A Formal Proof of Correctness of Construct Association from PROMELA to *Java*

Suprapto, *Member, IAENG,* Retantyo Wardoyo, Belawati H. Widjaja, and Reza Pulungan

***Abstract*—The association between the subset of PROMELA's constructs (or statements) and the subset of *Java*'s constructs is intended to provide a collection of rules that can be used as a reference in developing a model of code translator from a PROMELA model to a *Java* program. The idea arises from the fact that, both PROMELA model and *Java* program are built (or composed) by various elementary elements called constructs. Although this kind of association has already been introduced in some previous researches, they provided no proofs about its correctness.**

In this paper we propose a formal proof of association's correctness by showing the equivalence (or similarity) of the program graphs for every two associated constructs in the association. The correctness of association means that every two associated constructs in association have equivalent semantics. In addition, at the end of this paper we also introduce a translator tool we have developed based on this association's definition to translate PROMELA model to *Java* program.

***Index Terms*—Constructs association, PROMELA, *Java*, correctness, equivalence, preserving, program graph, similarity, semantics.**

I. Introduction

The widespread use of source-to-source translation for imperative languages would help software engineering if such translator could be written and if it were easier to translate an existing program into another language than building the program from scratch [5]. Source-to-source translation has been studied by various researchers, and is often used together with program optimization. It is sometime, however, used without program optimization. Translators that do not optimize programs but only preserve the same structure from one language to another have significant potential for software engineering.

A translator was also developed to detect deadlock existence on *Java* programs based on PROMELA and SPIN [2]. An abstract formal model expressed in PROMELA is generated from *Java* source using the Java2Spin translator. Then the model is analyzed by SPIN [7], [8], and possible error traces are converted back to traces of *Java* statements and reported to the user. An indirect way of model checking C programs was proposed in [11] by first translating the C code to PROMELA. The translator was developed by using syntax-directed translation techniques to perform the translations, and several tools and languages are involved.

PROMELA is one of the most widely used modeling language to model systems, especially distributed, reactive and concurrent ones [10]. On the other hand, some parts of *Java* language can be used in the translation of certain PROMELA properties. Therefore, *Java* is one of a few reasonable candidates that can be used as a target language for PROMELA translation [6].

Manuscript received June 09, 2015; revised July 15, 2015.

Suprapto, R. Wardoyo, and R. Pulungan are with the Department of Computer Science and Electronics, Faculty of Mathematics and Natural Sciences, Universitas Gadjah Mada, Yogyakarta, Indonesia e-mail: sprapto@ugm.ac.id, rw@ugm.ac.id, and pulungan@ugm.ac.id.

B.H. Widjaja is with the Faculty of Computer Science, Universitas Indonesia, Jakarta, Indonesia e-mail: bela@cs.ui.ac.id.

TABLE I: Construct association between a subset of PROMELA and *Java* constructs

PROMELA	*Java*
Expression	Expression
Assignment	Assignment
Send and receive	Two defined methods in separate classes together with the required channel (buffer) that can be invoked either by regular invocation or by socket programming.
$atomic$	$while \cdots switch$, namely a switch in a while loop.
d_step	$synchronize$
$if \cdots fi$	Generally, a program is specifically built to have a priority in selecting certain condition; in which case nondeterminisms can be removed. Otherwise, if nondeterminism is preserved (or at least imitated), $randomize$ is implemented.
Deterministic for	Fixed repetition for loop.
$do \cdots od$	Repetition of selection together with nondeterminism resolution as described for $if \cdots fi$.
$unless$	Exception handling $try..catch$.

A construct association from PROMELA to *Java* is an association from a subset of PROMELA's statements (or constructs) to a subset of *Java*'s statements associating every statement in the first subset to probably more than one statement in the second subset. According to their functionality, statements in PROMELA can be classified into five groups, *i.e.*, meta terms (9 statements), declarators (21 statements), control flows (8 statements), basic statements (6 statements), and predefined (21 statements) [17]. Of them, however, there are only ten constructs in PROMELA to be selected in the association. The selection was made by considering that some constructs in a PROMELA model are only used for verification purpose, hence, they are not required in the system (or *Java* program) development. In addition, some bigger constructs can be composed by several elementary ones. An informal definition of construct association is given in Table I.

Even though this kind of associations has already been introduced in previous researches [3], [6], [16], [17], however, the proof that can guarantee the association's correctness has never been provided. This research proposes a formal proof of association's correctness by showing the equivalence of semantics for pair of associated constructs, and this equivalence in turn is proven by their program graphs similarity. The proof is performed by executing the following steps:

1. For each two associated constructs in the association:
   1.1 Derive the program graph for PROMELA construct, and its associate *Java* construct.

1.2 Compare the two program graphs for their similarity.
1.3 If they are not similar, then modify its associate in *Java*, and go to 1.1.

2. If there are still pairs of associated constructs in the association, then go to 1 for next pair of association. Otherwise, go to 3.
3. Stop, and the association's correctness is proved.

II. Preliminaries

A. Program Graph

A program graph (PG) over a set of typed variables is a digraph whose edges are labeled with conditions on these variables and actions. It is formally defined as follows [1]:

Definition 1. *A program graph PG over set Var of typed variables is a tuple $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$, with:*

- Loc is a set of locations,
- Act is a set of actions,
- Effect : $Act \times Eval(Var) \times Eval(Var)$ is an effect function,
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the conditional transition relation,
- $Loc_0 \subseteq Loc$ is a set of initial locations, and
- $g_0 \in Cond(Var)$ is the initial condition.

The program graph associated with a statement (or construct) $stmt$ formalizes the control flow when it is being executed. It means the substatements play the role of the locations, and a special location $exit$ must be provided in order to model termination. Roughly speaking, any guarded command $g \Rightarrow stmt$ corresponds to an edge with the label $g : \alpha$ where α stands for the first action of $stmt$ [1].

The notation $\ell \xhookrightarrow{g:\alpha} \ell'$ is used to concisely represent $(\ell, g, \alpha, \ell') \in \hookrightarrow$. The condition g is also called the *guard* of the conditional transition $\ell \xhookrightarrow{g:\alpha} \ell'$. If the guard is *tautology* (*e.g.*, $g = true$ or $g = ((x < 1) \vee (x \geq 1))$), then it is simply written as $\ell \xhookrightarrow{\alpha} \ell'$.

The behavior of location $\ell \in Loc$ depends on the current variable evaluation η. A nondeterministic selection is made between all transitions $\ell \xhookrightarrow{g:\alpha} \ell'$ that satisfy the condition g in evaluation η (*i.e.*, $\eta \models g$). The execution of action α changes the evaluation of variables according to $\text{Effect}(\alpha, \cdot, \cdot)$. Subsequently, the system changes into location ℓ'. If no such transition is possible, the system stops.

A location ℓ in a program graph is **ignorable** if the execution of action α does not change the evaluation of variables according to $\text{Effect}(\alpha, \cdot, \cdot)$ of the next location ℓ', provided that any variable being involved in the computation is completely new and independent. More formally, this concept might be stated in the following proposition. The ignorable locations is illustrated in Fig. 1.

Proposition 1. *Let L_A, L_B be any two locations in a program graph PG over variable Var, and $L_1, L_2, \ldots, L_m$ are locations induced by introducing any new variables $x' \notin Var$. Then the existence of $L_1, L_2, \ldots, L_m$ in between L_A and L_B is ignorable. Hence, locations $L_1, L_2, \ldots, L_m$ can be coalesced with L_1 into a new single location, and they (*i.e.*, locations $L_1, L_2, \ldots, L_m$) are called **coalesceable** locations.*

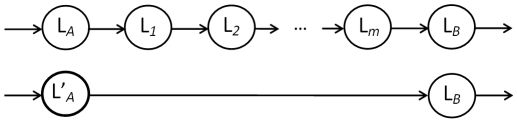

Fig. 1: An illustration of ignorable locations

B. Substatements

The set of substatements of a PROMELA statement (construct) $stmt$ is defined recursively [1]. The set of substatements for an atomic statement $stmt \in \{expr, x = expr, c?x, c!expr\}$ is $sub(stmt) = \{stmt, exit\}$, since an atomic statement only requires one-step of execution. For sequential composition $stmt_1; stmt_2$, the set of substatements is defined as:

$$sub(stmt_1; stmt_2) = \{stmt'; stmt_2 \mid stmt' \in sub(stmt_1) \setminus \{exit\}\} \cup sub(stmt_2).$$

As an illustration, consider the sequential composition in Listing 1.

Listing 1: Sequential composition

```
...
x = x + 3;
y = 2y + 2;
z = 3z + 1;
...
```

Since all constructs in the composition are assignment and in PROMELA assignments are always atomic, then $sub(x = x+3; y = 2y+2; z = 3z+1) = \{(x = x+3; y = 2y+2; z = 3z+1), (y = 2y+2; z = 3z+1), (z = 3z+1), exit\}$. These substatements determine a set of location, Loc, of program graph. In addition, assignments are always executable, so that the condition of transition is always $true$, $Act = \{x = x+3; y = 2y+2; z = 3z+1\}$, and $Loc_0 = \{(x = x+3; y = 2y+2; z = 3z+1)\}$. Then, the corresponding program graph is shown in Fig. 2.

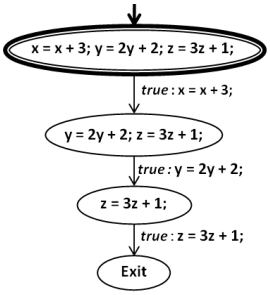

Fig. 2: Program graph for sequential composition in Listing 1

The set of substatements of $if \cdots fi$ selection statement is defined incrementally, as the set consisting of the $if \cdots fi$ statement itself plus substatements of its guarded commands. That is, let $iffi$ be $if \ :: g_1 \rightarrow stmt_1 \ldots :: g_n \rightarrow stmt_n \ fi$, then:

$$sub(iffi) = \{iffi, exit\} \cup \{ \bigcup_{1 \leq i \leq n} \{stmt' \mid stmt' \in sub(stmt_i) \setminus \{stm', exit\}\}\}$$

where stm' is the first action in $stmt_i$.

An example for a conditional statement $if \cdots fi$ is shown in Listing 2.

Listing 2: A conditional statement $if \cdots fi$

```
cond = if
       :: true -> x = x + 1; y = 2y + 1;
                  z = 3z + 1;
       :: x == 0 -> c?x; c!y;
       :: c?z -> y = 3z + 5;
       fi
```

According to the definition of substatements for conditional statements, $sub(cond) = \{cond, exit\} \cup \{y = 2y + 1;, z = 3z + 1;, z = 3z + 1\} \cup \{c!y\}$. There are three edges going out of the initial location $cond$ with label $true : x = x + 1$ to location $y = 2y + 1; z = 3z + 1$, then leaving this location with label $true : y = 2y+1$ for location $z = 3z+1$, and subsequently leaving this location with label $true : z = 3z + 1$ for location $exit$. Meanwhile, an edge from location $cond$ with label $x == 0 : c?x$ goes to location $c!y$, then leaves this location with label $\neg full(c) : c!y$ for location $exit$. In addition, out of location $cond$ there is an edge labeled $c?z : y = 3z + 5$ leaving for location $exit$. Since, there is always an executable guard (*i.e.*, $true$), then it does not block. Consequently, there is no edge returning to initial location $cond$. Graphically, the program graph for conditional construct $if \cdots fi$ is shown in Fig. 3.

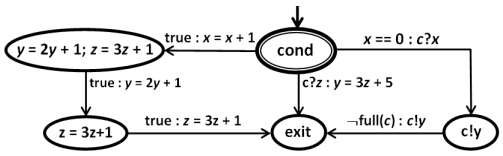

Fig. 3: Program graph for construct $if \cdots fi$ in Listing 2

Similarly, the set of substatements of $do \cdots od$ repetition statement is defined as the set containing of the $do \cdots od$ statement itself plus $exit$, and substatements of its guarded commands plus $loop$ minus $exit$.

Let $dood$ be $do \ :: g_1 \rightarrow stmt_1 \ldots :: g_n \rightarrow stmt_n \ od$, then:

$$sub(dood) = \{loop, exit\} \cup \{ \bigcup_{1 \leq i \leq n} \{stmt'; loop \mid stmt' \in sub(stmt_i) \setminus \{stm', exit\}\}$$

where stm' is the first statement in $stmt_i$.

According to the definition of the set of substatements of atomic, then $sub(atomic\{stmt_1, stmt_2, \ldots, stmt_n\}) = \{atomic\{stmt_1, stmt_2, \ldots, stmt_n\}, atomic\{stmt_{i+1}, stmt_{i+2}, \ldots, stmt_n\}, exit\}$. It strongly restates that atomic is indivisible. The same is true for d_step, hence $sub(d\_step\{stmt\}) = \{d\_step\{stmt\}, exit\}$.

For $unless$ statement, let $unless\_construct$ represent $\{stmt1_1, \ldots, stmt1_m\}$ $unless$ $\{stmt2_1, \ldots, stmt2_n\}$, then:

$$sub(unless\_construct) = sub(stmt2_1; \ldots; stmt2_n),$$

when $stmt2_1$ is executable for the first checking, then there is no statement in the main sequence executed, or:

$$sub(unless\_construct) = sub(stmt1_1; \ldots; stmt1_{i-1}) \cup sub(stmt2_1; \ldots; stmt2_n).$$

$stmt2_1$ is executable, when for some i, $2 \leq i \leq n-1$, $stmt1_i$ will be executed, or:

$$sub(unless\_construct) = sub(stmt1_1; \ldots; stmt1_m)$$

if $stmt2_1$ never executable.

Then, it can be seen that each element in the set of substatements of PROMELA as well as *Java* constructs corresponds to the locations of program graph for the corresponding construct. For example, the locations of program graph for atomic statement such as $assignment$ are $assignment$ itself and $exit$, namely $sub(assignment) = \{assignment, exit\}$. Similarly, the locations in program graph for $if \cdots fi$ selection statement is $if fi$ itself plus union of locations of each guarded command.

C. Semantics

As mentioned before, the intention of this research is to formally prove the semantics' equivalence of every two associated constructs in the association. Therefore, the following lemma helps explain the formal semantics of any PROMELA as well as *Java* statements.

Lemma 1. *The semantics for any statement of both PROMELA and* Java $stmt$ *is described by three rules that classify them into one-step statement, multi-step statement, and blocked statement. If the computation of* $stmt$ *terminates in one step by the execution of action* α*, then control of* $stmt$ *moves to exit after executing* α*:*

$$\frac{}{stmt \xrightarrow{g:\alpha} exit}.$$

If the first step of $stmt$ *leads to a location (or statement) different from exit, then the rule looks like:*

$$\frac{}{stmt \xrightarrow{g:\alpha} stmt' \neq exit}.$$

On the other hand, if the computation of $stmt$ *for some reasons cannot be performed (blocked), then control does not move. The following rule will satisfy:*

$$\frac{}{stmt \xrightarrow{g:\alpha} stmt}.$$

As a consequence, Lemma 1 leads to a more general form of sequential composition $stmt_1; stmt_2$, that is stated in the following corollary.

Corollary 1. *Sequential composition* $stmt_1; stmt_2$ *is defined by two rules that distinguish whether or not* $stmt_1$ *terminates in one step. If the computation of* $stmt_1$ *terminates in one step by executing action* α*, then control of* $stmt_1; stmt_2$ *moves to* $stmt_2$ *after executing* α*:*

$$\frac{stmt_1 \xrightarrow{g:\alpha} exit}{stmt_1; stmt_2 \xrightarrow{true:\alpha} stmt_2}.$$

If the first step of $stmt_1$ *leads to a location (or statement) different from exit, then the following rule applies* [1]:

$$\frac{stmt_1 \xrightarrow{g:\alpha} stmt_1' \neq exit}{stmt_1; stmt_2 \xrightarrow{g:\alpha} stmt_1'; stmt_2}.$$

If the computation of $stmt_1$ *for some reasons cannot be performed (blocked), then control does not move, and the following rule will satisfy:*

$$\frac{stmt_1 \xrightarrow{g:\alpha} stmt_1}{stmt_1; stmt_2 \xrightarrow{g:\alpha} stmt_1; stmt_2}.$$

Inference rules for both PROMELA and *Java* constructs, such as $expression$, $assignment$, $send$, $receive$, etc., will be derived from both Lemma 1 and Corollary 1. Subsequently, these rules are used to generate program graphs for the corresponding constructs.

D. Program Graph Equivalence

The definition of equivalence (or similarity) between two program graphs is inspired by the definition of digraph isomorphism. The only difference is that the bijective function f maps two locations representing statements from two different program graphs. For any two locations ℓ_1 and ℓ_2, $f(\ell_1) = \ell_2$ if and only if both ℓ_1 and ℓ_2 represent two semantically equivalent statements or substatements. The formal definition of program graph similarity is stated in the Definition 2.

Definition 2. *Let* $PG_1 = (Loc_1, Act_1, \text{Effect}_1, \hookrightarrow_1, Loc_{0,1}, g_{0,1})$ *and* $PG_2 = (Loc_2, Act_2, \text{Effect}_2, \hookrightarrow_2, Loc_{0,2}, g_{0,2})$ *be two PGs over variables* Var_1 *and* Var_2*, respectively.* PG_1 *and* PG_2 *are called* equivalent *(or similar), denoted* $PG_1 \simeq PG_2$*, only if there are two bijective functions* $f : Loc_1 \to Loc_2$ *and* $g : Loc_1 \times Loc_1 \to Loc_2 \times Loc_2$*, such that:*

- *for any* $\ell_1 \in Loc_1$ *there is* $\ell_2 \in Loc_2$ *such that* $f(\ell_1) = \ell_2$*; and*
- *for any* $(\ell_{1_i}, \ell_{1_j}) \in Loc_1 \times Loc_1$ *there is* $(\ell_{2_m}, \ell_{2_n}) \in Loc_2 \times Loc_2$ *such that* $g(\ell_{1_i}, \ell_{1_j}) = (\ell_{2_m}, \ell_{2_n})$*, where* $f(\ell_{1_i}) = \ell_{2_m}$*,* $f(\ell_{1_j}) = \ell_{2_n}$*.*

III. Discussion

The list of construct associations between the subset of PROMELA and *Java* constructs in Table I can be formally defined in the Definition 3.

Definition 3. *Let* P *and* J *be the subset of PROMELA constructs and Java constructs, respectively, in* CA *(construct association).* CA *is defined as a relation from* P *to* J*, namely* $CA : P \to J$*, such that for any construct* $Cp_i \in P$*, and* $Cj_k \in J$*,* $(Cp_i, Cj_k) \in CA$ *only if there is a PROMELA construct* Cp_i *associated with a Java construct* Cj_k*. This is illustrated in Fig. 4.*

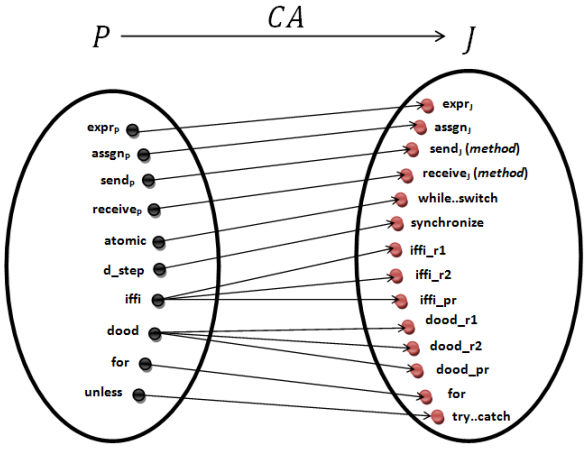

Fig. 4: Association diagram from set P to J

The correctness of association defined in Definition 3 is proved by showing the similarity between two program graphs of every two associated constructs in association.

Theorem 1. *The association* CA *is* ***correct*** *only if for every association* $(Cp_i, Cj_k) \in CA$*,* $Cp_i \in P$*, and* $Cj_k \in J$*, then* $PG(Cp_i) \simeq PG(Cj_k)$*, where* $PG(Cp_i)$ *is a program graph for* Cp_i*, and* $PG(Cj_k)$ *is a program graph for* Cj_k*.*

The proof of Theorem 1 is given in the following subsection.

A. The proof of Construct Association's Correctness

The proof is carried out for all association $(Cp_i, Cj_k) \in CA$; $Cp_i \in P$, $Cj_k \in J$ (see Fig. 4).

According to the Definition 1, a program graph consists of a set of locations Loc, a set of actions Act, an effect function *Effect*, a conditional transition relation $\hookrightarrow$, a set of initial locations Loc_0, and an initial condition g_0.

Loc is determined by the corresponding set of substatements, while the conditional transition relation is provided by inference rules represented in Structured Operational Semantics (SOS) notation determining the transition from one location to other locations. In the level of program graph, actions are normally in the form of expression, assignment, and send or receive. The effect function could be any evaluation function that has any possibility to change variable's value in the construct. In graphical representation of program graph, the initial location will be denoted by double-line circle (or ellipse) pointed by an arrow.

1) ***Expression****:* In PROMELA, an expression $expr_P$ is the most elementary construct in modeling, and it is an atomic statement [1]. Therefore, it only needs one step of execution when the value of $expr_P$ is not zero [9], it means that there is a transition from an initial location to the next location $exit$. This transition is depicted by the following inference rule.

$$\frac{}{expr_P \xhookrightarrow{value(expr_P) \neq 0} exit}$$

On the other hand, if the value of $expr_P$ is zero, then there is no transition to the next location. Since the expression $expr_P$ blocks, the execution cannot be continued, namely it has to wait until the value of $expr_P$ is not zero. The inference rule for this transition is as follows:

$$\frac{}{expr_P \xhookrightarrow{value(expr_P) = 0} expr_P}.$$

The set of locations Loc of a program graph for $expr_P$, $PG(expr_P)$ is determined by its substatement, so that $Loc = sub(expr_P) = \{expr_P, exit\}$. Act is the evaluation of $expr_P$, conditional transition relation is represented by its inference rules, the initial location, $Loc_0 = \{expr_P\}$, and the initial condition, g_0 is $val(expr_P) \neq 0$. Then, the program graph for $expr_P$, $PG(expr_P)$, is shown in Fig. 5.

In *Java*, an expression $expr_J$ has a similar form with the one in PROMELA. Unlike expression in PROMELA, expression in *Java* is not atomic—especially for the long one. However, the synchronization in *Java* can be used to ensure that there is no other process interfering the result of expression evaluation. In this way, the atomicity of $expr_J$ can be preserved. In *Java*, this implementation is carried out by defining some methods, such as lock() and unlock() in a class containing global variables. Suppose header is the name of the intended class, then the implementation of locking is shown in Listing 3.

Listing 3: The usage of lock and unlock

```
...
header.lock();
expr_J;
header.unlock();
...
```

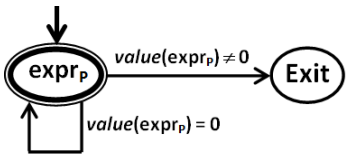

Fig. 5: A program graph for a PROMELA expression, $expr_P$

This implementation guarantees atomicity, and the program graph for $expr_J$ can be generated in similar way. For example, transition to the next location $exit$ will occur only if the value of $expr_J$ is not zero. See the following inference rules:

$$\frac{}{expr_J \xhookrightarrow{value(expression_J)\neq 0} exit}.$$

Otherwise, transition will move back to the initial location, $expr_J$, as it is shown by the following inference rule:

$$\frac{}{expr_J \xhookrightarrow{value(expr_J)=0} expr_J}.$$

Formally, a program graph for $expr_J$, $PG(expr_J)$ consists of a set of locations, $Loc = sub(expr_J) = \{expr_J, exit\}$, a set of actions, Act containing the evaluation of $expr_J$, conditional transition relation is represented by its inference rules, a set of initial location, $Loc_0 = \{expr_J\}$, and an initial condition, g_0 is $val(expr_J) \neq 0$. Then, the program graph for $expr_J$ is shown in Fig. 6.

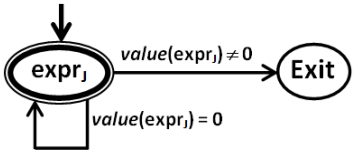

Fig. 6: A program graph for a *Java* expression, $expr_J$

The last step is proving that $PG(expr_P) \simeq PG(exprJ)$ is satisfied. It can be seen from Fig. 5 and Fig. 6 that there must be two bijective functions f and g, such that $f(expr_P) = expr_J$, $f(exit) = exit$; and $g(expr_P, exit) = (expr_J, exit)$, $g(expr_P, expr_P) = (expr_J, expr_J)$. Hence, it is proven that $PG(expr_P) \simeq PG(exprJ)$. ■

2) ***Assignment****:* Given the association $(assgn_P, assgn_J) \in CA$, and we prove that $PG(assgn_P) \simeq PG(assgn_J)$.

In PROMELA, an assignment has the form of $x = expr_P$, and like an expression, an assignment is atomic [1]. In addition, it is always executable provided that the value of $expr_P$ and x are compatible, i.e., $dom(value(expr_P)) \subseteq dom(x)$. The effect of its execution is that the value of $expr_P$ is stored to variable x. Since an assignment is atomic, it only requires one step of execution. The transition occurs from the initial location $assgn_P$ to the next location $exit$ as depicted by the following inference rule:

$$\frac{}{assgn_P \xhookrightarrow{dom(value(expr))\subseteq dom(x):x\leftarrow value(expr)} exit}.$$

A program graph for $assgn_P$ consists of $Loc = \{assgn_P, exit\}$, Act contains evaluation of $x = expr_P$, conditional transition relation is represented by inference rules, $Loc_0 = \{assgn_P\}$, and $g_0 = true$ (an assignment is always executable). Graphically, a program graph for $assgn_P$ is then shown in Fig. 7.

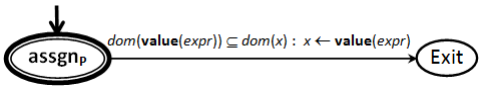

Fig. 7: A program graph for a PROMELA assignment, $assgn_P$

In *Java*, an assignment $assgn_J$ also has the form of $x = expr_J$. It is always executable, provided that x and $expr_J$ are compatible, i.e., $type(value(expr_J)) \subseteq type(value(x))$. The effect of its execution is that the value of $expr_J$ is stored to x. Because *assignment* contains *expression*, and in *Java* expression is not atomic, then *assignment* is not atomic. However, by giving it the same handling as in expression (i.e., $header.lock(); x = expr_J; header.unlock();$), it can be made atomic. Hence, the transition occurs from the initial location $assgn_J$ to the next location $exit$ as described by the following inference rule:

$$\frac{}{assgn_J \xhookrightarrow{type(value(expr_J))\subseteq type(value(x))} exit}.$$

Formally, a program graph for $assgn_J$ consists of $Loc = \{assgn_J, exit\}$, Act contains evaluation of $x := expr_J$, conditional transition relation is inference rule, $Loc_0 = \{assgn_J\}$, and $g_0 = true$ (an assignment is always executable). And graphically, it is shown in Fig. 8.

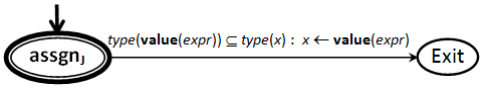

Fig. 8: A program graph for a *Java* assignment, $assgn_J$

The last step is proving the similarity between $PG(assgn_P)$ and $PG(assgn_J)$. From Fig. 7 and Fig. 8, it can be seen that these two program graphs are exactly the same. Therefore, there must be two bijective functions f and g, such that $f(assgn_P) = assgn_J$, $f(exit) = exit$; and $g((assgn_P, exit)) = (assgn_J, exit)$. Hence, it is proven that $PG(assgn_P) \simeq PG(assgn_P)$. ■

3) ***Communications****:* PROMELA has two kinds of communication operations, *i.e.*, *send* and *receive* statements. They are atomic [1]. Both $send$ and $receive$ operations assume that the capacity of communication media (or *channel*) is greater than zero (buffered, or asynchronous) [9].

In PROMELA, $send$ operation has two forms: $c!expr$ and $c!!expr$. These operations can be performed when the channel c is not full, otherwise it will block (*i.e.*, it has to wait until there is a space in the channel). In addition, $expr$ and c must be compatible. Each of these $send$ operations, $c!expr$ and $c!!expr$, gives different effect to the channel. $c!expr$ and $c!!expr$ place the value of $expr$ in the rear, and in the front of channel, respectively.

This operation requires one step of execution, so that when all conditions are met the transition occurs from initial

location $send_P$ to the next location $exit$. Their transitions are depicted by the following inference rules:

$$\frac{\neg full(c)}{c!expr \xrightarrow{dom(expr)\subseteq dom(c):c.rear \leftarrow value(expr)} exit},$$

and

$$\frac{\neg full(c)}{c!!expr \xrightarrow{dom(expr)\subseteq dom(c):c.front \leftarrow value(expr)} exit}.$$

A program graph for $send$ consists of $Loc = sub(send) = \{c!expr, exit\}$, Act contains expression, and assignment forms, conditional transition relation is its inference rules, $Loc_0 = \{c!expr\}$, and g_0 is $\neg full(c)$. Graphically, a program graph for $send$ is then shown in Fig. 9.

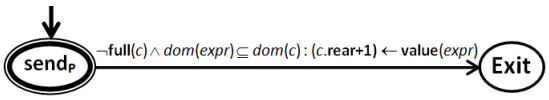

Fig. 9: A program graph for a $send$ operation, $c!expr$

As mentioned above, the only difference between $c!expr$ and $c!!expr$ is on the effect to the channel. Therefore, they have the same program graph (see Fig. 9).

On the other hand, $receive$ statement is used for receiving messages from *channels*. Unlike $send$ statement, it has four forms: $c?x$, $c??x$, $c? < x >$, and $c?? < x >$, where c and x denote the name of the *channel* and the list of argument(s) used to receive the messages, respectively.

The first and third forms of the statement (written in ?) are executable if the first message in the channel matches the pattern from the receive statement. While, the second and fourth forms of the statement (written in ??) are executable if there exists at least one message anywhere in the channel that matches the pattern from the receive statement.

A match of a message is obtained if all message fields that contain constant values in the receive statement equal the values of the corresponding message fields in the message. If no angle brackets are used, the message is removed from the channel buffer after the values are copied, otherwise (angle brackets are used), the message is not removed and remains in the channel.

For any form of $receive$ statement, it should be assumed that the channel c is not empty, otherwise it blocks (it has to wait until there is a value in the channel). $c?x$ needs an additional requirement: c and x must be compatible. The effect of $c?x$ operation is that the value of the front channel (*i.e.*, $c.front$) will be taken from c and stored into x.

When all requirements are met, there will be a transition from the initial location $c?x$ to the next location $exit$ as depicted in the following inference rule:

$$\frac{\neg empty(c)}{c?x \xrightarrow{dom(c)\subseteq dom(x):x \leftarrow c.front} exit}.$$

The compatibility requirement for $c??x$ is softer than the one for the previous statement, which is there must be at least one i, $front \leq i \leq rear$ such that $c.i$ and x are compatible. The effect of $c??x$ operation is that the value of the i-th position in c (*i.e.*, $c.i$) will be taken and stored into x.

When all requirements are met, there will be a transition from the initial location $c??x$ to the next location $exit$ as depicted in the following inference rule:

$$\frac{\neg empty(c)}{c??x \xrightarrow{dom(c.i)\subseteq dom(x):x \leftarrow c.i} exit},$$

for some i, $front \leq i \leq rear$.

The program graph for the $receive$ statements $c?x$, $c? < x >$, $c??x$, or $c?? < x >$ is, respectively, defined formally with $Loc = sub(c?x) = \{c?x, exit\}$, $Loc = sub(c? < x >) = \{c? < x >, exit\}$, $Loc = sub(c??x) = \{c??x, exit\}$, and $Loc = sub(c?? < x >) = \{c?? < x >, exit\}$, Act contains expression, assignment forms. The conditional transition relation is inference rule, $Loc_0 = \{c?x\}$, $Loc_0 = \{c? < x >\}$, $Loc_0 = \{c??x\}$, and $Loc_0 = \{c?? < x >\}$, and g_0 is $\neg empty(c)$. Based on the way the value moves from the channel to the variable, the program graph for $c?x$ and $c??x$ are very much the same. Therefore, the program graphs for $receive$ statements are sufficiently depicted one in Fig. 10.

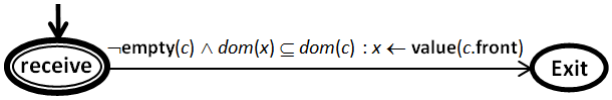

Fig. 10: A program graph for a $receive$ statement

Communication operations between two or more processes in *Java* (*i.e.*, both $send$ and $receive$), either they are in the same computer or different computers are implemented by defining a separate class containing at least a buffer and two methods. Particularly, when two or more communicating processes are in different computers, then either socket programming or Remote Method Invocation (RMI) should be implemented.

In *Java*, socket programming is the most widely used concept in networking [14], [15]. Sockets provide the communication mechanism between two computers using TCP (Transmission Control Protocol). A client program creates a socket on its end of the communication and attempts to connect that socket to a server. When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket [14], [15]. When a communication is performed via socket, the success of both operations actually does not only depend on contention of the buffer, but also on the network connection. The sending and receiving can only proceed when the buffer is not full and is not empty, respectively, plus connection is still maintained.

In accordance with the implementation discussed above, $send$ and $receive$ statements in PROMELA are implemented by a method $send$ and $receive$ respectively in *Java*. In this way, they can be defined in a similar way to $send$ and $receive$ statements in PROMELA. Hence, the program graph for $send$ statement in PROMELA and $send$ method in *Java*, $receive$ statement in PROMELA and $receive$ method in *Java* are equivalent (or similar). ■

4) ***Weak Atomicity****:* Given $(atomic, while - switch) \in CA$, for $atomic \in P$, $while - switch \in J$, and we prove that $PG(atomic) \simeq PG(while - switch)$.

In PROMELA, an *atomic* construct is expressed in a form of $atomic\{stmt_1, \ldots, stmt_n\}$, in which $stmt_i$ for $1 \leq i \leq$

n can be any construct. The effect of an *atomic* evaluation toward the change of variable's values is postponed until the last statement $stmt_n$ is completely evaluated. In the execution of an *atomic* construct, there might be $stmt_j$ for some j, $1 \leq j \leq n$, that blocks, and control flow stays in the location where the list of arguments $stmt_1, \ldots, stmt_{j-1}$ for some j, $1 \leq j \leq n$ has completely been executed. The rest of arguments list $stmt_j, \ldots, stmt_n$ will be treated exactly the same as an *atomic* except with shorter length of the list. Hence, during an *atomic* construct's execution, it is possible that other process(es) take control until $stmt_j$ is executable.

The inference rule of an *atomic* construct if all statements in the argument list are executable during the execution is defined as:

$$\frac{\forall i, 1 \leq i \leq n, stmt_i \ executable}{atomic\{stmt_1, \ldots, stmt_n\} \longrightarrow exit}.$$

However, if for some i, $1 \leq i \leq n$, $stmt_i$ blocks, it is defined as:

$$\frac{\exists i, stmt_i \ block}{atomic\{stmt_1, \ldots, stmt_n\} \longrightarrow atomic\{stmt_i, \ldots, stmt_n\}}.$$

In the latter case, the transition stops in the "temporary" location $atomic\{stmt_i, \ldots, stmt_n\}$. Whenever $stmt_i$ becomes executable, it will be treated similarly as the previous one except with the shorter list of arguments.

Formally, a program graph for an *atomic* $PG(atomic)$ consists of a set of locations, $Loc = sub(atomic) = \{atomic\{stmt_1, \ldots, stmt_n\}, atomic\{stmt_i, \ldots, stmt_n\}, exit\}$; a set of actions, Act consists of actions are in the form of an expression, assignment, send or receive, $Loc_0 = \{atomic\{stmt_1, \ldots, stmt_n\}\}$, and g_0 is $stmt_1$ executable. Therefore, a program graph for an *atomic* construct is graphically depicted in Fig. 11.

The execution of a sequence $stmt_1, stmt_s, \ldots, stmt_n$ inside $atomic$ might be interleaved by a certain number of other processes because of blocking. The consequence is that an *atomic* is partitioned into several *subatomics*. Even so, the effect of variable valuation will be accumulated at the execution of the last statement in the list of arguments.

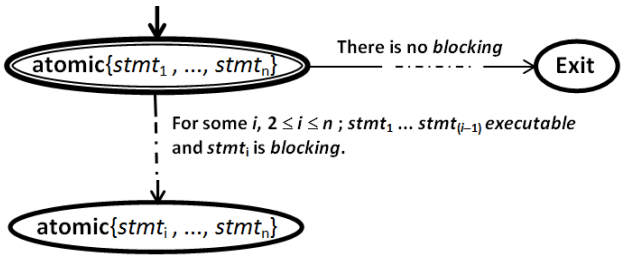

Fig. 11: A program graph for an *atomic* construct

In *Java*, an $atomic$ construct is implemented by a $switch$ construct inside of a $while$ loop. The $switch$ construct is used to accommodate a number of guards, while the $while$ loop is used to make the flow of program keeps returning to the loop until the last statement in the $atomic$ is completely executed. A *Java* code fragment of $switch - while$ implementation is depicted in Listing 4.

Listing 4: An implementation of $atomic$ in *Java*

```
...
int stmt_nmbr = 1;
while (stmt_nmbr <= nmbr_of_stmt) {
   header.lock();
   switch (stmt_nmbr) {
      case 1:
           if (!stmt_1) break;
             stmt_1;
             stmt_nmbr=2;
      case 2:
           if (!stmt_2) break;
             stmt_2;
             stmt_nmbr=3;
               ...
               ...
      case n:
           if (!stmt_n) break;
             stmt_n;
             stmt_nmbr=nmbr_of_stmt+1;
   }
   header.unlock();
}
...
```

The loop of a $while$ is exited when $stmt\_nmbr > nmbr\_of\_stmt$—when all statements in the argument list of an $atomic$ had completely been executed ($nmbr\_of\_stmt$ denotes the number of statements are in an *atomic* construct). The block of $switch$ is surrounded by $header.lock()$ and $header.unlock()$ to ensure the atomicity of a $stmt_i$ execution in each case. According to the $switch$ definition in Listing 4, for any i, $1 \leq i \leq n$, if $stmt_i$ is false then exit from $switch$ before updating the value of $stmt\_nmbr$. If $i \leq n$, then $stmt\_nmbr \leq nmbr\_of\_stmt$ and program control is still in $while$ loop. However, once exit the $switch$, $header.unlock()$ is executed—the lock is released. This means that other processes are allowed to take a control flow until $stmt_i$ becomes true. It is similar with what happen in an *atomic* construct when $stmt_i$ is not executable.

The inference rules for $while - switch$ are defined as follows: If $stmt_i$ is executable for all $i, 1 \leq i \leq n$ during the execution, then:

$$\frac{\forall i, 1 \leq i \leq n, stmt_i \ executable}{[while - switch]\{stmt_1, \ldots, stmt_n\} \longrightarrow exit}.$$

However, when there is i, $1 \leq i \leq n$, and $stmt_i$ is not executable, then

$$\frac{stmt_i, 2 \leq i \leq n \quad block}{[while - switch]\{stmt_1, \ldots, stmt_n\} \longrightarrow [while - switch]\{stmt_i, \ldots, stmt_n\}}.$$

Formally, a program graph for a $while - switch$ construct consists of a set of locations, $Loc = sub([while - switch]) = \{[while - switch]\{stmt_1, \ldots, stmt_n\}, [while - switch]\{stmt_i, \ldots, stmt_n\}, exit\}$, a set of actions, Act consists of either an expression, assignment, send or receive. $Loc_0 = \{[while - switch]\{stmt_1, \ldots, stmt_n\}\}$, and g_0 is $stmt_1$ executable. And, the corresponding program graph is shown in Fig. 12.

To prove the equivalence (or similarity) between the program graph for *atomic* construct and one for $while - switch$ construct in *Java* is done by comparing the program graph in Fig. 11 and one in Fig. 12, and it is evident that $PG(atomic) \simeq PG(while - switch)$. ■

5) ***Strong Atomicity****:* Given $(d\_step, synchronize) \in CA$, for $d\_step \in P$, $synchronize \in J$, and we prove that $PG(d\_step) \simeq PG(synchronize)$.

d_step introduces a deterministic code fragment that is executed indivisibly [17]. Syntactically, it is like $atomic$ construct, except some differences : (1) goto cannot come into or go out of a d_step sequence; (2) the $sequence$ is executed deterministically, if non-determinism occurs, it is carried out in deterministic manner, for example, by always selecting the first true (or executable) guard in every selection and repetition structure; and (3) if the execution of any statement inside the sequence can block, it is an error. For this reason, in most cases $send$ and $receive$ statements cannot be used inside d_step sequence.

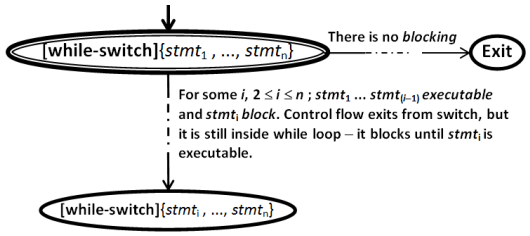

Fig. 12: A program graph for a $while - switch$ construct in *Java*

Formally, a program graph for a d_step construct consists of a set of locations, $Loc = sub(d\_step) = \{d\_step\{stmt_1, \ldots, stmt_n\}, exit\}$; a set of actions, Act is in the form of an expression, assignment, or send and receive; a set of initial locations, $Loc_0 = \{d\_step\{stmt_1, \ldots, stmt_n\}\}$; and an initial condition, g_0 which is the executability of $stmt_1$ (*i.e.*, the first statement in the sequence). And graphically, a program graph for a d_step construct is shown in Fig. 13.

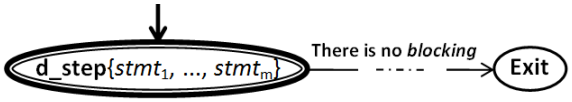

Fig. 13: A program graph for a d_step construct

The above situation can only be achieved when a d_step construct is in the verified model—all statements in the argument list are already proved executable. Otherwise, the program graph for d_step will consist of a set of locations, $Loc = sub(d\_step) = \{d\_step\{stmt_1, \ldots, stmt_n\}, error, exit\}$; a set of actions, Act is in the form of an expression, assignment, send or receive; a set of initial locations, $Loc_0 = \{d\_step\{stmt_1, \ldots, stmt_n\}\}$; and an initial condition, g_0 which is the executability of $stmt_1$. And graphically, a program graph for a d_step with the possibility of error is presented in Fig. 14.

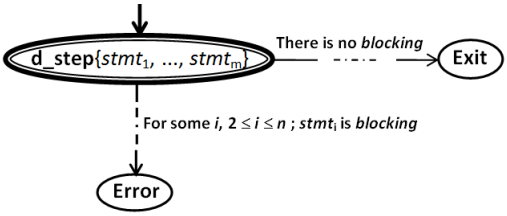

Fig. 14: A program graph for a d_step construct with the possibility of error

Java provides a $synchronized$ keyword to methods that cause only one invocation of a synchronized method on the same object at a time. Every object has an intrinsic lock associated with it. A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it is done with them. A *Java* method may be synchronized, which guarantees that at most one thread can execute the method at a time. Other threads wishing access are forced to wait until the currently executing thread completes [14], [15].

There is also a synchronized statement in *Java* that forces threads to execute a block of code sequentially. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock.

Synchronize in *Java* might be used to implement d_step construct in PROMELA. In accordance with d_step construct's behavior, however, synchronized statement is more appropriate than synchronized method.

The inference rule for a $synchronize$ is defined as:

$$\frac{}{synchronize\{stmt_1, \ldots, stmt_n\} \longrightarrow exit}.$$

Formally, a program graph for a $synchronize$ consists of a set of locations, $Loc = sub(synchronize) = \{synchronize\{stmt_1, \ldots, stmt_n\}, exit\}$; set of actions, Act consists of either an expression, assignment, send or receive; set of initial locations, $Loc_0 = \{synchronize\{stmt_1, \ldots, stmt_n\}\}$; and go_0 is the executability of $stmt_1$. And the program graph for a *synchronize* is shown in Fig. 15.

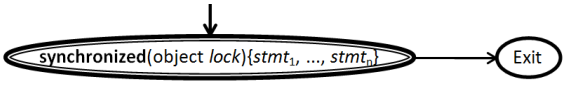

Fig. 15: A program graph for a *synchronize* in *Java*

To prove the equivalence (or similarity) between the program graph for a d_step and one for a $synchronize$ is done by comparing these two program graphs from Fig. 13 and Fig. 15 respectively. It is evident that there must be two bijection functions f, and g such that $f(d\_step) = synchronize$, $f(exit) = exit$, and $g(d\_step, exit) = g(synchronize, exit)$. Hence, it is proven that $PG(d\_step) \simeq PG(synchronize)$. ■

The synchronize's implementation in Fig. 15 is taken by assuming (or assuring) that every $stmt_i$, $1 \leq i \leq n$, is executable. Otherwise, it is error as described in the following inference rule:

$$\frac{\exists i; 1 \leq i \leq n,\ stmt_i\ block}{synchronize\{stmt_1, \ldots, stmt_n\} \longrightarrow error}.$$

Formally, a program graph for a $synchronize$ with error has a set of locations, $Loc = \{synchronize\{stmt_1, \ldots, stmt_n\}, error, exit\}$; a set of actions, Act is in the form of an expression, assignment, send or receive. The conditional transition relation is its inference rules; a set of initial locations $Loc_0 = \{synchronize\{stmt_1, \ldots, stmt_n\}\}$, and the initial condition g_0 is the executability of $stmt_1$. The program graph for a synchronize with error is shown in Fig. 16.

The only difference with a program graph in the previous case (i.e., one with no error) is an additional location $exit$. Therefore, in this case there is also an equivalence (or a similarity) between a program graph for a d_step with error and one for $synchronize$ with error. ■

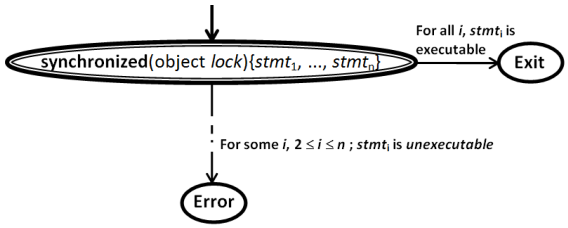

Fig. 16: The program graph for a *synchronize* with *Error* in *Java*

6) ***Selection*** $iffi$*:* In PROMELA, a selection construct $iffi$ has a unique start and stop state. Each option sequence inside the construct defines outgoing transitions for the start state, leading to the stop state. By default, the end of each option sequence leads to the control state that follows the construct [17].

The selection construct $iffi$ must have at least one option sequence (for some integer i, $1 \leq i \leq n$). A sequence, $stmt_i$ for some i, $1 \leq i \leq n$, can be selected for execution only when its guard statement g_i is executable. If more than one guard statements are executable, one of them will be selected in non-deterministic manner. Otherwise, if there is no executable guard statement, the selection construct as a whole blocks. This means that a non-deterministic selection construct $iffi$ as a whole is executable only if there is at least one guard inside it is executable.

Listing 5: A syntax of the selection construct, $iffi$

```
if
 {
  :: g_1 -> stmt_1;
  :: g_2 -> stmt_2;
     ...
     ...
  :: g_n -> stmt_n;
 }
fi
```

The syntax of the selection construct $iffi$ is expressed in a form as shown in Listing 5.

The inference rules of a selection construct $iffi$ are defined as:

$$\frac{stmt_i \xrightarrow{h:\alpha} stmt_i' \neq exit}{iffi \xrightarrow{g_i \wedge h:\alpha} stmt_i'},$$

when the corresponding statement $stmt_i$ of the selected guard g_i requires more than one step of execution, or:

$$\frac{stmt_i \xrightarrow{h:\alpha} stmt_i' = exit}{iffi \xrightarrow{g_i \wedge h:\alpha} exit},$$

when the corresponding statement $stmt_i$ of g_i is one-step of execution, or:

$$\frac{}{iffi \xrightarrow{\neg g_1 \wedge \neg g_2 \ldots \wedge \neg g_n} iffi},$$

when there is no guard holds, the selection blocks.

Formally, a program graph for a nondeterministic selection construct $iffi$ has a set of locations, $Loc = sub(iffi) = \{iffi, exit\} \cup \bigcup_{1 \leq i \leq n} \{sub(stmt_i) \setminus \{stm', exit\}\}$; a set of actions, Act is normally in the form of an expression, assignment, and send or receive; conditional transition relation is all inference rules mentioned above; a set of initial locations, $Loc_0 = \{iffi\}$; and an initial condition, g_0 is guard for selected option g_i's. And graphically, it is shown in Fig. 17.

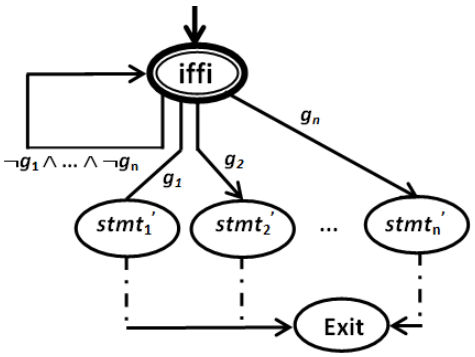

Fig. 17: A program graph for a nondeterministic selection construct $iffi$

In *Java*, a nondeterministic selection construct $iffi$ is implemented in two approaches: (i) randomize, and (2) priority. The behavior of the randomize approach is closer to the one of selection construct $iffi$ in PROMELA than the priority. Since the priority approach behaves like most general purpose (programming) languages including *Java*.

Furthermore, the randomize approach is implemented in two versions (i.e., $iffi\_r_1$, $iffi\_r_2$). The first version of randomize initially generates a random number $i, 1 \leq i \leq n$ for some positive integer n (in which n is number of guards inside selection construct $iffi$). Then the value of i is assigned to a variable $choice$ to select the matching case in the $switch$ construct, and evaluate the guard g_i. If the guard g_i holds (or executable), the corresponding sequence $stmt_i$ will be executed. Otherwise, the function will generate another random number j repeatedly until finds a guard g_j for some j, $1 \leq j \leq n$ holds. As normal, $header.lock()$ and $header.unlock()$ are used to guarantee an atomicity. Then, a *Java* code fragment for this implementation is shown in Listing 6.

Listing 6: A *Java* code fragment of random selection in *Java* version one

```
...
boolean if_flag=false;
while (!if_flag) {
   int choice=1+Math.random()*nmbr_of_opts;
   switch(choice) {
      case 1:
           header.lock();
           if (!g_1) {
              header.unlock();
              break;
           }
           if_flag=true;
           stmt_1;
           break;
      case 2:
           header.lock();
           if (!g_2) {
              header.unlock();
```

```
                break;
            }
            if_flag=true;
            stmt_2;
            break;
        ...
        ...
        case n:
            header.lock();
            if (!g_n) {
                header.unlock();
                break;
            }
            if_flag=true;
            stmt_n;
            break;
    }
}
header.unlock();
...
```

There are two levels of $break$ statement used in this implementation; one is inside if statement and the other is outside. The former is used to exit from $switch$ when $guard_i$ is not true, and allow another process to take the program control (by unlocking). While the later is used to exit from the $switch$ when a guard g_i is true, after executing $stmt_i$. Therefore, a statement $header.unlock()$ is placed outside the $while$ loop.

The semantics of a *Java* selection construct using the randomize approach version one is formally explained by the following inference rules: If a randomly selected guard g_i satisfied, and the corresponding action $stmt_i$ is not one-step statement, then there is a transition to the location $stmt_i^{'}$:

$$\frac{i \longleftarrow random(1 \ldots n) \wedge sequence_i^{'} \neq exit}{iffi\_r_1 \xrightarrow{guard_i} sequence_i^{'}}.$$

If a randomly selected guard g_i satisfied, and the corresponding action $stmt_i$ is one-step statement, then there is a transition to location $exit$:

$$\frac{i \longleftarrow random(1 \ldots n) \wedge sequence_i^{'} = exit}{iffi\_r_1 \xrightarrow{g_i} exit}.$$

If a randomly selected guard, g_i is not satisfied, the transition returns to the location $while$:

$$\frac{i \longleftarrow random(1 \ldots n)}{iffi\_r_1 \xrightarrow{\neg guard_i} iffi\_r_1}.$$

A program graph for an $iffi\_r_1$ is formally derived by determining its components: $Loc = sub(iffi\_r_1) = \{iffi\_r_1, switch, exit\} \cup \{sub(stmt_i) \setminus \{stmt', exit\}\}$, Act containing an expression, assignment and send or receive, $Loc_0 = \{iffi\_r_1, switch\}$, and g_0 is guard for selecting $stmt_i$. Since a new location $switch$ does not give any effect toward the value changes of any other variables in the construct, this location can be ignored. Graphically, a program graph for construct $iffi\_r_1$ is represented in Fig. 18.

The last step is proving $PG(iffi) \simeq PG(iffi\_r_1)$. Unlike the previous constructs, the number of locations between these two compared program graphs is different. This different is caused by an additional location $switch$ in $PG(iffi\_r_1)$ to facilitate a random selection. However, in accordance with Proposition 1, location $switch$ in $PG(iffi\_r_1)$ can be coalesced with the initial location $iffi\_r_1$ becomes one new location, say $iffi\_r_1^{'}$. And the result of this coalescing process is a program graph shown in Fig. 19.

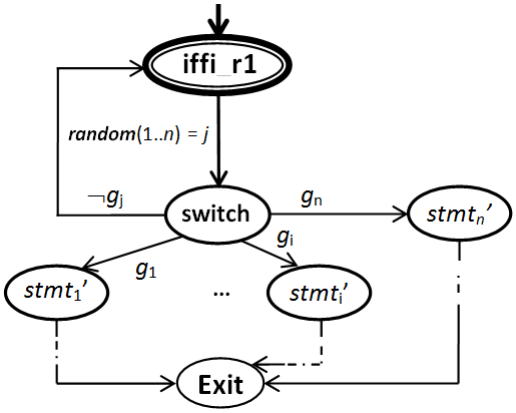

Fig. 18: A program graph for a random selection $iffi\_r_1$ version one

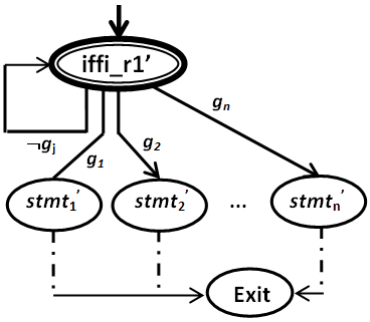

Fig. 19: The result of coalescing locations in a program graph Fig. 18

From Fig. 19, it can be observed that number of locations in $PG(iffi\_r_1^{'})$ is equal to number of locations in $PG(iffi)$. Then, there must be two bijective functions f and g mapping locations and transitions respectively from $PG(iffi)$ to $PG(iffi\_r_1)$. Hence, this proves $PG(iffi) \simeq PG(iffi\_r_1)$. ■

Unlike the first version, this second version of implementation works by first checking all guards g_i in the $iffi$ construct. If a g_i is executable, then the number i is stored in the corresponding position of a List variable $options$ (i.e., $options.add(i)$). If there is no executable guard (i.e., $options.size() = 0$), then blocks and waits until there is at least one executable guard; otherwise (i.e., $options.size() > 0$) generates a number $j, 1 \leq j \leq options.size()$ randomly to select the corresponding case in the $switch$ construct. Because all of $options.size()$ guards in the $options$ variable are executable, so that any number $j, 1 \leq j \leq options.size()$ generated by the random function will cause an execution of $stmt_j$. Therefore, only one level of $break$ statement needed in this implementation. Then, the *Java* code fragment of this implementation is shown in Listing 7.

Listing 7: A *Java* code for $iffi\_r_2$ implementation

```
...
boolean if_flag=false;
while (!if_flag) {
```

```
    List<Integer> options=new List();
    header.lock();
    if (g_1) { options.add(1); }
    if (g_2) { options.add(2); }
    ...
    if (g_n) { options.add(n); }
    header.unlock();

    if (options.size()>0) {
      if_flag=true;
      int choice=Math.random()*
                        options.size();
      choice=options.get(choice);
      switch(choice) {
         case 1 : stmt_1;
                  break;
         case 2 : stmt_2;
                  break;
                ...
                ...
         case options.size() :
                  stmt_options.size();
                  break;
      }
    }
}
...
```

The semantics of a randomize implementation version two for the selection construct, $iffi\_r_2$ is formally explained by the following inference rules:

$$\frac{i \leq n}{stmt_j \longleftarrow stmt_i; j++}.$$

This inference rule is to store $stmt_i$ with satisfied guard g_i into new variable $list_j$, or

$$\frac{j \longleftarrow random(1 \ldots m) \wedge stmt_j^{'} \neq exit}{iffi\_r_2 \overset{true}{\longrightarrow} stmt_j^{'}},$$

when $list_j$ is not a one-step statement, *i.e.*, $list_j^{'} \neq exit$, or:

$$\frac{j \longleftarrow random(1 \ldots m) \wedge stmt_j^{'} = exit}{iffi\_r_2 \overset{true}{\longrightarrow} exit},$$

when $stmt_j$ is a one-step statement, *i.e.*, $stmt_j^{'} = exit$.

Formally, a program graph for the $iffi\_r_2$ construct consists of a set of locations, $Loc = sub(iffi\_r_2) = \{iffi\_r_2, for, switch, exit\} \cup \{sub(stmt_i) \setminus \{stm', exit\}\}$; a set of actions, Act normally contains an expression, assignment, send or receive; a set of initial locations, $Loc_0 = \{iffi\_r_2, for, switch\}$, and the initial condition g_0 is the executability of g_i for selecting $stmt_i$. And the program graph for the $iffi\_r_2$ construct is graphically shown in Fig. 20.

The last step is proving $PG(iffi) \simeq PG(iffi\_r_2)$. The only different with the first version is that in $PG(iffi\_r_2)$ there are two additional locations: for to mark all satisfied guards, and $switch$ to randomly select one among all marked guards. According to Proposition 1, locations for and $switch$ in $PG(iffi\_r_2)$ can be coalesced with the initial location $iffi\_r_2$ becomes one new location, say $iffi\_r_2^{'}$. The next argumentation is similar to the one in the version one, i.e., construct of $iffi\_r_1$. Hence, this concludes that $PG(iffi) \simeq PG(iffi\_r_2)$ is proven. ■

The third implementation of the selection construct $iffi$ is by using conditional construct $else\ if$. Therefore, the

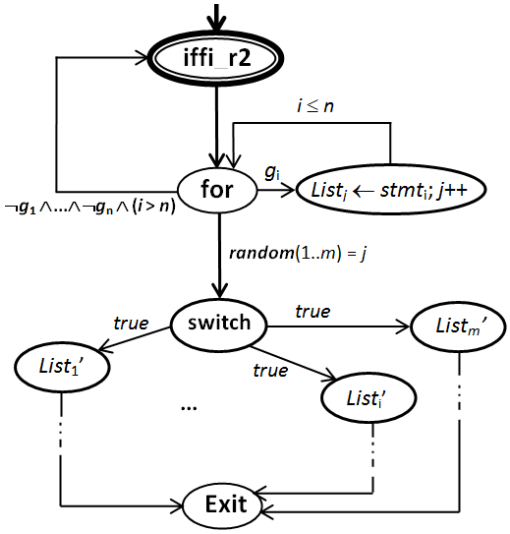

Fig. 20: A program graph for the randomize implementation, $iffi\_r_2$ in *Java*

flow of execution is similar with one of $else\ if$ in *Java*, except the process is repeated until one of condition in $else\ if$ is executable. Then, the *Java* code fragment for this implementation is shown in Listing 8.

Listing 8: A *Java* code for the priority selection, $iffi\_pr$

```
...
boolean if_flag=false;
while (!if_flag) {
   int choice=-1;
   header.lock();
   if (g_1) {
     choice=1;
   } else if (g_2) {
     choice=2; }
   ...
     else if (g_n) {
     choice=n; }
   header.unlock();

   switch(choice) {
        case 1:
             if_flag=true;
             stmt_1; break;
         case 2:
             if_flag=true;
             stmt_2; break;
               ...
               ...
         case n:
             if_flag=true;
             stmt_n; break;
   }
}
...
```

The evaluations in an $else - if$ construct is done sequentially starting from the first condition until find one satisfied, and start to execute a corresponding option $stmt$. If, however, there is no condition found satisfied, the evaluation of conditions is carried out repeatedly until find a satisfied one. It is why this approach is also called by the priority selection.

The semantics of the *Java* priority selection construct $iffi\_pr$ is formally explained by the following inference

rules:

$$\frac{for \;\; some \;\; i, 1 \leq i \leq n \wedge stmt_i^{'} \neq exit}{iffi\_pr \xrightarrow{g_i} stmt_i^{'}},$$

when there is a guard g_i for some i satisfied, but $stmt_i^{'} \neq exit$, or:

$$\frac{for \;\; some \;\; i, 1 \leq i \leq n \wedge stmt_i^{'} = exit}{iffi\_pr \xrightarrow{g_i} exit},$$

when there is a guard g_i for some i holds, and $stmt_i^{'} = exit$, or:

$$\frac{}{iffi\_pr \xrightarrow{\neg g_1 \wedge \neg g_2 \wedge \ldots \wedge \neg g_n} iffi\_pr},$$

when there is no guard satisfied, then there is no transition occurs (i.e., it is still in the location $iffi\_pr$).

Formally, a program graph for the $iffi\_pr$ consists of a set of locations, $Loc = sub(iffi\_pr) = \{iffi\_pr, exit\} \cup \{sub(stmt_i) \backslash \{stm', exit\}\}$; a set of actions, Act containing an expression, assignment, and send or receive, a set of initial locations, $Loc_0 = \{iffi\_pr\}$, and an initial condition, g_0 is the first satisfied guard g_i's. And graphically, a program graph for priority selection $iffi\_pr$ is shown in Fig. 21.

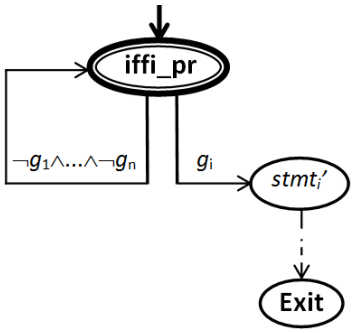

Fig. 21: A program graph for the priority selection, $iffi\_pr$

The last step is proving $PG(iffi) \simeq PG(iffi\_pr)$. In this implementation there is no additional locations is required. In accordance with the behavior of $if - else$ construct, however, there is always only one option (or path) corresponding to the first executable guard taken. Therefore, a set of locations in $PG(iffi\_pr) \subseteq$ a set of locations in $PG(iffi)$. It means, any path taken in $PG(iffi\_pr)$ there is always a corresponding path in $PG(iffi)$. Hence, it is also proven that $PG(iffi) \simeq PG(iffi\_pr)$. ■

7) ***Deterministic repetition*** *for:* In PROMELA, the deterministic repetition construct for is represented in three version of forms (or syntaxes).

(i) **for** $(var : expr_1 \ldots expr_2)\{stmt\}$, the repetition keeps running providing $var \geq expr_1$ && $var \leq expr_2$. Starting with $expr_1$, the value of var is incremented by one each time the repetition is taken, and the repetition will stop when the value of var reaches the value of $expr_2$.

The semantics of the deterministic repetition for version one is formally explained by the following inference rules:

$$\frac{true : var++}{for \xrightarrow{var \geq expr_1 \&\& var \leq expr_2} for},$$

or:

$$\frac{true : var++}{for \xrightarrow{var < expr_1 \| var > expr_2} exit}.$$

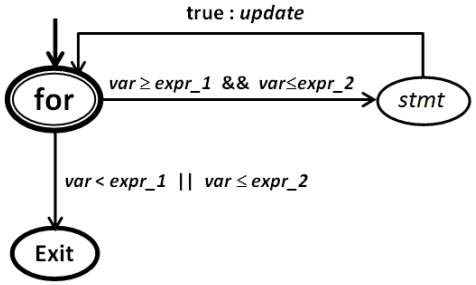

Fig. 22: A program graph for the deterministic repetition for version one

A program graph for the deterministic repetition for version one is formally derived by determining its components: $Loc = sub(for) = \{for, exit\} \cup \{sub(stmt) \setminus \{exit\}\}$, Act contains an expression, assignment, and send or receive, $Loc_0 = \{for\}$, and g_0 is $var \geq expr_1$ && $var \leq expr_2$. The graphical representation of a program graph for the deterministic repetition for is shown in Fig. 22.

(ii) **for** $(var \;\; in \;\; array)\{stmt\}$, the value of var starts from 0, and the repetition will keep running providing the value of var is less than the length of the array. The value of var is incremented by one each time the repetition is taken.

The semantics of the deterministic repetition for version two is formally explained by the following inference rules:

$$\frac{true : var++}{for \xrightarrow{var < length(array)} for},$$

or:

$$\frac{true : var++}{for \xrightarrow{var = length(array)} exit}.$$

A program graph for the deterministic repetition for version two is formally derived by determining its components: $Loc = sub(for) = \{for, exit\} \cup \{sub(stmt) \setminus \{exit\}\}$, Act contains an expression, assignment, and send or receive, $Loc_0 = \{for\}$, and g_0 is $var < length(array)$. A program graph for this version is shown graphically in Fig. 23.

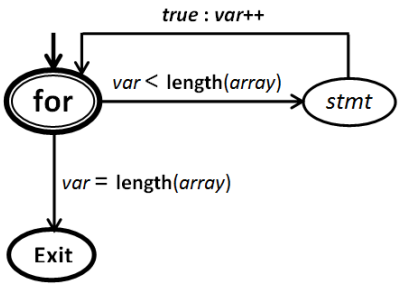

Fig. 23: A program graph for the deterministic repetition for version two

(iii) **for** $(var \;\; in \;\; channel)\{stmt\}$, this third use of the for statement is to retrieve all messages from a channel sequentially. For doing this, the channel must be defined in a special way, with a single user-defined type as its contents. The repetition is carried out by initializing counter variable var associated with the first location of first value stored in the channel by one, and var is incremented each time the repetition is taken. The repetition keeps running when the value of var is less than or equal to the position of the last value stored in the channel; otherwise it stops.

The semantics of the deterministic repetition for version three is formally described by the following inference rules:

$$\frac{true : var++}{for \xrightarrow{var \leq len(channel)} for},$$

or:

$$\frac{true : var++}{for \xrightarrow{var > len(channel)} exit}.$$

A program graph for the deterministic repetition for version three is formally derived by determining its components: $Loc = sub(for) = \{for, exit\} \cup \{sub(stmt) \setminus \{exit\}\}$, Act contains an expression, assignment, and send or receive, $Loc_0 = \{for\}$, and g_0 is $var < len(channel)$. The graphical representation of a program graph is shown in Fig. 24.

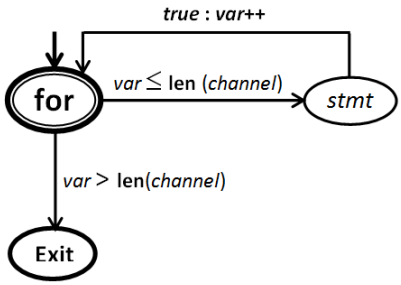

Fig. 24: A program graph for the deterministic repetition for version three

In *Java*, the three versions of deterministic repetition for in PROMELA are implemented by a similar construct **for** $(init; boolean\_expr; update)\{stmt\}$. In this construct, the initialization step $init$ is executed first, and only once. Then, the boolean expression $boolean\_expr$ is evaluated. If it is true, $stmt$ is executed, otherwise, $stmt$ is not executed and flow of control jumps to the next statement past the for loop. After $stmt$ being executed, the flow of control jumps back up to the update statement $update$. It allows updating any loop control variables, and $boolean\_expr$ is again evaluated. The repetition keeps running until find the $boolean\_expr$ is false.

The semantics of *Java*'s repetition for is formally explained by the following inference rules:

$$\frac{true : update}{for \xrightarrow{boolean\_expr} for},$$

or:

$$\frac{true : update}{for \xrightarrow{\neg boolean\_expr} exit}.$$

A program graph for the deterministic repetition for in *Java* is formally derived by determining its components: $Loc = sub(for) = \{for, exit\} \cup \{sub(stmt) \setminus \{exit\}\}$, Act contains an expression, assignment, and send or receive, $Loc_0 = \{for\}$, and g_0 is $boolean\_expr$. The graphical representation of a program graph is shown in Fig. 25.

The last step is proving three equivalences (or similarities) $PG(for\_P_1) \simeq PG(for\_J)$, $PG(for\_P_2) \simeq PG(for\_J)$, and $PG(for\_P_3) \simeq PG(for\_J)$, where for_P_1, for_P_2, and for_P_3 are deterministic for in PROMELA version one, two, and three respectively; and for_J is deterministic for in *Java*.

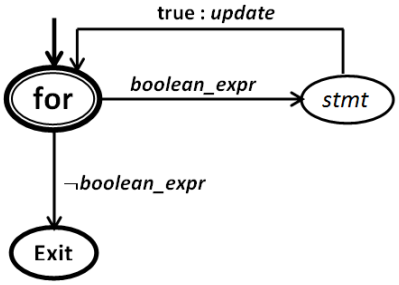

Fig. 25: A program graph for the deterministic repetition for in *Java*

From Fig. 22, 23, 24, and Fig. 25, it is evident that their program graphs are equivalent (or similar). In other word, it can be concluded that $PG(for\_P_1) \simeq PG(for\_J)$, $PG(for\_P_2) \simeq PG(for\_J)$, and $PG(for\_P_3) \simeq PG(for\_J)$ are proven. ■

8) ***Repetition - do****:* A repetition construct $dood$ has a single start and stop state. Each option sequence inside the construct defines outgoing transitions for the start state. The end of each option sequence transfer control back to the start state of the construct, allowing for repeated execution. The stop state of the construct can only be reached via a $break$ statement from inside one of its option sequence [17].

The syntax of the repetition construct $dood$ is expressed in a form as shown in Listing 9.

Listing 9: A syntax of the repetition construct $dood$

```
do
 {
  :: g_1 -> stmt_1;
  :: g_2 -> stmt_2;
     ...
     ...
  :: g_m -> stmt_m;
 }
od
```

The repetition construct must have at least one option sequence. An option can be selected for execution only when its guard statement is executable. If more than one guard statement is executable, one of them will be selected nondeterministically, otherwise, the repetition construct as a whole blocks. This means a repetition construct as a whole is executable if and only if there is at least one guard inside the construct is executable.

The execution flow of the repetition $dood$ is similar to the one of the selection $iffi$, except in the repetition $dood$ it is repeated until finds statement $break$. Semantics of the repetition construct $dood$ is described by its reference rules as the following.

If the corresponding sequence $stmt_i$ of the satisfied guard g_i requires more than one step of execution, then:

$$\frac{stmt_i \xrightarrow{h:\alpha} stmt_i' \neq exit}{dood \xrightarrow{g_i \wedge h:\alpha} stmt_i'; dood}.$$

Otherwise, if $stmt_i$ requires more than one step of execution, then:

$$\frac{stmt_i \xrightarrow{h:\alpha} exit}{dood \xrightarrow{g_i \wedge h:\alpha} dood}.$$

If the corresponding sequence $stmt_i$ of the satisfied guard g_i is either equal to or contains $break$ statement, then the control flow will exit from the repetition (see the following inference rule):

$$\frac{break \in stmt_i}{dood \xrightarrow{g_i \wedge h:\alpha} exit}.$$

A program graph for the repetition construct $dood$ is formally generated by determining its components: a set of locations, $Loc = sub(dood) = \{dood, exit\} \cup \{sub(stmt_i) \setminus \{stm', exit\}\}$; a set of actions Act containing an expression, assignment, send or receive; $Loc_0 = \{dood\}$, and g_0 is there at least one satisfied g_i. And the graphical representation of a program graph is shown in Fig. 26.

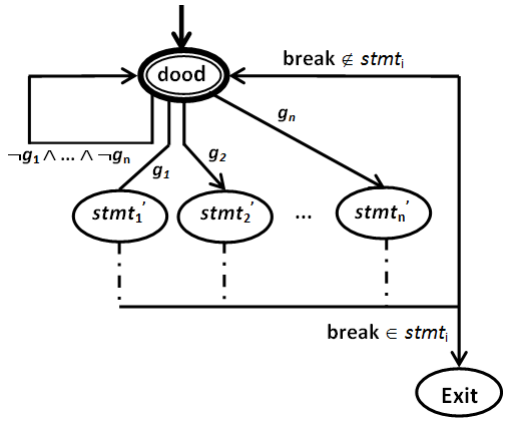

Fig. 26: A program graph for the nondeterministic repetition $dood$

.

When $stmt_i$ is completely executed and finds $break$ statement, then exit from the loop (repetition). Otherwise, it returns to the loop and performs the similar process.

Like a nondeterministic selection construct $if fi$, a nondeterministic repetition construct $dood$ is also implemented in two approaches (*i.e.*, (1) randomize, and (2) priority) in *Java*.

The first version of randomize implementation of $dood$ initially generates a random number $i, 1 \leq i \leq n$ for some positive integer n (*i.e.*, n is number of guards inside repetition construct $dood$). Then the value of i is assigned to a variable $choice$ to select the matching case in $switch$ construct, and get the lock ($header.lock()$) before evaluating the corresponding guard g_i. If the guard g_i holds (or executable), then release the lock ($header.unlock()$) before executing the corresponding sequence $stmt_i$, and if during executing $stmt_i$ finds $break$ statement, the repetition terminates. Otherwise, if g_i does not hold, release the lock and exit from $switch$. Because the value of variable do_flag is unchanged, the program control is still inside the $while$ loop and try to generate another random number j repeatedly until finds a guard g_j for some j, $1 \leq j \leq n$ holds.

The $break$ statement at the end of $stmt_i$ associated with each $case$ inside the $switch$ is ensuring that the flow of control will not fall through to subsequent cases. A *Java* code fragment of randomize implementation version one is shown in Listing 10.

Listing 10: A *Java* code for implementation of $dood\_r_1$

```
...
boolean do_flag=false;
while (!do_flag) {
   int choice=1+Math.random()*nmbr_of_opts;
   switch(choice) {
      case 1:
            header.lock();
            if (!g_1) {
               header.unlock();
               break;
            }
            header.unlock();
            stmt_1;
            do_flag=true; /* a break ? */
            break;
      case 2:
            header.lock();
            if (!g_2) {
               header.unlock();
               break;
            }
            header.unlock();
            stmt_2;
            do_flag=true; /* a break ? */
            break;
      ...
      ...
      case n:
            header.lock();
            if (!g_n) {
               header.unlock();
               break;
            }
            header.unlock();
            stmt_n;
            do_flag=true; /* a break ? */
            break;
   }
}
...
```

The semantics of the *Java* repetition construct using randomize version one is formally explained by the following inference rules. If a randomly selected guard g_i satisfied, and the corresponding action $stmt_i$ is not one-step statement, then there is a transition to location $stmt_i^{'}$:

$$\frac{i \longleftarrow random(1 \ldots n) \wedge stmt_i^{'} \neq exit}{dood\_r_1 \xrightarrow{g_i} stmt_i^{'}}.$$

If a randomly selected guard g_i satisfied, and the corresponding action $stmt_i$ is one-step statement, then there is a transition to location $exit$:

$$\frac{i \longleftarrow random(1 \ldots n) \wedge stmt_i^{'} = exit \wedge break \in stmt}{dood\_r_1 \xrightarrow{g_i} exit}.$$

If a randomly selected guard g_i is not satisfied, the transition return to location $while$:

$$\frac{i \longleftarrow random(1 \ldots n)}{dood\_r_1 \xrightarrow{\neg g_i} dood\_r_1}.$$

A program graph for the $dood\_r_1$ is formally derived by determining its components: a set of locations, $Loc = sub(dood\_r_1) = \{dood\_r_1, switch, exit\} \cup \{sub(stmt_i) \setminus \{exit\}\}$; a set of actions Act containing an expression, assignment and send or receive; $Loc_0 = \{dood\_r_1, switch\}$; and g_0 is guard for selected option g_i's. Then, a graphical representation of the program graph for the randomize repetition construct version one, $dood\_r_1$ is shown in Fig. 27.

The last step is proving $PG(dood) \simeq PG(dood\_r_1)$. Fortunately, the similar argumentations used in proving the equivalence (or similarity) between $PG(iffi)$ and $PG(iffi\_r_1)$ also applies in this case. Hence, it is proven that $PG(dood) \simeq PG(dood\_r_1)$. ■

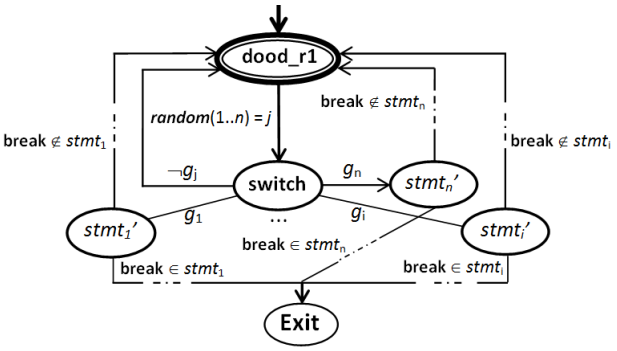

Fig. 27: A program graph for the randomize repetition version one, $dood\_r_1$

In Fig. 27, g_i is the first guard found satisfied in the selection implementation.

The second version of nondeterministic repetition $dood$ implementation works similarly with one for nondeterministic selection $iffi$ version two plus the mechanism of determining whether exit from or return to the loop. A *Java* code fragment for the implementation is depicted in Listing 11.

Listing 11: A *Java* code for the implementation of $dood\_r_2$

```
...
boolean do_flag=false;
while (!do_flag) {
   List<Integer> options=new List();
   header.lock();
   if (g_1) { options.add(1); }
   if (g_2) { options.add(2); }
   ...
   if (g_n) { options.add(n); }
   header.unlock();

   if (options.size()>0) {
     if_flag=true;
     int choice=Math.random()*
                    options.size();
     choice=options.get(choice);
     switch(choice) {
        case 1:
             stmt_1;
             do_flag=true; /* a break ? */
             break;
        case 2:
             stmt_2;
             do_flag=true; /* a break ? */
             break;
        ...
        ...
        case options.size():
             stmt_options.size();
             do_flag=true; /* a break ? */
             break;
     }
   }
}
...
```

The semantics of the randomize implementation for repetition construct $dood$ version two in *Java* is formally depicted by the following inference rules.

To mark $stmt_i$ with the corresponding satisfied guard g_i into a new list variable $options.add(i)$ for $i \geq 1$:

$$\frac{i \leq n}{stmt_j \longleftarrow stmt_i; j++}.$$

If $list_j$ is not a one-step statement, i.e., $list_j^{'} \neq exit$, then:

$$\frac{j \longleftarrow random(1 \ldots m) \wedge stmt_j^{'} \neq exit}{dood\_r_2 \overset{true}{\longleftrightarrow} stmt_j^{'}}.$$

Otherwise, if $stmt_j$ is a one-step statement, i.e., $stmt_j^{'} = exit$, then:

$$\frac{j \longleftarrow random(1 \ldots m) \wedge stmt_j^{'} = exit}{dood\_r_2 \overset{true}{\longleftrightarrow} exit}.$$

A program graph for the $dood\_r_2$ is formally derived by determining its components: a set of locations, $Loc = sub(dood\_r_2) = \{dood\_r_2, for, switch, exit\} \cup \{sub(stmt_i) \setminus \{stm', exit\}\}$; a set of actions Act containing an expression, assignment, send or receive; $Loc_0 = \{dood\_r_2, for, switch\} = \{dood\_r_2^{'}\}$; and g_0 is guard for selected option g_i's. The graphical representation of program graph for construct $dood\_r_2$ is shown in Fig. 28.

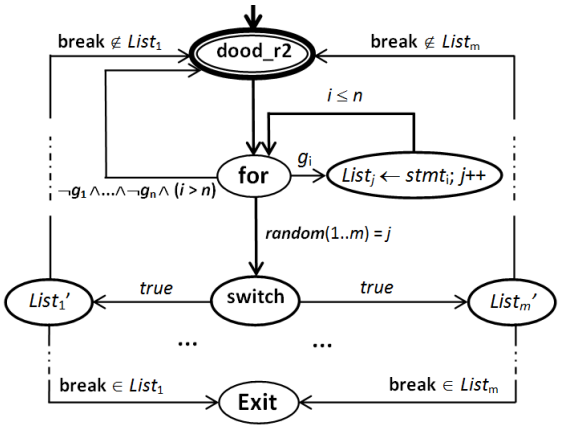

Fig. 28: A program graphs for the randomize version two in *Java*, $dood\_r_2$

The last step is proving $PG(dood) \simeq PG(dood\_r_2)$. To do this, the argumentations used in proving $PG(iffi) \simeq PG(iffi\_r_2)$ will apply again in this case. Hence, it is proved that $PG(dood) \simeq PG(dood\_r_2)$. ■

The third implementation of a nondeterministic repetition construct $dood$ is by using a conditional construct $else\ if$. In this implementation, the flow of execution is similar to one of $esle\ if$ in *Java*, except the process is repeated until one of condition in $else\ if$ (or from guard in $dood$) is executable. Then, a *Java* code fragment of implementation is depicted in Listing 12.

Listing 12: A *Java* code fragment of $dood$ priority implementation

```
...
boolean do_flag=false;
while (!do_flag) {
   int choice=-1;
```

```
header.lock();
if (g_1) {
  choice=1;
} else if (g_2) {
  choice=2; }
...
  else if (g_n) {
  choice=n; }
header.unlock();

switch(choice) {
    case 1:
      stmt_1;
      do_flag=true; /* a break ? */
      break;
    case 2:
      stmt_2;
      do_flag=true;
      break;
    ...
    ...
    case n:
      stmt_n;
      do_flag=true;
      break;
  }
}
...
```

The conditions in $else-if$ construct are sequentially evaluated starting from the first until find one is satisfied, and start to execute a corresponding $stmt$. If, however, there is no condition found satisfied, the evaluation of condition is carried out repeatedly until find a satisfied one.

The semantics of *Java* priority selection construct $dood\_pr$ is formally explained by the following inference rules. If there is a guard g_i for some i satisfied, but $stmt_i^{'} \neq exit$, then:

$$\frac{for\ \ some\ \ i, 1 \leq i \leq n \wedge stmt_i^{'} \neq exit}{dood\_pr \xhookrightarrow{g_i} stmt_i^{'}}.$$

If there is a guard g_i for some i holds, and $stmt_i^{'} = exit$, then:

$$\frac{for\ \ some\ \ i, 1 \leq i \leq n \wedge stmt_i^{'} = exit}{dood\_pr \xhookrightarrow{g_i} exit}.$$

Otherwise, if there is no guard holds, then there is no transition occurs (it is still in the location $dood\_pr$), then:

$$\frac{}{dood\_pr \xhookrightarrow{\neg g_1 \wedge \neg g_2 \wedge \ldots \wedge \neg g_n} dood\_pr}.$$

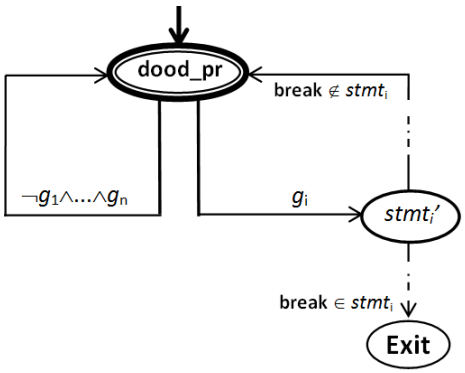

Fig. 29: A program graph for the priority repetition $dood\_pr$

A program graph for the $dood\_pr$ is formally derived by determining its components: a set of locations, $Loc = \{dood\_pr, exit\} \cup \{sub(stmt_i) \setminus \{stm', exit\}\}$; a set of actions Act containing an expresser assignment, send or receive, $Loc_0 = \{dood\_pr\}$, and g_0 is the first satisfied guard g_i's. Program graph for priority repetition ($dood\_pr$) construct is also shown graphically in Fig. 29.

To prove that $PG(dood) \simeq PG(dood\_pr)$ is sufficiently to show that a set of location in $PG(dood\_pr)$ is subset of a set of locations of $PG(dood\_pr)$. From their program graphs definition, Fig.ure 26, and Fig.ure 29, it can be seen that a set of location in $PG(dood\_pr)$ is subset of a set of locations of $PG(dood\_pr)$. Hence, it is proven that $PG(dood) \simeq PG(dood\_pr)$. ■

*9) **Exception Handling**:* In PROMELA, an $unless$ construct defines an exception handling routine. Similar to the repetition and selection constructs, the unless construct is not really a statement, but a method to define the structure of the underlying automaton and to distinguish between higher and lower priority of transitions within a single process. It can appear anywhere a basic PROMELA statement can appear [17].

The $unless$ construct has syntax:

$$\{ms_1, ms_2, \ldots, ms_n\}\textbf{unless}\{es_1, es_2, \ldots, es_m\},$$

in which $\{ms_1, ms_2, \ldots, ms_n\}$ and $\{es_1, es_2, \ldots, es_m\}$ are called $\{main\_sequence\}$ and $\{escape\_sequence\}$ respectively.

The guard of either sequence can be either a single statement, or it can be an $if\,fi$, $dood$, or lower level $unless$ construct with multiple guards and options for execution. The way of how to execute the $unless$ construct can be explained as follows. Anytime to execute statement listed in $main\_sequence$, the guard (first statement in $escape\_sequence$) es_1 will be checked first. If it is executable, the flow of control will move to $escape\_sequence$, and will never return to $main\_sequence$. So that, if the first statement is executable in the time the-ith statement in $main\_sequence$ being executed, starting the ith statement until the last statement in $main\_sequence$ will never be executed.

The semantics of the $unless$ construct is explained by the following inference rules: For some i; $i = 1, 2, ..., n-1, n$:

$$\frac{ms_i^{'} = exit}{ms_i; ms_{i+1}; \ldots; ms_n \xhookrightarrow{\neg es_1} ms_{i+1}; ms_{i+2}; \ldots; ms_n},$$

when es_1 is not executable, and ms_i is one-step execution statement ($ms_i^{'} = exit$), then there is a transition from $ms_i; ms_{i+1}; \ldots; ms_n$ to $ms_{i+1}; ms_{i+2}; \ldots; ms_n$, or:

$$\frac{ms_i^{'} \neq exit}{ms_i; ms_{i+1}; \ldots; ms_n \xhookrightarrow{\neg es_1} ms_i^{'}; ms_{i+1}; \ldots; ms_n},$$

when es_1 is not executable, but ms_i is multi-step execution statement ($ms_i^{'} \neq exit$), then there is a transition from $ms_i; ms_{i+1}; \ldots; ms_n$ to $ms_i^{'}; ms_{i+1}; \ldots; ms_n$, or:

$$\frac{}{ms_i; ms_{i+1}; \ldots; ms_n \xhookrightarrow{es_1} es_1; es_2; \ldots; es_m},$$

when es_1 is executable, the control flow moves to $es_1; es_2; \ldots; es_m$ and never comes back.

A program graph for the $unless$ construct is derived by determining its components: a set of locations, $Loc = (unless) = \{unless, exit\} \cup \{sub(stmt_i) \setminus \{exit\}\}$; a set of actions, Act containing an expression, assignment, send or receive; a set of initial locations, $Loc_0 = \{unless\}$; and an initial condition, g_0 is es_1. Then, the graphical representation of the program graph is shown in Fig. 30.

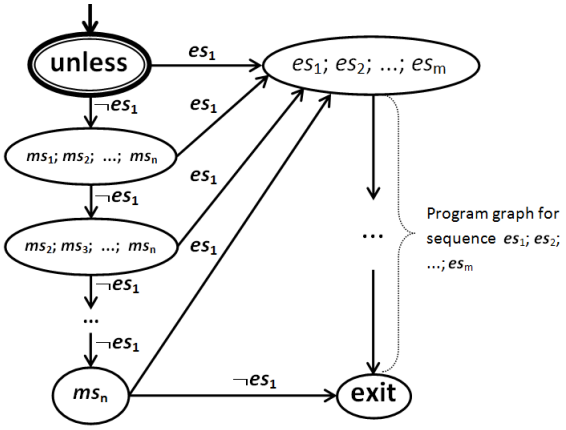

Fig. 30: A program graph for the $unless$ construct

Based on the $unless$ construct's flow of execution, and there is no direct corresponding construct in *Java*, then the $unless$ construct is implemented by $try - catch$ construct. In order to simulate the $unless$ construct's flow of execution, a *Java* code fragment of this implementation is described in Listing 13.

Listing 13: A *Java* code fragment of $try - catch$ implementation for $unless$

```
...
try {
      if (es_1 != 0) throw New Exception;
        ms_1;
      if (es_1 != 0) throw New Exception;
        ms_2;
      ...
      if (es_1 != 0) throw New Exception;
        ms_n;
    }
catch(Exception ex) {es_1; es_2; ...; es_m}
...
```

Each time ms_i, $i \in \{1, 2, ..., n\}$ is being executed, es_1 must be checked first. If es_1 is executable, the flow of control moves to the $catch$ section. Once it reaches the $catch$ section, it will never return. This means, it starts executing all statements in the $catch$'s section. In another word, all of $ms_1, ms_2, \ldots, ms_n$ will be executed when es_1 is never executable during the execution of $ms_1, ms_2, \ldots, ms_n$.

The flow of control of $try-catch$ (*i.e.*, implementation of $unless$) is depicted by its program graph shown in Fig. 31.

The semantics of the $try - catch$ construct is explained by the following inference rules. If for some i, $i = 1, 2, ..., n-1, n$, and ms_i is one-step statement and es_1 is still not executable, there is an intermediate transition from $ms_i; ms_{i+1}; \ldots; ms_n$ to $ms_{i+1}; ms_{i+2}; \ldots; ms_n$, then:

$$\frac{ms_i^{'} = exit}{try\{ms_i; \ldots; ms_n\} \xrightarrow{\neg es_1} try\{ms_{i+1}; ms_{i+2}; \ldots; ms_n\}}.$$

If for some i, ms_i is multi-step statement and es_1 is still not executable, an intermediate transition is from $ms_i; ms_{i+1}; \ldots; ms_n$ to $ms_i^{'}; ms_{i+1}; \ldots; ms_n$, then:

$$\frac{ms_i^{'} \neq exit}{try\{ms_i; \ldots; ms_n\} \xrightarrow{\neg es_1} try\{ms_i^{'}; ms_{i+1}; \ldots; ms_n\}}.$$

Otherwise, once es_1 executable, the transition occurs from the location $ms_i; ms_{i+1}; \ldots; ms_n$ to location $es_1; es_2; \ldots; es_m$, and:

$$\frac{}{try\{ms_i; ms_{i+1}; \ldots; ms_n\} \xrightarrow{es_1} es_1; es_2; \ldots; es_m}.$$

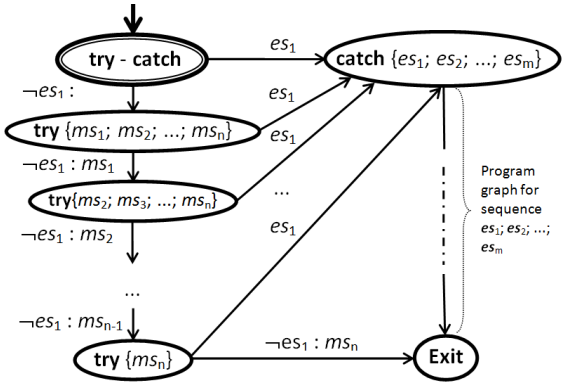

Fig. 31: A program graph for the $try - catch$'s construct in *Java*

The last step is proving that $PG(unless) \simeq PG(try - catch)$. Based on Fig. 30, and Fig. 31, there must be two bijective functions f, and g such that $PG(unless) \simeq PG(try - catch)$. ■

IV. The P2J Translator Tool

The work of proving the association's correctness is a part of the code translator tool development. This code translator tool translates a model of PROMELA to a *Java* program. The correctness of the association will guarantee the preserving semantics. The developed translator is named by P2J stand for PROMELA to *Java*. It is developed based on the association construct CA defined from a subset of constructs of PROMELA and a subset of constructs of *Java*.

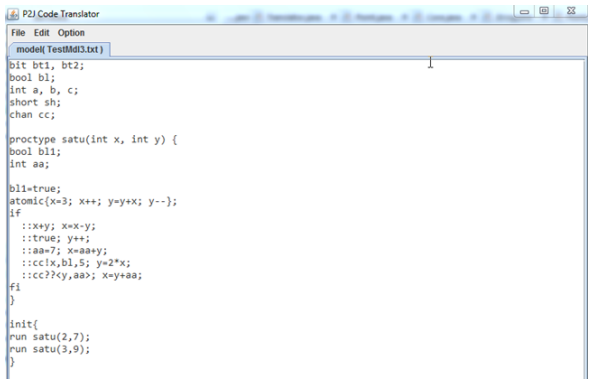

Fig. 32: An example of PROMELA model to be translated

Fig. 32 shows a simple example of model consisting of the declaration of global variables including channel, and two processes (*proctype*), i.e. $satu$ and $init$. The process $satu$ is

composed by some elementary constructs, i.e., *expression*, *assignment*, *atomic*, and nondeterministic selection $iffi$. And the *init* process consists of two statements *run*.

The model in Fig. 32 is translated to five *Java* classes—$HeaderInterface$ (stored in a Library), $HeaderImpl$ (header implementation), $ChannelLauncher$ (to run class channel), *satu* and *init* (see Fig. 33).

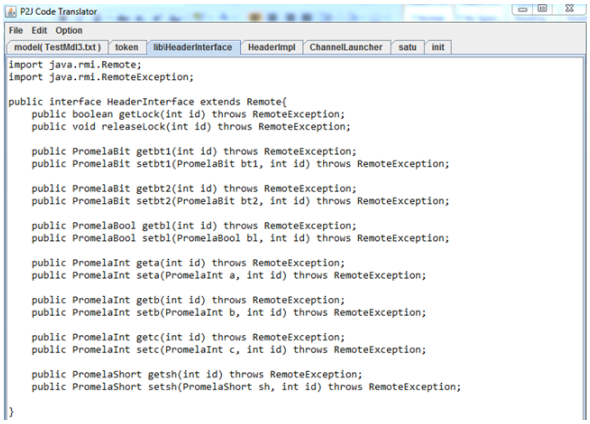

Fig. 33: The list of classes as the result of translation

These classes correspond to respectively the global declaration variables, channel declaration, and two processes *satu* and *init* of the translated PROMELA model. The result of this translation is depicted by their class definitions as shown in Fig. 33, 34, 35, 36 and 37, respectively.

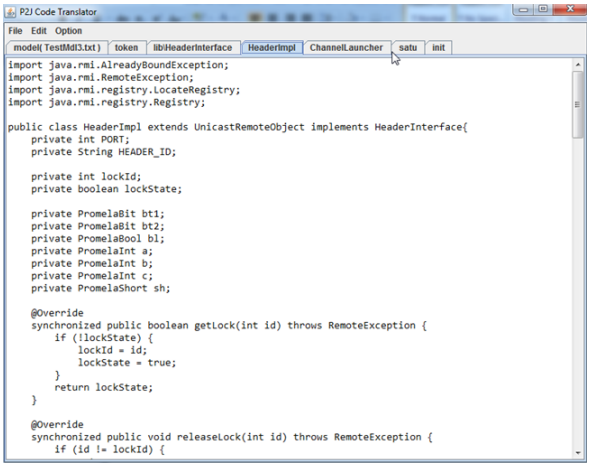

Fig. 34: The class definition of $HeaderImpl$

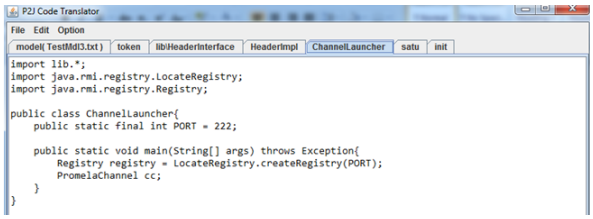

Fig. 35: The class definition of $ChannelLauncher$

Channel is run similarly as header, so that the method $main()$ not only runs header itself but also runs the channels in the global declaration. This will work provided all channels reside in the same machine as header. In case the channels are separated in different machines from header, each machine needs launcher for channel.

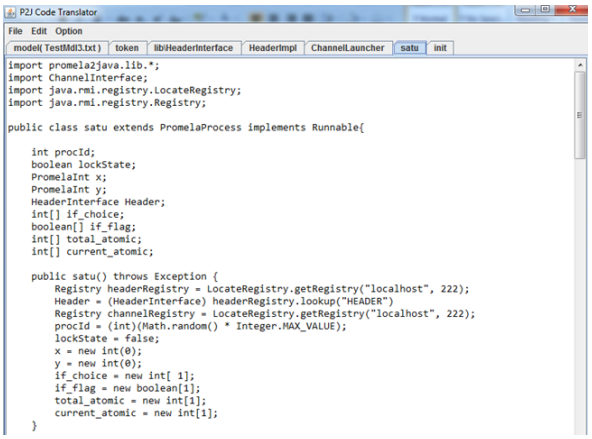

Fig. 36: The class definition of *satu*

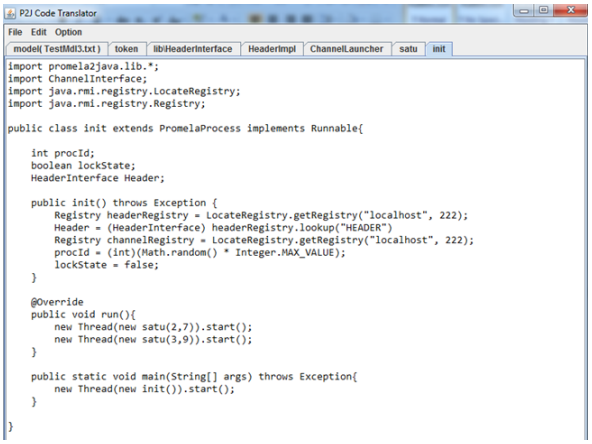

Fig. 37: The class definition of *init*

V. Conclusion

This paper has proved the correctness of association from a subset of PROMELA constructs to a subset of *Java* constructs. The association's correctness is proved by showing the equivalence (or similarity) of program graph for every pair of associated constructs in the association. The similarity of two program graphs is determined by the equivalence of corresponding locations between the two program graphs. The number of locations of program graph for any construct corresponds to the number of elements in the substatement of construct (or statement).

Several pairs of associations have exactly the same program graphs, and some do not. For those pairs that have exactly the same program graphs, their equivalence can be shown in a straightforward manner. On the other hand, those that do not have exactly the same program graphs need some additional steps to equalize their number of locations. This is done by coalescing *ignorable* locations together with the previous closest location into one new location.

The different number of locations in program graph for *Java* constructs are caused by two reasons: (i) most constructs in PROMELA do not have direct corresponding constructs in *Java*; hence combining more than one construct is required, and yields additional locations, and (ii)

PROMELA models need verification (or simulation), while *Java* programs need execution. In verification, all possible options (such as in selection and repetition) will be taken, while in execution only one is taken.

The last section has shown the implementation of the defined construct association in developing a source-to-source code translator from PROMELA models to *Java* programs, P2J Code Translator that preserves semantics.

REFERENCES

[1] C. Baier, and J. P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, Massachussets, London England, 2008.

[2] C. Demartini, R. Iosif, and R. Sisto. *Modeling and Validation of Java Multithreading Applications using SPIN*. Dipartimento di Automatica e Informatica, Politecnico di Torino corso Duca degli Abruzzi 24, 10129 Torino (Italy).

[3] C. Pronk. *Promela to Java - Automatic translation*. Slides of TU Delft course IN4023: Advanced Software Engineering, 2007. Adapted from slides of Twente by T.C.Ruys.

[4] C. Wimberger. *Source to Source Translator from C# to Java and Action Script*. Journal of Kepler University Linz (KJU), 2008.

[5] D. A. Plaisted. *Source-to-Source Translation and Software Engineering*. Journal of Software Engineering and Applications, 2013, 6, 30-40.

[6] E. Vielvoije. *Promela to Java, Using an MDA Approach*. Thesis, 2008.

[7] Gerald J. Holzmann. *The Model Checker SPIN*. IEEE Transactions on Software Engineering, 23(5):17, 1997.

[8] Gerald J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. ISBN 0 321 228628.

[9] I. Schaefer. *Software Engineering Using Formal Methods - Introduction to PROMELA*. Institute for Software Systems Engineering, TU Braunschweig, Germany.

[10] J. Magee, and J. Kramer. *Concurrency State Models Java Programming*. John Wiley & Sons, Ltd, 2006.

[11] K. Jiang. *Model Checking C Programs by Translating C to PROMELA*, 2009.

[12] L. Aceto, A. Ingolfsdottir, K. G. Larsen, and J. Srba. *Reactive Systems - Modeling, Specification, and Verification*. Cambridge University Press, 2007. ISBN 978-0-521-87546-2.

[13] M. Pawlan. *Essentials of the Java Programming Language*. A Hands-On Guide. Sun Microsystems, Inc. All right reserved.

[14] Sun Microsystems. *The Source for Java Developers*. The Java Homepage. http://java.sun.com/.

[15] Sun Microsystems. *Java Tutorial*. http://www.tutorialspoint.com/. Copyright ©tutorialspoint.com/.

[16] Suprapto, R. Wardoyo, R. Pulungan, and B. Wijaya *A Scheme of Construct Association from PROMELA model to Java program*. Proc. 4th FTRA, 2013, p. 11-12.

[17] T. Ruys. *A Tutorial Introduction to SPIN*. SPIN On-line References. The Homepage: spinroot.com/spin/Man/

[18] W. Nabialek, A. Janowska, and P. Janowski. *Translation of Timed Promela to Timed Automata with Discrete Data*. John Wiley Sons, Ltd, 2008.