Terms*Σ,X). We use *Terms*Σ,X represents the expressions on the label of arc f. We use L($Terms_{\Sigma,X}$) represents L(f, $Terms_{\Sigma,X}$) when there is no confusion in context. If $f < F$, $L(f) = \Phi$.
6. R is a mapping from transitions to a set of inscription formulae. The inscription on transition $t \in T$, $R(t)$, is a logical formula built from variables and the constants, operations, and relations in structure Φ, variables occurring free in a formula have to occur at an adjacent input arc of the transition.
7. $M0$ is the initial or current marking with respect to sort, which assigns a multi-set of tokens to each place p in P with the same sort, $M0 : P \to MCONs$.

The dynamic semantics of PrT nets are defined by the transition firings. Dynamic semantics of PrT nets are described as follows [19]:

1. A marking M of a PrT net N is a mapping function defined from set of places P to constants *MCONs*.
2. The enabling condition of a transition $t \in T$ under a marking M with a substitution $\alpha = \{x_i \leftarrow ci | xi \in X, ci \in MCONs\}$ is defined as follows:
   $\forall p \in P.\ (L(f) : \alpha) \subseteq M(p) \wedge R(t) : \alpha$.
3. If a transition $t \in T$ under a marking M with a substitution α is enabled, a marking $M0$ is obtained after the transition t is fired, then the firing condition of a transition t is defined as:

$\forall p \in P.\ M0(p) = M(p) - L(p, t) : \alpha \cup L(t, p) : \alpha$.

The enabling and firing condition can also be defined by preset and postset of transition. A preset of a transition t, denoted by *pre*(t) or $\bullet t$, is the set of all places that have outgoing relation from these places to the transition t.

If a transition is enabled, required tokens specified by the label expression of input arcs of the transition must be available in the preset of the transition. If a transition is fired, those required tokens are consumed and produce some tokens that satisfy the label expression of output arcs of the transition. Both consumed tokens and produced tokens must have the same sort of the preset of the transition and postset of the transition respectively. Let $L(f$, *Terms*Σ,$X)$ be label expression that associates with arc $f = (p, t) \in F \vee f = (t, p) \in F$. For any place $p \in P$, we can define the two functions for the consumed tokens (*consuming-token-p*($L((p, t)$, *Terms*Σ,X))) and produced tokens (*producing-token-q*($L(f$, *Terms*Σ,X))) for a transition t as follows.

$\forall p \in P.consuming - token - p(L((p,t), Terms_{\Sigma,X})) : p \to L((p,t), Terms_{\Sigma,X}) : \alpha$ where $L(f, Terms_{\Sigma,X}) : \alpha \in \varphi(p)$, $(p,t) \in F$, and $M'(p) = M(p) - consuming - token - p(L((p,t), Terms_{\Sigma,X}))$,

$\forall q \in P.producing - token - q(L(f, Terms_{\Sigma,X})) : p \to L((p,t), Terms_{\Sigma,X}) : \alpha$ where $L(f, Terms_{\Sigma,X}) : \alpha \in \varphi(p)$, $(p,t) \in F$, and $M'(p) = M(p) \cup producing - token - q(L((p,t), Terms_{\Sigma,X})$.

From above functions, we can see that for any transition t, the tokens consumed in the preset of the transition t can be described by a substitution of label expression in the function *consuming- token-p*($L(f$, *Terms*Σ,X)), if the substitution of label expression in the token set of preset of transition t; while the tokens produced in the postset of the transition t can be described by a substitution of label expression in the function *producing-token-q*($L(f)$), if the substitution of label expression satisfy the sort of postset of the transition t. The token set of preset and postset of transition *t* can be described by the substitution of sorts of preset and postset, i.e., $pre(t)(\varphi(p) : \alpha)$ and $post(t)(\varphi(p) : \alpha)$. Therefore, we have enabling and firing conditions as follows:

$Consuming\text{-}token\text{-}p(L(f, Terms_{\Sigma,X})) \in pre(t)(\varphi(p) : \alpha)$, and

$Producing\text{-}token\text{-}q(L(f, Terms_{\Sigma,X})) \in post(t)(\varphi(p) : \alpha)$.

We can define the interleaving semantics using the sequence of markings with the occurrence of corresponding transitions for each set of substitutions.
A sequence $\sigma = M_0[t_0{=}\alpha_0 > M_1[t_1{=}\alpha_1 > \ldots [t_{n-1}/\alpha_{n-1} > M_n$ with $n \geq 0$ is called a finite interleaving execution starting with M_0 iff $\forall i \in Nat$ and $0 \leq i \leq n$ and $M_{i-1} \rightarrow_{t_{i-1}/\alpha_{i-1}} M_i$, where $M_i : P \rightarrow \wp(St)$, α_i denotes a substitution for the variables in a guard condition of a transition t_i, St denotes set of sorts.

2.1.2 Temporal Logic

Temporal logic defines four future-time (past-time) operators in addition to the propositional logic operators. They are:

- Always in the future (past), symbolized as a box $\Box$ ($\boxdot$).
- Sometime in the future (past), symbolized as a diamond $\Diamond$($\diamond$).
- Until for the future (Since for the past), U (S).
- Next (Previous) for the future (past) , O($\odot$).

An example of a temporal logic formula $\Box(p \rightarrow \Diamond q)$ indicates that predicate *p* implies eventually *q* always happen.

2.2. Rewriting Logic

Rewriting logic [26] is a logic for concurrency. A rewrite theory R is a tuple (Σ, *E*, *L*, *R*), where (Σ, *E*) is an equational logic, *L* is set of labels, and *R* is a set of rewrite rules. A rewrite $P : M \rightarrow N$ means that the term *M* rewrites to the term *N* modulo E_R, and this rewrite is witnessed by the proof term *P*. Apart from general (concurrent) rewrites $P : M \rightarrow N$ that are generated from identity and atomic rewrites by parallel and sequential composition, rewriting logic classifies its most basic rewrites as follows: a one-step (concurrent) rewrite is generated by parallel composition from identity and atomic rewrites and contains at least one atomic rewrite, and a one-step sequential rewrite is a one-step rewrite containing exactly one atomic rewrite.
We often write .

$$l : [s] \rightarrow [t] \text{ if } [\overrightarrow{u}] \rightarrow [\overrightarrow{v}] \text{ for } l : [s] \rightarrow [t] \text{ if } [u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k]$$

For the unconditional rewrite, if clause is weaved and the form would be l : [s] $\rightarrow$ [t]. Where s and t are terms that may contain variables. A rule describes a local concurrent transition in a system, i.e., anywhere where a substitution instant $\alpha(s)$ of the lefthand side *s* is found, a local transition of that state fragment to the new local state $\alpha(t)$ can take place.
Rewriting logic therefore extends equational logic with rewrite rules, allowing one to derive both equations and rewrites. Deduction remains the same for equations, but the symmetry rule is dropped for rewrite rules. Rewriting logic is a framework for true concurrency [27]:

the locality of rules allows multiple rules to apply at the same time provided they don't modify the shared part.
The operational semantics of rewrite specification extends the operational semantics of membership equational specification by applying computational equations *E*R and rewrite rules *R*R modulo the structural equations *E*R. The process net for the behavior model – Petri nets – can be matched into the rewrite theory *R*. For instance, signatures and rewrite rules can be mapped into the net specification and places, and transitions with guard conditions and their pre- and post-sets. Instead of using operational semantics, the work [26] demonstrated the isomorphism between category Petri nets and category Rewrite theory.
Rewriting logic and membership logics are supported by Maude [3]. In Maude [2] the basic units are functional modules and system modules. A functional module is an equational style functional program with user-definable syntax in which a number of sorts, their elements, and functions on those sorts are defined. A system module is a declarative style concurrent program with user-definable syntax.
A functional module is declared in Maude using the keywords

```
fmod <ModuleName> is
        <DeclarationsAndStatements>
endfm
```

The <DeclarationsAndStatements> includes signatures (e.g. sorts, subsorts, kinds etc.), operations, and equations. In Maude, functional modules are equational theories in membership equational logic satisfying some additional requirements. Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. This is the reason why the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort decreasing.
A system module is declared in Maude using the keywords

```
mod <ModuleName> is
        <DeclarationsAndStatements>
endm
```

The <DeclarationsAndStatements> includes sorts and subsorts, operation, equation, rules, etc.. declaration.
Conditional rules has the form of

```
crl [label] : <left term> => <right term> if <condition or set of conditions> .
```

While unconditional rules has the form of

```
rl [label] : <left term> => <right term> .
```

The system modules specify the initial model T of a rewrite theory R in the membership equational logic variant of rewriting logic. These initial models capture nicely the intuitive idea of "rewrite systems" in the sense that they are transition systems whose states are equivalence classes [*t*] of ground terms modulo the equations *E* in R, and whose transitions are proofs $\alpha : [t] \rightarrow [t0]$ in rewriting logic, that is concurrent rewriting computations in the system described by the rules in *R*.

3. Translation from SAM Specification to Rewriting Logic

```
*** This is the algorithm to translate the SAM model to Maude program

*** M−SIGNATURE
for all $B \in C_k$, where $k \in Nat$
    for all $P \in B, St \in B$
        declare sort s,   $s_i \in St$ where $i \in Nat$
        sort Marking .
        for all  $p \in P$ and $s_i \in St$
        *** declare an operation
        $f_p : s_i \to Marking$ .

*** M−SYSTEM
    including M−SIGNATURE
    for all  $B \in C_k$, where $k \in Nat$
    for all  $T \in B$ and $G \in B$,
        rl [t] : $f_1 \to f_2$
            where $f_1$ is the set of operations mapped from $\bullet t$,
                  $f_2$ is the set of operations mapped from $t\bullet$.
        crl[t] : $f_1 \to f_2$ if g
            where $g$ is associated w $t$, and $g \in G$ and $t \in T$

*** M−PREDICATE
    including M−SYSTEM .
    including SATISFACTION .
    sort Marking < State .
    for all  $B \in C_k$, where $k \in Nat$
        for all $P \in B, St \in B$
        ***declare an operation
        $Pf_p : s_i \to Prop$ .
        $m_1, m_2 : Marking$ .
        *** define semantics using equations
        eq $m_1\ f_p\ m_2 \models Pf_p$ = true .
            where $i, j \in Nat$

*** M−MODEL−CHECK
    including M−SYSTEM .
    declare initial markings
    declare properties to be checked .
```

Fig. 1. Translation Algorithm from SAM to Maude

The SAM specification allows formal verification of a component system against system constraints and property specified on its abstraction. Here, verification means that the developer can animate the specification by providing initial markings and checking if the responses meet the expected results. Verification of SAM is based on the precise syntax and semantics of Petri net formal language and temporal logic.
In this work, we choose Maude [2] – a high level and high performance declarative programming language – as a model checking tool. Maude is based on rewriting logic and membership equational logic. During the verification, we found a seamless matching

between SAM specification and rewriting logic. We use *Nat* to denote the set of natural numbers. The translation algorithm is summarized in Fig. 1. The translation covers the topology and dynamic behavior of the SAM specification.
In Fig. 1, *B* denotes the set of behavior models, *Ck* denotes the set of components, connectors or compositions; *St*, *P*, *T*, and *R* the set of sorts, places, transitions, and constraints associated with transition in the behavior model *B*, respectively; each element of the set *St*, *P*, *T*, and *G* is represented by lower case letters *si*, *p*, *t*, *g* respectively. Following the rewriting logic notation in [2], each operation is denoted by *f* or *fi* where $i \in Nat$. We use *fpre* to denote the left side operations and *fpost* denote right side operations.
SAM specification is a hierarchical structure that provides the high level organization of the system topology. When no behavior refinement defined, the components and connectors are taken off so that a flattened architecture description with one behavior model (Petri net) is obtained. Petri nets of all components/connectors form an integrated Petri net through merging connected ports between components/connectors. This flattened SAM structure simplifies the analysis and verification. Therefore, the translation algorithm presented in Fig. 1 is based on the flattened version of SAM specification. In this section, we first present the translation algorithm and the corresponding Maude program in two groups – signature translation and system description translation. Comments in Maude syntax are started with three stars and ended at the end of the line.
In Maude syntax, there are two fundamental blocks defined – function module and system module. Where functional module in Maude denotes the signature of the system, system module describes the state changing under certain set of rules (conditions). The purpose of the translation algorithm is to map each building block in SAM specification to a corresponding one in Maude specification.
In Fig. 1, the signature of the system is described in a function module named *M-SIGNATURE* where *M* represents the system. To describe the system, three system modules, named *M-SYSTEM*, *M-PREDICATE*, and *M-CHECKING* must be defined. *M-SYSTEM* describes the system topology with structure and interactions. In this module we describe the state transition of the system using rewriting rules and the equations. *M-PREDICATE* describes the semantics of the operations and state transitions. *M-CHECKING* is a verification module of the system. This module specifies the initial conditions that are required for the model execution and properties to be checked. We summarize the translation algorithm in Fig. 1.

3.1 Mapping to Signature

Signature (sorts, operations) and equations of the behavior model (*B*) of SAM – Petri nets – can be mapped to the _-signature in the rewriting theory. In addition, each sort of a place and a port is considered as a local state, and defined as a sort *Marking*. The translated sorts in the rewriting logic include the set of sorts in Petri nets *St* as well as a new sort *Marking*.
Maude currently supports limited basic data types (int, bool, String) and operations on them in predefined module. The module for the basic sorts and operations can be loaded when the core Maude is running. To implement all user defined sorts in Petri nets, we need following two restrictions: a) specifying them as Cartisian product using the predefined sorts in Maude; and b) trying to specify the system using the basic sorts and avoid the complicated user defined sorts. We do not define any more equations in the signature translation. The equations are those used for the basic sorts defined in rewriting and membership equational logic for basic types.

3.2 Mapping to System Module

In the behavior model Petri nets, the state changings are described by marking updating through transition firing. Each time a transition t is fired, some tokens consumed in the preset ($pre(t)$) of the transition t, new tokens are generated to the postset ($post(t)$). This state changing can be described by a(n) (un)conditional rule *rl* in the rewriting theory R.
Let $id(t)$ denote the name of transition t. A transition $t \in T$ with its preset and postset of places can be mapped to an unconditional rule with following format:
$rl[id(t)] : fpre(t) \rightarrow fpost(t)$ if guard $g \in G$ associated with this transition is *true* or empty.
Otherwise, the transition $t \in T$ is mapped to a conditional rule

$$crl[id(t)] : fpre(t) \rightarrow fpost(t) \text{ iff } g \text{ for } g \in G.$$

This translation is shown in Fig. 1 from line 11 to 17.
In Maude program, we can express the above translation as follows:

```
including M-SIGNATURE .

vars vi : si .

rl [id] : <operations of pre(transition)> => <operations of post(transition)> .

crl [id] : <operations of pre(transition)> => <operations of post(transition)> if <guard of
transition> . ,
```

where *id* is the identifier of the transition, $pre(t)$ and $post(t)$ denote the preset and postset of transition t.

3.3 Mapping to State Predicates

After translating the signature and system model, to do model checking, the things left is for the state predicates and initial markings. Maude provides the conversion from system model to the Kripke Structure with internal modules based on rewriting theory. To specify state predicates what we need is to associate the kind of sorts of the system to the formulae in the Kripke structure. In Maude, the predefined module *LTL* converts the Linear Temporal Logic (LTL) formulae to the Kripke structure [2]. Module *SATISFACTION* associates the state to formula by declaring a sort *State* and a boolean operation on the sort *State*.
Sort *Marking* is defined as a subsort of predefined sort *State*. Based on the pre-exiting module definitions of Maude, we can have following translation. Each state predicate is defined as an operator of sort *Prop*. Then defines their semantics by means of a set of equations that specify for what states a given state predicate evaluates to true. This is also expressed the algorithm Fig. 1 in line 18 – 29. We have following code template in Maude:

```
protecting M-SIGNATURE .

including SATISFACTION .

subsort Marking < State .

vars m1 m2 : Marking .

op Pp : si -> Prop .

eq m1 Pp m2 |= Pid = true . ,
```

where *si* denotes the set of sorts of the place *p*, .

3.4 Mapping to Initial Markings & Properties

Finally, to define the initial marking and the properties to be verified (Fig. 1), upon previous translation algebra, we can simply define initial markings as equations of operations to initial conditions, and properties are equations from property operations to properties. Implementing in Maude, we have following template:

In Maude, the predefined module *MODEL-CHECKER* convert the system model defined by the rewriting logic to the Kripke structure so that we can apply model checking on the system [2]. The module *LTL-SIMPLIFIER* is used to define the linear temporal logic in rewriting logic. Based on these two modules, we define the following template Maude program:

```
protecting M-PREDICATE .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .
*** declare initial markings as operations that output sort Marking.
op initn : -> Marking .
*** declare operation for each property that output sort Formula.
op propertyn : -> Formula .
*** declare equations for each place that has initial markings.
eq init = pi .
*** declare needed variables.
eq propertyn = <propertyn> .
```

3.5 Discussion

The ports synchronization is a challenging issue. The semantics of each pair of merged ports are described as interface places that are shared by component and connector in SAM. Although ports are visible for communicated components/connectors, only when tokens are produced in the outgoing ports, the incoming ports of corresponding components/connectors can be executed. In the translation (Fig. 1) for the shared places of ports, the outgoing ports are translated to right side of a rule, while the corresponding incoming ones are the left side of the corresponding rule. On the other side, if the overlap rewriting rules can fire concurrently, the executions of the concurrent rule is nondeterministic.

This translation algorithm (Fig. 1) is applied on a case study – coffee machine and experimented using Maude program. The experimental result is demonstrated in Section 4.

4. Case Study

Fig. 2 shows the SAM specification of a simple coffee machine [29], which accepts requests and then either serves a cup of coffee or returns back money. Behavior models of all three

components *CoinHandler*, *BrewingFacility*, and *CMInter f ace* are demonstrated in Appendix A (Fig. 3(a)-Fig. 3(c)).
A request with a tuple of money and coffee type is accepted in the component *CMInter f ace*. A simplification of this model is that we assume money and coffee type are input at the same time instead of modelling input sequentially. Then the request flows into component *CoinHandler* through connector *CH-CMI*. After checking the price table kept in the *CoinHandler*, either a money return or a coffee request is issued through the corresponding port *coin_back_ch* or *coffee_request_ch*, respectively. The place *price* (Fig. 3(a)) outputs a token either to transition *enough* or *not enough*, which is used to check whether or not the input money is enough for a certain type of coffee. If there is enough money, coffee request is issued by port *coffee_request_ch*, and *coin_back_ch* gets a token when the money is more than that kept in the place *price*. If there is not enough money, *coin_back_ch* will eventually get a token to return the money.

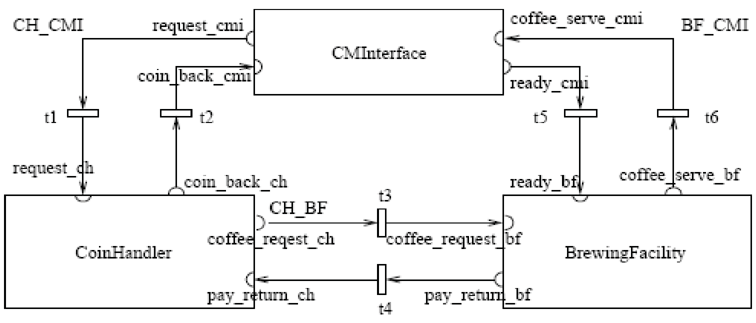


Fig. 2. SAM specification of Coffee Machine

The component *BrewingFacility* (Fig. 3(b)) keeps a coffee storage table and is responsible for cooking and providing coffee to *CMInterface*. *BrewingFacility* (Fig. 3(b)) serves coffee whenever two conditions are satisfied – a token with *true* is obtained in place *ready_bf,* and a token flows into place *coffee_request_bf* when requested coffee is not out of stock in place *storage*. Place *storage* keeps a table to count how many cups of coffee the machine has currently. If there is enough coffee, a user receives a cup of coffee in the *CMInterface*. If there is not enough coffee, then an exception is issued to *CoinHandler* through place *pay_return_bf*.

4.1 Translated Maude Code

In this section, we use a simple example to illustrate the experimental results of the translation algorithm as follows.

4.1.1 Mapping to Signature

Based on the above algorithm, we first translate the coffee machine example to obtain the function module. We consider one component *CoinHandler* as the example here. In the SAM specification, we define ports *request_ch* and *coin_back_ch* in the component *CMHandler*. The sorts of the port *request_ch* is defined as < *int*, *int* >, while the sorts of the port *coin_back_ch* is < *int* >. After the translation, we have following Maude program segment:

```
fmod CM-SIGNATURE is

protecting INT .
```

```
sort Marking .

op request-ch : Int Int -> Marking .

op coin-back-ch : Int -> Marking .

*** other operations are ignored.

endfm
```

4.1.2 Translation of System

In the coffee machine example, port *request_cmi*(*money*, *type*) is translated to an operation on sorts *int* and *Marking*. We want the data in this port go to the port *request_ch*(*money*, *type*) without condition. This behavior is controlled by the transition *enough* specified in the Petri net, and specified in the rewriting logic as the following formula:
$r1 : request\_cmi(money, type) \rightarrow request\_ch(money, type)$ if $money \geq cost$,
where operations *request_cmi*(*money*, *type*) and *request_ch*(*money*, *type*) are defined as follows:

$$request\_cmi(money, type) : int \times int \rightarrow Marking \text{ and}$$

$$request\_ch(money, type) : int \times int \rightarrow Marking.$$

In Maude, we have the following rule for this state changing:

```
mod CM is

including CM-SIGNATURE .

vars money type cost : Int .

crl [enough] : request-ch(money, type) price(type,cost) => coffee-request-ch(type)
save(money, type)

price(type,cost) if money >= cost .

*** other rules are ignored.
```

4.1.3 Mapping to State Predicates

After we have the translated function module and system module, we can define the state predicates with their semantics. In the Maude program, concerning the translation of the port *request_ch*(*money*, *type*) and *coin_back_ch*(*money*), we can have the following program segment by applying the translation algorithm:

```
mod CM-PREDS is

protecting CM .

including SATISFACTION .

subsort Marking < State .

***(CMHandler)

op Prequest-ch : Int Int -> Prop .
op Pcoin-back-ch : Int -> Prop .
vars money type cost capacity : Int .
vars m1 m2 : Marking .
eq m1 request-ch(money,type) m2 |= Prequest-ch(money,type) = true .
eq m1 coin-back-ch(money) m2 |= Pcoin-back-ch(money) = true .
*** others are ignored.
```

4.1.4 Mapping to Initial Marking & Property

Following is the segment of translated model checking module:

```
mod CM-CHECK is
including CM-PREDS .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .
vars money type : Int .
op init1 : -> Marking .
eq init1 = request-ch(75,2) price(1, 50) storage(1, 50) .
op property1 : -> Formula .
eq property1 = [] (Prequest-ch(money,type) -> <>(Pcoffee-request-ch(type) /\ Pcoin-back-
ch(money))) .
*** others properties are ignored
*** due to space limitation.
```

4.2 Discussion

We have examined several properties for this example, the running results evaluated from translate Maude program follow the expected results of the SAM model. This algorithm had been applied on several other examples such as cruise controller [5] and online shopping example [11]. All the experimental results demonstrated its efficiency and portability.

5. Translation Validation

In our translation, we simply flattened the SAM specification so that all compositions are dereferenced and all components and connectors are visualized by a Petri net. Petri nets of dereferenced compositions are connected together through merging shared ports. Then the whole system connects to one Petri net. This is shown in the M-SIGNATURE and M-SYSTEM module of the translation algorithm (Fig. 1). We translate the flattened SAM specification, i.e. its Petri nets into the rewriting logic. Therefore, we consider an individual flattened component/connector in the translation validation. To validate the translation, we first build a Petri net based on the rewriting logic. The semantics of the rewriting based Petri net can be expressed by a sequence of rewriting steps. Then we define the correspondence results between the operational semantics of Petri nets and the logic one, lifting it to the proved semantics.

5.1 Rewriting Theories and Deduction

The mapping of a Petri net model into a rewriting theory requires fairly sophisticated algebraic techniques. In the following we first recall the rewriting logic and membership equational logic [16].

5.1.1 Basic

Definition An equational logic is a pair (Σ, E), where Σ is a set of operations, also called its "syntax", and E is a set of equations of the form $\forall X.t = t'$ constraining the syntax, where X is some set of variables and t, t' are well-formed terms over variable set X and operations in Σ. Equational logics can be many-sorted (operations in Σ have arguments of specific sorts), or even order-sorted (sorts with a partial order on them). Equations can be conditional or unconditional. Conditions is a typically finite set of pairs $u = u'$ over set of variables X.
Term rewriting is an approach related to the equational logic (Σ, E) in which equations are oriented from left to right, with the form $\forall X.p \rightarrow q \text{ if } \wedge_i u_i \rightarrow u'_i$, and called rewrite rules. A rewrite rule can be applied to a term t at any position where p is matched. A pair (Σ, R), where R is a set of rewrite rules, is called a rewrite system.
Rewriting logic [26] is a logic for concurrency. A rewrite theory is a tuple (Σ, E, L, R), where (Σ, E) is an equational logic, L is set of labels, and R is a set of rewrite rules. Rewriting logic therefore extend equational logic with rewrite rules, allowing one to derive both equations and rewrites. Deduction remains the same for equations, but the symmetry rule is dropped for rewrite rules. Rewriting logic is a framework for true concurrency [27]: the locality of rules allows multiple rules to apply at the same time provided they don't modify the shared part. A formal definition of a labeled rewrite theory is given in the following.
Definition 1 An many-sorted labeled rewrite theory R is a 4-tuple $< \Sigma, E, L, R >$ where Σ is an many-sorted signature, E is a set of Σ-equations, L is the set of labels, and $R \subseteq L \times T\Sigma{,}E(X)^2$ is the set of labeled rewrite rules.
We often write.

$$l : [s] \rightarrow [t] \text{ if } [\overrightarrow{u}] \rightarrow [\overrightarrow{v}] \text{ for } l : [s] \rightarrow [t] \text{ if } [u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_k] \rightarrow [v_k].$$

Rewrite rules in R may be understood as the basic *rewriting steps* of a theory, the building blocks of the actual rewrite relation. More complex deductions can be obtained by a finite number of applications of inference rules. We introduce a suitable signature for building an algebra of labels, each element of the term algebra encoding a justification of a rewrite. Out of all possible different ways to introduce such a signature, we follow the lines of [4].
Definition 2 (rewriting step) Let R =< Σ, E, L, R > be a rewrite theory, let be the signature containing all the labels r as suitable operators, with the corresponding arity and sort given by the variables in R(r) A proof term α is a term of the algebra TR(X) = T_[(X) (We assume that there are no clashes of names between the sets of operators).

A rewriting step is a triple < α, [s], [t] > (usually written as α : [s] → [t]) where α is a proof term and [s], [t] 2 $T_{\Sigma,E}(X)$.

(reflexivity)	$\dfrac{[t] \in T_{\Sigma,E}(X)}{t : [t] \rightarrow [t]}$
(transitivity)	$\dfrac{\alpha : [s] \rightarrow [u], \beta : [u] \rightarrow [t]}{\alpha;\beta : [s] \rightarrow [t]}$
(congruence)	$\dfrac{f \in \Sigma_n, \alpha_i : [s_i] \rightarrow [t_i]}{f(\alpha_1, ..., \alpha_n) : [f(s_1, ..., s_n)] \rightarrow [f(t_1, ..., t_n)]}$
(replacement)	$\dfrac{\alpha_i : [w_i] \rightarrow [z_i], R \vdash r : [s] \rightarrow [t]}{r(\alpha_1, ..., \alpha_n, \alpha) : [s(\overrightarrow{w}/\overrightarrow{x})] \rightarrow [t(\overrightarrow{z}/\overrightarrow{x})]}$
(equality)	$\dfrac{E \vdash s = u, \alpha' : u \rightarrow u', E \vdash u' = t}{\alpha : s \rightarrow t}$
(composition)	$\dfrac{\alpha : [s] \rightarrow [u], \beta : [u] \rightarrow [t]}{\alpha;\beta : [s] \rightarrow [t]}$
(associativity)	$\dfrac{-}{\alpha; (\beta; \gamma) \equiv (\alpha; \beta); \gamma}$
(distributivity)	$\dfrac{f \in \Sigma}{f(\alpha_1;\beta_1, ..., \alpha_n;\beta_n) \equiv f(\alpha_1, ..., \alpha_n); f(\beta_1, ..., \beta_n)}$

Table 1. Inference Rules

As argued in [26], “a rewrite theory is just a static description of ‘what a system can do’; the behavior of the theory is instead given by the rewrite relation induced by the set of rules of deduction”. Given a set of rewriting rules, we can derive a series of rewriting theorems about the system. This procedure is called *entailment* and defined in the next definition.

Definition 3 (Entailment) Let R =< ∑, E, L, R > be a rewrite theory. We say that R entails the rewriting step α : [s] → [t], written as R ⊢ : [s] → [t], if and only if it can be obtained by a finite number of applications of the inference rules in Table 1.

5.2 A Theory for Petri nets

In Section 3, to describe local state represented by a place, we introduce a new sort *Marking* to rewriting logic. The sort *Marking* lifts the local state on sort description and relates to global state described by marking in Petri nets. Therefore, we use *Marking* as a sort of ∑ in the rewriting theory of Petri nets in the following.

Given a PrT net *N* = (*P*, *T*, *F*, ∑, *Eq*, φ, *L*, *G*, *M*0) [19], we define a rewriting theory R*PrT* = < ∑R, *E*R, *L*R, *R*R > for each ingredient of PrT nets as follows. If there is no confusion in the context, we refer R simply as R*PrT* . A rewriting theory of Petri nets can be defined as:

- SR = $St \cup \{Marking\}$ for $St \in \sum$ and OpR = $Op \cup Opp$, where $Opp = \{\varphi(p) \to Marking\}$ and $p \in P$. Then we have signature _R = (SR,OpR).
- ER = Eq.
- LR is set of labels, which is defined by a function such that LR : $P \cup T \to f\sum \cup R$R, where $f\sum$ denotes the operators on the signature $\sum$, if no confusion, we simply refer f as $f\sum$.
    - o $\forall p \in P$. LR(p) = $f\sum$, where f is defined as a mapping from sorts of p to the sort *marking*, i.e., $f : s \to$ m*arking*, where s is the sort of the place p.
    - o $\forall t \in T$. LR(t) = L(t), where $L \in N$.
    - o $\forall x \in X$. LR(x) = L(x), where $L \in N$.
    - o $\forall a \in F$. LR(a) = Φ. Label expression on each flow relation is used to instantiate tokens in a place.
    - o $\forall cons \in CONS$. LR($cons$) = $CONS$, where $CONs$ is a set of constants.
- RR is the set of rewriting rules defined by constraint set G that is associated with transitions $t \in T$. $\forall g \in G$, $t \forall T$,
    - o if $g = \{true\} \vee g = \Phi$ of the transition t, we have an unconditional rule with the form $l : [p] \to [q]$, where p, $q \in Op$R, OpR $\in \sum$R, with $p = Oppre(t)$ and $q = Oppost(t)$.
    - o otherwise, $g = \{TermOp,bool(X)\}$, then l is conditional rule with the form $l : [p] \to [q]$ *if* $\wedge i$ Ci where $Ci \in TermOp,bool(X)$ and $\forall vi \in Ci$. $vi \in X$, with $p = Oppre(t)$ and $q = Oppost(t)$ respectively.

It is worth to note that ER can be empty. We define a rewriting step for a Petri net based rewriting theory RPrT as follows.

Definition 4 (rewriting step) Let RPrT = < ∑R, ER, LR, RR > be a rewrite theory of a PrT, a proof term α is a term of the algebra TR(X) = T∑(X) (We assume that there are no clashes of names between the sets of operators).

A rewriting step is a triple < Δ, [s], [t] > (usually written as Δ : [s] $\to$ [t]) where α is a proof term and [s], [t] 2 T∑,E(X), s = pre(t)(φ(p) : α), and t = post(t)(φ(p) : α).

With the rewriting theory of Petri nets, we can give the following interleaving correspondence based on the defined rewriting step.

5.3 An Interleaving Correspondence

We state a claim that relates the semantics of PrT nets based on the rewriting theory RPrT . First we define a series of execution status of RPrT .

Definition 5 Let Δ be a closed proof term over the rewriting theory RPrT . Then we have Δ as

- *initial, if it is element of sort SPrT .*
- *enabled, if it is initial and contains occurrence of an operator in Op*R *and E*R $\neq \Phi$.
- *one step, if it is enabled and contains occurrence of M*(p) *with* $p \in P$ *and no occurrence of composition operator ;.*
- *fired, if it is one step.*
- *many steps, if* $\Delta = \Delta 1, \ldots, \Delta n$ *with* $1 \leq n \leq Nat$ *and each* Δi *is one step.*
- *rewriting step, if it is many steps and all the one-step rewriting step is enabled.*

In the following, we define the relation between an occurrence for a transition $t \in T$ of a PrT net and a rewriting step Δ.

Definition 6 An occurrence for a transition t ∈ T, a guard g ∈ G of a PrT net N is a rewriting step $\Delta t : [p] \rightarrow [q]$ if $\wedge i$ Ci with the following conditions:

- *a transition t is initial, if $g(t) \in Term(X)$ and Δt is initial.*
- *a transition t is enabled, if t is initial and Δt is enabled.*
- *a transition t is fired once under marking M1 and reach marking M2, if Δt is one step.*

This characterization is needed to prove the correspondence between PrT nets and rewriting theory R*PrT* : given a PrT net *N*, there is a one-to-one correspondence between the computation sequence of PrT net *N* and the rewriting theory R*PrT* .

Proposition 1 (One-step Correspondence) Let t ∈ T be a transition in PrT net N, M1 and M2 be two markings before and after t is fired, then a one step computation sequence for PrT net N is M1tM2.

If RPrT entails an initial, one-step rewriting step $\Delta : pre(t)(\varphi(p) : \alpha) \rightarrow post(t)(\varphi(p) : \alpha)$, then there is a computation sequence M1tM2 with $pre(t)(\varphi(p) : \alpha) \in M1$ and $post(t)(\varphi(p) : \alpha) \in M2$ defined for PrT nets.

Proof.

The proof has two steps on the definition of rewriting step. For any transition $t \in T$, from Subsection 2.1.1, with a one step computation sequence for PrT net *N M*1*tM*2,
∃consuming-token-p(L((p, t), Terms∑,X)) such that consuming-token-p(L((p, t), Terms∑,X)) ∈ M1, and ∃producing-token-q(L((t, q), Terms∑,X)) such that producing-token-q(L((t, q), Terms∑,X)) ∈ M2.
Based on the definition 1, we can conclude the above proposition.

endProof.

We already observed in previous section that each place defines an _-signature on the set of sorts and the fixed sort *marking* as the operation/function with arguments. The markings are connected by transition firing defined by the Petri net semantics [19]. Thus we can conclude this section by the correspondence result between Petri net computational sequence semantics and the rewriting theory semantics.

Corollary 1 Let Mi and Mj be two markings of a PrT net N. Then a computation sequence Mi…Mj entails a transition Mi to Mj iff RPrT entails a rewriting step $\Delta : Mi(p) \rightarrow Mj(p)$.

Finally, next result lifts the correspondence to computations.

Proposition 2 (Computational Correspondence) Let M_{c0} and Mc_j be two markings of a PrT net N. Then there exists a proved computation $\sigma = M_{c0tc0} \ldots t_{c\ j}\text{-}1M_{cj}$ with source M_{c0} and target Mcj iff RPrT entails an initial one step sequential rewriting step $\Delta = \Delta 1, \ldots, \Delta m$ with $\Delta i : pre(tk)(\varphi(p) : \alpha) \rightarrow post(t)(\varphi(p) : \alpha)$ where $i \leq k \leq j$.

Proof.

This can be proved inductively on the computation sequence.

Base case: $k = c0$ we have the computation sequence $Mc0tc0Mc1$, from Proposition for the One-step Correspondence, we have $\Delta c0 : pre(tc0)(\varphi(p) : \alpha) \rightarrow post(tc0)(\varphi(p) : \alpha)$.

Hypothesis Assumption: Suppose it is true for $k = n$ where $i \leq n \leq j$ the proposition for computational correspondence holds. Then we have $\Delta = \Delta 1, \ldots, \Delta m$ with $\Delta i : (pre(tk))(\varphi(p) : \alpha) \rightarrow (post(tk))(\varphi(p) : \alpha)$ holds. Because R*PrT* entails an initial one step sequential rewriting step $\Delta = \Delta 1, \ldots, \Delta m$, there exists one rewriting step $\Delta I : \Delta 1, \ldots, \Delta i$ where $i \leq k \leq j$ and $\Delta i : pre(tk)(\varphi(p) : \alpha) \rightarrow post(tk)(\varphi(p) : \alpha)$. We need to show for $k = n + 1$ it still holds.
If $k = n + 1$, we need to consider two situations:

1) one is that $post(tn) = pre(tn+1)$;
2) $post(tn)$ is not $pre(tn+1)$. Because we use interleaving semantics for the PrT nets, and in the rewriting theory, each rewriting rule can be executed concurrently, *tn* and *tn*+1 may be in a causal relation.

Based on the one-step correspondence, for each transition firing, we can have a rewriting step, so for *tn*+1 under some marking *Mn*, we have $\Delta cn+1 : pre(tcn+1)(\varphi(p) : \alpha) \rightarrow post(tcn+1)(\varphi(p) : \alpha)$.
For case (1), under the marking *Mn*, based on the hypothesis assumption, we know that $(post(tn)) = (pre(tn+1))$. From transition rule, the proposition holds.
For case (2), there must be some transition *th* fired and make the proposition holds, i.e., let *th* be the transition in the hypothesis assumption, that is $th = tn$. Similarly, the proposition holds.
endProof.
The characterization above is summarized as follows: for each firing transition, there is in fact a one-to-one correspondence between proved computations starting from the preset of the transition to the postset of transition and rewriting step with substitution _. Furthermore, for a firing sequence under a series of markings, there is a correspondence between the execution of the constructed Petri net and the rewriting sequence. The proofs of the correspondence validate the correctness of translation in Section 3.

6. Conclusion

We discussed a rewriting semantics of a software architecture specification (SAM) by a translation from specification to a rewriting logic based declarative programming language Maude. In addition, to validate the translation, we have shown the correctness of the translation by a stepwise proof. A case study of a coffee machine modeled in SAM is applied on the translation algorithm. This paper provides an alternative concurrent semantics of a formal architecture specification (SAM) using rewriting logic and equational logic implemented on Maude. As a high performance declarative programming language platform, Maude is efficient in different types of systems.
Maude provides modules that convert model in rewriting logic to Kripke structure so that it is able to do model checking using Maude. Moreover, Maude supports the specification of nested processes, which is hard accomplished in other model checking systems such as SMV or SPIN. It is known that nested processes are the natural way to specify distributed systems, which are supported by SAM. The Maude can model check systems whose states involve data types in any algebraic data types [6]. Other model checkers only support limited data types in terms of which all other data must be encoded. However the resulting Maude program after translation is kind of large, which is hard to debug even with the support of Maude tool.

An important aspect of our work is the validation of translation. In addition to the fact that translation validation is an important pragmatic procedure, we believe it is quite interesting in the context of architecture specification verification and automatic code generation. Given a SAM component or connector, it can be fully automatically translated to a Maude system module on rewriting logic without the need of any information from other components in a given architecture description. As the only visible entity in the architecture model, ports are mapped to the operations in the signature from sorts to the local state representation *Marking* and proposition sort *Prop*.
The transitions are mapped to rules in the rewriting logic with the variable substitution mapped to the rewriting steps. The semantics consistency between Petri nets' and rewriting logic's is seamlessly established through these mappings.
We are also interested in a deeper treatment of specification and verification features with a special focus on model checking techniques to tackle the state space problem. We will also work on validating the usability and expressiveness of SAM specification by implementing encodings of various process calculi that can be developed using Petri net and temporal logic.

7. References

[1] R. J. Allen. A formal approach to software architecture. PhD thesis, 1997. Chair-David Garlan.
[2] M. Clavel, F. Dur´an, S. Eker, P. Lincoln, N. Mart´ı-Oliet, J. Meseguer, and C. Talcott. Maude manual, 2004.
[3] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. volume 4 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 1996.
[4] A. Corradini and F. Gadducci. Rational term rewriting. Lecture Notes in Computer Science, 1378:156–171, 1998.
[5] Z. Dong, Y. Fu, and X. He. Automated runtime validation of software architecture design. In Proceedings of Second International Conference of Distributed Computing and Internet Technology (ICDCIT 2005), volume 3816 of Lecture Notes in Computer Science, pages 446–457. Springer, 2005.
[6] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude LTL model checker. In Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA'02), volume 71 of Electronic Notes in Theoretical Computer Science, Amsterdam, September 2002. Elsevier.
[7] C. C. Frederiksen. Correctness of classical compiler optimizations using CTL. In Proceedings of theETAPS 2002 :European Joint Conference on Theory and Practice of Software, pages 41–55, 2002.
[8] Y. Fu, Z. Dong, G. Argote-Garcia, L. Shi, and X. He. An Approach to Validating Translation Correctness From SAM to Java. In Proceedings of The Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE2007), 2007.
[9] Y. Fu, Z. Dong, and X. He. A Methodology of Automated Realization of a Software Architecture Design. In Proceedings of The Seventeenth International Conference on Software Engineering and Knowledge Engineering (SEKE2005), 2005.

[10] Y. Fu, Z. Dong, and X. He. An Integrated Runtime Monitoring Framework for Software Architecture Design. In Proceedings of the Software Engineering and Applications (SEA'05), 2005.
[11] Y. Fu, Z. Dong, and X. He. An Approach to Web Services Oriented Modeling and Validation. In Proceedings of the 28th ICSE workshop on Service Oriented Software Engineering (SOSE2006), 2006.
[12] Y. Fu, Z. Dong, and X. He. A method for realizing software architecture design. In Proceedings of the Sixth International Conference on Quality Software(QSIC'06), pages 57–64, Washington, DC, USA, 2006. IEEE Computer Society.
[13] Y. Fu, Z. Dong, and X. He. A translator of software architecture design from sam to java. International Journal of Software Engineering and Knowledge Engineering, 17(6):1–54, 2007.
[14] H. J. Genrich. Predicate/Transition Nets. Lecture Notes in Computer Science, 254, 1987.
[15] S. Glesner, R. Gei¨s, and B. Boesler. Verified code generation for embedded systems. In Proceedings of the COCVWorkshop (Compiler Optimization meets Compiler Verification), volume 65. Electronic Notes in Theoretical Computer Science (ENTCS), 5th European Conferences on Theory and Practice of Software (ETAPS 2002),, April 13 2002.
[16] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science, 105(2):217–273, 1992.
[17] K. Havelund. Using runtime analysis to guide model checking of java programs. In Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, pages 245–264, London, UK, 2000. Springer-Verlag.
[18] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In Proceedings of Tools and Algorithms for Construction and Analysis of Systems, pages 342–356, 2002.
[19] X. He. A formal definition of hierarchical predicate transition nets. In Proceedings of the 17th International Conference on Application and Theory of Petri Nets, pages 212–229, London, UK, 1996. Springer-Verlag.
[20] X. He, J. Ding, and Y. Deng. Model checking software architecture specifications in sam. In Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE'02), volume 27 of ACM International Conference Proceeding Series, pages 271–274, New York, NY, USA, 2002. ACM Press.
[21] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng. Formally analyzing software architectural specifications using sam. Journal of Systems and Software, 71(1-2):11–29, 2004.
[22] K. Korenblat, O. Grumberg, and S. Katz. Translations between textual transition systems and petri nets. In Proceedings of the Third International Conference on Integrated Formal Methods (IFM'02), pages 339–359, London, UK, 2002. Springer-Verlag.
[23] L. Lu, X. Li, Y. Xiong, and X. Zhou. Xm-adl: An extensible markup architecture description language. In IEEE, pages 63–67. IEEE Press, 2002.
[24] D. Luckham, J. Kenney, and L. A. et al. Specification and analysis of system architecture using rapide. volume 21 of IEEE Transactions on Software Engineering, pages 336–355, 1995.

[25] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. Software Engineering, 26(1):70–93, 2000.

[26] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. volume 96, pages 73–155, 1992.

[27] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Proceedings of International Conference on Concurrency Theory, pages 331–372, 1996.

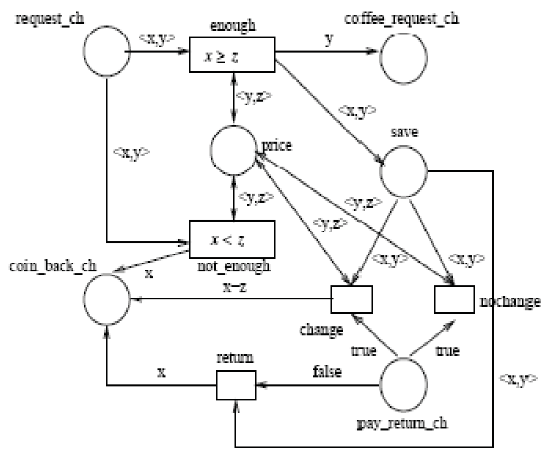
[28] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. IEEE Transactions on Software Engineering, 22(6):390–406, 1996.

[29] W. M. P. van der Aalst, K. M. van Hee, and R. A. van der Toorn. Component-based software architectures: A framework based on inheritance of behavior. Science of Computer Programming, 42(2-3):129–171, 2002.

[30] J. Wang, X. He, and Y. Deng. Introducing Software Architecture Specification and Analysis in SAM through an Example. Information and Software Technology, 41(7):451–467, 1999.

[31] F. Xie, V. Levin, R. P. Kurshan, and J. C. Browne. Translating Software Designs for Model Checking. In Proceedings of 7th International Conference Fundamental Approach to Software Engineering (FASE), volume 2984, pages 324–338. Springer-Verlag, 2004.

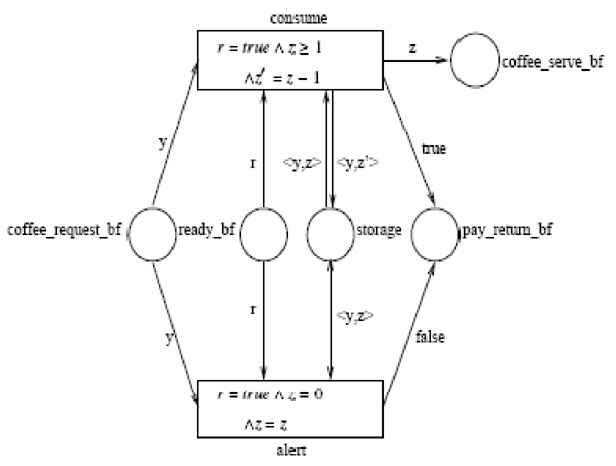
A. Behavior Model of Components in Coffee Machine
The behavior model of each component in the coffee machine example is shown in the Fig. 3.



$$\varphi = \{request\_ch \mapsto <int, int>, price \mapsto <int, int>, coffee\_request\_ch \mapsto int, coin\_back \mapsto int, save \mapsto <int, int>, pay\_return\_ch \mapsto boolean\}$$

$$M_0 = \{request\_ch \mapsto \{\}, price \mapsto \{<1, 50>, <2, 75> <3, 100>\}, coffee\_request\_ch \mapsto \{\}, coin\_back\_ch \mapsto \{\}, save \mapsto \{\}, pay\_return\_ch \mapsto \{\}\}$$

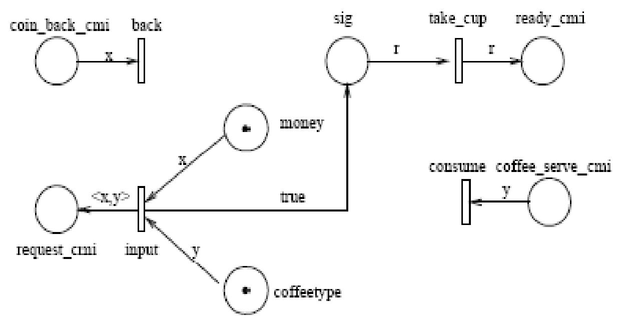
(a) Component CoinHandler



$$\varphi = \{coffee\_request\_bf \mapsto int, ready\_bf \mapsto boolean, storage \mapsto <int, int>, pay\_return\_bf \mapsto boolean, coffee\_serve\_bf \mapsto int\}$$

$$M_0 = \{coffee\_request\_bf \mapsto \{\}, ready\_bf \mapsto \{\}, storage \mapsto \{<1, 50>, <2, 50>, <3, 50>\}, pay\_return\_bf \mapsto \{\}, coffee\_serve\_bf \mapsto \{\}\}$$

(b) Component BrewingFacility



$$\varphi = \{money \mapsto int, coffee \mapsto int, request\_cmi \mapsto <int, int>, coin\_back\_cmi \mapsto int, sig \mapsto boolean, ready\_cmi \mapsto boolean, coffee\_serve\_cmi \mapsto int\}$$

$$M_0 = \{money \mapsto 75, coffeetype \mapsto 2, sig \mapsto \{\}, ready\_cmi \mapsto \{\}, coin\_back\_cmi \mapsto \{\}, request\_cmi \mapsto \{\}, coffee\_serve\_cmi \mapsto \{\}\}$$

(c) Component CMInterface

Fig. 3. Behavior of Subcomponents in CoffeeMachine

Petri Nets Applications
Edited by Pawel Pawlewski

ISBN 978-953-307-047-6
Hard cover, 752 pages
Publisher InTech
Published online 01, February, 2010
Published in print edition February, 2010

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Yujian Fu, Zhijiang Dong, Phil Bording and Xudong He (2010). Towards Rewriting Semantics of Software Architecture Specification, Petri Nets Applications, Pawel Pawlewski (Ed.), ISBN: 978-953-307-047-6, InTech, Available from: http://www.intechopen.com/books/petri-nets-applications/towards-rewriting-semantics-of-software-architecture-specification