An Approach for Modeling and Analyzing Crosscutting Concerns

Yujian Fu, Junhua Ding, Phil Bording

Abstract— **Aspect-oriented software development (AOSD) is a promising technique for modeling crosscutting concerns, but formal specification and analysis of AOSD concerns is not well exploited. In this paper, we propose an approach for formally modeling and analyzing crosscutting concerns in software. We designed an aspect-oriented Petri net with AOSD mechanisms for identifying and modularizing crosscutting concerns. In order to analyze concern interactions and other properties, we developed an automated approach for formally analyzing the software design using a model checking technique. We demonstrate the effectiveness and feasibility of our approach through modeling and analyzing a clinical diagnostic system.**

I. Introduction

Software development often addresses many concerns. A concern in software defined as is anything a developer may consider as a conceptual unit such as features, security requirements, and design idioms [13]. The concern that is scattered and tangled throughout multiple modules is called a crosscutting concern [13]. Security concerns are representative crosscutting concerns because it is difficult to separate them from the tangling with other modules. AOSD was developed to identify, isolate and modularize crosscutting concerns through introducing the aspects concepts so that hard-coded tangled connections with crosscutting concerns are not needed, instead crosscutting concerns are separated and encapsulated as aspects to be composed with other modules automatically. AOSD is an extension of object-oriented (OO) technology [1] and/or other software development methodologies via integrating linguistic mechanisms from aspect-oriented programming (AOP) [3]. Comparing to OO models, AOSD models are easily to be understood, reused and maintained thanks to the isolation and modularization of crosscutting concerns. While the ability to modularize crosscutting concerns appears to improve the quality, AOSD does not assure the correctness by itself. Aspects may be used in a harmful way that invalidates desired properties and even destroys the conceptual integrity of systems [7] [18]. For example, we assume different aspects are superimposed on the same join point this may cause problems if the execution orders or dependency of these aspects cannot be determined. To assure the trustworthiness of an AOSD design, formal analysis of the design is highly desirable. However, much work on AOSD focus on modeling or programming of crosscutting concerns, formal analysis of AOSD concerns is not well exploited. Due to the complexity of formal analysis of an AOSD model, a feasible automated approach for rigorously analyzing AOSD design is extremely necessary [18]. Because formal analysis uses mathematical methods to analyze software, a formal language for modeling software will minimize the gap between the software specification and analysis. In this paper, we first exploit formal modeling of crosscutting concerns by defining a formalism called aspect-oriented Predicate/Transition nets (AOPrT), which are an extension of Predicate/Transition (PrT) nets with integrating essential AOSD mechanisms such as aspects and pointcuts. Then we investigate formal analysis of crosscutting concerns using model checking technique. In our review of a clinical diagnostic algorithm, we show the approach is effective for assuring quality of AOSD design specifications.

Yujian Fu is with the Department of Computer Science, Alabama A&M University, Normal, AL 35762, USA yujian.fu@aamu.edu

Juhua Ding is with the Department of Computer Science, East Carolina University, Greenville, NC 27858, USA dingj@cs.ecu.edu

Phil Bording is with the Department of Computer Science, Alabama A&M University, Normal, AL 35762, USA phil.bording@aamu.edu

The main contributions of this paper include: a. A new AOPrT net for modeling crosscutting concerns in software design. Comparing to existing aspect-oriented Petri nets, the new AOPrT nets provide a more feasible approach for weaving aspect nets with their base nets under different situations. A set of rules is developed for resolving the dependency and conflict issues among aspects. In addition, the weaving algorithm in the new AOPrT nets is fairly simple and easily to be automated thanks to the elegance of advice types and the simplicity of join points in the AOPrT nets. b. An automated approach using model checking technique for analyzing AOPrT net models. The analysis approach provides a practical solution to assure the correctness of an AOSD design. We report a case study on a clinical diagnostic system that demonstrates the feasibility and effectiveness of our approach.

In the next section, we first briefly describe PrT nets, and then we discuss the design of aspect-oriented PrT nets for modeling crosscutting concerns. Section III discusses the approach for analyzing semantic conflicts and interactions between crosscutting concerns. In section IV, we discuss the specification and analysis approach through studying a clinical diagnostic system. We conclude with a discussion of related work in section V and a summary of the paper in section VI.

II. Modeling Crosscutting Concerns

In order to formally analyze crosscutting concerns in software, we need an AOSD language for formally modeling the concerns. The desired modeling language should be a mathematical language with graphical notations so that the semantics of the language are well defined, and its models are easily understood. In addition, the language should support AOSD modeling, and the language should be executable

978-1-4244-3541-8/09/$25.00 ©2009 IEEE

so that simulation of software design is possible. Based on above requirements, we develop an AOSD modeling language called AOPrT nets, which are a formal language with graphical notations to support AOSD modeling, via incorporating PrT nets with fundamental features of AOSD. Petri nets [11] are a graphical and mathematical modeling language well suited for defining and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. PrT nets are a type of high-level Petri nets where tokens carry data types, labels on arcs filter incoming or outputting tokens, and constraint expressions on transitions restrict the enabling of transitions. The details on PrT nets can be found at [8].

Definition 1 (PrT Nets). A PrT net consists of: (1) a finite net structure (P, T, F), (2) an algebraic specification $SPEC$, and (3) a net inscription (φ, L, R, M_0) [8, pp. 459-476]. P and T are the set of predicates and transitions, respectively, where $P \cap T = \emptyset$. F is the flow relation where $F \subseteq P \times T \cup T \times P$. $SPEC$ is a meta-language to define the tokens, labels, and constraints of a PrT net. The underlying specification $SPEC = (S, OP, Eq)$ consists of a signature $\Sigma = (S, OP)$ and a set Eq of Σ-equations. S is a set of sorts and OP is a family of sorted operations. Tokens of a PrT net are ground terms of the signature Σ, written $MCON_S$. The set of labels is denoted by $Label_S(X)$, where X is the set of sorted variables disjoint with OP. Each label can be a multiset expression of the form $\{k_1x_1, ..., k_nx_n\}$. Constraints of a PrT net are a subset of first order logic formulas containing the S-terms of sort *bool* over X, denoted as $Term_{OP,bool}(X)$.

A. Definitions for Aspect-Oriented PrT Nets

AOPrT nets are PrT nets extended with aspect-oriented modeling mechanisms including join points, pointcuts, and aspects. In addition, algorithms for automatically weaving aspects with join points based on pointcuts have to be defined. An AOPrT model includes a group of base nets and a group of aspects. Base nets are regular PrT nets, and an aspect includes pointcuts and advice nets with their advice types. Advices nets can be automatically composed with join points based on pointcuts. The weaving algorithm defines how an aspect is composed with a join point.

Definition 2 (AOPrT Nets). An aspect-oriented PrT net is a 2-tuple: $AOPrT = (BN, AN)$, where: $BN = (P, T, F)$ is a base net, which is a regular PrT net. $AN = (AN_1, AN_2, ..., AN_m)(m \geq 0)$ is a group of aspects.

Definition 3 (Aspects). An aspect AN_i in AOPrT net is a structure $AN_i = < P, A >$, where P is a set of pointcuts, A is a set of advice nets. Each advice net A includes an advice name AI, an advice type $< advicetype >$ $(< parameters >)$, and a regular PrT net AT.

Definition 4 (Join points). A joint point is the position where advice nets may be composed with. A joint point is a transition in base nets.

Definition 5 (Pointcuts). A pointcut defines rules in an aspect for locating join points in base nets. A pointcut PT is an expression $PT = < pointcutname > (< parameters >) < basenet > . < transition >$, where $pointcutname$ identifies a pointcut, $parameters$ define formal parameters for chosing join points, and $< basenet > . < transition >$ refers to a transition selected from the based net by the pointcut.

Definition 6 (Advice Types). There are four advice types for weaving advice nets: *before*, *after*, *around* and *concurrent*. *before* means the aspect will be composed before the matched join points, *after* means the aspect will be composed after the matched join points, *around* means the aspect will be composed to replace the matched join points, and *concurrent* means the aspect will be composed concurrently to the matched join points. Each *advice type* is followed by its *parameters*, which define the weaving between an advice net and its join points.

B. Weaving Aspects

The parameters of an advice type decide the connection of an advice net and a join point. The following description defines how an advice net is connected with a join point under four different advice types with parameters, where A represents an advice net in an aspect, p_i represents a place in A, t_j or t'_jrepresents a transition in A, and l_k, and l'_q represent labels in arcs that connect the base net with the advice net.

1) Advice type *before*: $before(A.t_1 < l_1 >, A.t_2 < l_2 >, ..., A.t_n < l_n >; A.p_1 < l'_1 >, A.p_2 < l'_2 >, ..., A.p_m < l'_m >)$. The first half before the ";" defines the transitions $\{A.t_1, ..., A.t_x\}$ ($\{A.t_1, ..., A.t_x\} \subseteq \{A.t_1 ... A.t_m\}$) at advice net A to be connected with the input places of the matched join point t. All input places of the join point t become the input places of each transition at $\{A.t_1, ..., A.t_x\}$, and the input places are disconnected from the join point t. The second half after the ";" defines the places $\{A.p_1, ..., A.p_y\}$ ($\{A.p_1, ..., A.p_y\} \subseteq \{A.p_1, ..., A.p_m\}$) at A to be connected with the join point t, so that each place at $\{A.p_1, ..., A.p_y\}$ serves as an input place of the join point t.
2) Advice type *after*: $after(A.p_1 < l_1 >, A.p_2 < l_2 >, ..., p.t_n < l_n >; A.t_1 < l'_1 >, A.t_2 < l'_2 >, ..., A.t_m < l'_m >)$. The first half before the ";" defines the places $\{A.p_1, ..., A.p_y\}$ ($\{A.p_1, ..., A.p_y\} \subseteq \{A.p_1, ..., A.p_m\}$) at advice net A to be connected with the matched join point t as output places of t. The connection between the join point t and its original output places are disconnected. The second half after the ";" defines the transitions $\{A.t_1, ..., A.t_x\}$ ($\{A.t_1, ..., A.t_x\} \subseteq \{A.t_1, ..., A.t_m\}$) at advice net A to be connected with the original output places of the join point t, so that each original output place of the join point t becomes one of the output places of each transition at $\{A.t_1, ..., A.t_x\}$.
3) Advice type *around*: $around(A.t_1 < l_1 >, A.t_2 < l_2 >, ..., A.t_n < l_n >; A.t'_1 < l'_1 >, A.t'_2 < l'_2 >, ..., A.t'_m < l'_m >)$. The first half before the ";" defines the transitions $\{A.t_1, ... A.t_x\}$ ($\{A.t_1, ..., A.t_x\} \subseteq$

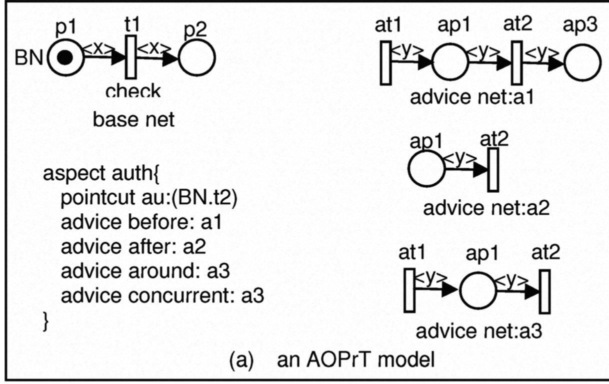

Fig. 1. An AOPrT model

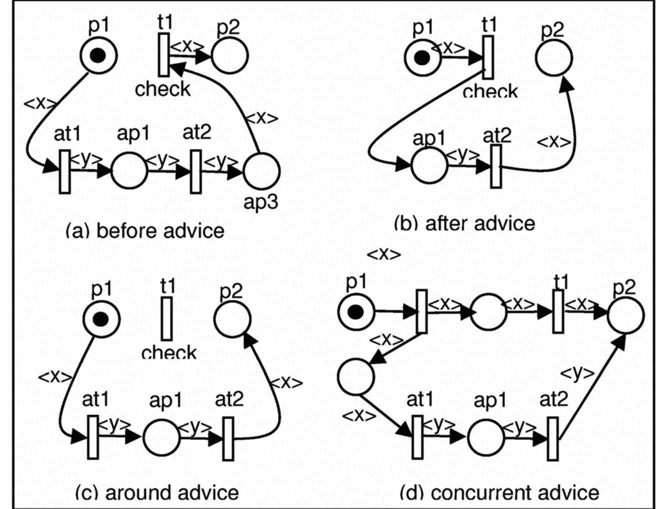

Fig. 2. Weaving an advice net (a) before advice. (b) after advice. (c) around advice. (d) concurrent advice.

$\{A.t_1, ..., A.t_n\}$) at advice net A to be connected with the input places of the matched join point t. All input places of the join point t become input places of each transition at $\{A.t_1, ..., A.t_x\}$, and the input places are disconnected from the join point t. The second half after the ";" defines the transitions $\{A.t_1, ...A.t_y\}$ ($\{A.t_1, ..., A.t_y\} \subseteq \{A.t_1, ..., A.t_m\}$) at advice net A to be connected with the output places of the join point t, so that each output place of the join point t becomes one of output places of each transitions at $\{A.t_1, ..., A.t_y\}$. The join point t and its original output places are disconnected.

4) Advice type *concurrent*: $concurrent(A.t_1 < l_1 >, A.t_2 < l_2 >, ..., A.t_n < l_n >; A.t'_1 < l'_1 >, A.t'_2 < l'_2 >, ..., A.t'_m < l'_m >)$. The first half before the ";" defines the transitions $\{A.t_1, ...A.t_x\}$ ($\{A.t_1, ..., A.t_x\} \subseteq \{A.t_1, ..., A.t_n\}$) at advice net A to be connected with the input places of the matched join point t. All input places of the join point will be duplicated as the input places for each transition at $\{A.t_1, ..., A.t_x\}$. Duplication of a place p in a base net is implemented through adding p as an input place to a new transition t, which has two new output places p_1, p_2, which are identical to the place p. p_1 is connected to the join point t as an input place, and p_2 is connected to transitions at $\{A.t_1, ..., A.t_x\}$ as an input place. The label on each new arc is same as the label on the arc that originally connects p to the join point t. The second half after the ";" defines the transitions $\{A.t_1, ...A.t_x\}$ ($\{A.t_1, ..., A.t_x\} \subseteq \{A.t_1, ..., A.t_m\}$) at advice net A to be connected with the output places of the join point t, so that each output place of the join point becomes one of the output places of each transition at $\{A.t'_1, ..., A.t'_y\}$.

The parameters for a advice type are optional. If there is only one scenario for weaving an advice net with its join points, the *parameters* for the advice type is not needed.

Figure 2 shows the four different weaving scenarios for weaving an advice net with a join point at Figure 1.

In order to analyze or simulate an AOPrT model, it is necessary to weave all aspects with their join points. Here we define an algorithm for weaving an aspect with its join points. We denote the pointcuts, advice types, and advice nets of aspect A as $A.P$, $A.T$, and $A.N$, respectively.

Algorithm 1 (Weaving). Given a based net B and aspect A. For each poincut in $A.P$, denoted by $tcut(x_1, ..., x_n)$. The advice nets in $A.N$ are denoted as $\{A.N_1 A.T_1, ..., A.N_m A.T_m\}$. The weaving mechanism for weaving aspect A with base net B is defined as follow:

1) For each pointcut in $A.P$, say $tcut_i$, find its corresponding advice net at $A.N$, say $A.N_j A.T_j$.
2) Find all join points at base net B using pointcut expressions $\{x_1, ..., x_n\}$ at $tcut_i$.
3) For each join point, say $B.t_k$, composing $A.N_j$ with $B.t_k$ according to advice type $A.T_j$ using the procedures defined at above section.
4) As soon as every pointcut in $A.P$ is processed, it may need rename places, transitions, labels or expressions in the woven net to resolve the name conflict.

C. Identification of Conflicts between Aspects

In AOPrT nets, multiple advice nets with same advice type can be woven with the same join point, which is referred as shared join point (SJP) [12]. The composition of multiple aspects at the SJP raises several problems like the execution order and the dependency between aspects [12]. Following the same definitions in [12], we define four different relations between aspects at SJP. A and B are aspects that can be composed with the same join point, then the relation between A and B is one of the four cases defined in aspect relations at SJP.

Definition 7 (Aspect Relations at SJP). There are four and only relations among aspects at SJP: $A \parallel B$, $A \rightarrow B$, $A \rightharpoonup B$, and $A \mid B$.

1) $A \parallel B$ means the execution sequence between A and B does not matter.

2) $A \rightarrow B$ means B can never be executed until A has been executed.
3) $A \rightharpoonup B$ means the execution results of A decide whether B will be executed or not.
4) $A|B$ means A or B will execute but not both.

From Defintion 7, it is not difficult to show that the relation $\rightarrow$ and $\rightharpoonup$ are both transitive. If $A \rightarrow B$, and $B \rightarrow C$, then $A \rightarrow C$, and if $A \rightharpoonup B$, and $B \rightharpoonup C$, then $A \rightharpoonup C$. The combination relations between above two relations are also transitive. If $A \rightarrow B$, and $B \rightharpoonup C$, then $A \rightharpoonup C$. If $A \rightharpoonup B$, and $B \rightarrow C$, then $A \rightharpoonup C$.

The definition of the aspect relations at SJP solves the problem on the execution order of aspects, but the dependency between aspects is not obvious, an algorithm is required to detect the dependency between aspects. The basic idea is to check each SJP to build a dependency path based on the aspect execution orders and transitive relations at the SJP, and then the dependency among aspects can be deduced based on the dependency path. If an aspect depends on itself, such as $A \rightharpoonup A$, or $A \rightarrow A$ can be deduced from the dependency path (i.e., same aspect appears in the dependency path at least twice), the relation is called conflict [5].

III. Analyzing Aspects

The challenge issue on analysis of aspects is related to semantic interference between aspects. These kinds of issues are extremely hard to detect because these are syntactically sound and only exposed when the composed model executes [9]. In order to analyze the semantic interference between aspects, the model checking technique [2] is used to perform the analysis. Model checking can automatically check whether a model meets given properties through algorithmically analyzing the state graph of the model. Properties can be defined using temporal logic formulas [2]. Before we can perform model checking, an AOPrT model might need to be woven, syntactic errors have to be corrected, and execution sequences at SJP have to be determined. Then the properties to be analyzed are defined. Finally the woven net and properties are input to the model checking tool for checking. Depending on capacities of the model checking tools, the woven nets and properties might need to be converted to models that are acceptable to the model checker. We chose a formal analysis tool called PROD [17], which is used for reachability analysis or model checking of PrT nets, to analyze AOPrT models. Because a woven AOPrT net model is a PrT net, it is a straightforward work for using PROD to check an AOPrT model. Analyzing an AOPrT model may check the conflicts of aspects at SJP, the aspect interferences, and conflicts between a base net and aspects. We discuss analysis in the following three aspects.

1) Analyzing conflicts of aspects at SJP. If an aspect depends itself, the dependency is easily found during building the dependency path, which is completed as soon as a woven net is generated. Checking an aspect itself can be applied to the advice net directly because it is a regular PrT net. Before an aspect can be composed with its base net, it should be checked using the model checker first. If two aspects are composed with the same join point, then these two aspects (actually their advice nets) should be composed together based on their dependency relations before the analysis can be applied to the composed net.
2) Analyzing aspect interferences. The interference between aspects that do not share the SJP only can be checked through composing these aspects with the base nets.
3) Analyzing conflicts between a base net and aspects. Base net itself is a regular PrT net, therefore, base net can be analyzed without composing with aspects. In order to analyze the conflicts between an base and aspects, the base net and advice nets should be checked first, and then the woven net composing the base net with advice nets is used for checking the interaction between the base net and aspects.

IV. A Case Study

Donor screening system is a type of clinical diagnostic systems used for screening virus in donated blood. Data integrity is one of the most important issues in donor screening systems, which means the data consistency has to be verified with historical and other correlated information every time when any data is added or updated. Data integrity is a crosscutting concern because many modules have to check the data integrity, such as an analysis instrument may add data, doctor may manually make an analysis conclusion and a technician may delete some records. Here is the simplified description for checking the data integrity for analyzing HIV at human blood samples.

In order to test HIV in the blood from a donor, five blood samples from the donor are tested using different techniques. Before an HIV result is added, it still needs referring other test records to decide the status of the current record. The following constraints are used for testing blood samples and adding or updating results:

1) Two samples are analyzed using the same primary testing techniques at the same time. If both test results of the two samples are negative (or NEG), then the conclusion of the HIV test result is automatically made as NEG. Then HIV result of the donor can be added to the laboratory information system (LIS). All five samples are recorded as NEG in the LIS.
2) If any of the two primary tests is positive (or POS, i.e., the donor is an HIV carrier), then further tests needed to be performed on the remaining three samples. If all three remaining test results are NEG, then NEG result of the donor will be added to the LIS. Results of the five samples are recorded in the LIS. If any of the three remaining test result is POS, then POS result of the donor will be added to the LIS. Results of the five samples are recorded in the LIS.
3) If any result of the five samples is missing, then the HIV conclusion of the donor cannot be added to the

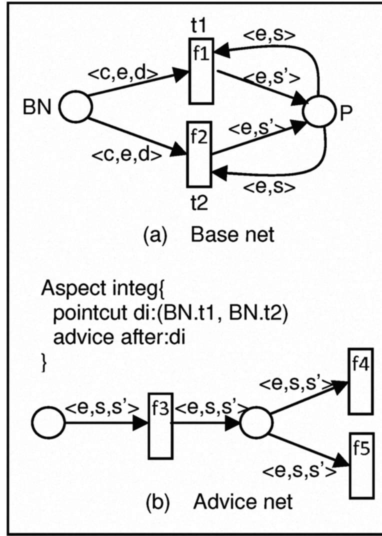

Fig. 3. An AOPrT model (a) adding or modifying data (base net). (b) checking data integrity (aspect).

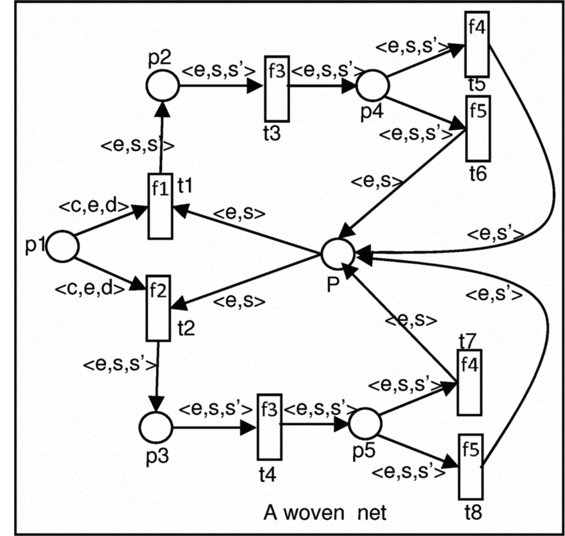

Fig. 4. A woven net for the case

symbols	Description
$< c, e, d >$	c represents add or $modify$; e is a donor id; d is data.
$< e, s/s' >$	e is a donor id; s/s' is records of e in P.
f_1	$(c == "add") \;\&\; (s' = s + d)$.
f_2	$(c == "modify") \;\&\; (s' = s + d)$.
P	records at LIS.
f_3	checking data integrity.
f_4	$(s' \neq \phi)$ (means the change is valid).
f_5	$(s' == \phi)$ (means the change is invalid); $s' = s$.

TABLE I
DESCRIPTION OF FIGURE 3

LIS.

4) Each HIV record for a donor has a status value. Before a new HIV record can be added to the LIS, the system checks whether the donor has previous HIV records in the LIS. If the donor has not a previous HIV record, the status of the record is flagged as status 1, which means the blood cannot be used for transfusion to other patients. If the donor has a POS result, then the record is flagged as status -1, which means the donor will not be selected in the future. If the donor has a NEG record, the status of the current record is flagged as 0, which means the blood can be used for transfusion.
5) Deleting one HIV record of a donor in the LIS may affect the status of other records of the same donor, such as there are only two records for one donor in the LIS, if one of them is deleted, then status of the left record cannot be 0 because the historical information of this record is missing (If it is 0, then it has to be updated to 1).
6) Above data integrity checking is also applied for modifying results.

A. Modeling Aspects using AOPrT Nets

We model the data integrity checking as an aspect using AOPrT nets. We model a scenario that a technician is adding or modifying HIV results in the LIS, which has to satisfy the data integrity constraints. Figure 3 (a) is the base net for a technician adding or modifying data in LIS. Figure 3 (b) is the aspect model for checking data integrity. Figure 4 is a woven net composing the base net with advice nets defined at Figure 3. We renamed names for some places and transitions and some labels on arcs at Figure 4 after the weaving to unify the names in the woven net.

Table I briefly describes the AOPrT models at Figure 3.

B. Model Checking Conflicts using PROD

The key idea for analyzing semantic conflicts at AOPrT models is to compose the model as a PrT net using the method discussed at previous sections and then define properties to be checked as temporal logic formulae.

1) Model Checking Procedures: A PROD program consists of a PrT net description and properties to be verified. The net description language is the C preprocessor language extended with net description directives [17]. Properties to be checked are defined using the $\#verify$ statement. PROD includes a module for generating executable reachability graph, a module for verification or on-the-fly verification of a property, and some utility modules for debugging or querying information. PROD performs on-the-fly verification of linear time temporal properties [2] with the aid of the stubborn set [16] method for reducing the size of state space during verification. Although PROD also provides the capacity for verification of branching time temporal properties [2], these properties have to be verified after state space generation and the verification does not support the stubborn set method. In this paper, we chose on-the-fly verification and reachability analysis method for analyzing AOPrT models.

We use numbers instead of real commands or data in the

PROD program to reduce the complexity of the program but it still well illustrates the analysis approach.

2) Model Checking the Properties: We performed reachability analysis and on-the-fly verification of the model at Figure 4, which is composed with the data integrity aspect and the base net used for adding or modifying donor information. In the reachability analysis, we checked the deadlock and livelock properties using the *tester* approach in PROD. We checked composition or system properties using the on-the-fly verification approach.

1. The model is deadlock-free after aspects were woven with the join points. The following codes in the PROD net description is used to check the deadlock-free property.

```
#place tester lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester tester deadlock(<.0.>)
```

First, we generate the executable reachability graph, and then run the executable graph to check the deadlock-free property. The model is deadlock-free. We also can check the reachability graph information using the *Probe* utility.

2. The model is livelock-free after aspects are woven with the join points. The following codes in the PROD net description is used to check the livelock-free property.

```
#place tester lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester tester livelock(<.1.>)
```

The model is livelock-free.

3. Adding donor data will be completed with success if the data integrity is not violated, or failure if the data integrity is violated. We perform the on-the-fly verification on the woven net through defining the property as a linear temporal logic formula at *verify* section in the PROD net description:

```
#verify (p1==<.0,0,1.> and p==<.0,0.>)
implies (eventually (p4 ==<.0,1,1.> or
                p4 == <.0,0,1.>));
```

The property is verified as *true*.

4. Modifying donor data will be completed with success if the data integrity is not violated, or failure if the data integrity is violated. We perform the on-the-fly verification on the woven net through defining the property as a linear temporal logic formula at *verify* section in the PROD net description:

```
#verify (p1==<.1,1,1.> and p==<.1,1.>)
implies (eventually (p3 ==<.1,1,1.> or
                p4 == <.1,0,1.>));
```

The property is verified as *true*.

Through analyzing the reachability graph, we fixed some errors in the original PROD net description program for the net at Figure 4. Through changing the statement at `tester` or `verify` in the net description, we can verify other properties as well. Based on above analysis, we conclude model checking is an effective and easy way for analyzing semantic conflicts, execution orders and dependency among aspects.

C. Discussion

Although modeling checking has been widely used for analyzing systems with finite states [2], researches on modeling checking aspect-oriented systems are still rare due to complexity caused by crosscutting nature of this type of systems [14] [10] [18]. There are two key issues for model checking an AOSD design: (1). How to specify an AOSD design. Because modeling checking is a formal analysis method, the gap between model checking and modeling is minimal if we model an AOSD design using a formal language. However, formal specification of an AOSD system is a fairly challenge task to many software developers. Therefore, many researchers prefer modeling systems using a non-formal language such as UML, and then transforming the model into a model that is acceptable by a model checker. During the transformation, it is difficult to guarantee the semantic consistency between the two models. In this paper, an AOSD system is directly modeled using the formal language AOPrT nets, and model checking is directly applied to AOPrT models. (2). How to check an AOSD model. One way is to check an AOSD model using an existing model checker, and then the AOSD model has to be pre-processed before it can be checked. The pre-processing includes tasks such as weaving aspects with join points, resolving aspect execution orders at SJP. Weaving aspects, resolving conflicts or aspect dependency is complex, therefore, approaches for automatically pre-processing AOSD model is critical important for model checking an AOSD system. In this paper, we proposed ways on how to pre-process an AOPrT model for model checking or analysis. The second way is to model check an AOSD model directly, then a new model checker may have to be built. We chose PROD for model checking AOPrT models because PROD can check PrT nets directly.

V. Related Work

Many AOSD modeling approaches are based on UML such as work at [15] and [6], which extended UML notations with AOSD mechanisms especially the linguistic constructs from AspectJ such as aspects and pointcuts. Although UML-based AOSD modeling provides a nice solution for general developers to modeling crosscutting concerns, formal analysis of UML-based models is challenge due to the informal nature of UML. In [19], PrT nets were extended with AOP facilities for modeling security concerns. However, concern conflicts at SJP were not considered in the work. Our work refers to the work at [19], but our work simplified pointcuts in the modeling language (one type of pointcuts instead of three types of pointcuts) so that the composing aspects with base nets are more feasible. In our work, the weaving algorithm has general meaning so that composing complex advice nets with base nets is possible thanks to the introduction of four different advice types. In addition, concern conflicts were resolved in our work through modeling the interference among aspects. Our work provides an automated solution for

analysis of AOSD design specifications, but the solution was not at [19].

Formal proof, testing, simulation and model checking have all been tried for checking AOSD designs or programs. But most analysis approaches were applied to aspect-oriented programs. Krishnamurthi and Fisler developed some theoretical work at [10] on incrementally model checking aspect-oriented programs, and the work can be extended for model checking AOSD design specifications. In [18], Xu developed an approach for model checking state-based specification of AOSD design. In order to model check a design specification, an AOSD state model has to be converted into the input model of a model checker. Sihman [14] discussed an approach for model checking an aspect-oriented program, where the model checking input was automatically generated. Model checker SPIN was used for model-checking a concurrency control aspect at [4], where the AOSD models had to be manually converted into the input programs of SPIN. Model checking at above work was applied to programs directly or model checking was applied to a design model indirectly via transforming the design model into a model that can be accepted by a model checker. However, model checking programs may easily suffer the state space explosion issue. In addition, indirectly checking AOSD design specifications also has some problems due to the transformation from design specifications to input models of model checking. Not only converting an AOSD design model into an input model of a model checker is challenge, but also the conformance between the two models is an issue. In our work, the formal modeling language with graphic notations is easy to use for modeling software design, and the model checking is directly applied to design specifications so that converting models is not necessary.

VI. Summary and Future Work

AOSD aims at improving the identification, separation and modularity of concerns especially crosscutting concerns in software. Formal specification and analysis of crosscutting concerns is highly effective for assuring the quality of software design. In this paper, we developed a formal approach for modeling and analyzing AOSD design specifications. Crosscutting concerns are separated and modularized as aspects, and software is modeled using AOPrT nets, which are an extension of PrT nets with aspect concepts. System properties and interference among aspects in AOPrT models are formally analyzed using the model checking and reachability analysis tool PROD. To experiment our approach, we modeled a clinical diagnostic algorithm using AOPrT nets, and successfully analyzed the AOPrT models. In conclusion, our approach is a powerful and practical solution for modeling and analyzing crosscutting concerns in software. In the future, we are going to extend our approach for modeling and analyzing software requirements.

VII. Acknowledgments

This work is supported by Title III grant under awards P031B085057-08. The authors gratefully acknowledge the contribution of National Research Organization and reviewers' comments.

References

[1] G. Booch, R. A. Maksimchuk, M. W. Engel, and B. J. Young. *Object-Oriented Analysis and Design with Applications (3rd Ed.).* Addison-Wesley, 2007.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, MA, 2000.

[3] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools.* Addison Wesley, 2004.

[4] G. Denaro and M. Monga. An experience on verification of aspect properties. In *IWPSE '01: Proc. of the 4th Int. Workshop on Principles of Software Evolution*, pages 186–189, 2001.

[5] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of 3rd Int. Conf. on Aspect-Oriented Software Development*, pages 141 – 150, Lancaster, UK, March 2004.

[6] L. Fuentes and P. Sánchez. Towards executable aspect-oriented uml models. In *AOM '07: Proc. of the 10th int. workshop on Aspect-oriented modeling*, pages 28–34, 2007.

[7] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph based approach to modeling and detecting composition conflicts related to introductions. In *Proc. of 6th Int. Conf. on Aspect-Oriented Software Development*, pages 85 – 95, Vancouver, Canada, 2007.

[8] X. He and T. Murata. *High-Level Petri Nets - Extensions, Analysis, and Applications.* Electrical Engineering Handbook (ed. Wai-Kai Chen). Elsevier Academic Press, 2005.

[9] ir. P.E.A. Durr, ir. T. Staijen, D. L. Bergmans, and P. M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[10] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.

[11] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, Apr. 1989.

[12] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proc. of Int. Conf. Net.ObjectDays (NODe)*, pages 19 – 38, Erfurt, Germany, 2005.

[13] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[14] M. Sihman and S. Katz. Model checking applications of aspects and superimpositions. In *Foundations of Aspect-Oriented Lang.*, pages 51 – 60, 2003.

[15] D. Stein, S. Hanenberg, and R. Unland. A uml-based aspect-oriented design notation for aspectj. In *AOSD '02: Proc. of the 1st int. conf. on Aspect-oriented software development*, pages 106–112, 2002.

[16] A. Valmari. On-the-fly verification with stubborn sets. In *CAV '93: Proc. of the 5th Int. Conf. on Computer Aided Verification*, pages 397–408, London, UK, 1993. Springer-Verlag.

[17] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. Prod reference manual. Technical report, Helsinki University of Technology, Dept. of Computer Science and Eng., Digital Systems Lab., 1995.

[18] D. Xu, I. Alsmadi, and W. Xu. Model checking aspect-oriented design specification. *Computer Software and Applications Conference, Annual International*, 1:491–500, 2007.

[19] D. Xu and K. E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Trans. on Software Eng.*, 32(4):265 –278, 2006.