Internship at Kansas State University

February-August 2001

Adviser : David Schmidt

Élodie-Jane Sims.

M.M.F.A.I.

Chapter 1

Introduction

Martin Rinard is doing pointer analysis which transform a program in an object language into a double labeled graphs. Then with this kind of graph he wants to check some correctness properties.

Daniel Jackson is working in a language called Alloy in which we can define a model and then check some property of the model.

Our global goal was to code Rinard's graphs in Alloy, to translate some linear-logic syntaxes into Alloy and then check property of the graph. So our first work was with Alloy but we didn't reach all our goals and we have just translated CTL into Alloy. This first work is presented in chapter 2. Then we have studied Martin Rinard's pointer analysis and this work is presented in chapter 3.

Chapter 2

Alloy

2.1 Introduction

We have been considering ways of representing state-transition systems (STSs) in Alloy and checking the systems for temporal-logic-like properties. The motivation for this work is that Martin Rinard's shape graphs are a form of STS, and the forms of correctness properties one wants to deduce on such systems are often neatly written as temporal-logic formulas.

We have writing the STSs and coding temporal logic in Alloy (a language presented in section 2.2). The STSs are stated rather reasonably, but there have been some interesting challenges in writing the temporal logics and their semantics: CTL formulas can be coded more or less in a straightforward fashion, but modal mu-calculus has proved problematic. After some struggles, we noted that the semantics of Alloy's some operator was influencing the effort—we were trying to define the notion of "least-fixed point" in Alloy, and we found that the usual set-theoretic definition (that a least-fixed point of a functional is the intersection of all the functional's prefixed points) could not be written in a natural way. We have writing a letter to Daniel Jackson which explain our work and problem, it is presented in section 2.4 and his answer is presented in section 2.5.

2.2 Presentation of Alloy

Alloy is a language with a tool for analyzing object models. It has three key ingredients: a language of constraints, much like first-order logic with sets and relations, but with special forms for declarations; some structuring mechanisms for organizing constraints so they can be used to describe the static and dynamic properties of systems; and a simple typing scheme.

Alloy model contains the notions of set, quantifier, transitive closure and relation.

The Annexe G show an example of an Alloy file with all the feature of Alloy, the syntaxe is pretty intuitive.

Short description of Alloy syntaxe

- define some primitive sets : `domain`
- define some subsets in `state` with some key words like `partition, static, !, +, *`
- define relations `where :  Equipment -> Location!`
- define indexed relations `adj[Place]:  Location -> Location`
- express constraints formula with quantifier `all, some, sole, one, no`

The main utility of Alloy is to describe a system and then to check some property of this system.
The main problem is that Alloy is not a decidable language and the answer it give are always in a finite *scope*, this come from the fact that first-order logic is undecidable.
An other problem is that it translates the problem to be analyzed into a usually huge boolean formula and it can't analyze model in big scopes.

I have learn Alloy in the course *Software Specification (CIS 771)* : `http://www.cis.ksu.edu/ hatcliff/771/`.

There is a general description of Alloy in [1].

Memo of [1] for myself : *Alloy is relational, declarative. assertions. Alloy can check that an operation preserves an invariant, that one operation refines another, or that one invariant implies a second. invariant. operations. not a decidable language. finite scope. output is an instance. no instance was found or counterexample. Alcoa is essentially a compiler. translates the problem to be analyzed into a boolean formula. SAT solver. ...can, unlike Alcoa, investigate elaborate temporal properties. Alcoa's counterexample may be a transition from an unreachable state. modular can analyze partial descriptions. a theorem 'refuter'. for everyday use. include quantifiers. implicit frame conditions. graphical display of instances*

There is a detailed description of Alloy in [4].

Memo of [4] for myself : *domain. fixed. state. static. definition. invariants. in. no. some. one. sole. condition. assertions. operations. simulation. checking. set. relation. function. unit (an element is (e, unit) a set is a set of (e, unit), there is no set of set.). navigation expression if a = {(e1, f1); (e2, f2)} ˜ a= {(f1, e1); (f2, e2)} and a relation from {(e1, unit); (e2, unit)} to {(f1, unit); (f2, unit)} is type of {(e1, f1); (e1, f2), (e2, f2)}. counterexample. scope. subsequent. fixed. static. indexed relation.*

In the paper [5] we can find how Alloy tool transform the Alloy file in a boolean formula for SAT.

Memo of [5] for myself : *First order logic is undecidable. check consistency of a formula or check the validity of a theorem. checking that one constraint follows from another, that one operation refines another, that an operation preserves an invariant. The meaning of a problem is the collection of well-formed environments in which its formula evaluates to true. Normalization of the relational formula, morgan's laws. conjunctive normal form.*

2.3 Linear logic

In this section, we recall some definitions.

Definition 1 *A state-transition system* **STS** *is a triple,* $< \Sigma, \rightarrow, l >$ *such that :*
- Σ *is a finite set of* states
- $\rightarrow \subseteq \Sigma \times \Sigma$ *is a* transition relation
- $l : \Sigma \rightarrow AProp$ *maps each* $s \in \Sigma$ *to the set of atomic propositions that hold true for it*

Definition 2 *Syntax of* **computational-tree logic** *:*

$$\phi ::= p \mid \neg\ \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid AX\phi \mid EX\phi \mid AG\phi \mid EF\phi \mid A[\phi_1 U \phi_2] \mid E[\phi_1 U \phi_2] \mid AF\phi \mid EG\phi$$

Definition 3 *A modal transition system* **MTS***is a tuple,* $< \Sigma, \mathtt{Act}, \mathtt{AProp}, \rightarrow, L >$ *such that :*
- Σ *us a finite set of* states
- $\mathtt{Act}$ *is a finite set of action symbols*
- $\mathtt{AProp}$ *is a finite set of atomic propositions*
- $\rightarrow \subseteq \Sigma \times \mathtt{Act} \times \Sigma$ *is a* transition relation
- $L : \Sigma \rightarrow AProp$ *is a* labeling function

Definition 4 *Syntax of* $\mu - calculus$ *:*

$$\phi ::= \top \mid \bot \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

Definition 5 *Syntax of* **linear-time logic** *:*

$$\phi ::= p \mid \neg\ \phi \mid \phi_1 \vee \phi_2 \mid X(\phi) \mid U(\phi_1, \phi_2) \mid F(\phi) \mid G(\phi)$$

In the paper [9] we can find some definition of the semantic of some linear-logic that we have used and a description of the automata-approach that we have though to implement in Alloy for checking LTL formulas. We have abandoned the idea of checking LTL's formulas in Alloy because it appears that the only way for it would have been the automaton method and Alloy was not interesting for this since it has limited scopes. The try is in Annexe H.

The book [3] was useful for linear-logic definitions and examples.

The paper [11] compare CTL to LTL.

Memo of [11] for myself :
- *Neither of CTL or LTL is included in the other*
- *CTL hard for programmers*
- *CTL $O(nm)$, n size of system, m size of formula*
- *LTL $O(n2^m)$*
- *BUT if we check for properties that can be expressed in both the superiority of CTL disappears.*
- *CTL is not adequate for modular verification*
- *LTL advantages : expressiveness, compositionality, semi-formal verification (c.f. $\simeq$ check emptiness), property-specific abstraction, combined methods, uniformity, bounded model checking*

There are classical examples of the "incompatibility" of CTL and LTL :
- the LTL formula $FG\ p$ is not expressible in CTL
- the CTL formula $AFAG\ p$ is not expressible in LTL.

2.4 Letter to Daniel Jackson

This section transcript a letter we have send to Daniel Jackson to explain our try of translating $\mu - calculus$ to Alloy.

This note lists several approaches at expressing the semantics of the modal μ-calculus in Alloy. In each of the approaches, difficulties are encountered when attempting to write the semantics of the least-fixed point operator.

Please recall that the least-fixed point operator is used to write recursive propositions. For example, the CTL construction, $\mathsf{EU}(\phi_1, \phi_2)$, can be coded in mu-calculus as

$$\mu Z.\phi_2 \vee (\phi_1 \wedge \Diamond Z)$$

When a finite-state transition system is checked for such recursive properties, there is simple definition of the semantics of μ:

$$s \models \mu Z.\phi \text{ iff there exists } i \geq 0 \text{ such that } s \models \phi_i, \text{ where } \begin{cases} \phi_0 = \mathit{false} \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$$

It would be nice to encode this semantics in Alloy, but this is not so easy. (See Approach 3 below.) In any case, the standard definition of μ is written with a semantic mapping, M : Proposition $\rightarrow$ $\mathit{Environment} \rightarrow \mathcal{P}(\mathit{State})$, such that $M(\phi)\rho$ maps ϕ to the set of states for which ϕ holds true. (Of course, ρ : Variable $\rightarrow \mathcal{P}(\mathit{State})$, gives the meaning of free variables in ϕ.)

Here is the "standard" semantics of μ:

$$M(\mu X \cdot \phi)_\rho = \cap\{x \subseteq \mathit{State} \mid M(\phi)_{\rho[X \mapsto x]} \subseteq x\}$$

That is, the least-fixed point is the intersection of all pre-fixed points, x.

The coding of μ in this style is attempted in Approach 1, below. (The modelling of a modular state-transition system (MTS) is in Annexe E.)

First approach: Annexe C

The difficulty comes from the fix-points.

To write this translation we define the operators μ and ν :if
- S is the set of states
- ρ is the environment
- $M(\phi)_\rho$ is the subset of states that satisfy ϕ in the environment ρ

then :

$$M(\mu X \cdot \phi)_\rho = \cap\{x \subseteq S \mid M(\phi)_{\rho[X \rightarrow x]} \subseteq x\}$$

$$M(\nu X \cdot \phi)_\rho = \cup\{x \subseteq S \mid M(\phi)_{\rho[X \to x]} \supseteq x\}$$

As usual, we write $s \models \phi$ as the predicate that holds true exactly when $s \in M(\phi)_\rho$.

Now, how might we express $s \in M(\mu X \cdot \phi)_\rho$ (similarly, ν)?

The first approach was just a manual translation into Alloy of the above definitions: $s \in M(\mu X \cdot \phi)_\rho$ translates to this predicate:

$$[s \models \mu X \cdot \phi] = all\, x \mid s\, in\, x \mid\mid \{s2 \mid [s2 \models \phi](X.rho/x)\}\, !in\, x$$

And similarly, $s \in M(\nu X \cdot \phi)_\rho$ might be translated as

$$[s \models \nu X \cdot \phi] = some\, x \mid s\, in\, x\, \&\&\, x\, in\, \{s2 \mid [s2 \models \phi](X.rho/x)\}$$

This translation would have been correct if the semantics of *some x* would allow x to be any set instead of being only a singleton set.

To understand this point, consider the the semantics of Alloy in the paper *Alloy:A Lightweight Object Modelling Notation* [4]. On Page 8 we have this semantics of the construction, $some\, v : t \mid F$:

$$X[some\, v : t \mid F]e = \vee\{M[F](e \oplus v \mapsto x) \mid (x, unit) \in e(t)\}$$

To make the typing exactly correct, we can restate the above to read :

$$X[some\, v : t \mid F]e = \vee\{M[F](e \oplus v \mapsto \{(x, unit)\}) \mid (x, unit) \in e(t)\}$$

Because in Alloy everything is a set, one might claim that this semantics is a bit counterintuitive. Indeed, we were expecting to see the following as the semantics of "some" in Alloy :

$$X[some\, v : t \mid F]e = \vee\{M[F](e \oplus v \mapsto x) \mid x \subseteq e(t)\}$$

This alternative semantics would have make our translation of the fixed-point operators in μ-calculus correct. (There is a similar issue with the semantics of "all".)

Is there some aspect of the semantics that we are misunderstanding? Or, for that matter, is there a neater way to represent the two definitions of least and greatest fixed point portrayed earlier? Finally, it might be helpful to us to understand the motivations and rationale for the current semantics of "some".

Second approach: Annexe D

Because the semantics of *some x* lets us discuss only singleton sets and not arbitrary sets, we tried next to circumvent the situation by manually defining relations that define the least and greatest fixed points. Each relation is defined from a state to a set of states, giving us the set we are searching for. Then, to check for the fixed-point property, we just check if the set is empty (there is no such state) or not.

The implementation is:

```
domain {State,...}
state {
...
Rel_nu_phi : State -> State
}
def Rel_nu_phi{
all s1 | s1.Rel_nu_phi = {s2 | s1 in s1.Rel_nu_phi &&
([s2 satisfy phi](X.rho/s1.Rel_nu_phi))}
}
```

and

`s in` $M(\nu X \cdot \phi)_\rho \implies$ `some s.Rel_nu_phi`

This attempt doesn't work, because nothing forces `s1.Rel_nu_phi` to be nonempty whenever possible. Is there some way to force the relation to be a nonempty set?

Third approach

Next, we tried to use the transitive closure operator of Alloy: $*$. In Alloy, if $R : S_i \to S_{i+1}$ then $S_0 \cdot *R = \cup S_i$.

But if we take this definition of μ :

$$M(\mu X \cdot \phi)_\rho = \cup\{M(\mu X \cdot \phi)_{\rho[X \to \emptyset]^i} \mid i \geq 0\}$$

there is a problem at the outset, because the $*$ operator doesn't take an empty set to start—we can't see how to define the relation so that it uses transitive closure to compute (and union) the successive approximations to the least fixed point.

About the "definition" construction in Alloy

In the paper *Alloy:A Lightweight Object Modelling Notation* [4], there is no semantics given for a definition. The Alloy verifier seems to treat definitions like invariants.

Is there no semantics for the definition construct? This is not an idle question—here's why:

This naive definition of a "set" is paradoxical:

$$P = \{d \in D \mid d \notin P\}$$

because, of course, for each $d \in D$, we have $d \in P$ iff $d \notin P$. Conventional set theory would not admit the definition, because it is nonwell-founded (P is defined in terms of itself), but we can insert the definition of P into Alloy as follows:

```
model basic{
domain {D}
state {
P : D -> D
}
def P {all d | d.P = {d2| d2 !in d.P}}
}// end model
```

When the definition is executed by the Alloy checker, the definition seems to be treated just like this invariant :

```
model basic{
domain {D}
state {
P : D -> D
}
inv unnamed_invariant {all d | d.P = {d2 | d2 !in d.P}}
}// end model
```

In the second example, the semantics seems to be clear: the subformula `{d2| d2 !in d.P }` has a value, and we expect that Alloy's tool will find no instance of the model.

But in the first example, if we think that we examined all `d` in `D` one after one and defined for each the value of `d.P`, then `{d2| d2 !in d.P}` should have a value. But it doesn't, because we are defining a model and not checking an assertion.

The underlying problem is the recursion : there is not a unique solution of the definition, and we don't know (and cannot choose) which solution Alloy's tool calculates. In contrast, the set defined by the invariant in the second example appears to have an unambiguous value.

In summary, if we allow recursion within a definition, it would be nice to say something like "the biggest set such as". There might already be a way to state this, but we are unaware how.

Translation from CTL to Alloy

Because it is less expressive, CTL can be coded in Alloy. A state transition system can be axiomatized as in Annexe B, and the coding of CTL is seen in Annexe A.

2.5 Answer of Daniel Jackson

here are some initial reactions to the note you sent me. i haven't had as much time as i'd like to really think about encoding modal mu calculus in Alloy, but i thought it would be better to send some rather general responses anyway, and then perhaps work on it more when i understand your problem better.

1. first, i don't understand properly what you're trying to achieve with Alloy. writing your theory in a form that allows some of its theorems to be automatically checked? investigating the limitations of Alloy in relation to these calculi?

2. sorry about the gaff in the definition of quantification in the Alloy paper. yes, your restatement is exactly right.

3. the reason that quantifiers bind a singleton in Alloy, in apparent contradiction to the general set-based viewpoint of the language as a whole, is that i wanted the language to remain first order. in this form, the quantifiers are completely standard first order quantifiers, and the set and relation operators (with the exception of closure) can be defined away. allowing quantification over a variable bound to a set would introduce a second order element to the language, and would make analysis much less tractable. in a scope of 4, with 4 variables quantified, that gives 64 instantiations of the formula. if the variables were bound to sets instead, we'd have 2^16 instantiations.

4. that said, we decided to include higher order quantifiers in the new version of the language. our intent was to support formulas in which the quantifier would be skolemized away prior to analysis. for example, to express the theorem that union is commutative, you'd write

```
        assert {all s, t: set X | s + t = t + s}
```

in the new language. the tool negates this and skolemizes, giving the first order formula

```
        s + t = t + s
```

in which s and t are constants to be determined. so you can now write

```
        some s: set X | f
```

etc, but we're not promising that the tool will always analyze it.

5. the problem with defining a least fixed point is not a surprise, because you really need a second order constraint. however, when we've wanted to minimize over a set (as in sarfraz's specification of intentional naming), we've written it rather inelegantly in first order form. here's an example in the new language:

```
        fun lfp (m: S -> S, x: set S) {
          x.m = x
          all e: x | (x-e).m != x-e
          }
```

the keyword "fun" introduces a formula (more generally a function -- but for now just view it as a formula). it constrains a relation m from S to S and a set x. we constrain x to be a fixed point of m, and also to be minimal: that is, if you take an element away, you no longer have a fixed point.

6. the problem with using transitive closure is, again, the issue of trying to express something second order. +r, the transitive closure of a relation r: S -> S, is not the same as the transitive closure of the function R: (set S) -> (set S) that maps sets to sets, and which cannot be declared in Alloy because it's not first order. unfortunately, i think it's this function that you want. you may be able to encode it in Alloy by the following trick. declare a relation that maps predicates to states:

```
        states: Pred -> State
```

and define the relation over predicates

```
        R: Pred -> Pred
```

now write a constraint that canonicalizes the predicates, so that two predicates with the same states must be equal:

```
        all p1, p2: Pred | p1.states = p2.states => p1 = p2
```

and then add some axioms that constrain R so that it's monotonic over states, and so on. i'm not sure if this will work. it's certainly tedious, but perhaps it can be done once and the details hidden.

7. about definitions. in Alloy, a definition differs from an invariant not in its mathematical meaning but in how it is used. this is much like the distinction Larch makes between "imports" and "assumes". the motivation for this was two fold. first, to recognize an important methodological distinction that is too often not recorded: this issue is discussed at length in my father's work; see for example the topics "definitions" and "designations" in Software Requirements and Specifications, Addison, 1995. second, to exploit the distinction in the automatic analysis. we intended to build checks that would ensure that a definition of x (1) gave at least one value of x for any combination of values of the other variables (ie, totality) ; and (2) gave at most one such value (determinism). we never got round to implementing the check. the russell's paradox example would fail check (1), since the formula

```
        s = {x: X | x !in s}
```

always evaluates to false. the checker does in fact exploit the marking of a formula as a definition in several ways. first, it uses the distinction in generating the formulas for checking invariant preservation, so that you get something like

```
        inv (s) && def (s) && op (s, s') && def (s') => inv(s')
```

in which the definitions are always included in the post-state. second, when it performs a dependence analysis to see which formulas to include, if a certain user-determined flag is set, it will take on faith that the definitions are well formed, and will only include a definition of x if there is a mention of x itself in the formula to be analyzed (or in some formula it depends on). in contrast, an invariant is included if any of its variables are mentioned, since it is assumed that an invariant, unlike a definition, can constraint all its variables.

i don't understand the issue in the mts2 example. it's a bit tricky to read formulas that mix Alloy and denotational semantics. perhaps we should look at a less general problem first, such as expressing the reachable states of a machine using the least fixed point of a predicate transformer?

there's a paper on the new version of the language at

http://sdg.lcs.mit.edu/~dnj/publications.html#micromodularity

comments v welcome! it should be more suitable for what you're trying to do, as it's simpler and more general.

2.6 Conclusion

We have been able to translate CTL into Alloy but $mu - calculus$ and LTL were to expressive for the version of Alloy we had. One of our objectif to identify some subset of Alloy or Alloy notational patterns that are temporal-logic like and can be used to check typical properties of STSs has also been abandoned since it appeared that to express some simple temporal-logic property in Alloy we need to use several pattern. Even if the results of our work didn't reach our first goals this work has offered to me the possibility to learn about temporal-logic and became more familiar with the meaning of CTL, LTL and $mu - calculus$'s syntaxes.

Chapter 3

Compositional pointer analysis

3.1 Introduction

We have studied the paper *"Compositional Pointer and Escape Analysis for Java Programs"* of John Whaley and Martin Rinard [13]. Pointer analysis build an abstraction of the objects and fields of a program in an object-oriented language. This abstraction is called a points-to graph. The algorithm of the paper build a points-to escape graph. It contains escape information which characterizes how objects allocated in one region of the program can escape to be accessed by another region.

The principal interesting property of their algorithm is that it is a compositional analysis - it analyses each method independently of its caller and can integrate or not the informations about the methods it invokes. The main difficulty is when we want to integrate the result of the analysis of a method in the analysis of an other method which invokes it. This is the point we have looked at, it is presented in the section 3.3.

3.2 Definition of the points-to escape graph

Definition of the points-to escape graph The algorithm consist in the construction of a points-to escape graph for each method. The nodes represent objects; the edges represent references between objects or between a variable and the object it represent.
There are two kind of edges :

- *Inside edges* : represent references created inside the currently analyzed method
- *Outside edges* : represent references created outside the currently analyzed method

There are three kind of nodes :

- *Inside nodes* N_I : represent objects created inside the currently analyzed region and accessed from inside edges
- *Outsides nodes* N_O : represent objects that are either created outside the currently analyzed region or accessed via outside edges
- *Return nodes* N_R : represent the return value of a skipped method invocation site

The graph keep also the escape information - for each node it says if the object it represents can *escape*. An object can *escape* if it is:

- reachable from a parameter or is the result value of the currently analyzed method
- reachable from a parameter or the result value of an invoked method, and we don't know what the invoked method do with it

- reachable from a static class variable or a runnable object

An object is *captured* if it does not *escape*.

Definition of the abstraction The points-to escape graph is an abstraction of the objects and references created during the execution of the program. A single concrete object in the execution of the program may be represented by multiple nodes in the points-to escape graph. This abstraction insure that the absence of edges between captured nodes guarantees the absence of references between the represented objects, and that captured objects are inaccessible when the method returns.

The statements The algorithm represents the computation of each method using a control flow graph. The nodes of the control flow graphs are statements of the form *(the algorithm assumes the program has been preprocessed to have only this statements)*:
- l = v
- l_1 = l_2.f
- l_1.f = l_2
- return l
- l = new cl
- l = l_0.op(l_1,..., l_k) : *method invocation site*

Where l, v, l_k are variables, f is a field, op is a method and cl is a class.

The general algorithm The algorithm starts by building an initial points-to graph with :
- a node for each parameter (call parameter-node)
- an inside edge from each parameter to its corresponding parameter-node
- no outside edge
- no return node

Then at each point of the control flow graph, the algorithm join the points-to graphs of the predecessors points and modify this joined points-to graph. The result is the points-to graph at the end of the control flow graph.
For the statement of type l = l_0.op(l_1,..., l_k) the modification is explain in section 3.3.
For the other statement the algorithm do what we could expect, its effects are represented in Figure 3.1 and the details are in the paper[13].

3.3 Analysis of the mapping rules

To proceed the statement of type l = l_0.op(l_1,..., l_k) the first way is to skip the call by :
- creating a new return node in N_R for this call
- removing all inside edges from l
- adding an inside edge from l to the new return node

If we want to integrate the informations from the called method, the algorithm use a mapping described in section 3.3.1.

3.3.1 Description of the mapping rules

The mapping algorithm take :
- the graph of the caller method with the escape function and with the call skipped : $< O, I, e, r >$, O are the outside edges, I are the inside edges, e is the escape function, r the set of nodes that represent objects that may be returned by the method
- the graph of the callee with the escape function : $< O_R, I_R, e_R, r_R >$

and it build a mapping from the nodes of the callee graph to the nodes of the caller, and build a new graph $< O_M, I_M, e_M, r_M >$.

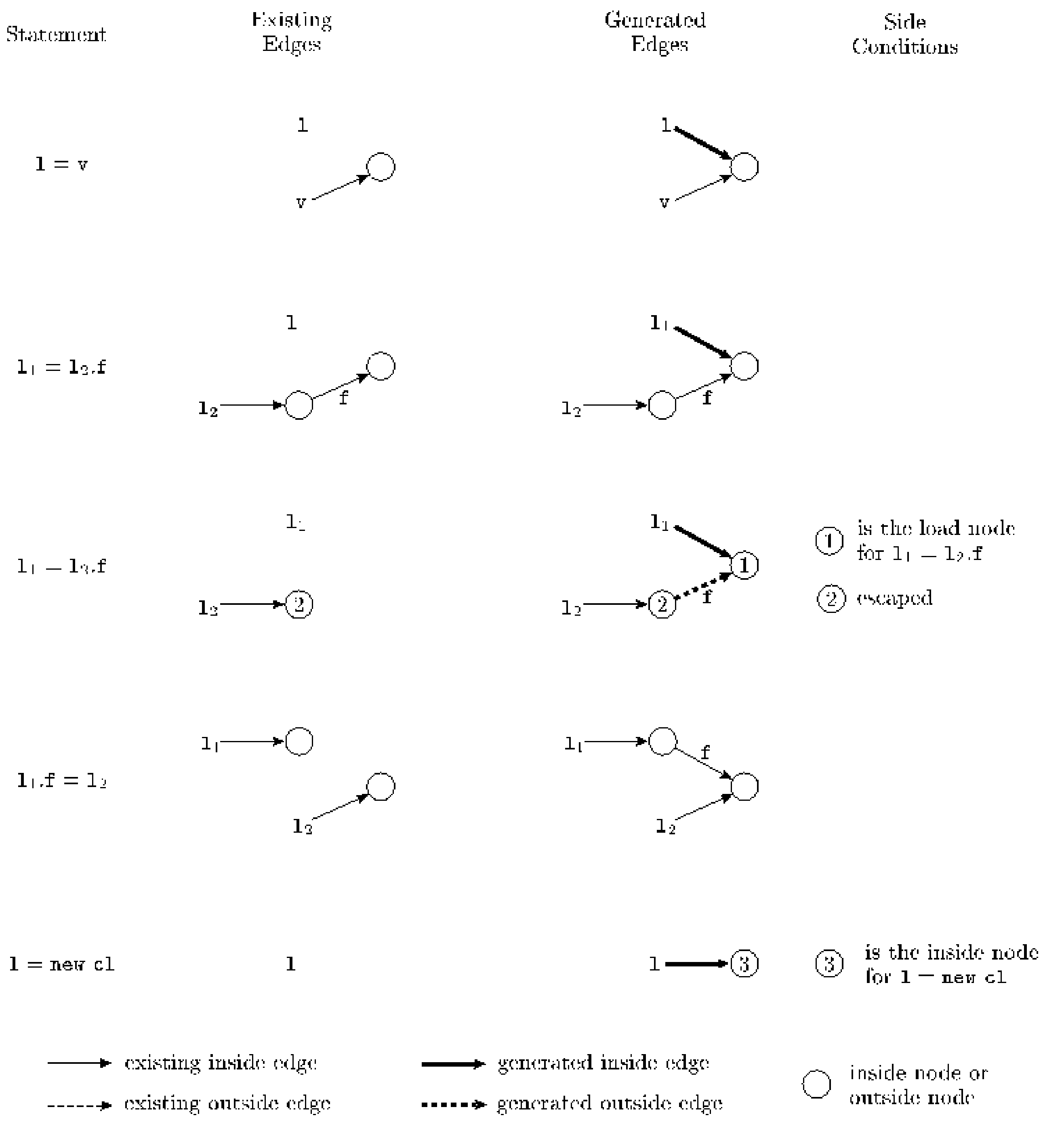

Figure 3.1: Generated Edges for Basic Statements

The mapping is a kind of equivalence[1] between nodes of the callee and nodes of the caller.

The mapping rules of the paper are in the Figure 3.2. We won't speak about the e, e_M and e_R here, it is just an "implementation trick" to know if a node escape without having to recompute it. So we won't explain the [`rules 9, 10`] and the part of the [`rule 4`] who speak about e.

[1]kind of equivalence and not equivalence because the mapping is only done for some interesting nodes, it's not an equivalence but just the subset useful for the computation

$$\frac{0 \le i \le k, n \in I(\mathtt{l}_i)}{n \in \mu(n_{\mathtt{p}_i})} \tag{1}$$

$$\frac{\mathtt{cl} \in \mathbb{CL}}{n_{\mathtt{cl}} \in \mu(n_{\mathtt{cl}})} \tag{2}$$

$$\frac{\begin{array}{c}\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, \langle\langle n_3, \mathtt{f}\rangle, n_4\rangle \in I, \\ n_3 \in \mu(n_1), n_1 \notin N_I\end{array}}{n_4 \in \mu(n_2)} \tag{3}$$

$$\begin{array}{cc} O \subseteq O_M & I - \text{edgesFrom}(I, \mathtt{l}) \subseteq I_M \\ e(n) \subseteq e_M(n) & r \subseteq r_M \end{array} \tag{4}$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R}{(\mu(n_1) \times \{\mathtt{f}\}) \times \mu(n_2) \subseteq I_M} \tag{5}$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R, n \in \mu(n_1), n_2 \in N_I \cup N_R}{n_2 \in \mu(n_2)} \tag{6}$$

$$\frac{\begin{array}{c}\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, n \in \mu(n_1), \\ \text{escaped}(\langle O_M, I_M, e_M, r_M\rangle, n)\end{array}}{\langle\langle n, \mathtt{f}\rangle, n_2\rangle \in O_M, n_2 \in \mu(n_2)} \tag{7}$$

$$\frac{n \in r_R \cap (N_I \cup N_R)}{n \in \mu(n)} \tag{8}$$

$$\frac{n' \in \mu(n)}{e_R(n) - \mathtt{P} \subseteq e_M(n')} \tag{9}$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_M \cup O_M}{e_M(n_1) \subseteq e_M(n_2)} \tag{10}$$

$$\cup\{\mu(n).n \in r_R\} \subseteq I_M(\mathtt{l}) \tag{11}$$

Figure 3.2: Rules for μ and $< O_M, I_M, e_M, r_M >$

We won't speak about the [`rule 2`] which is just for the static class variables.

- *Initialization :* To build the new graph the algorithm first copy the caller points-to graph but remove from it the insides edges from l (because l will be assigned) [`rule 4`].

- *Parameters :* Then it links the actual parameters with the formal parameter of the callee [`rule 1`].

- *Return values + assign l :* We said that the mapping is a kind of equivalence between the nodes of the callee and the nodes of the caller. By the [`rule 8`], the algorithm map the result nodes of the callee to themselves if they are inside nodes of the callee or if they are return nodes of the callee *(i.e. nodes created by a skipped call when computing the points-to graph of the callee).* Mapping a node to itself "add this node into the new graph", if we look at the [`rule 11`], which is used at the end of the algorithm, it add an inside edge from l to all the nodes "equivalents" of a result node of the callee, so the [`rule 8`] is a way to make l point to the result nodes which are created by the callee.

- *Add edges using the mapping :*

  - [`rule 5`] add an inside field edge to the new graph using the mapping equivalence

  - [`rule 7`] add an outside edge to the new graph (after having complete the mapping)

- *Complete the mapping :*

  - [`rule 3`] complete the mapping to add an equivalence between an outside edge of the callee and an inside edge of the caller

  - [`rule 6`] complete the mapping to add an equivalence between an inside edge of the callee and an inside edge of the caller *(with the help of the [`rule 5`])*

  - [`rule 7`] complete the mapping to add an equivalence between an outside edge of the callee and an outside edge of the caller

3.3.2 Implementation of the mapping algorithm

The paper give an algorithm to apply the mapping rules in a pseudo-language. We have implemented it in ocaml to see how it run and to see if our intuition of what should be done for some example was confirmed.

3.3.3 Problems founded

3.3.3.1 Mismatches in the algorithm and the specification :

The implementation helped us to find a mismatch between the specification of the rules (Fig. 3.2) and the algorithm.

The [`rule 6, 7`] are applied in the algorithm just in the case $n = n_1$. That means that those rules are first applied for the classes nodes and for the returned nodes, and because themselves implies their utilization by mapping the node n_2 to itself, we can resume that those rules if we add an class node or a return node then they add all the "chains" following it.

We have run an example to find this in which the [`rule 7`] would have had a wrong node and field, this example is the addElement method of the paper with a call to the insert method. The Figure 3.3 show this example and the wrong application of [`rule 7`] is circled in red.

!!!!! just for memo not right place here !!!!!!! In this example we can also see that the method addElement given by the paper is not exactly conform of the statement treated, "element" is not a variable like we expect for an assignment but a field of "this". To consider it as a variable have helped us to find the next problem studied in 3.3.3.2. If the program had had been recompiled to satisfy this condition the problem would not be there

3.3.3.2 Case $l = l_0.op(l_1, ..., l_i)$ where there is a k such that $l = l_k$

With the last example it show us a problem when in the statement $l = l_0.op(l_1, ..., l_i)$ there is a k such that $l = l_k$.

We have written a simple example to show this (see Figure 3.4):
l : res, l_0 : res, op : callee

```
class list{
list next;
list(){}

synchronized list callee (){
next = new list();
return(next);
}
synchronized void caller (){
res = this;
res = res.callee();
}
}
```

In this example we see that what we would expect is not computed by the paper's algorithm.

The problem are :
- the algorithm first cut the edges from the variable l to the right node and then make l point to a new "return"

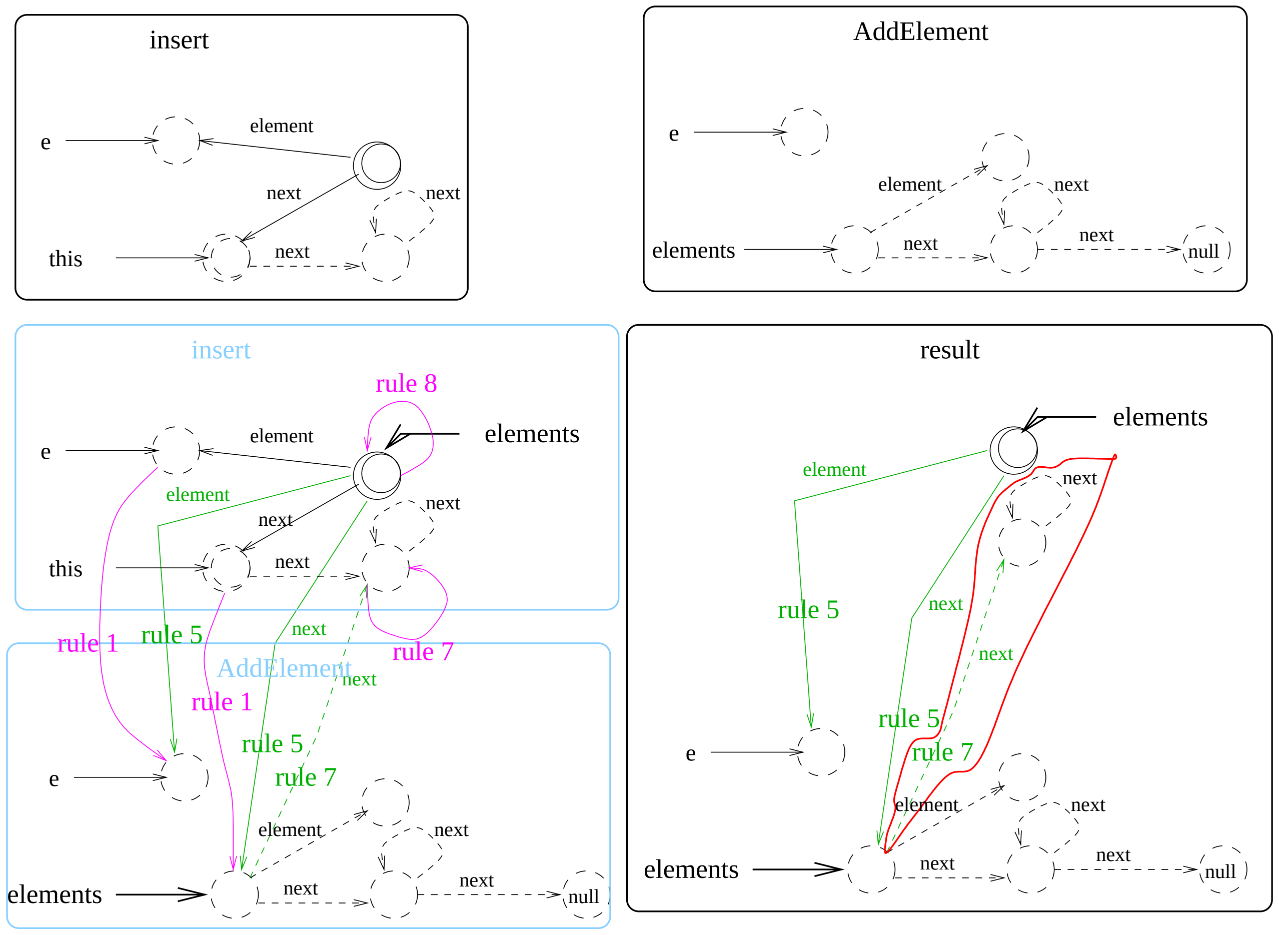

Figure 3.3: Example of addElement in a list

node (build for skipping the method) so when it need the old edges for the l_k it's lost
- to link the parameters the algorithm use the old caller points-to graph which contain the skipping "return" node and then include it in the new graph, which should not be

The way to avoid those problem would be :
- not allowed a l_k to be same as l by adding a statement in the precompilation, or
- not to suppress the edges from l when skipping (they would be suppressed after if treating the call), this is replace in the new paper (cf. 3.4) by rememoring the "situation" before treating a call statement
- and always replace N_R by N_R without the return node of the callee treated presently

3.3.3.3 Coherence if a call is treated after all the method has already been treated :

```
class list{
list next;
list(){}

synchronized list callee (){
next = new list();
return(next);
}
synchronized void caller (){
res = this;  [1]
```

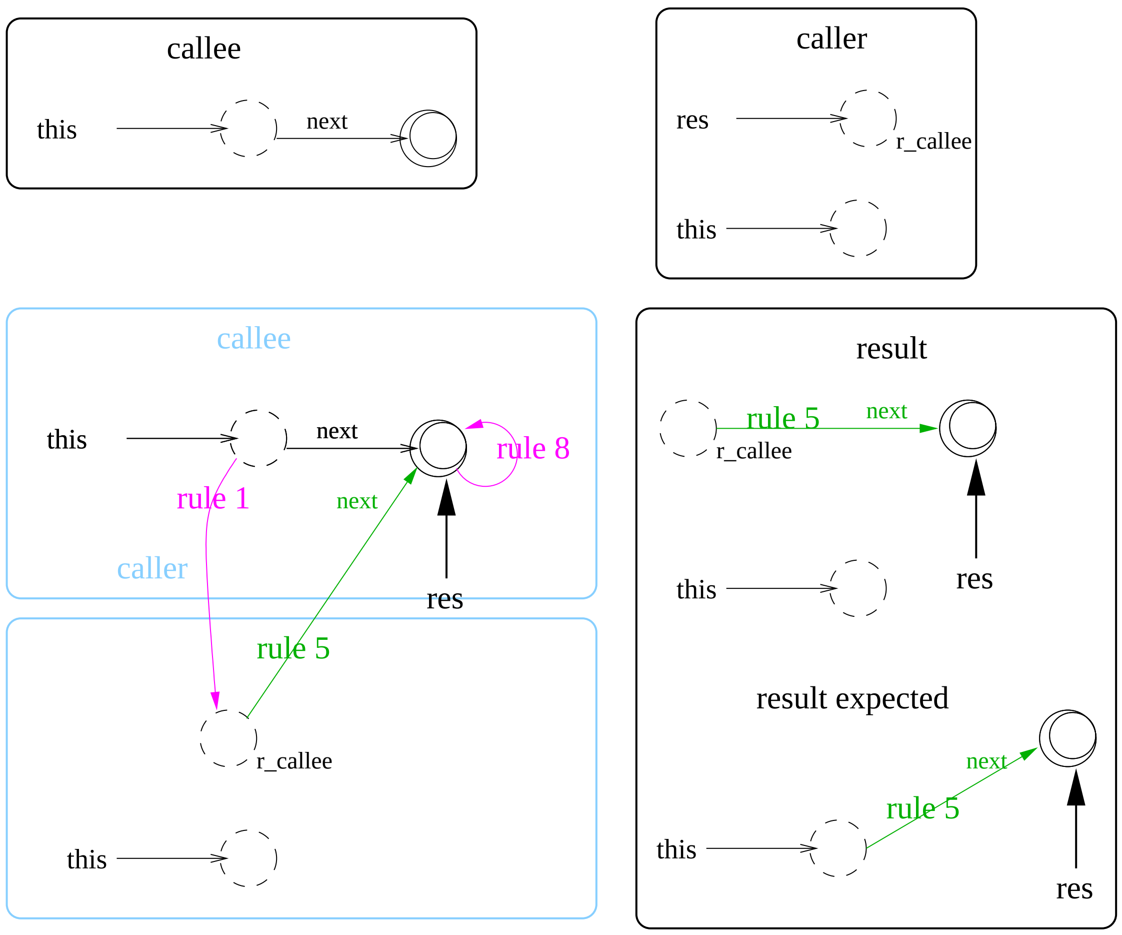

Figure 3.4: Example of $l = l_0.op(l_1, ..., l_i)$ where there is a k such that $l = l_k$

```
res = res.callee(); [2]
res = new list();  [3]
}
}
```

In this example, if we first skipped the call and after if we want to analyze it we have first to remember the points-to graph at point [1] then analyze it and then analyze the point [3].

The paper doesn't say how they find the points-to graph at point [1] without restart completely the analysis of the method from the beginning and for the following of the analysis they just say :"The analysis can incrementally increase the precision by analyzing method invocation sites that it originally skipped. The algorithm will then propagate the new, more precise result to update the analysis results at affected program points.“.

This seems to be quite expensive to analyses a call site after having already analyzed all the method.

3.4 New paper : resolve some problems we found in the last paper

We have next studied the paper *"Incrementalized Pointer and Escape Analysis"* of Frédéric Vivien and Martin Rinard [12].

They avoid the problem found with the case $l = l_0.op(l_1, ..., l_i)$ where there is a k such that $l = l_k$.
They don't have the same statements as in the other paper and they don't have method with return value so the statements for call sites are type of $l_0.op(l_1, ..., l_k)$. It's same since we can just replace $l = l_0.op(l_1, ..., l_k)$ by

$l_0.op(l_1, ..., l_k, l)$ but it avoid to lose the value of l before using it and it avoid to create return node when skipping a call statement.

They maintain the coherence if a call is treated after all the method has already been treated by recording the set of skipped call sites and for each recording the initial parameter and the points-to graph before and after skip and the set of the call sites skipped before and after the call site.
They say "this algorithm generates all of the new edges that a complete reanalysis would generate." but they don't show it and in the rules they show it seems that when they treat a call site they kind of loose precision by using only unions (c.f. 4.5).

3.5 Conclusion

* seems to be good idea, they have implemented it so it should be working
* doesn't detail what they do, specially in the second paper they just affirm it works
* the information from the expression of a condition is lost
(* what about a recursive method, specially if the method doesn't end)

Bibliography

[1] Daniel Jackson; Ian Schechter an dIlya Shlyakhter. Alcoa: The alloy constraint analyzer.

[2] William Adjie-Winoto; Elliot Schwartz; Hari Balakrishnan and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM SOSP*, Kiawah Island, SC, Dec 1999.

[3] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 199.

[4] Daniel Jackson. Alloy:a lightweight object modelling notation. Technical Report Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, Feb 2000.

[5] Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, November 2000.

[6] Michael Huth; Radha Jagadeesan and David Schmidt. Modal transition systems: a foundation for three-valued program analysis.

[7] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyser.

[8] M. Sagiv; T. Reps and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, San Antonio, TX, Jan 1999.

[9] M. Muller-Olm; D. Schmidt and B. Steffen. Model-checking: A tutorial introduction. In Springer, editor, *Proc. 6th Static Analysis Symposium, G. File and A. Cortesi*, number 1694 in LNCS, pages 330–354, 1999.

[10] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic.

[11] Moshe Y. Vardi. Branching vs. linear time: Final showdown.

[12] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, Utah, June 2001.

[13] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.

[14] Eran Yahav. Verifying sfety properties of concurrent java programs using 3-valued logic.

Chapter 4

Conclusion

Appendix A

Translation CTL to Alloy

`sts.tex`

$[s \models p]$ = p in s.L

$[s \models \neg\phi]$ = not ($[s \models \phi]$)

$[s \models \phi_1 \wedge \phi_2]$ = ($[s \models \phi_1]$) && ($[s \models \phi_2]$)

$[s \models \phi_1 \vee \phi_2]$ = ($[s \models \phi_1]$) || ($[s \models \phi_2]$)

$[s \models \phi_1 \rightarrow \phi_2]$ = ($[s \models \phi_1]$) $\rightarrow$ ($[s \models \phi_2]$)

$[s \models AX\phi]$ = all s2 | s2 in s.transition $\rightarrow$ ($[s2 \models \phi]$)

$[s \models EX\phi]$ = some s2 | s2 in s.transition && ($[s2 \models \phi]$)

$[s \models AG\phi]$ = ($[s \models \phi]$) &&
    (all s2 | s2 in s.∗transition $\rightarrow$ ($[s2 \models \phi]$))

$[s \models EF\phi]$ = ($[s \models \phi]$) ||
    (some s2 | s2 in s.∗transition && ($[s2 \models \phi]$))

$[s \models A[\phi_1 U \phi_2]]$ = ($[s \models \phi_2]$)[1] ||
(
      ($[s \models \phi_1]$)[2]
&& (all s2 | s2 in s.∗ Rel_($\phi_1 \wedge \neg\phi_2$)[3] $\rightarrow$
    (s2 !in s2.∗ Rel_($\phi_1 \wedge \neg\phi_2$)[4]
    && some s2.transition[5]
    && (all s3 | s3 !in s2.transition || ($[s3 \models \phi_1]$) || ($[s3 \models \phi_2]$))[6]   )
)

$[s \models E[\phi_1 U \phi_2]]$ = ($[s \models \phi_2]$) ||
    (($[s \models \phi_1]$)

[1] case s satisfy ϕ_2
[2] otherwise s satisfy ϕ_1
[3] s2 come from s by a path satisfying ϕ_1
[4] no path with always ϕ_1 i.e. no cycle
[5] no stop state after a path satisfying ϕ_1 and not ϕ_2
[6] after transition if we don't satisfy ϕ_1 we must satisfy ϕ_2

```
    && (some s2 | some s3 | [$s3 \models \phi_2$] &&
    s2 in s1.*Rel_$\phi_1$ && s3 in s2.transition))

[$s \models AF\phi$] = ([$s \models \phi$]) ||
     (no s2 | s2 in s.* Rel_$\neg\phi$ && (s2 in s2.* Rel_$\neg\phi$ || no s2.transition))

[$s \models EG\phi$] = ([$s \models \phi$]) &&
     (some s2 | s2 in s.* Rel_$\phi$ && (s2 in s2.* Rel_$\phi$ || no s2.transition))
```

```
State {
    ...
    Rel_$\phi$ : State $\rightarrow$ State
}

def Rel_$\phi$ {
    all s1 | s1.Rel_$\phi$ = {s2 | s2 in s1.transition && ([$s2 \models \phi$])}
}
```

[$s \models \top$] and [$s \models \bot$] are part of [$s \models p$]

k

Appendix B

State transition system in Alloy

```
// sts-simple.all
// Elodie-Jane Sims, March, 2001

model STS {
domain { States, Prop}
state {
//definiton of States
partition State_0, State_1, State_2: States!

Top, Bot : Prop!
Atom : Prop
partition  p, q, r:Atom!

transition : States -> States
L : States -> Prop

// all the relation for the CTL's formula
Rel_neg_r : States -> States
      }
def Rel_neg_r{
all sa1| sa1.Rel_neg_r = {sa2 | sa2 in sa1.transition && r !in sa2.L}
}

inv def_of_the_sts {
State_0.transition = {sa | sa = State_1 || sa = State_2}
State_1.transition = {sa | sa = State_0 || sa = State_2}
State_2.transition = State_2

State_0.L = {pr| pr = Top || pr = p || pr = q}
State_1.L = {pr| pr = Top || pr = r || pr = q}
State_2.L = {pr| pr = Top || pr = r}
}

inv def_of_Top_Bot{
// definition of Top
all sa| Top in sa.L
// definition of Bot
all sa| Bot !in sa.L
}

assert p_and_q_State_0{
p in State_0.L && q in State_0.L
}

assert AFr_s0{
r in State_0.L || (no s2 | s2 in State_0.*Rel_neg_r && ( s2 in
s2.*Rel_neg_r|| no s2.transition))
}

}//end model sts
```

Appendix C

Translation $\mu - calculus$ to Alloy *(unworking version1)*

`mts.tex`

$[s \models \top]$ = Top in s.L

$[s \models \bot]$ = Bot in s.L

$[s \models \phi_1 \wedge \phi_2]$ = ($[s \models \phi_1]$) && ($[s \models \phi_2]$)

$[s \models \phi_1 \vee \phi_2]$ = ($[s \models \phi_1]$) || ($[s \models \phi_2]$)

$[s \models [a]\phi]$ = all s2 | all act |
    act != a || s2 !in s.transition[act] || ($[s2 \models \phi]$)

$[s \models \langle a \rangle \phi]$ = some s2 | all act |
    act != a || (s2 in s.transition[act] && ($[s2 \models \phi]$))

$[s \models X]$ = s in X.rho

$[s \models \mu X.\phi]$ = all x | s in x || {s2 | $[s2 \models \phi]$(X.rho/x)} !in x

$[s \models \nu X.\phi]$ = some x | s in x && x in {s2 | $[s2 \models \phi]$(X.rho/x)}

———

$[s2 \models \phi]$(X.rho/x) means $[s2 \models \phi]$ where X.rho is replaced by x

———

"$[s \models \phi]$ = F" means that ϕ holds for the state s in our model iff the Alloy's formula "F" is true in our model

Appendix D

Translation $\mu - calculus$ to Alloy *(unworking version2)*

`mts2.ps`

$[s \models \top]$ = Top in s.L

$[s \models \bot]$ = Bot in s.L

$[s \models \phi_1 \wedge \phi_2]$ = ($[s \models \phi_1]$) && ($[s \models \phi_2]$)

$[s \models \phi_1 \vee \phi_2]$ = ($[s \models \phi_1]$) || ($[s \models \phi_2]$)

$[s \models [a]\phi]$ = all s2 | all act |
    act != a || s2 !in s.transition[act] || ($[s2 \models \phi]$)

$[s \models \langle a \rangle \phi]$ = some s2 | all act |
    act != a || (s2 in s.transition[act] && ($[s2 \models \phi]$))

$[s \models X]$ = s in X.rho

$[s \models \mu X \cdot \phi]$ = no s.Rel_$\nu \neg X \cdot \neg \phi$

$[s \models \nu X \cdot \phi]$ = some s.Rel_$\nu X \cdot \phi$

$[s2 \models \phi]$(X.rho/x) means $[s2 \models \phi]$ where X.rho is replaced by x

---

"$[s \models \phi]$ = F" means that ϕ holds for the state s in our model iff the Alloy's formula "F" is true in our model

---

```
State {
    ...
    Rel_$\nu X \cdot \phi$ : State $\rightarrow$ State
}
def Rel_$\nu X \cdot \phi$ {
    all s1 | s1.Rel_$\nu X \cdot \phi$ = {s2 | s1 in s1.Rel_$\nu X \cdot \phi$ && ($[s2 \models \phi]$(X.rho/s1.Rel_$\nu X \cdot \phi$))}
}
```

Appendix E

Modal transition system in Alloy

```
// mts-simple.all
// Elodie-Jane Sims, April, 2001

model MTS {
domain { States, Act, Prop, Var}
state {


transition [Act] : States -> States
L : States -> Prop

//definiton of States
partition State_s, State_t, State_u, State_v: States!
partition act_a, act_b : Act!

partition Top, Bot, holds_A, holds_F, holds_F_bis,
holds_G: Prop!

//all the relation for the mu-calculus's formula
X_F : States -> States
X_G : States -> States

      }

def X_F {all sa | sa.X_F = {s2 | (some s2_2 | all act | act != act_b
 || (s2_2 in s2.transition[act] && s2_2 in sa.X_F)) && sa in sa.X_F}}

def X_G {all sa | sa.X_G = {s2 | ((some s2_2 | all act | act != act_b
 || s2_2 in s2.transition[act])
&&(some s2_3 | all act | act != act_a
        || (s2_3 in s2.transition[act] && s2_3 in sa.X_G)))
 && sa in sa.X_G}}

inv def_of_the_mts {
all act|
(act = act_a ->
  (   no State_s.transition [act]
   && no State_t.transition [act]
   && State_u.transition [act] = State_v
   && no State_v.transition [act])
)
&&
(act = act_b ->
  (   no State_s.transition [act]
  && State_t.transition [act] = {sa | sa = State_s || sa = State_u}
  && State_u.transition [act] = State_u
  && State_v.transition [act] = State_u)
)
}

inv def_of_Top_Bot {
// definition of Top
all sa| Top in sa.L
// definition of Bot
all sa| Bot !in sa.L
}

inv prop {
{sa |
some s2 | all act| act != act_a || (s2 in sa.transition[act])
}
= {sa |holds_A in sa.L}

{sa |
some sa.X_F
}
= {sa |holds_F in sa.L}

{sa |
some sa.X_G
}
= {sa |holds_G in sa.L}


{sa |
some x | sa in x &&
x in {s2 | some s2_2 | all act | act != act_b || (s2_2 in
s2.transition[act] && s2_2 in x
)}
}
= {sa |holds_F_bis in sa.L}
}
}//end model
```

Appendix F

Implementation of the mapping rules's algorithm in points-to escape analysis

```
(**********************************************************)
(* graph.ml *)

open List;;

type op = string;;
type variable_name = string;;
type variable = op * variable_name;;

type field_name = string;;

type class_name = variable_name;;

type node_out =
CL of class_name
|Pa of variable * int
|L of op * int
|R of op * int;;

type node_in =
T of int
|I of op * int;;

type node =
Out of node_out
|In of node_in;;

type field = F of field_name;;

type start_point =
Field of (node * field)
| Var of variable;;
```

```
type edges = (node * field) * node;;
type o = edges list;;

type edges_in = start_point * node;;
type i = edges_in list;;

type escape_data =
 P of int
|Cl of int
|T of int
|M of int;;

type e = (node * escape_data list )list;;

type r = node list;;

type graph = {mutable o:o; mutable i:i; mutable e:e; mutable r:r};;

type map = (node * node list) list;;
type delta = (node * (node * field) list) list;;
type d = (node * node) list;;

type w_list = (node * edges) list;;

type work_list = {mutable we:w_list; mutable wi:w_list;
 mutable wo:w_list};;

(* some fonction to union list without double element *)
let rec add l1 = function
[] -> l1
|h::t -> if (mem h l1) then add l1 t
else add (h::l1) t;;

(* add_map add a set of node n1_list to the mu of a node n1 *)
let rec add_map_list n1_list n1= function
[] -> [(n1, n1_list)]
|(n, n_list)::t when (n = n1) -> (n, add n_list n1_list)::t
|h::t -> h::(add_map_list n1_list n1 t);;
(* end of some fonction to union list without double element *)

(********************************************************)
(* escape.ml *)

open List;;
open Graph;;

let escape (g: graph) (n : node) =
let e_n = try (assoc n g.e) with Not_found -> [] in
(e_n != []) || (mem n g.r);;
```

```
(*******************************************************)
(* propagate.ml *)

open List;;
open Graph;;

let rec union_e info1 = function
[] -> info1
|h::t -> if (mem h info1) then union_e info1 t
 else h::(union_e info1 t);;

let rec succ_o n = function
[] -> []
|((n1, f) , n2)::t when (n1=n) -> n2::(succ_o n t)
|h::t -> succ_o n t;;
let rec succ_i n = function
[] -> []
|(Field (n1, f) , n2)::t when (n1=n) -> n2::(succ_i n t)
|_::t -> succ_i n t;;
let succ g n =
let osucc = succ_o n g.o in
let isucc = succ_i n g.i in
union_e osucc isucc;;

let rec change n new_e_n = function
[] -> (match new_e_n with
[] -> []
|_ -> [(n, new_e_n)])
|(m, old_e_n)::t when (m = n) -> (match new_e_n with
[]-> t
|_ -> (n, new_e_n)::t)
|h::t -> h::(change n new_e_n t);;

let rec union_special chgmt e_n2 = function
[] -> chgmt:= false; e_n2
|h::t -> if (mem h e_n2) then union_special chgmt e_n2 t
 else let res = (h::(union_special chgmt e_n2 t))
      in chgmt:= true; res;;

let rec update_2_special (e_M:e) n1 n2 (chgmt: bool ref) =
let e_n1 = try (assoc n1 e_M) with Not_found -> [] in
let e_n2 = try (assoc n2 e_M) with Not_found -> [] in
let new_e_n2 = union_special chgmt e_n2 e_n1 in
change n2 new_e_n2 e_M;;

let rec propag (g : graph) chgmt = function
[] -> ()
|n1::t1 -> updating chgmt n1 g t1 (succ g n1)
and updating chgmt n1 g t1 = function
[] -> propag g chgmt t1
|n2::t2 -> g.e <- (update_2_special g.e n1 n2 chgmt);
  if ((!chgmt) = true) then
updating chgmt n1 g (n2::t1) t2
```

```
  else updating chgmt n1 g t1 t2;;

let propagate (g : graph) nodes =
let chgmt = ref false in
propag g chgmt nodes;;

(************************************************************)
(* worklist.ml *)

open List;;
open Graph;;

let rec make_wk n = function
[] -> []
|e::t -> (n, e)::(make_wk n t);;

(*the last parameter is the work list*)
let rec union_work_list n (edges: edges list) = function
[] -> (match edges with [] -> [] |_ -> make_wk n edges)
|(n_w, e)::t when (n_w=n) -> if (mem e edges) then
      union_work_list n edges t
     else (n_w, e)::(union_work_list n edges t)
|h::t -> h ::(union_work_list n edges t);;
(************************************************************)
(* edgesFrom.ml *)

open List;;
open Graph;;

let rec edgesFrom_in n = function
[] -> []
|(Field (n1, f) , n2)::t when (n1=n) -> ((n1, f) , n2)::(edgesFrom_in n t)
|h::t -> edgesFrom_in n t;;

let rec edgesFrom_out n = function
[] -> []
|((n1, f) , n2)::t when (n1=n) -> ((n1, f) , n2)::(edgesFrom_out n t)
|h::t -> edgesFrom_out n t;;
```

```
(*********************************************************)
(* addEdges.ml *)

open List;;
open Graph;;
open Propagate;;

let rec make_edges_in n = function
[] -> []
|h::t -> (Field (h) , n)::(make_edges_in n t);;
let rec make_edges_in_list edg_list = function
[] -> []
|h::t -> (make_edges_in h edg_list)@(make_edges_in_list edg_list t);;
let rec make_edges_out n = function
[] -> []
|h::t -> (h, n)::(make_edges_out n t);;
let rec make_edges_out_list edg_list = function
[] -> []
|h::t -> (make_edges_out h edg_list)@(make_edges_out_list edg_list t);;

let rec insert_inside_edge in_edge = function
[] -> [in_edge]
|(Field(n1, f) , n2)::t -> (match in_edge with
(Field (n1_add, f_add) , n2_add) when
((n1=n1_add) && (f=f_add) && (n2=n2_add)) ->
(Field(n1, f) , n2)::t
|_ -> (Field(n1, f) , n2)::(insert_inside_edge in_edge t))
|(Var (v), n)::t -> (match in_edge with
   (Var (v_add), n_add) when
   ((v=v_add)&&(n=n_add)) -> (Var (v), n)::t
   |_ -> (Var (v), n)::(insert_inside_edge in_edge t));;
let rec union_edges_in edges_in_list = function
[] -> edges_in_list
|h::t -> union_edges_in (insert_inside_edge h edges_in_list) t;;

let rec insert_outside_edge (out_edge: edges) = function
[] -> [out_edge]
|((n1, f), n2)::t -> (match out_edge with
((n1_add, f_add), n2_add) when
((n1=n1_add) && (f=f_add) && (n2=n2_add)) ->
((n1, f), n2)::t
|_ -> ((n1, f), n2)::(insert_outside_edge out_edge t));;
let rec union_edges_out edges_out_list = function
[] -> edges_out_list
|h::t -> union_edges_out (insert_outside_edge h edges_out_list) t;;


let addInsideEdges g edg_list n_list=
let starting_nodes = fst(split edg_list) in
let new_edges = make_edges_in_list edg_list n_list in
g.i <- (union_edges_in g.i new_edges);
propagate g starting_nodes;;
let addOutsideEdges g edg_list n_list=
let starting_nodes = fst(split edg_list) in
let new_edges = make_edges_out_list edg_list n_list in
g.o <- (union_edges_out g.o new_edges);
propagate g starting_nodes;;

(*******************************************************)
(* mapNode.ml *)

open List;;
open Graph;;
open Propagate;;
open AddEdges;;
open EdgesFrom;;
open Worklist;;

(* function for the update of mu *)
let rec add_map n_add n1 = function
[] -> [(n1, n_add::[])]
|(n, n_list)::t when (n = n1) -> if (mem n_add n_list) then (n, n_list)::t
 else (n, n_add::n_list)::t
|h::t -> h::(add_map n_add n1 t);;
(* end of function for the update of mu *)

(* function for the update of D *)
let rec add_d (n1:node) (n:node) = function
[] -> [(n1, n)]
|(n1_2, n_2)::t when (n1_2 = n1) && (n_2 = n) -> (n1_2, n_2)::t
|h::t -> h::(add_d n1 n t);;
(* end of function for the update of D *)

(* function for the update of e_M *)
let rec change n new_e_n = function
[] -> (match new_e_n with
[] -> []
|_ -> [(n, new_e_n)])
|(m, old_e_n)::t when (m = n) -> (match new_e_n with
[]-> t
|_ -> (n, new_e_n)::t)
|h::t -> h::(change n new_e_n t);;

let rec un_P = function
[] -> []
|P (a)::t -> un_P t
|h::t -> h::(un_P t);;

let update_e (e_M:e) (e_R:e) n n1 =
let e_M_n = try (assoc n e_M) with Not_found -> [] in
let e_R_n1 = try (assoc n1 e_R) with Not_found -> [] in
match (un_P e_R_n1) with
[] -> e_M
|new_e_R_n1 ->
(let new_e_M_n = union_e e_M_n new_e_R_n1 in
change n new_e_M_n e_M);;
```

```
(* end of function for the update of e_M *)

(* function for updating delta *)
let rec updt_delta (n2: node) (new_delta_n2 : (node * field) list) = function
[] -> (match new_delta_n2 with
[] -> []
|_-> [(n2, new_delta_n2)])
|(n, v)::t when (n=n2) -> (match new_delta_n2 with
[] -> updt_delta n2 new_delta_n2 t
|_-> (n2, new_delta_n2)::(updt_delta n2 new_delta_n2 t))
|h::t -> h::(updt_delta n2 new_delta_n2 t);;

let update_delta n2 new_delta_n2 (delta: delta ref) =
delta := updt_delta n2 new_delta_n2 !delta;;

(* function for updating delta *)

(* function for the step of adding inside egdges *)
(*the last parameter is I_R*)
let rec step_add_in (g_M:graph) (n1:node) (n:node)
(delta : delta ref) (mu: map ref) = function
[] -> ()
|(Field (n1_l, f) , n2_l)::t when (n1_l=n1) ->
(addInsideEdges g_M [(n, f)] (try (assoc n2_l !mu) with Not_found -> []);
update_delta n2_l (add [(n,f)] (try (assoc n2_l !delta)
with Not_found -> [])) delta;
step_add_in g_M n1 n delta mu t
)
|h::t -> step_add_in g_M n1 n delta mu t;;
(* function for the step of adding inside egdges *)

let mapNode (n1: node) (n:node) (mu : map ref) (d : d ref)
(delta : delta ref) (g_M: graph) (g_R : graph) (wk : work_list)=
if not(mem (n1, n) !d) then
    (mu:= add_map n n1 (!mu);
    d:= add_d n1 n (!d);
    g_M.e <- (update_e g_M.e g_R.e n n1);
    propagate g_M [n];
    addInsideEdges g_M (try (assoc n1 !delta) with Not_found -> []) [n];
    step_add_in g_M n1 n delta mu g_R.i;
    wk.we <- union_work_list n (edgesFrom_out n1 g_R.o) wk.we;
    if (n1=n) then
(wk.wi <- union_work_list n (edgesFrom_in n1 g_R.i) wk.wi;
wk.wo <- union_work_list n (edgesFrom_out n1 g_R.o) wk.wo);
)
else ();;

(*********************************************************)
(* mapping.ml *)

open List;;
open Graph;;
open MapNode;;
open Escape;;
open AddEdges;;

(* function to initialise the I_M *)
let rec init_i v = function
[] -> []
|(Var(v2), n)::t when (v=v2) -> (init_i v t)
|h::t -> h::(init_i v t);;
(* end of function to initialise the I_M *)

(* function mapping parameter nodes *)
let rec get_param_node (number: int) (callee : op) = function
[] -> []
|Out(Pa((op, v), n))::t when (op=callee)&&(number=n)
-> Out(Pa((op, v), n))::(get_param_node number callee t)
|h::t -> get_param_node number callee t;;

let rec assoc_list n = function
[] -> []
|(n2,m)::t when (n2=n) -> m::(assoc_list n t)
|h::t -> assoc_list n t;;

let rec make_couple l1 = function
[] -> []
|h2::t2 -> (mk_couple h2 l1)@(make_couple l1 t2)
and mk_couple h2 = function
[] -> []
|h1::t1 -> (h1, h2)::(mk_couple h2 t1);;

let map_one_node g caller callee mu d delta g_M g_R wk nodes_set li =
let vari = fst li in
let number = snd li in
let n_pointed_by_li = (try (assoc_list (Var vari) g.i) with Not_found -> []) in
let n1 = get_param_node number callee nodes_set in
let map_func mu d delta g_M g_R wk = function
(n1, n) -> (mapNode n1 n mu d delta g_M g_R wk) in
iter (map_func mu d delta g_M g_R wk) (make_couple n1 n_pointed_by_li);;

let map_parameter_nodes g caller callee mu d delta g_M g_R wk nodes_set l_i =
let f g caller callee mu d delta g_M g_R wk nodes_set =
map_one_node g caller callee mu d delta g_M g_R wk nodes_set in
iter (f g caller callee mu d delta g_M g_R wk nodes_set) l_i;;
(* end function mapping parameter nodes *)

(* function mapping class nodes *)
let f_class mu d delta g_M g_R wk = function
cl_name ->
(mapNode (Out(CL(cl_name))) (Out(CL(cl_name))) mu d delta g_M g_R wk);;
(* end of function mapping class nodes *)

(* function mapping return nodes *)
let f_recturn_value mu d delta g_M g_R wk = function
n -> (mapNode n n mu d delta g_M g_R wk);;
```

```
let rec select_return_node = function
[] -> []
|In(a)::t -> In(a)::(select_return_node t)
|Out(R(a,b))::t -> Out(R(a,b))::(select_return_node t)
|h::t -> select_return_node t;;

(* end of function mapping return nodes *)

(* function of W list *)
let rec do_e mu d delta g g_M g_R wk = function
[] -> do_i mu d delta g g_M g_R wk wk.wi
|(n3, ((Out(nd), f), n2))::t ->
wk.we <- t;
let n4_list = assoc_list (Field(n3, f)) g.i in
let map_func mu d delta g_M g_R wk = function
(n1, n) -> (mapNode n1 n mu d delta g_M g_R wk) in
iter (map_func mu d delta g_M g_R wk) (make_couple [n2] n4_list);
do_e mu d delta g g_M g_R wk wk.we
|h::t -> wk.we <- t; do_e mu d delta g g_M g_R wk wk.we
and do_i mu d delta g g_M g_R wk = function
[] -> do_o mu d delta g g_M g_R wk wk.wo
|(n, ((n1, f), In(a)))::t ->
wk.wi <- t;
mapNode (In(a)) (In(a)) mu d delta g_M g_R wk;
do_e mu d delta g g_M g_R wk wk.we
|(n, ((n1, f), Out(R(a,b))))::t ->
wk.wi <- t;
mapNode (Out(R(a,b))) (Out(R(a,b))) mu d delta g_M g_R wk;
do_e mu d delta g g_M g_R wk wk.we
|h::t -> wk.wi <- t; do_i mu d delta g g_M g_R wk wk.wi
and do_o mu d delta g g_M g_R wk = function
[] -> ()
|(n, ((n1, f), n2))::t when (escape g_M n) ->
wk.wo <- t;
addOutsideEdges g_M [(n, f)] [n2];
mapNode n2 n2 mu d delta g_M g_R wk;
do_e mu d delta g g_M g_R wk wk.we
|h::t -> wk.wo <- t; do_o mu d delta g g_M g_R wk wk.wo
;;
(* end of function of WE *)

(* fonction to update I_M at the end for the result *)
let rec get_result_nodes mu = function
[] -> []
|n::t -> add (try (assoc n mu) with Not_found -> []) (get_result_nodes mu t);;
(* used as get_result_nodes mu g_R.r *)

let rec make_new_l_nodes l = function
[] -> []
|n::t -> (Var(l), n)::(make_new_l_nodes l t);;
(* end fonction to update I_M at the end for the result *)
```

```
let mapping_intra (g : graph) (g_R : graph) (g_M : graph)
(caller:op) (l: variable) (callee : op) (l_i: (variable * int) list)
(nodes_set : node list) (cl_list : class_name list) =
let (wk : work_list) = {we=[]; wi=[]; wo=[]} in
let (d : d ref) = ref [] in
let (mu : map ref) = ref [] in
let (delta : delta ref) = ref [] in
g_M.o <- g.o;
g_M.i <- (init_i l g.i);
g_M.e <- g.e;
g_M.r <- g.r;
map_parameter_nodes g caller callee mu d delta g_M g_R wk nodes_set l_i;
iter (f_class mu d delta g_M g_R wk) cl_list;
iter (f_recturn_value mu d delta g_M g_R wk) (select_return_node g_R.r);
do_e mu d delta g g_M g_R wk wk.we;
g_M.i <- g_M.i@(make_new_l_nodes l (get_result_nodes !mu g_R.r));;

(* function to put the number to the parameters*)
(* and to peut the caller name to them *)
let rec do_parameters_intra caller num l_i =
match l_i with
[] -> []
|v::t -> (let n = !num in
(num:= !num + 1;((caller, v), n)::(do_parameters_intra caller num t)));;

let do_parameters caller l_i =
let num = ref 0 in
do_parameters_intra caller num l_i;;
(* end function to put the number to the parameters*)
(* and to peut the caller name to them *)


(* function to collect the nodes *)
let insert_node n l = if (mem n l) then l else n::l;;

let rec collect_nodes_in list = function
[] -> ()
|(Field(n1, f), n2)::t -> list:= insert_node n1 (insert_node n2 !list);
  collect_nodes_in list t
|(Var(_), n)::t -> list:= insert_node n !list; collect_nodes_in list t;;

let rec collect_nodes_out list = function
[] -> ()
|((n1, f), n2)::t -> list:= insert_node n1 (insert_node n2 !list);
     collect_nodes_out list t;;

let rec collect_nodes_r list = function
[] -> ()
|n::t -> list:= insert_node n !list; collect_nodes_r list t;;

let collect_nodes_intra g list =
collect_nodes_in list g.i;
```

```
collect_nodes_out list g.o;
collect_nodes_r list g.r;;

let collect_nodes g g_R =
let list = ref [] in
collect_nodes_intra g list;
collect_nodes_intra g_R list;
(!list);;

let rec collect_classes = function
[] -> []
|Out(CL(cl_name))::t -> cl_name::(collect_classes t)
|h::t -> collect_classes t;;
(* end of function to collect the nodes *)


let mapping (g : graph) (g_R : graph) (caller:op)
(l: variable_name) (callee : op) (l_i: variable_name list) =
let (g_M : graph) = {o=[];i=[];e=[];r=[]} in
let l2 = (caller, l) in
let l_i2 = (do_parameters caller l_i) in
let nodes_set = collect_nodes g g_R in
let cl_list = collect_classes nodes_set in
mapping_intra g g_R g_M caller l2 callee l_i2 nodes_set cl_list;
g_M;;
```

```
(********************************************************)
(* run3.ml *)

open List;;
open Graph;;
open Mapping;;
#use "load_all.ml";;

let (g: graph ) = {
o = [((Out(Pa(("addElement", "elements"), 0)), F("element")),
Out(L("addElement", 1)));
     ((Out(Pa(("addElement", "elements"), 0)), F("next")),
Out(L("addElement", 2)));
     ((Out(L("addElement", 2)), F("next")), Out(L("addElement", 2)));
     ((Out(L("addElement", 2)), F("next")), Out(L("addElement", 3)))
];
i = [(Var("addElement", "e"), Out(Pa(("addElement", "e"), 1)));
    (Var("addElement", "elements"), Out(Pa(("addElement", "elements"), 0)))
];
e = [(Out(Pa(("addElement", "e"), 1)), [ P(1) ]);
    (Out(Pa(("addElement", "this"), 0)), [ P(0) ]);
    (Out(Pa(("addElement", "elements"), 0)), [ P(0) ]);
    (Out(L("addElement", 1)), [ P(0) ]);
    (Out(L("addElement", 2)), [ P(0) ]);
    (Out(L("addElement", 3)), [ P(0) ])
];
r = []
};;

let (g_R: graph ) = {
o=[((Out(Pa(("insert", "this"), 0)), F("next")), Out(L("insert", 0)));
   ((Out(L("insert", 0)), F("next")), Out(L("insert", 0)))
];
i=[(Var("insert", "e"), Out(Pa(("insert", "e"), 1)));
   (Var("insert", "this"), Out(Pa(("insert", "this"), 0)));
   (Var("insert", "m"), Out(L("insert", 0)));
   (Field(In(I("insert", 0)), F("element")), Out(Pa(("insert", "e"), 1)));
   (Field(In(I("insert", 0)), F("next")), Out(Pa(("insert", "this"), 0)))
];
e=[(Out(Pa(("insert", "e"), 1)), [ P(1) ]);
   (Out(Pa(("insert", "this"), 0)), [ P(0) ]);
   (Out(L("insert", 0)), [ P(0) ]);
   (In(I("insert", 0)), [])
];
r=[Out(Pa(("insert", "this"), 0));
  In(I("insert", 0))
]
};;

let (caller : op) = "addElement";;
let (l : variable_name) = "elements";;
let (callee : op) = "insert";;
let (l_i : variable_name list) = ["elements"; "e"];;
let (g_M : graph) = mapping g g_R caller l callee l_i;;
```

Appendix G

An example of a model in Alloy

[1]

```
// Simple model of an airport's ground-based resources
// Elodie-Jane Sims, February, 2001

model GroundControl {
     domain {fixed Location, fixed Place, Equipment }
     state {
partition Gate, Runway, Taxiway : static Place
partition Plane, Vehicle : static Equipment

//where say in wich location is an equipment
//equipment say whate are the equipment in a place
// C1: mobile resources can be at one location at a time
where (~equipment) : Equipment -> Location!

//location give the locations which are the place
// place give say of wich place this location is a part of
//C3-begin : fixed resources consist of some number of locations
location (~place) : Place+ -> Location+

location_adj: static Location -> Location
// place_adj give for a place all the place wich have one location adjacent
place_adj: static Place -> Place
//adj[P] give for a location the adjacent location wich are in P
adj[Place]: static Location -> Location

intersect : Place -> Place
            }


def place_adj {all p | p.place_adj = p.location.~location_adj.place}
def adj {all P | all l | l.adj[P]= (l.location_adj&P.location)}

def intersect { all p | p.intersect = p.location.place - p}

inv Policy{
```

[1]This model has been writen for the homework#2 of the course Software Specification (CIS 771) at K.S.U

```
// adjacent is reflexive
all l | l in l.location_adj

//C2 : a location can hold at most one plane
all l | sole l.equipment&Plane

//C3-end : fixed resources consist of adjacent location
all p : Place| all l| l in p.location -> p.location = l.*adj[p]

//C4 : Gates consist of a single location
all g : Gate | one g.location
//C4 : that is adjacent to a single taxiway
&& one g.place_adj&Taxiway
//C4 : at a single location
&& one g.location.location_adj&(g.place_adj&Taxiway).location

//C5 : Runways consist of multiple locations
all r : Runway | not sole r.location

//C6 : of the fixed resources, only runways can intersect and at atmost one location
all p1 : Place - Runway |all p2 |  p1 != p2 -> no p1.location&p2.location
all p1,p2 : Runway | p1 != p2 -> sole p1.location&p2.location

//C7 : Taxiways are adjacent to other fixed resources at atmost one location
all t : Taxiway | all p | sole t.location.adj[p]

//C8 : Planes can reach a runway from any gate
all g :Gate | some g.*place_adj&Runway

//C10 : Only planes can be on runways
all e : Equipment - Plane | no e.where.place&Runway

//C11 : At most one plane can be on a runway
all r : Runway | sole r.location.equipment&Plane
//C12 : Gates can have a single plane
all g : Gate | sole g.location.equipment&Plane

//C13 : Gates can have multiple service vehicles
//nothing to specifie

//C14 : If a plane is at a gate, then a vehicle should be there to service it
all g : Gate | some g.location.equipment & Plane -> some g.location.equipment & Vehicle

   }

op Move(e: Equipment!, l : Location!){
//pre conditions

//C9 : Mobile resources can only move between adjacent locations
l in e.where.location_adj

// the location should be empty
no l.equipment
```

```
//post conditions
e.where' = l

//Frame condition
Equipment' = Equipment
all e1 : Equipment - e | e1.where' = e1.where'
all l1| l1.place' = l1.place
all p| p.location' = p.location
     }

op Takeoff (p : Plane!){
//pre conditions

//C15 : a plane can take off on a runway
p.where.place in Runway
//C15 : when all intersecting runways are empty
no p.where.place.intersect.location.equipment

//post condition
Plane' = Plane - p
all l | l.equipment' = l.equipment - p
Equipment' = Equipment - p

//Frame condition
all l| l.place' = l.place
all p1| p1.location' = p1.location
}


op Land (p : Plane'!, r : Runway){
//pre conditions
//C6 : of the fixed resources, only runways can intersect and at atmost one location
all p1 : Place - Runway |all p2 |  p1 != p2 -> no p1.location&p2.location
all p1,p2 : Runway | p1 != p2 -> sole p1.location&p2.location
//precondition put because the check of C10 and c11 show bugs


//end of precondition put because the check of C10 and c11 show bugs

p !in Plane

//C16 : a plane can land on a runway when it is empty
no r.location.equipment

//post condition
Plane' = Plane + p
Equipment' = Equipment + p
one l1 |all l2| (l1 in r.location && l1.equipment'=l1.equipment + p)

//Frame condition
&& ((l1 !=l2) -> l2.equipment' = l2.equipment )
//if r cross an other runway it should also be impty to care about C11
```

```
&& no l1.place.location.equipment

all l| l.place' = l.place
all p1| p1.location' = p1.location
}

cond  Realism {
//1 : at least one gate, runway, vehicle and plane
some Gate && some Runway && some Plane && some Vehicle

//2 : some intersecting runways
some r | some r.intersect //don't have to specify runway because of C6
      }

inv C10 {
//C10 : Only planes can be on runways
all e : Equipment - Plane | no e.where.place&Runway

}
inv C11 {
//C11 : At most one plane can be on a runway
all r : Runway | sole r.location.equipment&Plane
}

assert Gate_to_any_runway{
all g: Gate|all r : Runway| r in g.*place_adj
 }

} //end of model GroundControl
```

Appendix H

Try of automaton in Alloy for checking LTL's formulas

```
// STS model
// Elodie-Jane Sims, March, 2001

model STS {
domain { Sigma, States_G, Prop, States_phi,  States_int}
state {
//definiton of States_G
partition i_G,  State_G_0, State_G_1, State_G_2: States_G!
init_states_G : States_G
final_states_G : States_G

//definiton of States_phi
partition State_phi_0, State_phi_1, State_phi_2: States_phi!
init_states_phi : States_phi
final_states_phi : States_phi

//propositions
Top, Bot : Prop!
Atom : Prop
partition  p, q, r:Atom!

//definition of the path automata
transition_G : States_G -> States_G
L_G : States_G -> Prop

//definition of the phi automata
transition_phi : States_phi -> States_phi
L_phi[States_phi] : States_phi -> Prop

//definition of the intersection automata
map_G : States_int -> States_G!
map_phi : States_int -> States_phi!

init_states_int : States_int
final_states_int : States_int
```

```
transition_int : States_int -> States_int
L_int : States_int -> Prop
      }

inv def_of_the_intersection {
//one state for one couple
all sint1| all sint2| (sint1 != sint2)->
((sint1.map_G = sint2.map_G)-> (sint1.map_phi != sint2.map_phi))

//definition of init_states_int
all sint: init_states_int| init_states_int.map_G in init_states_G &&
init_states_int.map_phi in init_states_phi
all sg : init_states_G| all sp : init_states_phi | some sint|
sint.map_G = sg && sint.map_phi = sp

//definition of final_states_int
all sint: final_states_int| final_states_int.map_G in final_states_G &&
final_states_int.map_phi in final_states_phi
all sg : final_states_G| all sp : final_states_phi | some sint|
sint.map_G = sg && sint.map_phi = sp

// definition of the transition of intersection

all sint1| all sint2|
(  (sint2 in sint1.transition_int) ->
   ((sint2.map_G in sint1.map_G.transition_G)
    && (sint2.L_int = sint2.map_G.L_G)
    &&(sint2.map_phi in sint1.map_phi.transition_phi)
    && (sint2.L_int = sint2.map_phi.L_phi[sint1.map_phi])
   )
)
&&
( ((sint2.map_G in sint1.map_G.transition_G)
    && (sint2.L_int = sint2.map_G.L_G)
    &&(sint2.map_phi in sint1.map_phi.transition_phi)
    && (sint2.L_int = sint2.map_phi.L_phi[sint1.map_phi])
   ) ->
  (sint2 in sint1.transition_int)
)
}

inv def_of_the_path {
init_states_G = i_G
all s_G | s_G in final_states_G

State_G_0.transition_G = {sa | sa = State_G_1 || sa = State_G_2}
State_G_1.transition_G = {sa | sa = State_G_0 || sa = State_G_2}
State_G_2.transition_G = State_G_2

State_G_0.L_G = {pr| pr = Top || pr = p || pr = q}
State_G_1.L_G = {pr| pr = Top || pr = r || pr = q}
State_G_2.L_G = {pr| pr = Top || pr = r}
```

```
}

inv def_of_the_phi {
init_states_phi = {s_phi| s_phi = State_phi_2}
final_states_phi = {s_phi| s_phi = State_phi_2}

State_phi_0.transition_phi = {sa | sa = State_phi_1 || sa = State_phi_2}
State_phi_1.transition_phi = {sa | sa = State_phi_0 || sa = State_phi_2}
State_phi_2.transition_phi = State_phi_2

State_phi_0.L_phi = {pr| pr = Top || pr = p || pr = q}
State_phi_1.L_phi = {pr| pr = Top || pr = r || pr = q}
State_phi_2.L_phi = {pr| pr = Top || pr = r}
}

inv def_of_Top_Bot{
// definition of Top
all sg| Top in sg.L_G
all sp| Top in sp.L_phi

// definition of Bot
all sg| Bot !in sg.L_G
all sp| Bot !in sp.L_phi
}


assert non_phi_hold{
some sint: final_states_int| sint in init_states_int.transition_int
}

}//end model sts
```

Contents