

Designing Loop Condition Constraint Model for Join Point Designation Diagrams (JPDDs)

Bahram Zarrin
Master Student

Faculty of Computer Science & IT,
Universiti Putra Malaysia, Malaysia
Bahram.zarrin@gmail.com

Rodziah Atan
Doctor

Faculty of Computer Science & IT,
Universiti Putra Malaysia, Malaysia
rodziah@fsktm.upm.edu.my

Muhammad Taufik Abdullah
Doctor

Faculty of Computer Science & IT,
Universiti Putra Malaysia, Malaysia
taufik@fsktm.upm.edu.my

ABSTRACT

The specification of selection criteria (Join Point Selection) is a very important issue in the Aspect Oriented Software Development. Several models have been presented to visualize the selection criteria in the Aspect Oriented Modeling. JPDDs provides a visual means to constraint the selection of join points based on static and dynamic, structural and behavioral context. For some applications in order to call an advice, it is necessary to select the join points in programs, when specified message flow (or a block of them) should be repeated either for specified times or meeting a particular condition. this constraint cannot be visualized by JPDDs. No notation is defined to address loop structures constraint in these diagrams. This paper introduces a design model extension called Loop Condition Constraint Model (LCCM) to support these constraints in JPDDs. A scenario is used to explain this extension.

Categories and Subject Descriptors

D.3.2 [Language Classification]: Design Languages -UML

General Terms

Design, Languages

Keywords

Joint Point Selection, JPPDs, Combined Fragment Diagrams, Aspect Oriented Model

1. INTRODUCTION

The Aspect Oriented Software Design (AOSD) aims to solve crosscutting concerns in object oriented software development. There are some concerns which cannot be implemented in separate modules (like security, logging, etc) and their implementation crosscuts the other modules which led to occur code tangling and code scattering in the implementation. AOSD introduces some new concepts and artifacts to solve the problem. Concerns which have crosscutting (like logging and security) with the other concerns and cannot be implemented in separate module are called an Aspect. A part of the code in an aspect which needs to be injected in none crosscutting modules is called an Advice. The places of the program which an advice needs to be injected in none crosscutting concerns are called a Join Point. A set of join points is known a Pointcut. In recent years there are a lot of research interests to separate crosscutting in the design level and a lot of works done to design model based on UML to support crosscutting separation in this stage.

Aspect Oriented Modeling (AOM) attempts to separate crosscutting concerns in the earliest step of software development. A lot of models have been presented based on UML diagrams which most of them are the same in presenting aspect model and base model. The challenging area is to specify join point selection and design model for pointcuts. Queries on join points are an essential part of AOSD. Join point queries are necessary to identify all relevant points in a program (i.e. in its code, or during its execution) at which aspectual adaptations need to take place [2]. Finding appropriate means to designate such sets of relevant join points is a highly active field of research in AOSD [3, 4, 5, and 6]. JPDDs are means that visualize join point queries graphically and separately from the adaptation specification. They provide a visual means to constrain the selection of join points based on static and dynamic, structural and behavioral context [1, 2].

JPDDs do not provide any mechanism to visualize loop structure in the interaction between objects. When a message flow (or block of them) needs to be called for specified times in order to call an advice, the message flow/s should be drawn in the behavior section of a JPDD repeatedly as much as number of iteration needed. This task is time consuming and cumbersome especially when the repetition number or the number of message flows is increased.

On the other hand, when the number of repetitions is not constant and either depends on runtime values or a specified condition, it is impossible to visualize the constraint by a JPDD, because the number of iteration is not available at design time.

In this paper, a design model is introduced which is called Loop Condition Constraint Model (LCCM) to support specification of this constraint in JPDDs. This paper is organized as following. In section 2, JPDDs diagrams are shortly explained by an example (interested readers can study [7, 8, and 9] for more information). In section 3, a scenario is introduced for loop constraint, and then notation to visualize this constraint is introduced. In section 4, advantages of using this model in JPDDs are summarized. Finally in section 5, notation is compared with JPDDs equivalent diagram for the scenario.

2. JOIN POINT DESIGNATION DIAGRAMS (JPDDS)

JPDDS aim to specify the selection of join points in AOSD in a graphical notation. They are a mechanism for expressing sophisticated join point selections in a platform-independent way. They can provide a general notation to specify crosscutting concerns independently of any languages. They can be used by the developers who use different aspect oriented languages to communicate their join point selections [7].

A JPDD contains a description of structural and behavioral constraints. The structural part is described by mixing object and class diagram which the behavioral part is described by sequence diagrams. Collaboration diagrams are used to compose both parts.

General format of JPDDs is presented (see Figure 1). Here, the JPDDs notations are described by using the following example.

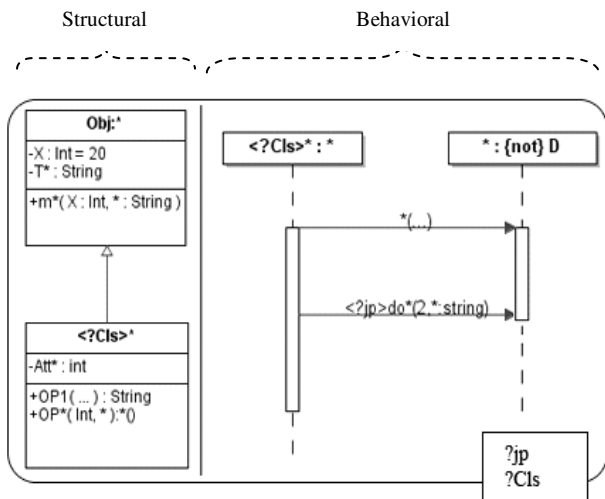


Figure 1. General Format of JPDD.

As shown in Figure 1 there are two diagrams surrounded by a rounded rectangle and separated by a line. The right diagram is a sequence diagram which specifies the behavioral constraints. The left diagram is an object diagram which defines structural constraints and object state constraints. Both structural and behavioral constraints should be matched to select a join point.

Two objects are presented in the object diagram in Figure 1. The name of the first object (the object in the left top of the diagram) should be matched with “Obj”. Type of the object is defined as wildcard “*”. It means that there is no constraint defined on the object type and any types can be used. The matched objects should have at least two attributes, the name of the first one should be exactly as “X” and its type should have been defined as “int” and the attribute value should be 20 at this time. The other attribute’s name is defined as “T*”. It means that attribute’s name should be started with “T” and its type should be defined as “String” and there is no constraint defined for its value. The object should also have a method which its name starts with “m” and should have two parameters. The first parameter’s name should be matched with “X” and it should be defined as “Int”. The “*” wildcard is used for the name of the other parameter, it

means there is no constraint on the name of the second parameter and the type of the second parameter must be defined as “String”.

The wildcard “*” is used as the name of the second object, it means any objects with any names can be selected but the object should have at least one attributes of type “int” which its name is started with “Att”. The object should have at least two operations. The first operation’s name must be defined exactly as “OP1” and the second operation’s name must be started with “OP”. Wildcard “...” is used as a parameter for the first operation it means there is no constraint on the number of parameters and the operation can have any numbers of parameters with any names and any types. The second operation should have two parameters. The type of the first parameter should be defined as “int” and no constraint is defined on the second parameter, any parameters with any types and any names can be used.

The other structural constraint defined in this example is that the second object (the object in the left bottom of the diagram) should be inherited from the first object because of the generalized relationship between these objects. In this example there is a direct generalized relationship. The other kind of relationships can be used in the diagrams (see Figure 2). For more information refer to [7, 8, and 9].

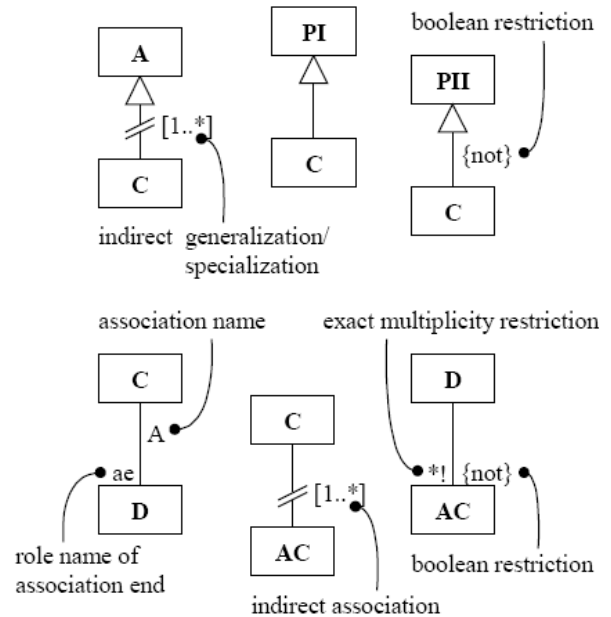


Figure 2. Different relations of constraints which defined in JPDD

On the right, a sequence diagram with two object lifelines is presented. There are two message calls from the first object with arbitrary name and arbitrary type of name to the other object with arbitrary name and any types except type of “D”. The first call designator selects all message calls with arbitrary name and arbitrary number of parameters from the object 1 to the object 2. It means that any message calls can occur. The second call designator selects message calls which their names are started

with “do” and their first parameter value is “2” and the second parameter type is “String”. Here, the second message call should be called immediately after the first one. If this notation ($\#$) is used in the object life line it means that between the two message calls one or more message calls can occur.

Each element specified within a JPDD, such as classes, objects, attributes, operations, and parameters may be given an identifier. Identifiers are denoted by a preopening question mark (?) and are enclosed in angle brackets (“< >”) [7, 8]. In Figure 1 <?cls> and <?jp> are Identifiers and they should be listed in the small rectangle on the right corner.

Identifiers are used to connect the behavioral and structural part of JPDDs. This is required if an object participating in an interaction on the right side has to be constrained by a structural definition on the left, as well. To connect both definitions, the same identifier is used on both sides [7, 8]. For example in Figure 1 <?cls> is used to connect both sides. In the left diagram it is assigned to the bottom object in the diagram and in the right diagram it is designated to the first object life line, as well.

Identifiers may be used to designate those elements of a crosscut specification that are exposed for further processing [7, 8]. In Figure 1, identifier <?jp> is used to expose the desired join point in the JPDD.

In this section, JPDDs are shortly explained because of space limitation of the paper. Interested users can refer [7, 8] for more information about these diagrams. In the next section a scenario is presented for Loop Condition Constraint and then a notation is introduced to visualize this constraint.

3. LOOP CONDITION CONSTRAINT MODEL (LCCM)

For some applications it is necessary to call an advice in a point of a program when specified message flow (or a block of them) is repeated either for specified times or until a specific condition is true. For example, in a security system when a user enters wrong password three times, an advice like block user or lock system should be called. To support specification of these constraints, a Loop Condition Constraint Model (LCCM) notation is introduced in this paper which can be used in JPDDs. This notation is derived from Combined Fragments Diagrams with a Loop operator.

The security example is used as a scenario to explain the design notation for loop condition constraint. This scenario is visualized by the design notation in Figure 3. In this figure, the notation is introduced to support loop condition constraint. A Combined Fragment is used with loop operator in the JPDD to define a loop condition constraint. The loop box is divided into two sections. The top section is used to define the loop body. The sequence of the message calls which is in the loop body should be repeated as much as the value in the bracket (in this example, the number is defined as three times). Variables and constants can be used to

specify the iteration number for the loop. In the bottom section, the list of JPDD identifiers which must be exposed immediately after the last message call in the last iteration in the loop body, should be listed and presented. (See Figure 4 as a JPDD equivalent of the Figure 3, it notes that <?jp> is applied on the last “False” message in the message calls).

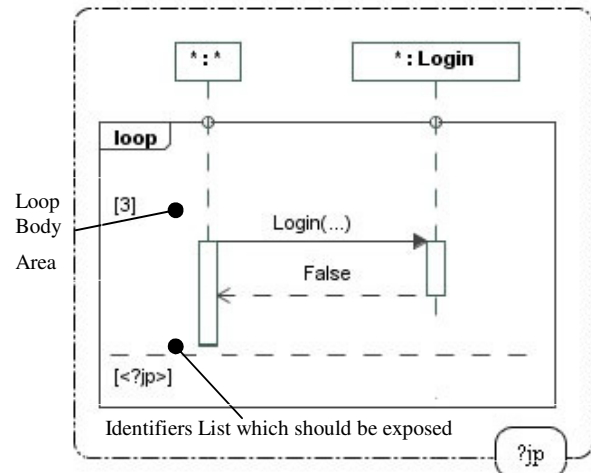


Figure 3. The Security Scenario Implemented by Using Loop Condition Constraint design model in JPDD.

In this scenario, if the “login” method of an object from type of “Login” is called with a “false” replay message for three times, <?jp> identifier will be exposed after the last “false” message and desired advice assigned to this identifier will be called. The scenario means that when a wrong password is entered three times for login method of a login object, the user’s access to the system should be blocked or system should be locked if the user is not existed. A JPDD equivalent of Figure 3 is presented in Figure 4.

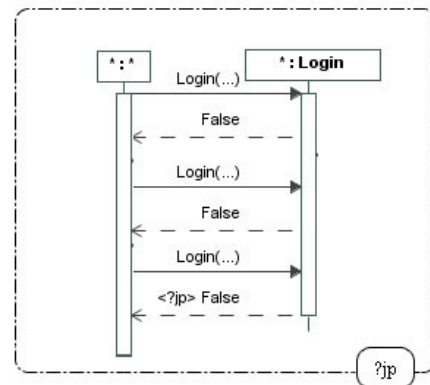


Figure 4. JPDD equivalent of Figure 3

The list of the identifier in the bottom of the loop box can be omitted, and the Loop Constraint can be used just as a precondition for the other Message flow. Following example shows a loop constraint without presenting the list of exposed elements in the loop box.

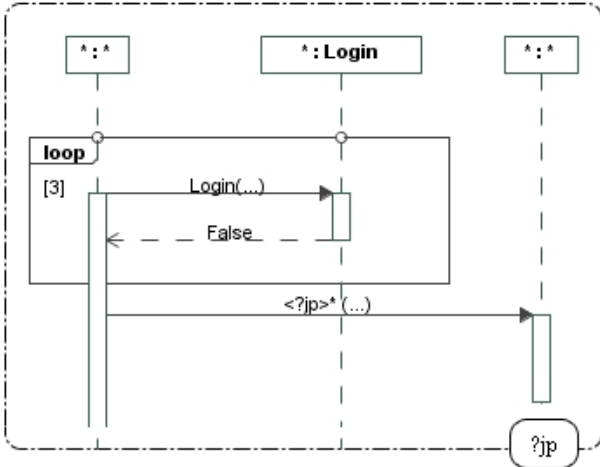


Figure 5. The Security Scenario Implemented by Loop Condition Constraint design model in JPDD without Identifiers list.

This diagram exposes the `<?jp>` identifier when a message with arbitrary name and arbitrary parameters from an arbitrary object to another arbitrary object, calls after three times of failed login. Note that here the Loop Condition Constraint is used as a precondition for the followed message flow constraint which is specified by the `<?jp>` identifier. This identifier will not be exposed immediately after the last message call in the last iteration of the loop Condition constraint. (A JPDD equivalent of Figure 5 is presented in Figure 6).

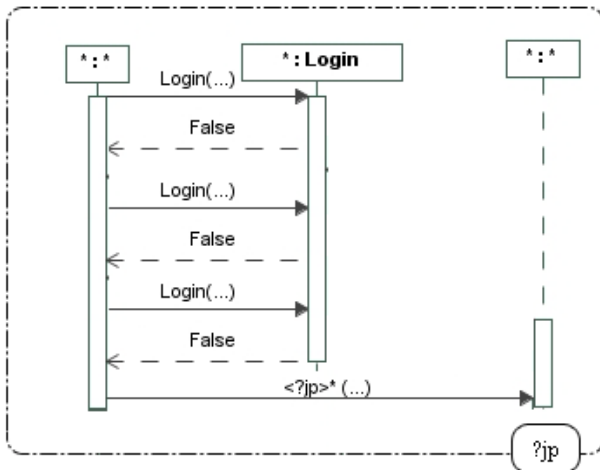


Figure 6. JPDD equivalent of Figure 5

Conditional loops can be also presented with these notations. For the conditional loops a Boolean expression is defined as the loop condition. This expression can include:

- Constant values (such as numbers, strings)
- The JPDD's identifiers such as Object Identifiers, Object Attributes' Identifiers, and Message call Identifiers.
- Comparative operators such as =, >, <, !=, etc.
- Logic operators such as AND, OR, XOR and NOT.
- Mathematic operators such as +, -, /, %, (,), etc.

The Loop Condition Constraint design model notation is the same for conditional and incremental loops. For conditional loops an expression will be used for the loop condition instead of a fixed number, and UML Stereotype can be used to specify the loop types.

Following example is related to the daily withdrawal limitation mechanism in banking system. Control Limit service is a part of the security module in most of the banking systems. This service makes a restriction on the total withdrawal amount per day that means the user can not withdraw or transfer money more than particular amount in a day. ControlLimit module can be identified as an aspect with two advices called UpdateWithdrawanAmount and MaximumWithdrawalExceeded. Each time user transfers or withdraws money from the account UpdateWithdrawanAmount advice should be called and when the total money withdrawn by the user in a day exceeded from the maximum daily withdrawal amount, the MaximumWithdrawalExceeded advice should be called to lock the account.

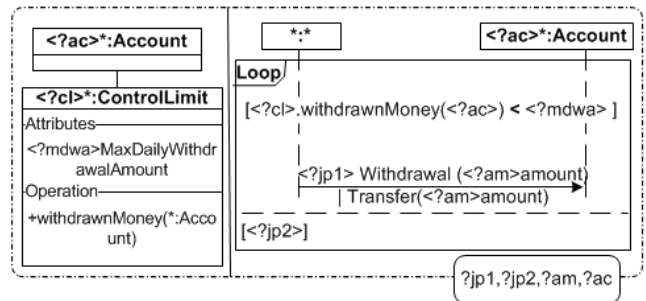


Figure 7. Daily Withdrawal Limit Example

In Figure 7, a loop condition constraint model has been used with a Boolean expression as the loop condition. In this diagram each time a message invocation (`<?jp1>`) with the name of "Withdrawal" or "Transfer" with exactly one argument which its name is "amount" (identifier `<?am>` is assigned to this argument) received by an object of class "Account" the identifiers `<?jp1>` and `<?am>` will be exposed (UpdateWithdrawanAmount advice will be called) until the loop condition is true.

Once the loop condition evaluated as false identifier `<?jp2>` would be exposed. (MaximumWithdrawalExceeded advice will be called). The loop condition will be calculated by this expression:

$$\langle ?cl \rangle . withdrawnMoney(\langle ?ac \rangle) < \langle ?mdwa \rangle$$

In this expression `<?cl>` is an identifier which refers to a "ControlLimit" object. Identifier `<?ac>` refers to the account object which is receiver of the withdrawal message and `<?mdwa>` refers to the MaxDailyWithdrawalAmount attribute of the ControlLimit object which is the maximum withdrawal limitation per day.

The identifier `<?jp1>` is assigned to the message flow in the loop body. In this case while the condition of the loop is true this identifier (`?jp1`) will be exposed.

4. SUMMARIZED ADVANTAGES

The Introduction of Loop Condition Constraint Model (LCCM) will affect some factors of JPDDs which are described and discussed in the following table (Table1):

Table1. Summarizing the impact of using LCCM in JPDDs

Factor	Description
Flexibility	Using LCCM in JPDDs can increase the flexibility of join point selection in JPDDs. All the loop condition constraint can be visualized by LCCM. With LCCM when the repetition times increased, it just needs to adjust the iteration number in the loop body and there is no need to draw more message flows in the diagram while without them the sequence message flows have to be redrawn to fill the required repetition times. In cases when the sequence is large or the repetition time is more, drawing the message flows in the diagrams is time consuming and cumbersome. When the number of iterations depends on the runtimes values or variables, it is impossible to specify the constraint in JPDDs without using LCCM. The Conditional Loop Constraint can be also visualized with LCCM. Without LCCM, specification of conditional loop constraint is impossible.
Readability	The developers and Designers are familiar with Loop Structure. Using explicit Loop condition can increase readability of the diagram for them. They can easily find out the behind concept of the diagram and infer a meaningful constraint from the diagram with LCCM. Without LCCM, in some cases when it is needed to repeat a large sequence of message calls, it is hard for them to find out the repetition embedded in the sequences. LCCM makes it clearer by applying explicit Loop.
Diagram Size	Using LCCM will reduce the size of JPDDs Diagrams because there is no need to repeat the message flow sequences more in the Diagram.

Regarding the Table 1, it can be observed that LCCM can improve readability and flexibility of the JPDDs in addition of reducing the size of JPDDs.

5. CONCLUSION AND FUTURE WORKS

By comparing Figures 3 and 5 (Diagrams which use the loop Condition Constraint Model) with Figures 4 and 6 (the JPDDs equivalent diagrams), it is observed that the first diagrams are simpler and more understandable for designers and developers because they are familiar with loop structures and this model is more meaningful for them and they can easily find out the concept behind of the constraint. On the other hand, if the loop iteration

number is increased, in the second diagrams more message calls are needed to be drawn to meet the iteration (in the incremental loop constraint) which leads the diagram to be more complicated and in some cases when the iteration number is more or the number of message calls in loop body is increased, it makes it time consuming and cumbersome to draw them without LCCM Diagram.

The conditional Loop constraint cannot be implemented with JPDDs because they need to meet the loop condition. Because the iteration number is not defined they cannot be visualized without LCCM. In this paper, the loop constraint model is introduced in the design level, and the notation presented to be supported in JPDDs. As a future work, the loop constraint model will be designed at the implementation level for AspectJ.

6. ACKNOWLEDGMENT

All who guided the authors are appreciated; also the anonymous reviewers are thanked for their insightful comments.

7. REFERENCES

- [1] Stein D., Hanenberg S., Unland R., Query Models, Workshop: The Unified Modelling Language: Modelling Languages and Applications, Lisbon, Portugal, October 11-15, 2004.
- [2] Stein D., Hanenberg S., Unland R., Visualizing join point selection for stateful aspects, European Interactive Workshop on Aspects in Software, September Vrije Universiteit Brussel, Belgium, 2005.
- [3] Gybels, K., Bricchau, J., Arranging language features for more robust pattern-based crosscuts, in: Proc. of AOSD'03, March 17-21, 2003, Boston, MA, ACM, pp. 60-69
- [4] Hanenberg, S., Hirschfeld, R., Unland, R., Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving, in: Proc. of AOSD '04 (Lancaster, UK, March 2004), ACM, pp. 46-55
- [5] De Fraigne, B., Vanderperren, W., Suvée, D., Bricchau, J., Jumping Aspects Revisited, DAW Workshop, at: AOSD 2005, March 2005, Chicago, IL
- [6] Ostermann, K., Mezini, M., Bockisch, Chr., Expressive Pointcuts for Increased Modularity, in: Proc. of ECOOP'05, Glasgow, UK, July 2005, ACM
- [7] Stein, D.; Hanenberg, S.; Unland, R.: Join Point Designation Diagrams: A Graphical Representation of Join Point Selections. In: Journal of Object Technology(guest editors); International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 16 (2006) 3 , S. 317-346.
- [8] Hanenberg, S.; Stein, D.; Unland, R.: From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In: de Moor, O. (Hrsg.): Proc. of 6th International Conference on Aspect-Oriented Software Development (AOSD 2007), Vancouver, BC, Canada, March 12-16, 2007. ACM, Vancouver, BC, Canada 2007.
- [9] Stein, D.; Hanenberg, S.; Unland, R.: Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In: Masuhara, H., Rashid, A. (Hrsg.): Proc. of 5th International Conference on Aspect-Oriented Software Development (AOSD 2006). ACM, Bonn, Germany, March 20-24 2006, S. 15-26.