UNIVERSITY OF WEST BOHEMIA
FACULTY OF APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Diploma thesis

Design of Reliable Systems Using Formal Methods

Plzeň, 2012 Peter Cipov

I hereby confirm that the work submitted is entirely my own and does not involve any additional human assistance. All quotations and paraphrases but also information and ideas that have been taken from sources used are cited appropriately with the corresponding bibliographical references provided. The same is true of all drawings, sketches, pictures and the like that appear in the text, as well as of all Internet resources used.

In Plzeň 15.5.2012 Peter Cipov

Abstract

The formal methods have been researched over several decades and they became very powerful tools. There are varieties of approaches from purely theoretical to solutions that are nearly automatic. Even so, formal checker is not daily bread of every software developer. In the industry formal checkers are used only for very expensive, mission critical software where only a few program parts, separate algorithms or protocols are transformed to model which is under check. This thesis tries to use other promising way of formal verification that aims to be used on much bigger units than a single algorithm. This thesis is focused on using such checker for purposes of CORDET project which is oriented on development of onboard satellite systems. It examines and demonstrates current checker characteristics.

Table of Contents

Table of Figures

1. Introduction

This thesis tries to find a solution for checking of functional properties of software developed by using CORDET methodology. Most of those properties can be viewed as state machine constrains. Checker role is to guard those constrains report any violations.

This thesis aims to find proper functional checker and adapt this solution to be as much user friendly as possible.

1.1. Organization of the Thesis

In the chapter 2, we introduce CORDET project, its methodology and domain specific Action Language. We introduce what are formal properties and what types of them are used in CORDET.

Chapter 3 is more formal. It introduces model checking as mathematical area – it formally defines primitives of model checking like Kripke structure, Büchi automaton and Linear Temporal Logic (LTL) itself. It defines when LTL formula is broken and what is considered as a counter example. This chapter also mentions one of the biggest problems in this field – State Explosion Problem and describes approaches how to solve it.

Chapter 4 defines and summarizes processes of choosing best candidate for checking of CORDET functional properties. The selection considers requirements like active development, LTL support or OOP support. It describes the reasons for choosing Java Pathfinder (JPF) as the best candidate.

Chapter 5 describes how JPF works and more importantly how JPF-LTL – LTL module for Java Pathfinder works, what are its internals and limitations.

Chapter 6 introduces JPF integration to Eclipse environment.

In Chapter 7, results of test case are discussed. A Telecommand example was used as a demonstration example, on which formal properties were checked. Chapter contains discussion what features of checker are still missing and what are the next steps for development of JPF-LTL.

2. Domain Overview

2.1. Brief Introduction to CORDET

CORDET (Component Oriented Development Techniques) is a project that approaches software development with model-driven development way [1] with high focus on software reusability. Project defines a generic architecture for on-board satellite software applications. This process contains of two steps.

First step is the definition of reusable software assets organized in a framework. This framework is a set of instances and classes expressed as UML models that comply with two UML profiles:

- FW Profile – which covers functional aspects
- HRT-UML profile – which covers timing aspects

These features are captured to Family Model that captures all available features and their legal combinations. To the System Model are selected only those that can be used on particular system during customization.

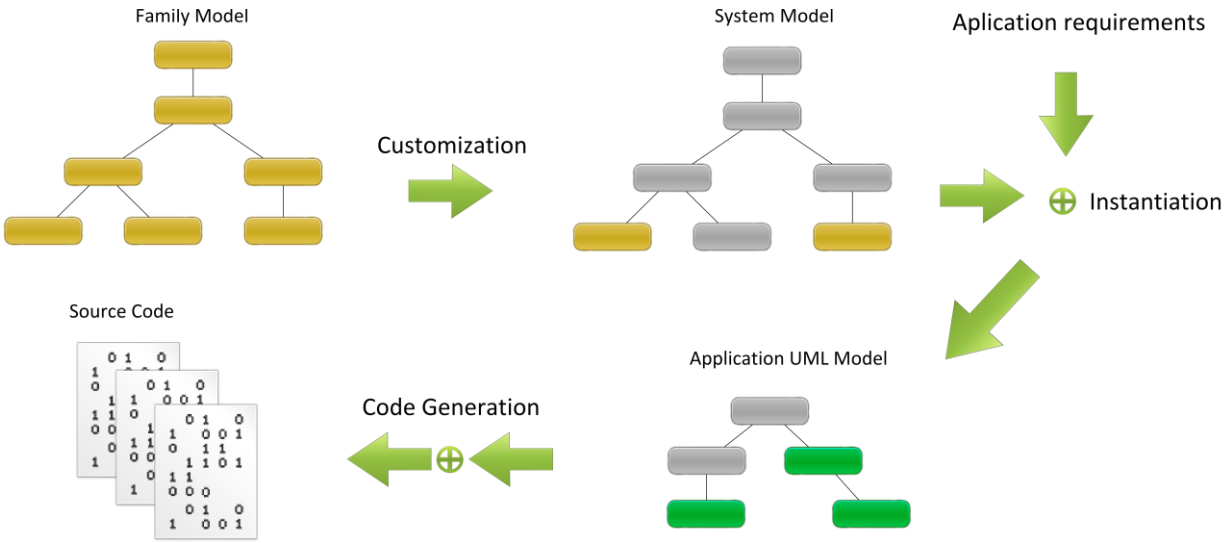

Figure 1 – Process of developing project by CORDET methodology.

The second step is the usage of those reusable building blocks according to application requirements in the Application UML Model. This is done by extension and implementation due to used OOP paradigm. Class-diagrams and state diagrams are result of this phase. Class diagrams defines skeleton and state diagrams defines behavior of the system. An action language acts as glue that connects these two diagrams. It is a fact pseudo-language that can be written directly to UML model properties (method bodies). Last stage is code generation to specified language. For more detailed discussion please refer to [2]

2.2. Action Language

UML model does not contain only UML diagrams (class diagrams, state chart diagrams). Models also contain attributes that are written implicitly in statically typed programming language called Action Language [3]. It's a simple programming language. It is not as expressive as Java or C++, but it supports OOP paradigm. It is not full-fledged OOP language, but it serves as glue between class diagrams and state charts. It is used to capture business logic.

```
1  for I in this.tcList'RANGE loop
2          if ( not(this.tcList[I] == null)) then
3                  if (this.tcList[I].isAborted  ||  this.tcList[I].isTerminated) then
4                          this.tcList[I].putOutOfUse();
5                          this.tcFactory.release(this.tcList[I]);
6                          this.tcList[I] := null;
7                  else
8                          //Print("TelecommandManager: EXECUTING branch entered...");
9                          this.tcList[I].accept();
10                         this.tcList[I].ready();
11                         this.tcList[I].execute();
12                 end if;
13         end if;
14 end loop;
```

Figure 2 – Example of action language from class TelecommandManager, body of a method runOneCycyle. [4]

As we can see in a Figure 2, Action language supports loops, if statements, virtual method execution, expression computation and assignments. It does not support declaration of local variables – it uses class or method input parameters. New object can be created by using keyword new. Other non-standard feature of this language is lack of support of nested method invocation, like a(b(c())). Parameters of method can only be non-calling expressions.

2.3. Formal Properties

In the above section we have defined methodology and processes. To check formal properties we have to somehow insert stage where we would define what kind of formal properties we want to check. In process line, visualized in Figure 1, there are two places where it would be preferable to have such options. It is in stage of preparing framework, where we want to check global properties. And also in the stage of design of application UML model, where typical use-case would be checking on more fine-grained level.

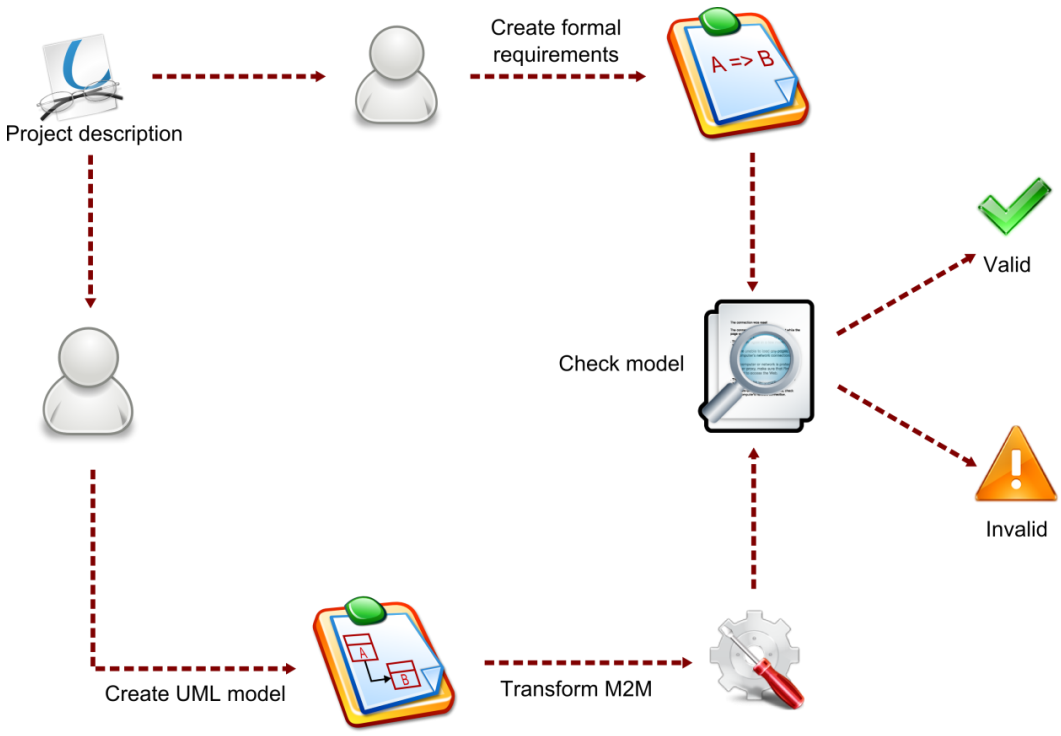

Figure 3 – Simplified visualization of checking formal properties

Figure 3 shows how model checking process would look like. Programmer has to create UML model where all properties of the system are stored. This is already done in these two CORDET stages we have mentioned a in section 2.1. User has to create formal specification of properties that need to be checked. An example of such requirement can be:

For state machine with three states: "initializing", "running", "stopped"

If component is in a state "running", then it can never switch back to "initializing".

If component is in a state "running", then method start cannot be executed again.

Such requirements and system UML model are transformed to DSL (domain specific language) of model checker. And checking is solely left on this tool. It returns positive result if checking was correct (formal properties hold) and negative result with counter-example (this example does not hold formal properties) if checking has found some mistakes.

3. Model Checking

Developing of software programs is in general connected with unwanted products features. We call them bugs. Programmers may devote more than 60% of their time on testing and debugging in order to increase reliability of software. A lot of automated tools has been created (PDM, FindBugs) that statically analyze code for the known and frequent problems. Yet, serious errors still afflict many computer systems including systems that are safety critical, mission critical, or economically vital. Some examples of famous failures caused by software bugs are:

- Ariane 5 floating point overflow (1996, approx $500 million) [5]
- Patriot missile failure (1991, 28 dead) [6]

More examples can be found here [7]. The US National Institute of Standards and Technology (NIST) has estimated that programming errors cost the US economy $60 billion annually [8]. According to NIST, 1/3 of the costs could be avoided by using better software development methods.

An alternative approach to classic testing method uses fact that programs and more generally computer systems may be viewed as mathematical objects. So one can try to give formal proof that the program does what its creators wants him to do. This line of study is referred to as formal methods.

At the beginning of model checking in 1980's, axiomatic verification was the mainly used paradigm. It consists of manual proofs of correctness for sequential programs. The basic principles of this approach were established by Robert W. Floyd in [9] and later extended by C. A. R. Hoare [10]. The Floyd-Hoare framework was a significant success and inspired many researches for further work, e.g. proof systems were proposed for new programming languages and constructs. Proofs were made for small programs.

However this framework turned out to be too complex for adoption as a mainstream technology. It did not scale up to "business programs". These proofs can involve the manipulation of extremely long logical formulae. It turned out that constructing of proof was tedious and error-prone work for humans. In practice it was unusable, because it turned to problem of "checking the proof that was checking the program". Therefore a different approach was proposed. It transforms algorithms to a graph with states and transitions that are traversed and checked over constraint written in Linear Temporal Logic (LTL). The state graph that is explored is called state-space. If the state-space is finite, it can be explored completely. State-space exploration with LTL constraints turned out to be the most successful strategy for analyzing and verifying concurrent systems.

Many different types of properties can be checked by exploring its state space: deadlocks, dead code, starvations, violation of user specified properties, etc. It has been studied intensively by scientists. Variety of different approaches was proposed over past 20 years. Moreover, verification by state-space exploration became fully automatic: no intervention of the designer is required. This is the crucial feature for further usage in the industry.

3.1. Temporal Logic

Originally it was developed by philosophers and later it was suggested that it might be useful in computer science [11]. As mentioned before temporal logic is more suitable for more complex programs, which are cyclic, non deterministic or concurrent. Its high degree of expressiveness and flexibility is considered to be the big advantage of this approach.

To describe behavior of a program and his transitions, temporal logic extends basic logic operator set (or, not, xor, and …) with those that can express time behavior (e.g. if variable A has value 5, variable B must be always greater than zero).

3.2. Kripke Structure

Kripke structure is form of non-deterministic automaton proposed by Saul Kipke [12]. Basic idea is to capture states of computing machine without adding unnecessary complexities. It is graph whose nodes represent reachable states of the system and whose edges represent state transitions. In more formal manner we can define it as:

$$K = (S, I, T, L)$$

(1) S is a finite non-empty set of states of K
(2) $I \subseteq S$ is non-empty set of initial states of K
(3) $T \subseteq S \times S$ is a set of transitions of K
(4) $L: S \to \Lambda$ is a labeling or interpretation function that assigns properties of system to automaton state.

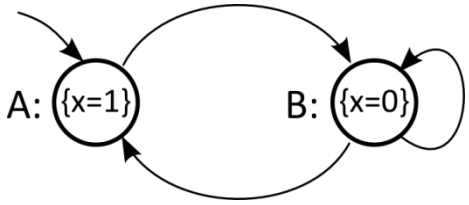

Figure 4 - example automaton for S ={A, B}; I={A}; T={(A,B); (B,B)}; L={ (A, {x=1}); (B, {x=0}) }
products for input path p = A, B, B, B execution word w = {x=1}, {x=0}, {x=0}, {x=0}

3.3. Computation Trees

Computation tree notation captures all possible behavior of Kripke structure. It is a representation for computation steps of non-deterministic automaton on a specified input. It can be represented as a tree graph where parent is labeled with current state S and it is pointed to children that are marked with label of all reachable states of S.

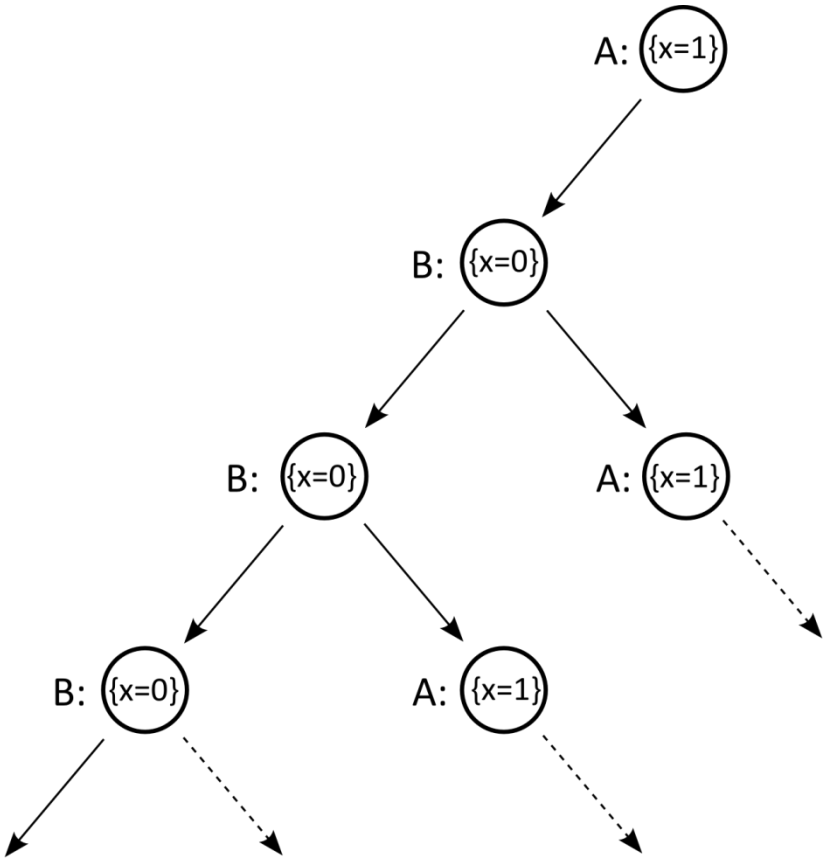

Figure 5 - Computational Tree for the Figure 4

3.4. Linear Temporal Logic

Linear Temporal Logic (LTL) is used to describe properties of a path in a Computation Tree. For example "for some state on a path", "for the next state", "for all states" can be expressed. We define new extended set of temporal operators:

- O A - A is true for the next state
- ◊ A - A is true for some state
- □ A - A is true for all states
- A Ʊ B - A is true for all following states until B is true
- A *R* B - B is true for all following states until it is released by A

The semantics of these operators is shown in Figure 6. In the simple way, they can be explained as follows.

O A – It means "next". It holds, if the next state holds on the path.

◊ A – It means "eventually". This formula holds if A eventually occurs, i.e., A holds at some state of the path.

□ A – It means "always". This formula holds if A holds on all states along the path.

A $\mathcal{U}$ B – It means "until". This formula holds, If A holds until B occurs, i.e., there is some state which holds B and all states before has to hold A.

A R B – It means "release". This formula holds if B holds globally on the path, or A occurs before the first state at which B is violated.

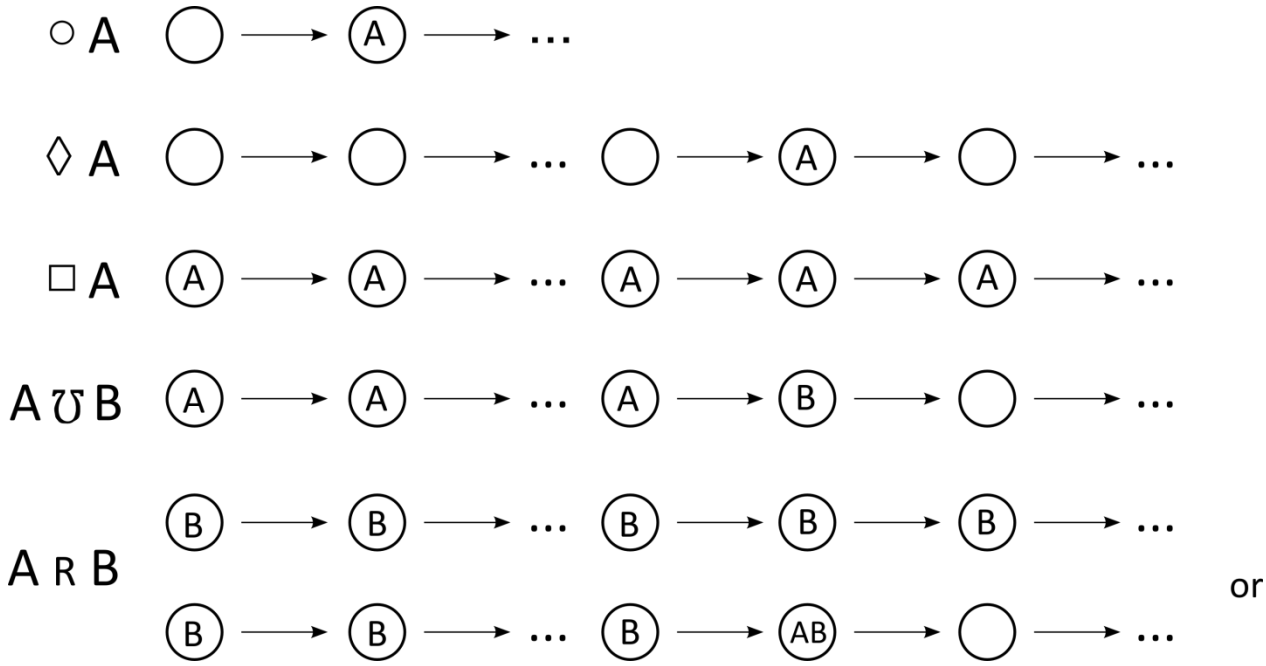

Figure 6 - Semantics of temporal operators

3.5. Büchi Automaton

Programs can be finite or infinite, therefore we have to use ω-automaton that is capable to recognize infinite words, i.e. Σ^{ω} , where Σ is finite alphabet, and ω denotes infinite number of iterations. The simplest class of ω-automata is Büchi automaton [13]. It is a fivetuple:

$$\mathcal{B} = (\Sigma, S, \Delta, I, F)$$

(1) Σ is the finite alphabet
(2) S is finite set of states
(3) $\Delta \subset S \times \Sigma \times S$ is the transition relation
(4) $I \subseteq S$ are starting states
(5) $F \subseteq S$ are accepting states

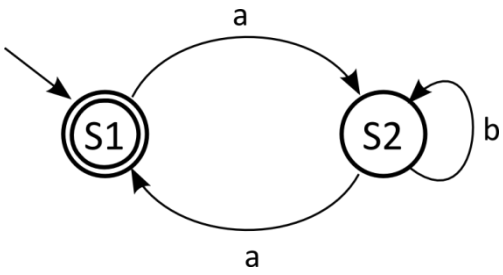

Figure 7 – A Büchi automaton

Figure 7 shows an example of Büchi automaton for Σ= {a, b}; S= {S_1, S_2}; I= {S_1}; F= {S_1}; $\Delta = \{S_1 \xrightarrow{a} S_2;\ S_2 \xrightarrow{b} S_2;\ S_2 \xrightarrow{a} S_1\}$. It can be also written using the ω-regular expression $(ab^*)^\omega$. Namely, an infinite repetition of single a, followed by finite (possible empty) sequence of b.

The language accepted by Büchi automaton is called ω-regular, because such language can be described using expressions containing letters from alphabet Σ, and the operators + (union), concatenation, * (iteration) and ω (infinite iteration).

3.6. LTL Checking

The biggest advantage of approach of using automata to describe both systems – model and temporal specification, is that they are represented in the same way, so it is much simpler to understand and compute. Kripke structure $K = (S, I, T, L)$ where $L: S \to \Lambda$ can be directly transformed to Büchi automaton:

$$\varkappa = (\Lambda;\ S \cup \{a_0\};\ \Delta;\ \{a_0\};\ S \cup \{a_0\})$$

Δ is set of transitions constructed from:

- $(s, \sigma, s') \in \Delta\ if\ (s, s') \in T;\ \sigma = L(s')$
- $(a_0, \sigma, s') \in \Delta\ if\ s' \in I;\ \sigma = L(s')$

Also LTL expression has to be translated to Büchi. But this transformation is much more complex and therefore out of scope for this work. For more discussion refer to [14]. In further text I will refer to transformed automaton as Ψ.

The system $\varkappa$ satisfies specification Ψ when $\mathcal{L}(\varkappa) \subseteq \mathcal{L}(\Psi)$ – language produced by automaton $\varkappa$ is a subset of language produced by automaton Ψ. This mean, each behavior of modeled system is included in behavior set of specification Ψ. The inclusion above can be also rewritten as:

$$\mathcal{L}(\varkappa) \cap \overline{\mathcal{L}(\Psi)} = \emptyset$$

Where $\overline{\mathcal{L}(\Psi)}$ is the complement of $\mathcal{L}(\Psi)$, i.e., $\Sigma^\omega \backslash \mathcal{L}(\Psi)$. This expression means there is no behavior in $\varkappa$ that is disallowed by Ψ. If the intersection is not empty, any element in it corresponds to counter-example – it is a word C, i.e. Σ^ω, that can be accepted by both languages: $C \in \mathcal{L}(\varkappa) \ \wedge\ C \in \overline{\mathcal{L}(\Psi)}$

This formula also gives us an elegant recipe how to check LTL property on-fly in a running program. At the beginning we will construct negated Büchi automaton Ψ. Program instructions are input for Ψ. After every instruction we ask automaton whether it is in accepting state. If no, program is correct and execution continues. If yes, this means intersection is not empty, therefore path to this point can be considered as counter example.

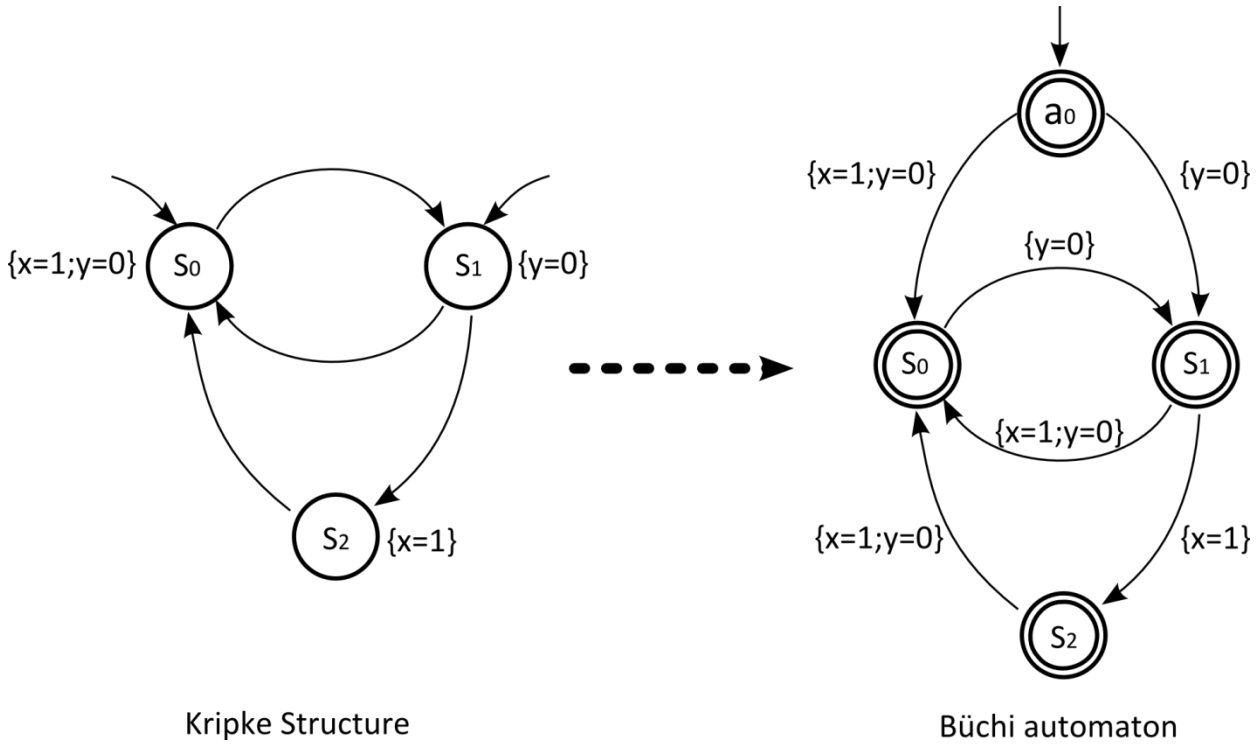

Figure 8 – Kripke structure and its corresponding Büchi automaton after transformation.

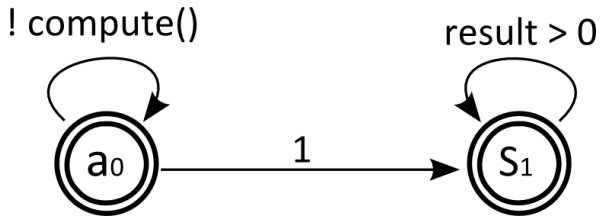

Figure 9 – Büchi graph of transformed LTL specification: $\square(\boldsymbol{compute}() \rightarrow \mathrm{O}(\boldsymbol{result} > 0))$.
It means: Check for all paths whether after calling method compute result is always > 0

3.7. State Explosion Problem

In real case scenarios, state space of computation tree can be huge. In this work it is marked as the language of all words that holds the Kripke structure limitations, i.e., $\mathcal{L}(\mathcal{K})$. But this space can grow exponentially when checker is traversing program, especially in concurrent systems, with many threads, that share the same critical section. It can be so huge that it can be easily bigger than memory of a computer. At first sight, the state

explosion problem looks so formidable that it seems to make state space methods useless for verification of real systems. However, the great advantages of state space traversal have motivated researchers to find a way to reduce the problem.

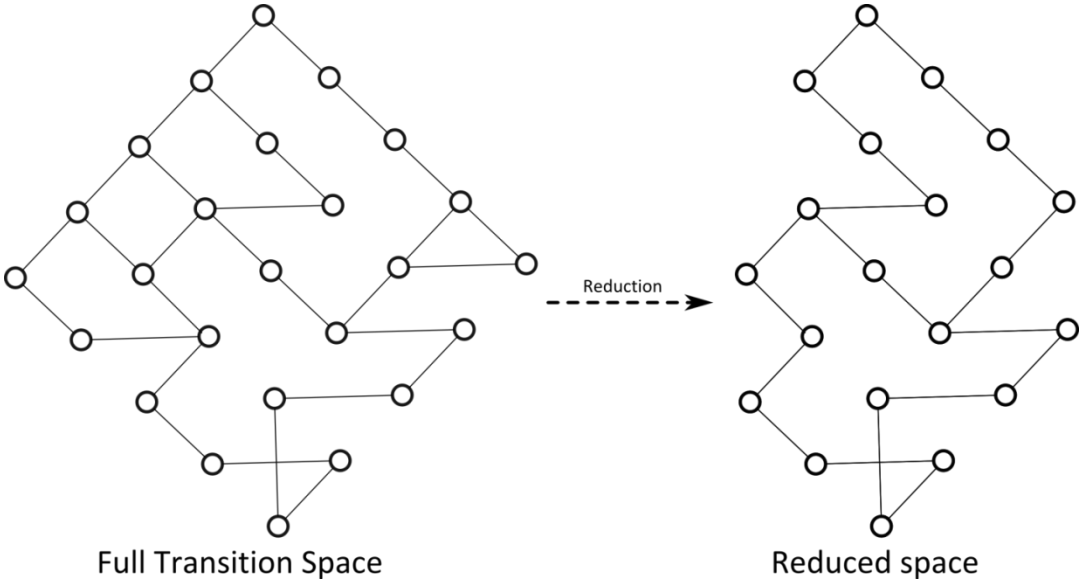

Figure 10 – Majority of checked model spaces have to be reduced to equivalent space. A lot of redundant spaces and transitions can be removed.

3.7.1. Symbolic Analysis – Binary Decision Tree Approach

Fundamental breakthrough in field of state space reduction was mad by Ken McMillan in his doctoral thesis [15]. McMillan argued that larger systems could be handled if transition relations were represented implicitly with order binary decision diagrams (OBDDs) [16]. By using the original model checking algorithms with the new representation of system transitions, he was able to verify some examples that had more than 10^{20} states.

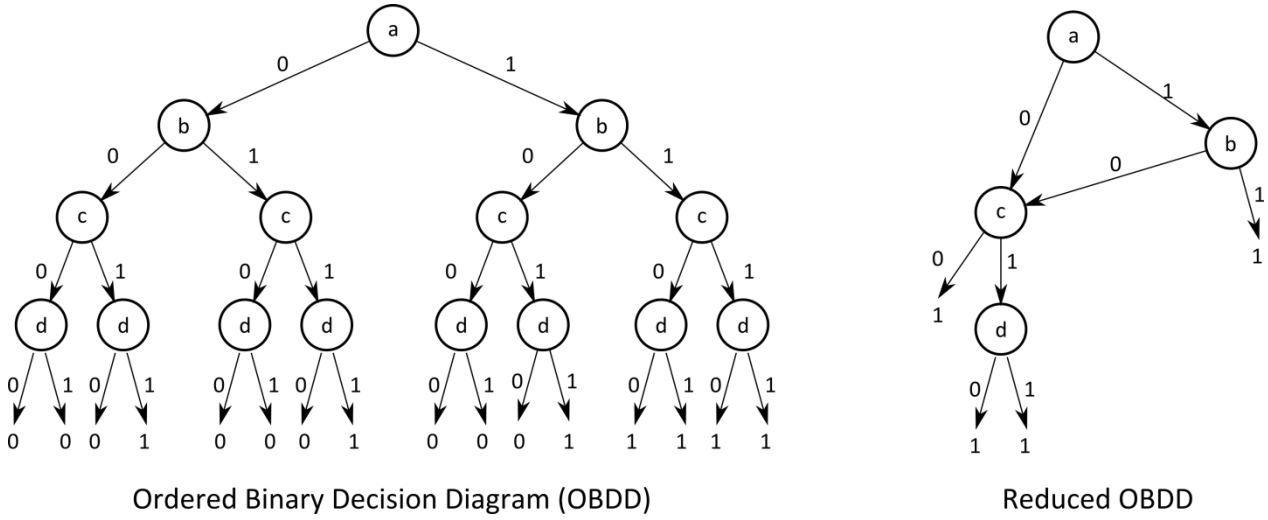

Figure 11 – Representation of Boolean function f= ab + cd with OBDD. It is an equivalent of table with all possible inputs and results. It can be reduced to equivalent diagram but with far less nodes.

McMillan represented Kripke structure with Boolean functions and then he reduced them with OBDD. The key idea of this method is to combine equivalent subclasses.

However, reduction ration of this method is highly unpredictable and in some special cases it can result in complexity problems. [17]. This algorithm is a base idea of model checker SMV [18].

3.7.2. Partial Order Reduction

This method makes use of fact that there exist independent transitions in the program that are commutative, i.e., it does not matter in which order they are executed, and after execution verification always ends in the same state. Such transitions are common in concurrent systems, where every thread modifies its own independent memory part, so other threads are never influenced.

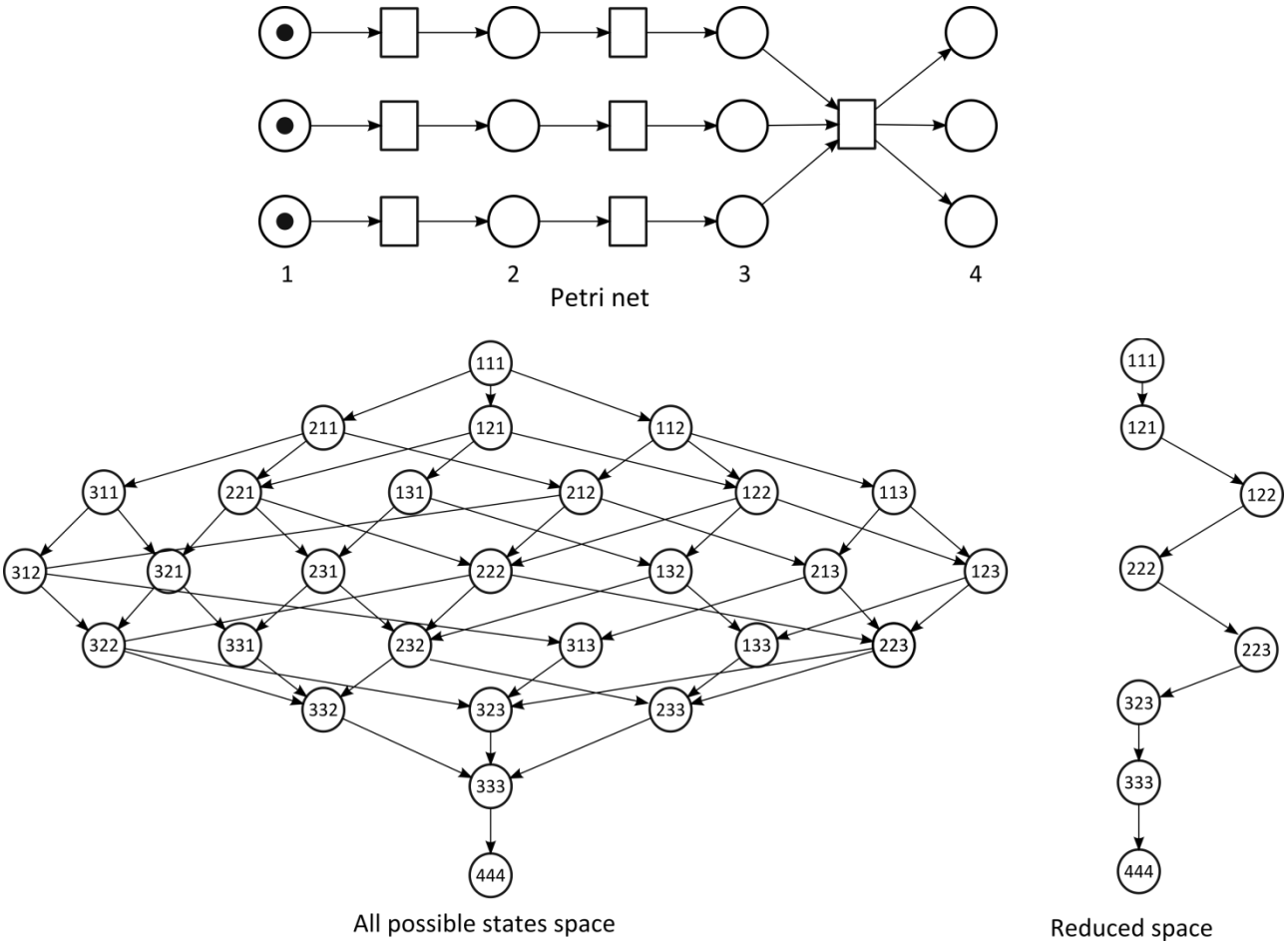

Figure 12 – En example of POR reduction on model of barrier and 3 threads (modeled with Petri net). States space contains huge amount of states and it will grow exponentially with more threads in the model. But if we apply POR, states count will dramatically drop.

As shown in Figure 12 we can prove correctness of those all possible states just by proving correctness of reduced space. Big advantage of this approach is that is scales very good with more and more threads in the system. Reduction ratio is better when system contains threads with independent parts. For more detailed discussion refer to [19].

4. Tools for Formal Verification

4.1. Checker Requirements

For selection of model checker I have selected these requirements:

- It has to be actively developed.
- It has to support LTL logic verification
- It should have some way to implement OOP without huge reimplementation of the model checker
- CORDET model contains action language (a pseudo language). There has to be some way to translate it to checker input model.
- model language should be easy to read, easy to write (nice to have feature)
- it should be already used on some industry case

Name	Development	LTL support	OOP	Model Language	Usage
Bandera	Discontinued	Yes	Yes	Java	Academic
SPIN	Actively	Yes	No	Promela	Industry
PAT	Actively	Yes	No	CSP#	Academic
Java PathFinder	Actively	Yes*	Yes	Java Bytecode	NASA
NuSMV	Sporadic	Yes	No	SMV	Academic
Prism	Actively	Yes	No	Prism	Academic
DiVinE Tool	Actively	Yes	No	DVE	Academic

Table 1 – List of model checkers

In Table 1 is a list of reviewed model checkers. A lot of them are solely academic attempts to demonstrate examined approaches and they are used as study examples for students. Two of them matured enough to be used in the industry. First of them it is SPIN. It is considered to be a leader in this area. It is widely used in industrial model checking [20].

The second one Java Pathfinder is interesting with its practical approach. It does not checks some model, but it checks directly Java Byte Code, that is pure program representation. With this approach already compiled java programs can be checked.

These two candidates I have chosen for more detailed selection.

4.1.1. Action Language Issues

As was mentioned in domain overview (section 2.2) model also contains "hard coded" parts in the Action language. This moves solution from standard model checking rather to runtime verification. Because of OOP paradigm polymorphism, we have to overcome additional nondeterminism that this code can contain. Static analysis fails in this case, so this code has to be somehow interpreted and executed. Keyword like "this" contains always different reference and what exactly kind of reference its clear only after the execution. Tools have to have solution for these parts.

4.2. SPIN

Simple Promela Interpreter is a model checker developed by Gerard J. Holzman for verifying communication protocols. As an input it uses model described in language Promela (Process Meta Language). Verification process (Figure 13) contains of describing of system in Promela language. Than this file is parsed by SPIN and C-source code is generated. The Code is compiled by C compiler and executed. The result of execution is a report that all computations are correct or some computations were incorrect and counter example is returned (trail). Trail can be used to analyze program.

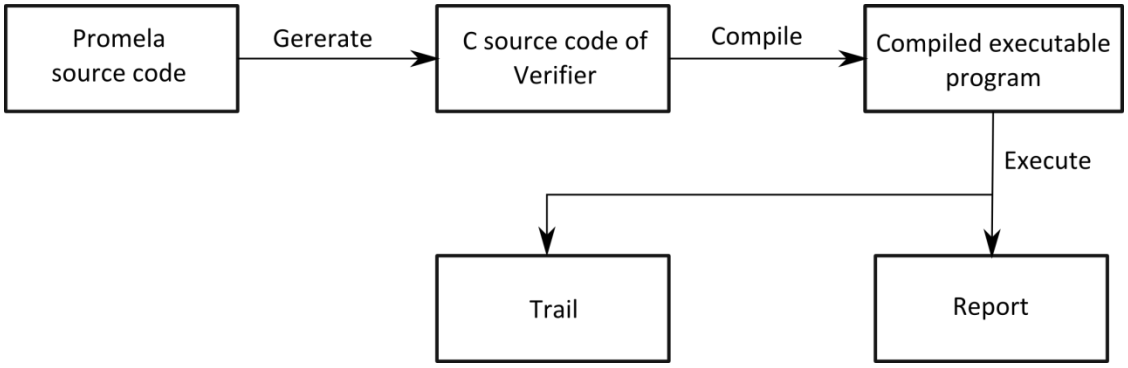

Figure 13 – Typical SPIN usage use-case.

4.2.1. Promela Language

Promela language could resemble C language. It has similar syntax, primitive types, but has several restrictions. There are no functions, procedures or methods in the language. Variables are defined only in global scope. There are only local variables in thread scope. Therefore writing some recursive code can be cumbersome. To imitate local variables we have to use an array to imitate stack or we have to run every recursion in separate thread and communicate via channels (shown in Figure 14). For better code folding, it is possible to use C-like macros that are expanded by spin preprocessor, or inline functions with only global variable scope. There are no object and OOP syntax notions. It means there is no polymorphism and inheritance in Promela. Program can be structured to threads, so concurrent problems like mutual exclusion, deadlock and starvation and be easily simulated.

```
1  proctype factorial(int n; chan result) {
2          int result;
3          if
4          :: (n <= 1) ->      result ! 1;                    /* push  1 result channel and finish*/
5          :: (n >= 2) ->      chan subresult  = [1] of { int };
6                              run factorial(n-1, subresult);  /* run sub task in separate thread */
7                              subresult  ? result;            /* wait for sub result */
8                              result ! n * result             /* count and push result */
9          fi
10 }
11 init {
12         int result;
13         chan subresult = [1] of { int };
```

```
14          run factorial(5, subresult);
15          subresult ? result;
16          printf("result: %d\n", result)
17  }
```

Figure 14 – Promela code example for factorial computation of 5!.

4.2.2. Linear Temporal Logic

SPIN also has a support for LTL formulas checking. Those formulas are composed from atomic prepositions and logic operator and temporal operators:

Operator	Math	SPIN
not	¬	!
and	∧	&&
or	∨	||
implies	→	->
equivalent	↔	<->
always	□	[]
eventually	◊	<>
until	Ʊ	U

Table 2 - Operators used in SPIN for describing temporal formulas

LTL property is passed to verifier via parameter -f in command console. For more detailed setup refer to [21].

4.2.3. Memory Management

To store a huge amount of states that can be generated while tracing of program, Spin uses these techniques:

- Partial Order Reduction – already mentioned in section 3.7.2
- Compressing state space – Byte masking: every state can be broken in a few constants that can be represented as a vector. In memory are stored only differences between already stored and new vector. It uses premise, that Promela do not encourage user to have many local states and lots of them are global. Those can be chunked and referenced via vector. With this approach, spin reduces 62% of memory usage with only 15% overhead [22].
- Bit hashing – SPIN is using byte array and hash function to store whether algorithm has already reached state in depth first state space.

4.2.4. Meeting Requirements

First and one of the most important disadvantages of Promela is lack of OOP semantics. Properties like inheritance and polymorphism has to be translated to Promela syntax. This process requires explicit declaration of table of virtual methods and simulation all OOP behavior in Promela.

Second problem is variable scoping. Promela contains much simpler scope mechanism that is not suited for scoping of variables of modern programming languages.

Dynamic memory allocation, exceptions, floating point numbers, method calls are not included in Promela.

Even so SPIN is de facto standard tool for model checking in the industry. On the one hand, it is a good solution for standard model checking, but on the other hand it is not suitable for models with OOP semantics. It is brings a lot of problems that SPIN was not designed for.

4.3. Java PathFinder

Java Pathfinder is an explicit model checker. Its input is pure Java Byte Code [23]. The first attempt was cross-compilation from Java to Promela. [24]. This approach has two drawbacks:

- **Language Coverage** – as mentioned in mitigation section of SPIN (4.2.4) each language feature of source, must have corresponding language. This is not true for UML model with Action Language and SPIN. For example Promela lacks support for floating point numbers. The only way is to extend Promela language and tweak SPIN compiler.
- **Libraries support** – Standard structures like List, Map, Set, LinkedList can be rewritten in Promela, but what about binary libraries distribution without source code?

The second attempt [25] abandoned Promela and SPIN model checker. To overcome language coverage they created custom JVM based model checker that executes Java Byte Code instead of parsing Java Source Code. This way they overcome both problems – language coverage and libraries support.

4.3.1. State checking

One can argue that this approach do not prove complete program correctness. To check correctness, this approach (run-time verification) needs to execute program. And in a single execution it checks only execution path. Let assume single threaded application implementing McCarthy91 algorithm [26]:

```
1  import gov.nasa.jpf.jvm.Verify;
2
3  public class McCarthy91 {
4
5    public static int mcCarthy91(int n) {
6      n = (n > 100)
7        ? n - 10
8        : mcCarthy91(mcCarthy91(n + 11));
9      return n;
10   }
11
12   public static void main(String[] args) {
13     // int rand = Verify.getInt(0, 101);
14     int rand = 101;
15     mcCarthy91(rand);
16   }
17 }
```

Figure 15 – McCarthy algorithm implementation for Java with single execution path

Program implemented as in Figure 15 will always be executed on single execution path and other possibilities (other than 101) are not checked. For these cases JPF comes with Verify API. By specifying rand as a result of Verify.getInt(0, 101), not directly 101, JPF will backtrack and checks all 102 possibilities. So ones argument

about runtime-verification is right, we cannot properly check programs with single value input. For proper check, tests has to contain all interval that verifier should check. But in case of float numbers it is a problem, because checker is traversing a lot of similar cases without major meaning. For this case JPF comes with heuristic choice generators that can be modified in the configuration.

```
1  code.java contains:
2  …
3  double s = Verify.getDouble("size")
4  …
5
6  in configuration.jpf:
7  …
8  size.class = gov.nasa.jpf.choice.DoubleThresholdGenerator
9  size.treshold = 13250
10 size.delta=500
11 ….
```

Figure 16 – Configuring JPF to use heuristics in choice generation of property size

Configuration shown in Figure 16 will generate set of 3 choices $C = \{t - \Delta; t; t + \Delta\}$. It is possible to create own generator, with own semantics.

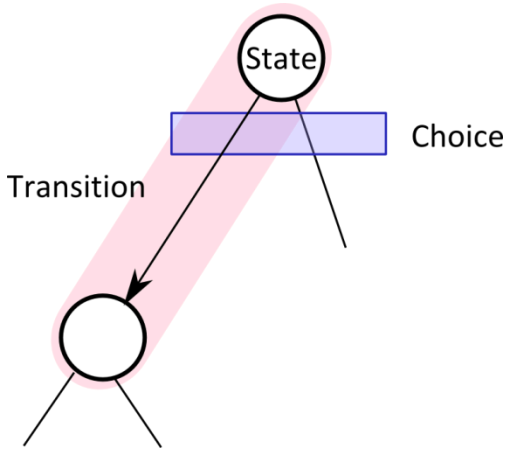

Figure 17 – State exploring in JPF. After every state "choice generator" is applied to choose next transition. If there are multiple states to be chosen, it backtracks and visits all choices.

In Figure 17 is a demonstration of JPF state exploration. It uses choice generators to create all "useful" states to check. If JPF executes instruction of Verify API, it will intercept and apply choice generator that will check all defined paths. This is the example of how powerful approach of JVM checker can be. Checker has total control of instruction flow.

4.3.2. State Space Exploration in Multithreaded Environment

As I mentioned in section 3.7 exploration of state space can be problematic because of tendency of space to be big and to be huge in the concurrent environment. For reducing state space, JPF uses "on-the-fly" variant of partial order reduction method (mentioned in section 3.7.2). JPF analyses instruction and decides whether it

is the one that can be thread "unsafe". Due to JVM is stack based executor only 10% of instruction are scheduling relevant.

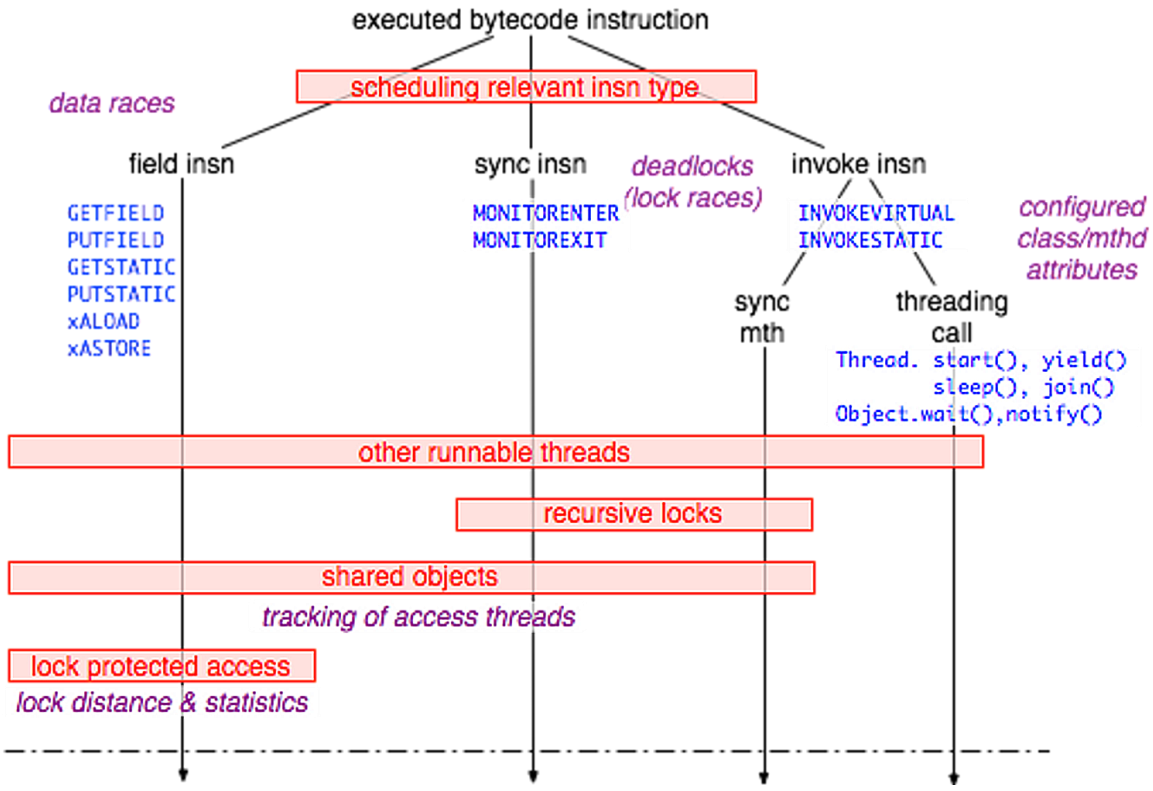

Figure 18 – JPF "on-the-fly" check for scheduling relevant instructions (27)

Scheduling relevance is considered via multiple levels.

- If there no other running thread, the instruction is scheduling irrelevant
- If thread is recursively entering its monitor (monitor inside monitor), the instruction is scheduling irrelevant
- If multiple threads share the same object, then instruction is scheduling relevant

4.3.3. Meeting Requirements

Input language of this checker is Java Byte Code that can be easily created by compiling Java sources, which brings us full power of one of the most popular OOP language that can be used as input language for checker.

Java PathFinder is developed and maintained by The NASA Ames Research Center. It is used for their internal usage [28] [29]. Fujitsu uses it for verification of business process of their web applications [30].

Current version of JPF lacks a proper LTL module. There are several external projects, but they are only proof of concept. They lack of maintenance and active development.

4.4. Tool Selection

For further work I have chosen Java PathFinder as the formal checker. The most interesting feature, I consider is the input language – Java Byte Code. It has included the OOP support. This is the biggest advantage from other tools. Other tools use language with very simple semantic and all OOP features like polymorphism and inheritance are unsupported. This has to be "translated" along model transformation, which would be a huge task itself. JPF in his first release tried to convert Java to Promela, but this way was discouraged due to too much incompatibility between them. Translating model to input language where floats, inharitance, functions are unknown features, is a way also discouraged by myself. Java has those features and a lot of more, like concurrent programming features and a big standard library of data types like lists map and sets.

The problem is in missing LTL module. Some work was already done [31] but project is discontinued for some time and is not working anymore. The project has several problems. It was not maintained for a long time so it was not compilable with current JPF code base and all its tests did not hold.

5. JPF-LTL module

5.1.Introduction to Java Virtual Machine

Java Virtual Machine (JVM) is stack based executor of Java Bytecode (see Figure 19). It means it does not make assumptions about registers and special functions of processor. Therefore it is very easy to implement for variety of hardware platforms. Its primary execution model is stack. All parameters of execution are pushed to the top of it. Every method call starts a new stack frame which holds start point of method execution (all local variable referencing is counted from this position). Then operation is called, which consumes (pops) all stored parameters and pushes result to the top. For example lines 5-10 of Figure 15 are translated in Bytecode shown on Figure 19 and snapshots of stack frame are shown on Figure 20. For detailed JVM description refer to [32].

```
1  public static int mcCarthy91(int);
2    0:  iload_0                    // pushes first parameter of function to the top
3    1:  bipush        100          // pushes constant 100 to the top
4    3:  if_icmple     13           // pops 2 from stack, if first <= second jumps to 13
5    6:  iload_0                    // pushes first parameter of function to the top
6    7:  bipush        10           // pushes constant 10 to the top
7    9:  isub                       // pops 2 from stack; subtraction is pushed to the top
8   10:  goto          23           // jumps to label 23
9   13:  iload_0                    // pushes first parameter of function to the top
10  14:  bipush        11           // pushes constant 11 to the top
11  16:  iadd                       // pops 2 from stack; addition is pushed to the top
12  17:  invokestatic  #2;           // calling int McCarthy91.mcCarthy91(int);
13  20:  invokestatic  #2;          // calling int McCarthy91.mcCarthy91(int);
14  23:  istore_0                   // pops 1 from stack and stores it to function parameter
15  24:  getstatic     #3;          // java.lang.System.out reference is pushed to the top
16  27:  iload_0                    // pushes first parameter of function to the top
17  28:  invokevirtual  #4;         // calling void println(int arg0); pops 1 from stack
18  31:  iload_0                    // pushes first parameter of function to the top
19  32:  ireturn                    // pops 1, returns the vale and pops all function parameters
```

Figure 19 – Java Bytecode of method mcCatrhy obtained from compiled source of Figure 15

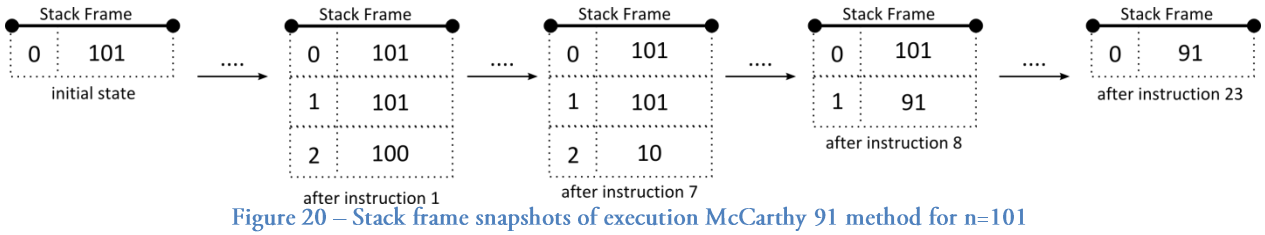

Figure 20 – Stack frame snapshots of execution McCarthy 91 method for n=101

5.2.Bytecode Analysis

JPF is a custom made virtual machine written in Java. It executes input Bytecode and provides variety of triggers and properties for modules, allowing them to intercept execution flow and allows them to react to every instruction execution. JPF-LTL module is also a standard JPF module implementing instruction listener.

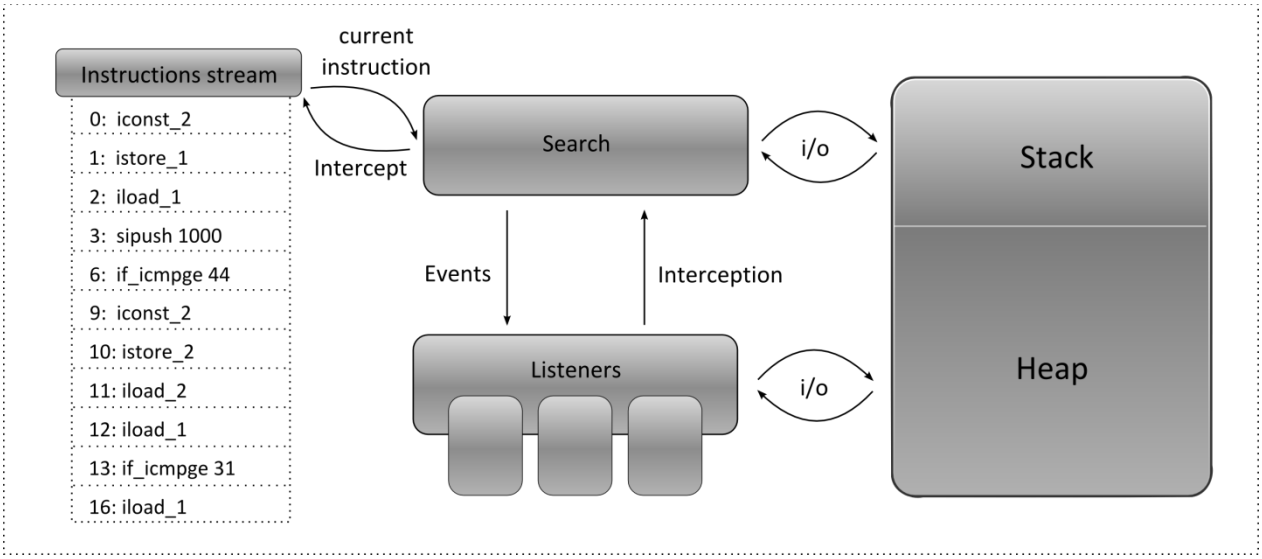

Figure 21 – JPF design. Search module is traversing computational tree and informs listeners about his progress. They can react or intercept current execution – they can intercept current instruction or state in memory.

JPF-LTL module is one of such listeners shown in Figure 21. It listens to JPF triggers and checks current computation tree traverse and memory state.

5.2.1. Temporal Logic Checking Idea

As mentioned in section 3.6 LTL check requires to test whether the following formula holds on every node of computational tree:

$$\mathcal{L}(\varkappa) \subseteq \mathcal{L}(\Psi)$$

$\mathcal{L}(\varkappa)$ can be interpreted as all states that search module can achieve and $\mathcal{L}(\Psi)$ are all states that Büchi automaton can achieve. Every instruction that is executed changes state in a memory. With a trigger after execution, module can check whether Büchi automaton and JPF search holds common state.

Lack of common state indicates breaking the checking formula. There has to be a mutual state accepted by automaton and execution. In a moment of breaking formula, current stack trace and memory can be considered as the counter example.

5.2.2. Module Overall Architecture

Module contains of three basic parts. Business code that contains checking logic, fields cache that holds references to stack (local variables), to heap (object references) or to permanent generation that holds statics, and negated Büchi automaton. Detailed view is shown in Figure 22.

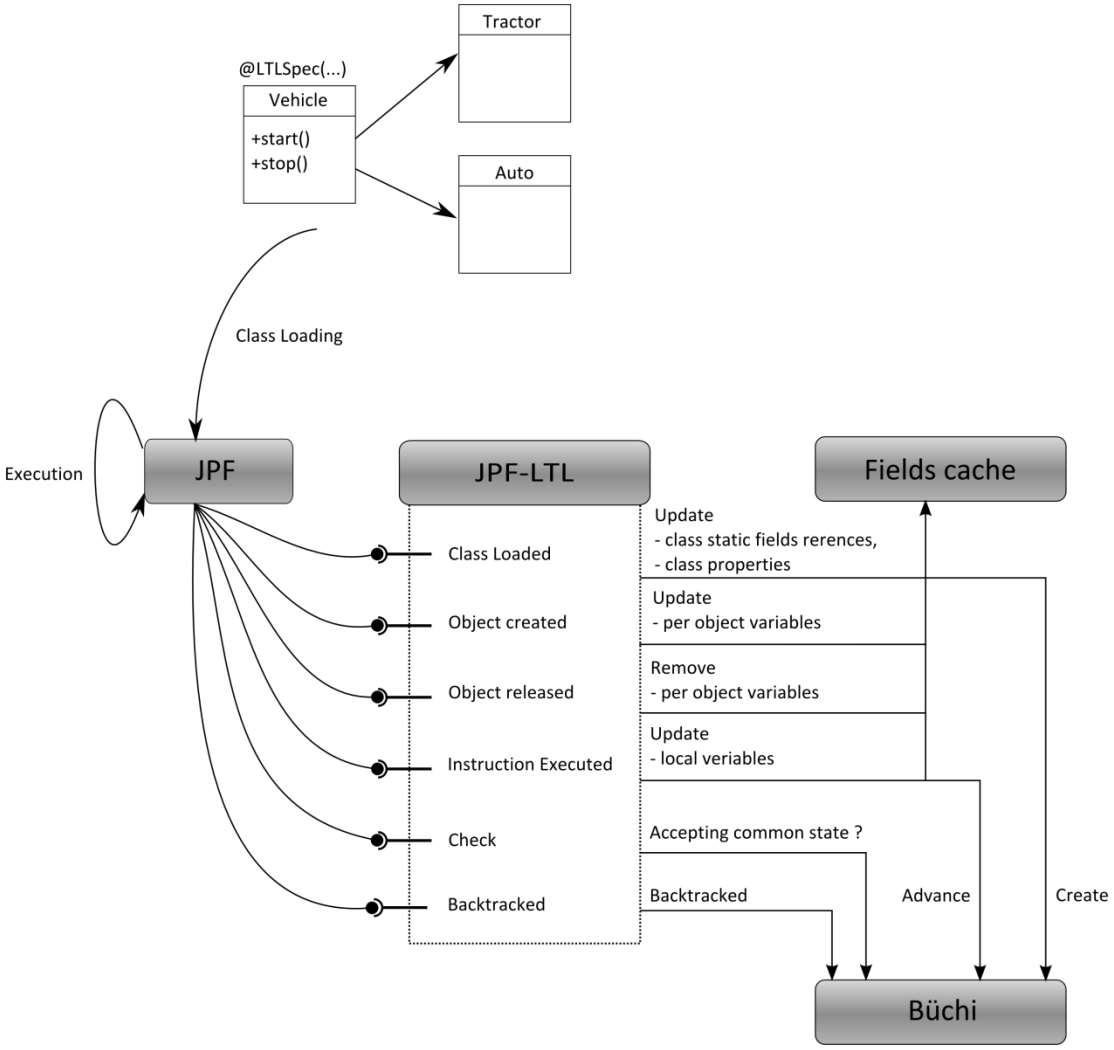

Figure 22 – JPF-LTL module architecture.

Module holds fields cache that holds reference pointers to all variables that are checked (specified in LTL formula). It is used as reference holder for Büchi automaton – it reduces redundant memory lookup. Class can be loaded at any time; hence cache has to be updated after every class load.

The class loading looks up the class for properties, static references and checks whether there is already some formula that refers to it. If a class is annotated with LTLSpec annotation, new formula is added to cache with the new Büchi automaton.

LTL formulae often hold dependencies to other classes that may not be loaded yet, e.g.:

$$\Diamond\ (A.property == B.property)$$

Cache has to be recreated after loading class A and class B, because it would not contain all properties that LTL formula refers to.

When new object is created, it is checked in the cache whether some formula is referring to it. If cache contains object descriptor, reference is added to the cache.

Releasing trigger does a similar action and removes object descriptor from cache.

Instruction execution triggers lookup of current stack frame and checks whether all local variables that are checked by LTL formulae have properly updated references to the stack frame. It also triggers advance of all stored Büchi automata.

After every instruction execution, module checks whether Büchi automaton transition is triggered. If some automaton does not move to the next state, it means temporal formula is broken and error is triggered. Second check is done after last instruction, where module checks whether automaton is in an accepting state. Non-accepting state in final stage means breaking of temporal formula. In term of mathematic formalism:

- The alphabet of language $\mathcal{L}(\varkappa)$ consists of ByteCode instructions
- Prefix is composed of all instructions in the execution order from the program start to the current executed instruction.
- Prefix is considered to be word, after execution of last instruction.
- In the first stage, after every instruction prefix is checked. It is assumed that if the whole word is accepted, then also its prefix can be translated to automaton states. This is not sufficient condition, but only necessary condition. With this simplistic approach most of the formula violations can be caught (like for *every state*, *next state* …). But conditions like *eventually* will be omitted.
- Therefore second stage check is done after every traversal path end. In this moment, it is checked whether word is accepted by Büchi automaton. In this stage full word has to be accepted, otherwise formula is broken.

Backtracking makes module to properly reinitialize automatons – move them a few steps back, so some new branch of computation tree can be checked. Heap and frame-stacks are rolled back automatically by JPF. This is one of the good sides of JPF, it snapshots computation of tree traversals, so after calling rollback trigger, it also rolls back memory state.

Büchi automaton has to be also modified to support backtracking. It is not necessary to copy whole automaton per snapshot, because states are always the same. Only actual states change per time. This information has to be stored in stack structure, every advance pushes new state and the backtracking pops one from the top. This way we can store all necessary traversal history that is required for checking.

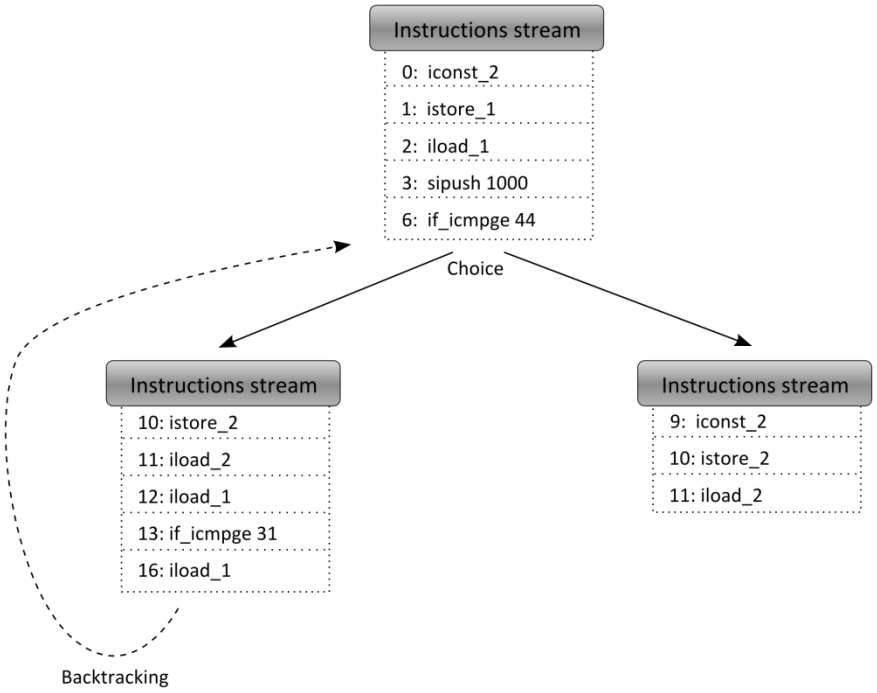

Figure 23 – JPF Snapshot System – Snapshots are not taken after every executed instruction, but before every choice that is made by search module. So it can easy backtrack and check all other computation paths.

5.3.Temporal Logic Formula Specification

Every class can be annotated by @LTLSpec annotations that hold linear time logic formula specification. In this string, it is possible to refer to:

- Method call: package.TargetClass.yourMethod(int,String[][],float)
- Variables
  - Class field– package.TargetClass.yourfield
  - Method variables– package.TargetClass.yourMethod(int,String[][],float).localInit
  - Reflection – sometimes class field is not a primitive time but object. For referencing to its properties reflection is needed: your.TargetClass.yourObjectField{subField1.subField2}
- Relations – (var1 + var2) * var3 - 4 <= 5.3 - var3

Those references can be connected with various operators:

Operator	Math	JPF-LTL
not	¬	!
and	∧	&&
or	∨	||
implies	→	->
equivalence	↔	<->
next	O	X

always	□	[]
eventually	◊	<>
until	ʊ	U
release	R	V

Table 3 – JPF-LTL formula operators

5.3.1. Grammar

LTL formula is specified by formal grammar (see Appendix A) and parser is generated according to the specification.

Every temporal logic specification formula is composed from atoms that refer to some system property like object variable k == 2 or method invocation. These atoms are combined with mathematical operators shown in Table 3.

5.4. LTL Formula Global Scope

All formulae have global scopes. That means per object checking is possible only if there is a single instance of a referenced object in the runtime. Otherwise in the case of multiple instances of referenced type, formula may not hold because Büchi automaton can be set to some unwanted state. This is caused by objects state *localness*. Example is show in Figure 24. Let's assume two instances (A, B) of a single class. Change in an instance A (setting k to value 2) would trigger transition of Büchi automaton (from S0 to S1). The same change in instance B also triggers change in Büchi automaton, but this leads to breaking of formula, because there is no (k==2) transition from S1. The formula assumes that there is only global scope, i.e., current state of Büchi automaton is stored only globally (per class, not per instance) and all instances of a same class can change it. Therefore verification of multiple instances can lead to ambiguous formula violations.

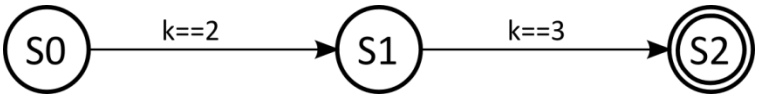

Figure 24 – Büchi automaton with a single accepting state.

5.5.Cache Limitations

The approach of continuous cache creation has its limitations due to class loading. Not all classes are loaded at once, therefore some LTL formulae can be loaded too late by which they can break in unpredictable states. Let's assume following example:

```
1  public class A {
2          int property;
3          public void foo() {
4                  property = 1; //expected break
5                  property = 0;
6                  new B();        // actually breaks here
7                  property = 1; // expected break
```

```
8           }
9           public static void main(String[] argv) {
10                  new A().foo();
11          }
12  }
13
14  @LTLSpec("[]A.property == 0")
15  public class B {    }
```

Figure 25 – Limitation example

Checker is checking that all states should always hold the formula A.property==0. We would assume that LTL formula will be broken on line 4 and definitely on line 7. But it would break after executing of object B creation after line 6. It is due to loading of B class dynamically (and also LTL formula annotation) after object A was made. Therefore there was no object referrer stored in cache after object A was created. Object A was created long before cache was instructed to store objects of class A. Module logic will always miss. This behavior limits annotations to contain only formulae that refer to already loaded classes. Problem in the Figure 25 would be solved by moving of annotation from class B to class A.

6. Tooling Integration

Java Pathfinder is a set of libraries and its configuration is not user friendly – it is complex and large. To fully understand whole capability of this tool takes time. Therefore its integration should be made as easy to use as possible. It should use common use cases. It should be integrated to the same toolset as other CORDET modeling tools.

6.1. Common Environment – Eclipse

Eclipse is a multi-language software development environment, with huge support of programming languages. It also provides RCP – rich client platform for creation of custom modules based on module framework OSGi [33].

6.1.1. Bundle System

OSGi framework uses concept of bundles. Every bundle can be considered as versioned black box with its own lifecycle (activation, computation, and dispose). It specifies dependencies to other bundles and its own interfaces that can be reused elsewhere. Framework handles the dependency graph and provides glue layer between bundles – it manages and provides dependencies. This approach has multiple advantages:

- There can be multiple copies of the same library in the system but in different version. Bundle A can have dependency to library L version 1. Bundle B has also dependency to L but version 2 and both versions are not compatible. In standard environment this problem could be overcome only with refactoring of A or B. In OSGi, these two libraries can be loaded at once.
- This approach tends to even more code reuse that is the number one requirement in huge projects like Eclipse.
- Dependencies are "injected" in the runtime [34]. This means dependencies are not hardcoded via static references and whole application structure is constructed at the beginning. This approach leads to better problem division and encapsulation. Then every bundle can be tested separately, without initialization of whole application.

6.1.2. EMF - Eclipse Modeling Framework

EMF is a modeling framework that is used store application structure, behavior or domain model. EMF differentiate meta-model and the actual model. The meta-model describes the structure of the model (defines class, interface, enum …). Model is the instance of meta-model (class Car, interface Vehicle, enum State). EMF uses xml to store model information. Framework is pluggable and allows describing model via multiple interfaces. First is mentioned XML or it can be annotated Java code. EMF also supports code generation from defined model. However EMF is only a framework. For modeling in UML via GUI, other bundles are needed which provides graphical user editor, like UML2 or Papyrus plug-ins.

6.1.3. Bundle Distribution

Plug-in development in Eclipse can be divided to separate layers:

- Development of bundles – all business logic is coded here.

- Feature composition – bundles can be considered as small pieces of puzzle. Feature is considered to be a set of bundles that provides some functionality that can be described with some use case (UML editor, git repo manager).
- Update site – provides a convenient way how to organize and distribute Features. It can be done by creating tree structure with features as its leaves. Result of update site building is a folder with all bundles and features that can be updated as is to some web hosting.

With update site URL, installation of plug-in is a standard routine, same for standard (eclipse provided) and custom made repositories.

6.2. CORDET Environment

CORDET uses custom domain specific model written using EMF framework to describe program components and state machines. All information is stored in XML files with model description. It uses its own UML profile *FW Profile* [35] that defines stereotypes for classes, interfaces and state machines and also multiple constraints on those stereotypes that can be validated by model checker.

FW Profile is distributed as set of plug-ins that contains:

- Profile containing stereotype meta-model written in EMF format
- Validator of profile constraints

6.3. Custom Stereotype

Stereotype is extension mechanism of UML that allows programmer to extend standard meta-model objects like class or method and specify there a new set of properties that will be added to each instance of those objects. This is a kind of mechanism allows us to add specific behavior to the model without significant refactoring.

In our case we have to "plug-in" formal specification. This can be easily done by creating custom stereotype that extends class object, so formal properties would be applicable to all classes in the model, i.e., formal properties can be applied per class. Programmers work would be the analysis of the domain and creating formal specification on particular classes.

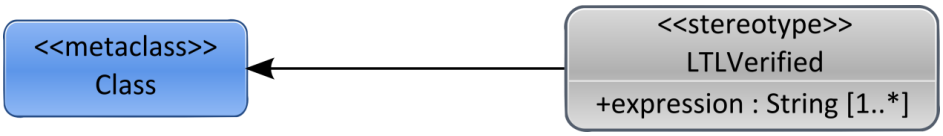

Figure 26 – Stereotype used for writing formal specifications in a UML model

Both CORDET software modeling stages (framework modeling, application instantiation) use UML to describe program behavior. Extending UML can be applied in both cases by using custom stereotypes. Figure 26 illustrates such stereotype. It extends meta-class Class, therefore it is applicable on every class in the model.

It contains property expression that can contain list of formal specifications. This stereotype is distributed as separate bundle. It is not part of FW Profile, but it has to be installed separately via an update site [36].

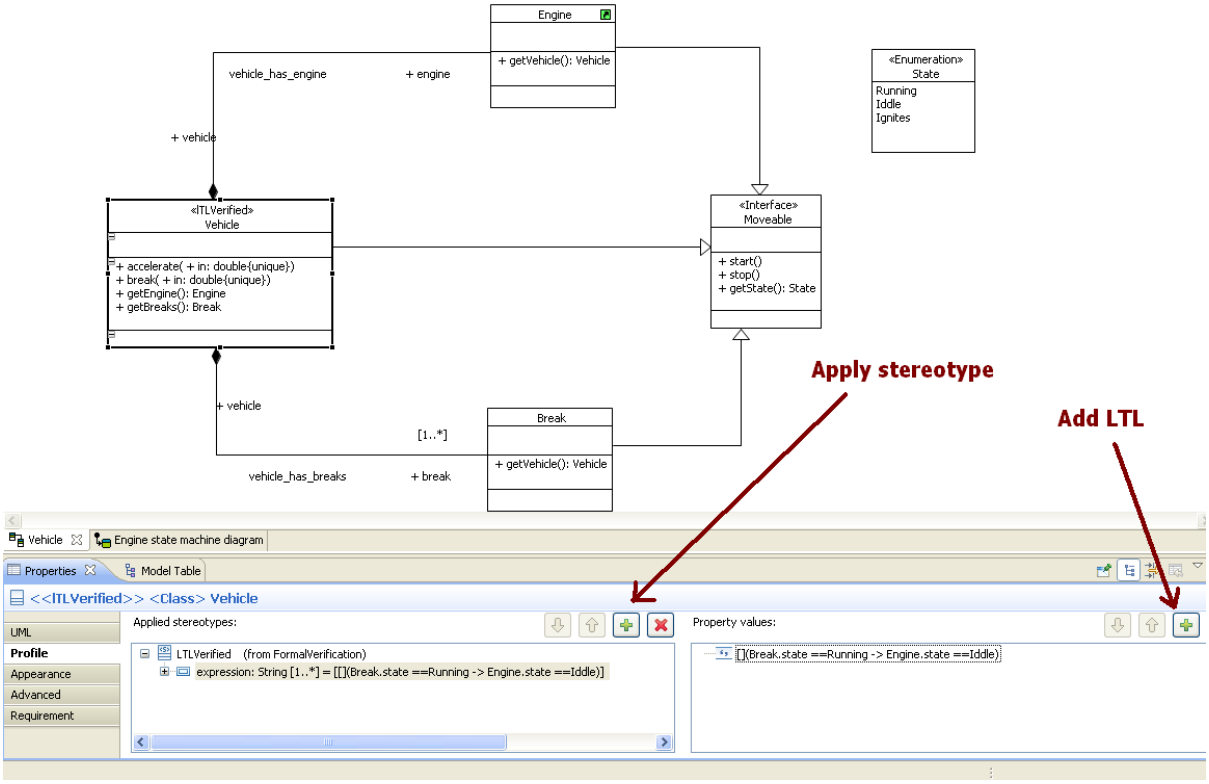

Figure 27 – Example of applying LTLVerified stereotype on Vehicle class in the Eclipse environment.

6.4. Integration of JPF

JPF environment consists of multiple libraries packed to the modules. Every module contains its own set of configuration parameters. This is first problem, that user can find. Framework contains many badly documented features, and it takes to figure it out alone from a source code.

Second problem is framework own distribution model. They distribute multiple modules with runtime dependencies between them. A deep knowledge about framework structure is required to prepare framework with non-trivial structure and setup initial configuration.

Third problem is lack of "user friendly" interface, at best with one big button "Validate".

Most significant problem is lack of any binary distribution. Sources can be found in mercurial repository [37], but no binary distribution is maintained. While developing jpf-ltl module I have encountered changes in source code base (some code was uncompilable) several times. Leaving this problem on users is unbearable.

6.4.1. Custom Distribution

For easy integration of JPF to Eclipse, it is necessary to omit any kind of compilation of framework by user. It brings unnecessary burden when compilation dependencies are broken by development delay of one of necessary module.

Several solutions can be applied, like creating custom Maven repository, or Linux repo. Both solutions are mature but they are unnecessary complex. Easier to maintain and more elegant solution can be used. In basis we can manage with simplistic solution with source code control system like GIT (or SVN, CVS) where the newest version of file can be accessed via simple link to repository (no version number is needed for head version). Plug-in contains hardcoded links to compiled modules (packed as zips). User is only obligated to trigger download framework by pressing button. Framework is then downloaded and configured.

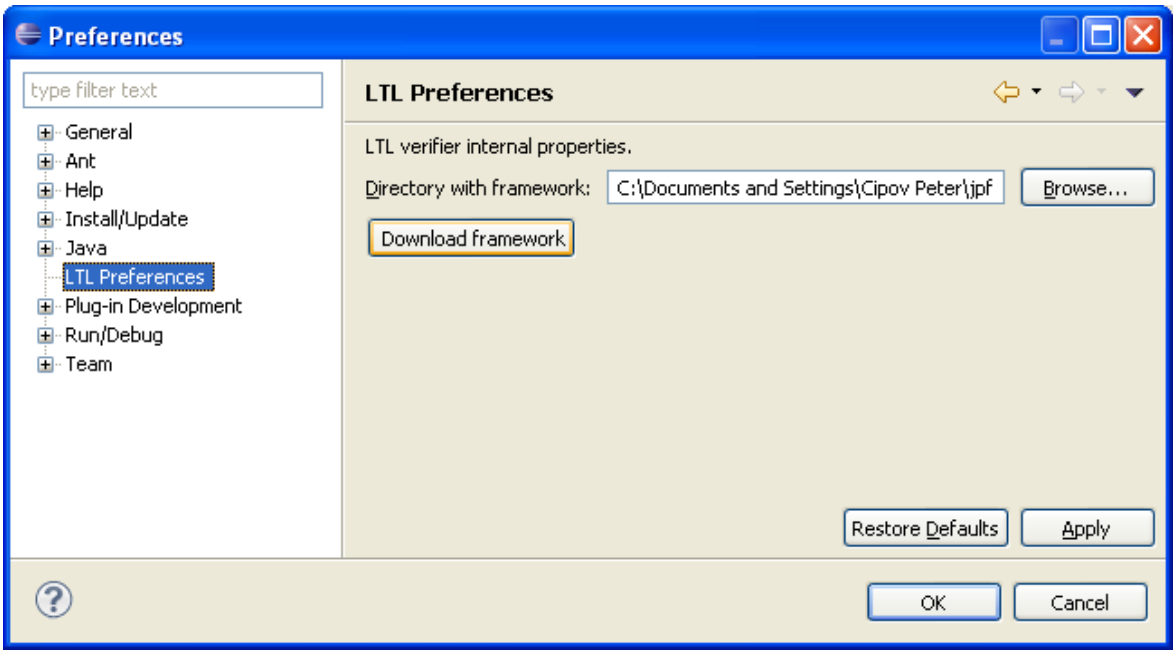

Figure 28 – Post plug-in configuration. User specifies folder and presses Download framework button. Plug-in downloads all modules extract them and configures framework to be fully operating.

Framework distribution consists of binary libraries, examples and source code of these modules:

- jpf-core – contains execution environment
- jpf-symbc – used for formulas evaluating. Its main purpose is symbolic analysis, not used in current version of jpf-ltl, but planned as further extension.
- jpf-ltl - module for verifying temporal properties in finite programs.

6.4.2. Custom Execution

JPF is designed as Java command line utility. It's not user friendly but on the other hand, it can be easily wrapped with proper GUI. The ideal integration would be in the form of one big button "Validate", without some configuration steps. All properties should be default or generated, and they can be changed by user if it is

necessary. "Convention over configuration" paradigm is used as much as possible. It's a design paradigm which seeks to decrease the number of decisions that user has to do, gaining simplicity, but not loosing flexibility.

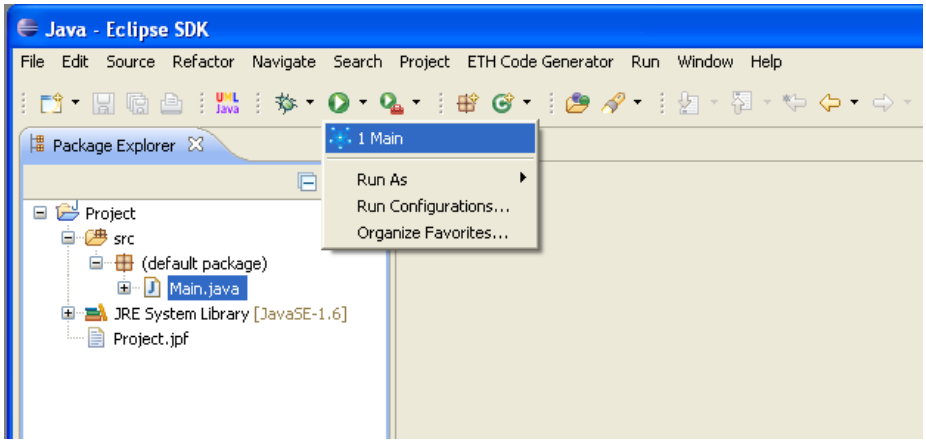

Figure 29 – Convention over configuration example. First validation will generate .jpf file with all configuration that can be altered by user.

To be as unobtrusive as possible, it is good to use "Eclipse way" of execution. There is the big green play button in the toolbar, which runs java execution, code generation, unit tests, plug-ins … Adding LTL validation seems fit to this methodology.

Eclipse contains set of various extension points, where programmer can extend almost every component in Eclipse. Extension points are declared in plug-in configuration xml file:

- org.eclipse.ui.preferencePages – adding custom plug-in preference page, shown on Figure 28. Extension class has to implement interface org.eclipse.ui.IWorkbenchPreferencePage, where it declares form dialog with preferences fields. Loading and storing configuration is led to standard Eclipse libraries that store them.
- org.eclipse.core.runtime.preferences – extension point that register property in Eclipse. By extending org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer class registers all properties and set default value.
- org.eclipse.debug.core.launchConfigurationTypes – creates custom launch configuration. This enables running jpf in a standard way via run configuration (shown on Figure 29).
- org.eclipse.debug.ui.launchShortcuts – enables running verification from .jpf file perspective.
- org.eclipse.debug.ui.launchConfigurationTabGroups – registers custom run dialog, where main class jpf-configuration file can be specified
- org.eclipse.debug.core.launchConfigurationTypes – registers run configuration on specified Java run mode (run or debug)

Execution of verification is executed in a separate process. Verification will consume a lot of memory and can fail due to memory limitation. It is more convenient and secure to run Eclipse and verification separately.

6.4.3. Temporal Logic Formula Visualization

Complex LTL formula can be ambiguous to read, and it can be quite cumbersome to search for mistakes. Also, it can be hard to understand how the final Büchi graph would look like. Because its Büchi graph that makes final decision, whether system is in an allowed state or formal property is broken. Visualization of constructed graph can be an enormous help to realize what exactly formula does.

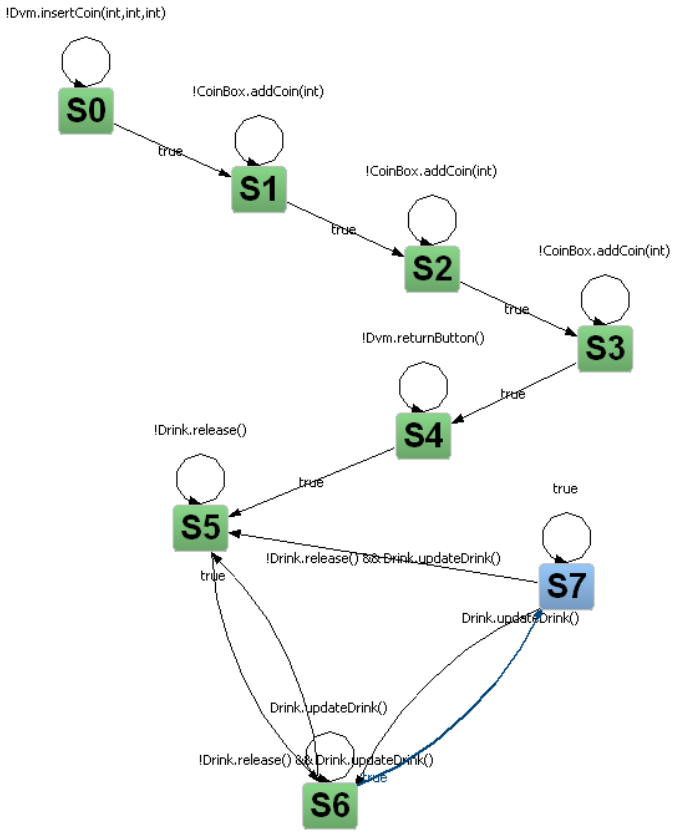

Figure 30 – Visualization screenshot from com.singularity.visualizer plug-in. Green nodes are accepting states. Blue are not accepting.

```
1  [] ( Dvm.insertCoin(int, int, int)
2          -> X ( [] ( CoinBox.addCoin(int)
3          -> X ( [] (  CoinBox.addCoin(int)
4          -> X ( [] ( CoinBox.addCoin(int)
5          -> X ( [] ( Dvm.returnButton()
6          -> X ( [] ( Drink.release()
7          -> X ( <>  Drink.updateDrink()
8          ) ) ) ) ) ) ) ) ) ) ) ) )
```

Figure 31 – An example of little bit more complex temporal logic example. It transformation to Büchi automaton is show in Figure 30

7. Evaluation

7.1. Telecommand Example

For demonstration of formal properties check (mentioned in section 2.3) Telecommand example [38] was chosen as an evaluation example.

This example covers the management of ESA's Packet Utilization Standard (PUS) telecommands in an on-board application. It can be seen as an example of handling commands that are sent to real-time application by an external operator.

As a Standard use-case can be considered: commands are prepared on the earth, then serialized and sent to the satellite over radio link. After reception signal is transformed to bytes. In these bytes satellite actions are encoded. Bytes are de-serialized and Telecommand instances are created. These instances are passed to the central execution component TelecommandManager that stores list of actual commands and executes them in a loop.

This example was originally created for demonstration purposes of CORDET project and therefore it is a good test case for solution proposed by this thesis.

7.1.1. Component Class

Component is meant to be a basic class for all non trivial objects. It defines three states: CREATED, INITIALIZED and CONFIGURED. These states describe the processes of configuration and initialization phase.

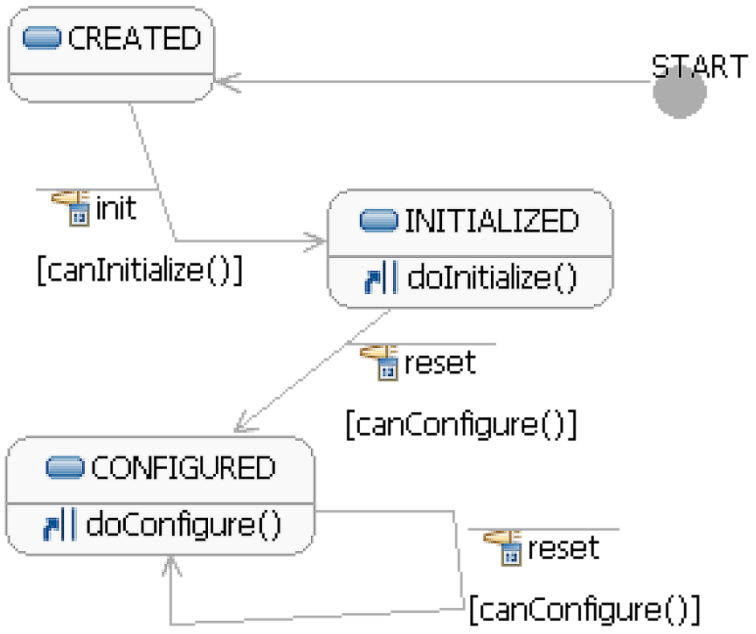

Figure 32 – Component state chart

Components are first *initialized* and then *configured*. Initialization is an irreversible process, where parameters can be set only once. Typically in the initialization internal data structure is allocated.

Configuration is a process where values of parameters can be dynamically changed. After reset, component is configured and it is ready to start its normal operation. Configuration parameters can be updated during normal operation but it will take effect after component reset.

Following properties should be checked on class Component:

1. A component cannot be configured if it has not been initialized.
2. A component can only be initialized if its initialization check is successful
3. A component can only be configured if its configuration check is successful.

7.1.2. ManagedMemory Class

Memory class extends class Component. This class is intended to be a base class for all components with dynamic allocation requirements. Class defines two states: IN_USE and OUT_OF_USE. When component is in a state IN_USE, then it is considered to be allocated.

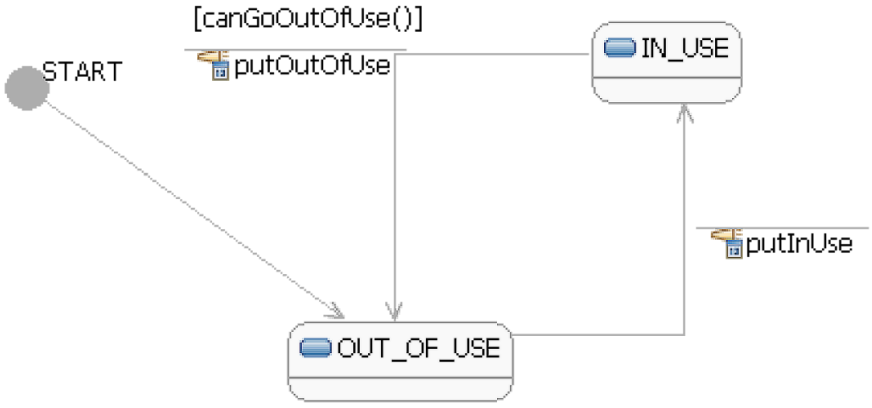

Figure 33 – State chart diagram for ManagedMemory class

Following properties should be checked:

1. A component can be put out of use if out-of-use check allows it

7.1.3. Telecommand class

Telecommand class extends ManagedMemory class, because it is meant to be created once in runtime by abstract factory and then reused.

Telecommand contains 7 internal states (shown on Figure 34 – Telecommand state chart) and performs many checks. Immediately after being loaded to TelecommandManager, an acceptance check is performed. If

acceptance check is passed, the telecommand enters the ACCEPTED state. It remains in this state until it is ready to start. Readiness to start execution is encapsulated in the ready check.

When ready check is passed, component enters the READY state and performs the start action. Then the telecommand performs start check that determines whether execution can start.

If start check is passed, component enters IN_PROGRESS state. It can be entered multiple times. On each entry doProgress action is executed. After each step progress-has-failed and progress check is executed.

The progress check determines whether telecommand has terminated execution and moves component to the COMPLETING state and performing doComplete action. Right after completion check is performed that determines whether telecommand can complete successfully.

At any stage if checks failed, component enters ABORTED state.

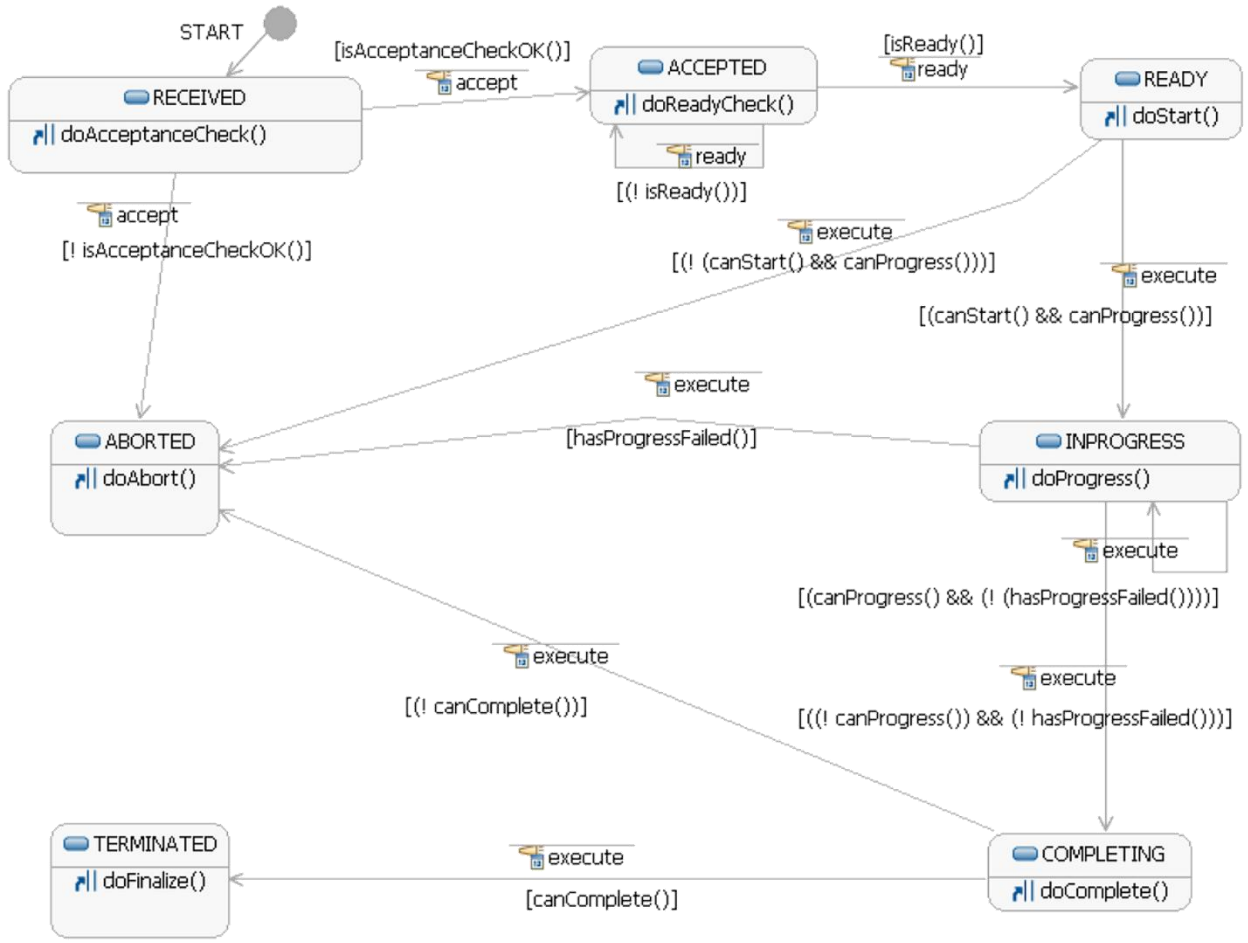

Figure 34 – Telecommand state chart

Following properties should be checked:

1. If the acceptance, ready, start and progress checks are passed, then a telecommand will go through the following states: ACCEPTED, READY, IN_PROGRESS (possibly more than once).
2. A telecommand can only complete if it has successfully terminated its progress actions.
3. If the ready check of a telecommand is passed, then the telecommand executes its start action.
4. If a telecommand fails its start check, then it is aborted.

7.1.4. TelecommandManager Class

This class hold list of telecommads and executes them. It extends Component class and defines two states: READY and OPERATING (shown on Figure 35). It defines trigger activate, that calls private method runOneCycle (see Figure 36) that executes all telecommands in the list.

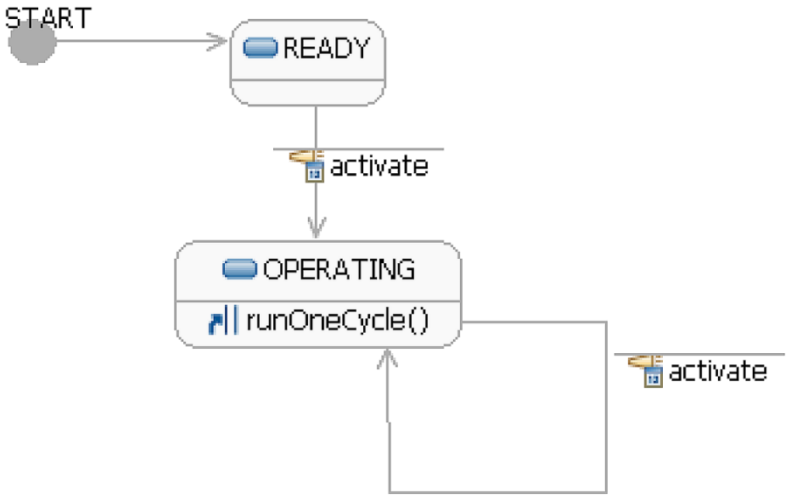

Figure 35 – TelecommandManager state chart.

```
1   public void runOneCycle() {
2       for  (Telecommand tc : tclist) {
3           if (tc.isAborted() || tc.isTerminated()) {
4               toBeRemoved.add(tc);
5           }  else if (tc.isAccepted() &&  ! tc.isReady()) {
6               tc.ready();
7           } else {
8               tc.execute();
9           }
10      }
11      for (Telecommand tc : toBeRemoved) {
12          tclist.remove(tc); }
13      toBeRemoved.clear();
14  }
```

Figure 36 – An example implementation of runOneCycleMethod

7.2. Functional Properties

For demonstration purpose I have chosen 3 formal properties from Telecommand class:

1. *If the acceptance, ready, start and progress checks are passed, then a telecommand will go through the following states: ACCEPTED, READY, IN_PROGRESS (possibly more than once).*

   - If the transition guards: isAcceptanceOK, isCheckReady, canStart, and canProgress are TRUE, then the telecommand state machine passes through the following states (in the order given): ACCEPTED, READY, IN_PROGRESS (possibly more than once).

```
1  !([](isAcceptanceCheckOk().can == true
2     -> X ([] (doReadyCheck()
3     -> X ([] (isCheckReady().can == true
4     -> X ([] (doStart()
5     -> X ([] (canStart().can ==true
6     -> X ([] (canProgress().can ==true
7     -> X ([] !(doProgress()
8  ))))))))))))))
```

Figure 37 – LTL formula for the first demonstration property.

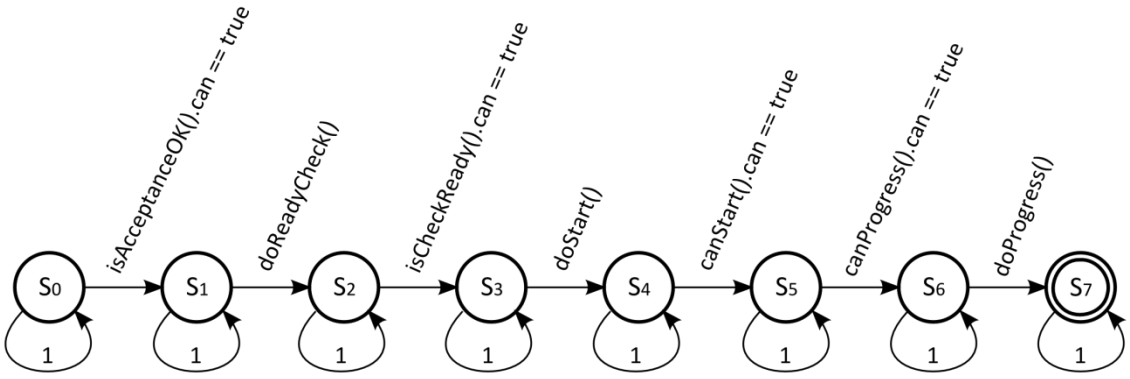

Figure 38 – Büchi automaton for LTL form shown on Figure 37

2. *A telecommand can only complete if it has successfully terminated its progress actions.*

   - If the telecommand has reached state IN_PROGRESS, then it can only enter state COMPLETED when both methods canProgress and hasProgressFailed return FALSE.

```
1  ! ( [] (doProgress()
2     -> X ( []  ( canProgress().can == false
3     -> X ( []  ( hasProgressFailed().can == false
4     -> X ( [] !( doComplete()))
5  ))))))
```

Figure 39 – LTL formula for the second demonstration property

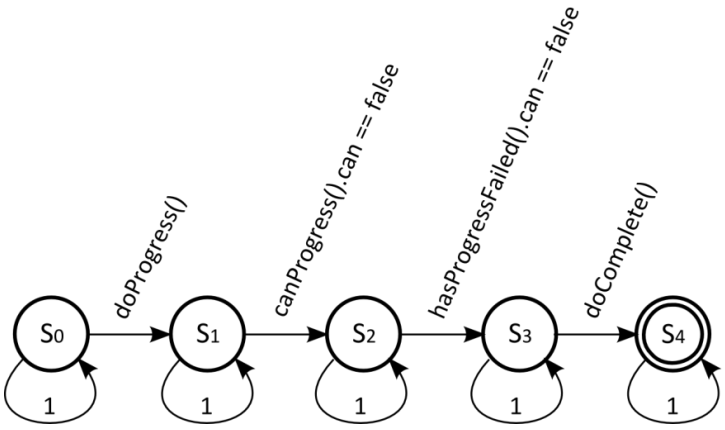

Figure 40 – Büchi automaton for LTL form shown on the Figure 39

3. *If the ready check of a telecommand is passed, then the telecommand executes its start action.*

   - If the transition guard isCheckReady returns TRUE, then method doStart is executed.

```
1  ! ( [](isCheckReady().can == true
2     ->X([] !(doStart()
3  ))))
```

Figure 41 – LTL formula for the third demonstration property

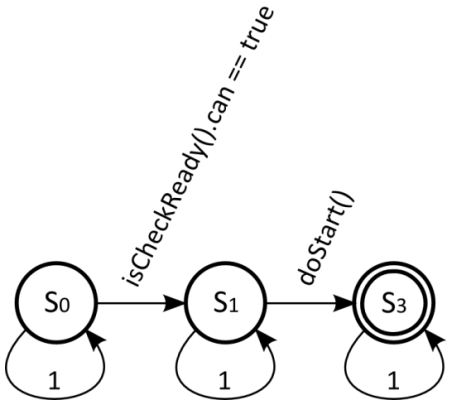

Figure 42 – Büchi automaton for LTL form shown on the Figure 41

7.3. Test results

7.3.1. Missing Localness

The review of Figure 40, Figure 41 and Figure 42 shows that they look nearly the same. It is always the same pattern: some instructions are done, then expected guard is satisfied and automaton transits to the next state with the same semantics. This is done to the last state which is accepting. With this automaton we declare that the only single state is accepting. Temporal logic offers more that simplistic transition diagrams. LTL provides powerful operator but in demo they were useless. The cause is localness of object state. In current state of jpf-ltl, it is possible only to check global state (as mentioned in section 5.4). But objects (like telecommand) conceive local state that is independent from global state and there can be multiple instances of single class that can have independent state. In test therefore I have always used only single telecommand instance. More challenging problem would be checking components properties, because Component class is extended by Telecommand manager and Telecommand and both of them would alter global Büchi automaton for Component properties. Therefore these properties were not included in the test case

This problem is considered to be a major problem of current jpf-ltl implementation. Creating scopes like global scope and object scope is crucial improvement for the next jpf-ltl release.

7.3.2. Function Return Value

Appendix C shows reference implementation of a telecommand. All methods that return some kind of value always make assignment to local variable at first. The reason for this redundant code is that there is no mean to check return value of function. The workaround is creating a local variable and assigning return value there. This variable may already be referenced in LTL formula.

Altering code to add referable points is not a correct use-case. This is the expected improvement of the next jpf-ltl version.

7.3.3. Cumbersome Formula Creation

While creating LTL formula, I have discovered that having some kind of LTL visualization tool is not only nice to have feature but it is necessity for two reasons:

1. Making LTL formula may be a big obstacle for people that are not familiar with this area of mathematics. Visualization is a big help for those kinds of users.
2. Also LTL formula can be easily translated to Büchi automaton with hundreds of states. Debugging of such automaton can be unbearable obstacle.

```
1  ([](isAcceptanceCheckOk().can == true
2     -> X (<> (doReadyCheck()
3     -> X (<> (isCheckReady().can == true
4     -> X (<> (doStart()
5     -> X (<> (canStart().can ==true
6     -> X (<> (canProgress().can ==true
7     -> X (<> !(doProgress()
8  ))))))))))))))
```

Figure 43 – Altering a few operators can lead to state explosion problem (compare with Figure 37).

Büchi automaton of Figure 37 contains 8 states and 15 transitions and 15 atoms. Büchi automaton of Figure 43 contains 176 states, 6466 transitions, 12306 atoms and this can be a big obstacle while evaluating which part of program has failed. It is easy to end up with hundreds of states. Then just compiling LTL to Büchi takes 5 minutes on 2 GHz core. Therefore it is better not to create one all checking rule, but it is recommended to divide problem to set of easier rules.

7.3.4. Symbolic Analysis

The test does not pass pass all possible execution paths without adding choices to the code (mentioned in section 4.3.1). This can be also considered as redundant code. Hopefully there is also another possibility how to check all states without refactoring in the source code. This can be done with symbolic analysis, where data are not important as symbols that represent them.

```
1   function (int x, int y) {
2      if (x > y)
3         a();
4      else
5         b();
6   }
7
8   function a() { ... }
9
10  function b(){ ... }
```

Figure 44 – Simple example with one if statement that divides execution into two execution paths

For example code shown in Figure 44, execution path would divide to two paths on line 2, one for x > y and one for x <=y. Exact values does not matter, just symbols and theirs operator. This way it is possible to traverse all paths without source code modification. Support for symbolic analysis is already implemented in official NASA module jpf-symbc [39] but symbolic LTL checking needs to be implemented.

This feature is considered as long term goal.

7.3.5. Infinite Model Checking

Satellite software can be hardly considered as finite (execution has some proper ending). It is software that is designed to run infinitely. Checking system from this perspective seems to be a proper use-case that deserves to implement. In a nutshell this is done by negating LTL Büchi automaton. This automaton contains states that are considered to be all possible invalid states. The trick is that automaton is preprocessed at first. Procedure looks for specific cycles in graph called strongly connected components that guarantees that there is always path that leads to the point of origin. If negated Büchi enters such cycle, it possible to declare that program will be always in invalid state.

This feature is considered as import for the next jpf-ltl release.

8. Conclusions

JPF-LTL module was repaired and improved and it can be now used to check formal properties on a global scope for whitch it works well. It is possible to refer to almost every class feature like variables values, method call and local method variables.

Module still lacks a lot of crucial features like localness or infinite model checking. Therefore it cannot be considered as tool ready for the industry. It is still in a prototype phase, but it is now very clear what features are exactly missing and what use cases it has to fulfill. Therefore next development should not focus on new features like symbolic analysis but stabilize core features.

For this purpose fork of the original JPF-LTL project was created and development is now separated. Current module source repository can be found at http://bitbucket.org/petercipov/jpf-ltl/.

JPF-LTL was integrated to Eclipse environment where it can be executed as other standard Eclipse features. Main idea while creating integration plug-in was simplicity and one click install of the checker framework. A part of the Eclipse integration is LTL visualiser that has proven his irreplaceable place while creating LTL formula. Creating proper LTL formula can be cumbersome and visualizing Büchi automaton can solve a lot of starters' problems. The source code of LTL visualization and runner can be found at http://code.google.com/p/singularity.

Bibliography

1. **Beydeda, Sami, Book, Matthias and Gruhn, Volker.** *Model-Driven Software Development.* s.l.: Springer, 2005.

2. **Pasetti, A., Rohlik, O. and Cechticky, V.** Software Framework Concept. [Online] 9 11, 2005. http://singularity.googlecode.com/git/docs/[ETH]%20Assert%20-%20deliverables/pdf/004033.DVT_ETH.DVRB.1.I1R2_D4.2.4-1_FrameworkConcept.pdf.

3. **Pasetti, A., Rohlik, O.** The FW Action Language. *The Model-to-Code Tranformation Project.* [Online] 12 2005. http://control.ee.ethz.ch/~rohliko/ceg/assert/model2code/The_FW_Action_Language.html.

4. **PNP-Software.** Class TelecommandManager. *TelecommandManager (CORDET Data Handling and Control Framework documentation).* [Online] 6 14, 2008. http://www.pnp-software.com/fwprofile/doc/com/pnp-software/cordet/dh/telecommand/TelecommandManager.html.

5. **Inquiry Board.** Ariane 5 - Flight 501 Failure. *ESRIN - European Space Agency.* [Online] July 19, 1996. http://www.niwotridge.com/Resources/Ariane5Resources/esa-x-1819eng.pdf.

6. **US General Accounting Office.** *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.* 1992.

7. **Huckle, Prof. Thomas.** Collection of Software Bugs. *Technische Universität München.* [Online] November 7, 2011. http://www5.in.tum.de/~huckle/bugse.html.

8. **RTI.** The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology.* [Online] May 2002. http://www.nist.gov/director/planning/upload/report02-3.pdf.

9. **Floyd, Robert W.** Assigning meanings to programs. [book auth.] J.T. Schwartz (Ed.). *Proceedings of a Symposium in Applied Mathematics, Vol. 19.* s.l. : A.M.S, 1967, pp. 19-32.

10. **Hoare, Charles Antony Richard.** An Axiomatic Basis for Computer Programming. *Communications of the ACM, Volume 12 Issue 10.* October 1969.

11. **Pnueli, Amir.** *The Temporal Logic of Programs.* 1977. pp. 46-57.

12. **Kripke, Saul.** Semantical Considerations on Modal Logic. [aut.] Philosophical Society of Finland. *Acta Philosophica Fennica 16.* Helsinki : North-Holland Publishing Company, 1962, s. 83-94.

13. **Khoussainov, Bakhadyr and Nerode, Anil.** *Automata Theory and Its Applications.* Boston : Birkhauser, 2001.

14. **Gerth, R., et al.** Simple On-the-fly Automatic Verification of Linear Temporal Logic. *In Protocol Specification Testing and Verification.* Poland : Chapman & Hall, 1995, pp. 3-18.

15. **McMillan, Kenneth.** *Symbolic Model Checking: An approach to State explosion Problem.* Pittsburgh : Carnege Mellon University, 1992.

16. **Bryant, Randal E.** *Graph-Based Algorithms for Boolean Function Manipulation.*1986. IEEE Transactions on Computers, C-35(8). pp. 677–691.

17. **Feigenbaum, J., et al.** Complexity of promlems of graphs represented as OBDD. *Chicago Journal of Theoretical Computer Science.* 1999.

18. **Specification and Verification Center at CMU.** School of Computer Science, Carnegie Mellon University. *The SMV System.* [Online] 1998. http://www.cs.cmu.edu/~modelcheck/smv.html.

19. **Godefroid, Patrice.** *Partial-Order Methods for the Verification of Concurrent Systems -- An Approach to the State-Explosion Problem.* s.l. : University of Liege, Computer Science Department, 1994.

20. **Holzmann, Gerard.** *Spin - Formal Verification.* [Online] January 20 , 2012. http://spinroot.com/spin/whatispin.html.

21. **Ben-Ari, Mordechai.** *Principles of the SPIN mocel checker.* London : Springer-Verlag, 2008.

22. **Holzmann, Gerard J.** The Model Checker SPIN. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 5.* May 1997.

23. **Lindholm, Tim and Yellin, Frank.** VM Spec The Structure of the Java Virtual Machine. *The JavaTM Virtual Machine Specification; Second Edition.* [Online] http://docs.oracle.com/javase/specs/jvms/se5.0/html/Overview.doc.html#7143.

24. **Havelund, Klaus and Pressburger, Thomas.** Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer, Vol. 2, No. 4.* 2000.

25. **Havelund, Klaus, et al.** Model Checking Programs. *Automated Software Engineering Journal, Volume 10, Number 2.* 2003.

26. McCarthy 91 function. *Wikipedia, Free Encyclopedia.* [Online] http://en.wikipedia.org/wiki/McCarthy_91_function.

27. **The NASA Ames Research Center.** On-the-fly Partial Order Reduction. *Java Path Finder.* [Online] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/partial_order_reduction.

28. **Brat, G., et al.** Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in Systems Design Journal . Volume 25, Number 2-3.* 2004.

29. **Penix, J., et al.** Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in Systems Design Journal. Volume 26, Number 2.* 2005.

30. **Ginbayashi, Jun, et al.** *New Approach to Application Software Quality Verification.* s.l. : Fujitsu, 2009.

31. **Raimondi, Franco.** JPF-LTL model checker. [Online] https://bitbucket.org/francoraimondi/jpf-ltl.

32. **Lindholm, Tim and Yellin, Frank.** The Java Virtual Machine Specification; Second Edition. *Oracle Documentation.* [Online] 1999. http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html.

33. **OSGi Alliance.** *OSGi Service Platform - Core Specification.* s.l. : OSGi Alliance, 2011.

34. **Fowler, Martin.** Inversion of Control Containers and the Dependency Injection pattern. [Online] January 23, 2004. http://martinfowler.com/articles/injection.html.

35. **Pasetti, A. and Rohlik, O.** Automated proof based System and Software Engineering for Real-Time Applications. [Online] http://www.pnp-software.com/fwprofile/pdf/004033.DVT_ETH.DVRB.2.I3R0_D4.2.2-1_AdaptationTechnique.pdf.

36. **Cipov, Peter.** Singularity update site. [Online] 2012. http://singularity.googlecode.com/git/SingularityUpdate/.

37. **NASA Ames Research Center.** jpf-core source repository. *Java Pathfinder.* [Online] http://babelfish.arc.nasa.gov/hg/jpf/jpf-core.

38. **Pasetti, A., Rohlik, O. and Egli.** V3 Demonstrator. *Automated proof based System and Software Engineering for Real-Time Applications.* [Online] May 10, 2007. http://singularity.googlecode.com/git/docs/[ETH]%20Assert%20-%20deliverables/pdf/004033.DVT_ETH.DVRB.9.I1R1_D4.2.4-4.3.V3Demonstrator.pdf.

39. **Pasareanu, Corina S., et al.** *Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software.*

Appendix A LTL Formula Grammar Specification

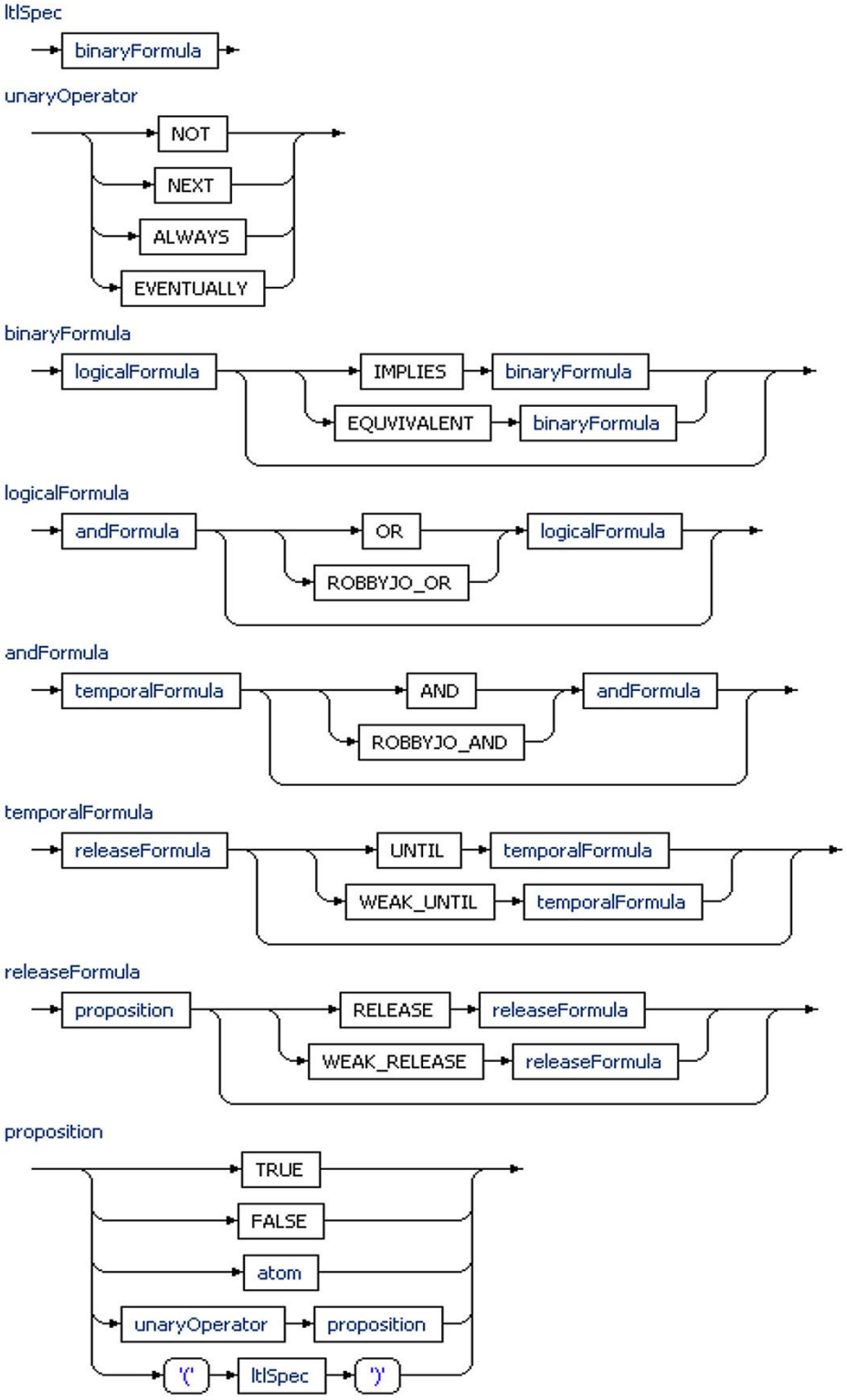

Figure 45 – LTL formula grammar specification

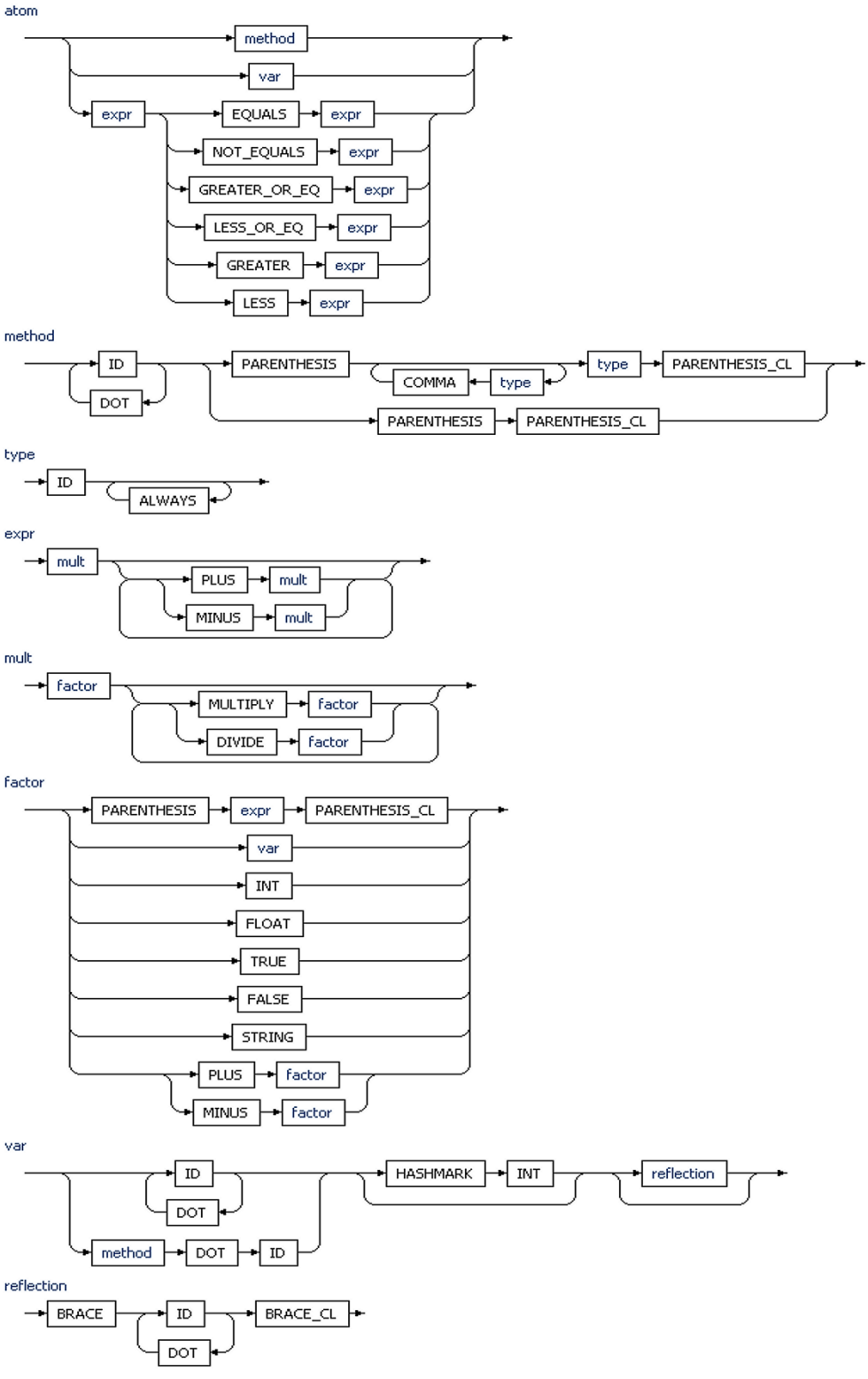

Figure 46 – Atom grammar specification

Appendix B Telecommand Class Diagram

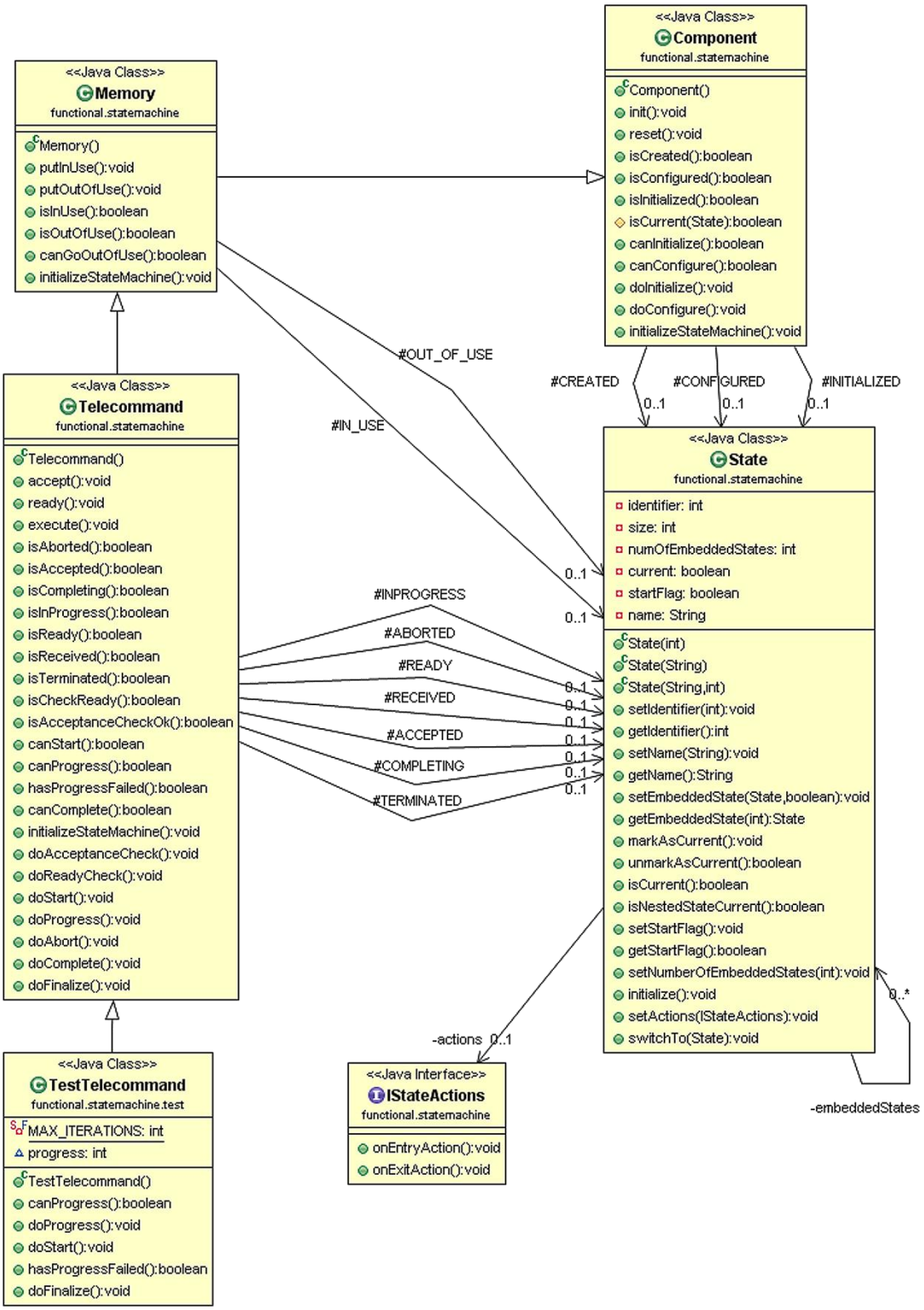

Figure 47 – Class diagram of Telecommand

Appendix C Reference implementation of Telecommand

```
1  @LTLSpec({
2          "!([](isAcceptanceCheckOk().can == true " +
3                          "-> X ([] (doReadyCheck() " +
4                          "-> X ([] (isCheckReady().can == true " +
5                          "-> X ([] (doStart() " +
6                          "-> X ([] (canStart().can ==true " +
7                          "-> X ([] (canProgress().can ==true " +
8                          "-> X ([] !(doProgress()" +
9          "))))))))))))))",
10
11         "!([](doProgress() " +
12                         "-> X ([] (canProgress().can == false  " +
13                         "-> X ([] ( hasProgressFailed().can == false  " +
14                         "-> X ([] !(doComplete()" +
15         "))))))))",
16
17         "!([](isCheckReady().can == true " +
18                         "->X([] !(doStart()" +
19         "))))"
20 })
21 public class TestTelecommand extends Telecommand {
22
23         private static final int MAX_ITERATIONS =5;
24         int progress = 0;
25
26         @Override
27         public boolean canProgress() {
28                 super.canProgress();
29                 boolean can  = progress < MAX_ITERATIONS;
30                 return can;
31         }
32
33         @Override
34         public void doProgress() {
35                 super.doProgress();
36                 progress++;
37         }
38
39         @Override
40         public void doStart() {
41                 super.doStart();
42                 progress = 0;
43         }
44
```

```
45
46
47          @Override
48          public boolean hasProgressFailed() {
49                  boolean can= super.hasProgressFailed();
50                  return can;
51          }
52
53          @Override
54          public void doFinalize() {
55                  super.doFinalize();
56          }
57
58          @Override
59          public void doComplete() {
60                  super.doComplete();
61          }
62
63          @Override
64          public void doReadyCheck() {
65                  super.doReadyCheck();
66          }
67
68          @Override
69          public boolean isAcceptanceCheckOk() {
70                  boolean can = super.isAcceptanceCheckOk();
71                  return can;
72          }
73
74          @Override
75          public boolean isCheckReady() {
76                  boolean can = super.isCheckReady();
77                  return can;
78          }
79
80          @Override
81          public boolean canStart() {
82                  boolean can =  super.canStart();
83                  return can;
84          }
85
86          @Override
87          public void doAbort() {
88                  super.doAbort();
89          }
90  }
```