A Study about Formal Verification of Imperative Programs

João Paulo C. C. de Queiroz
Departamento de Ciência da Computação
Universidade de Brasília, Brazil
Email: joao_paulo_c@yahoo.com

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília, Brazil
Email: flaviomoura@unb.br

***Abstract*—This work presents a formal verification study of a software written in Java programming language. The theoretical formalisms used include Hoare Logic, used to sketch properties on imperative languages; JML constructions, a specification language based on Hoare Logic which is used to specify expected properties from Java programs; Krakatoa software, used to convert JML specifications into proof obligations; and the Coq interactive proof environment, used to verify proof obligations.**

I. Introduction

The necessity of verifying the correction of algorithms can be seen in all the history of computation: firstly the developers, but also the users want to know whether a "product" attains the original specification. This means that nowadays it is not enough to know how to solve a problem, but it is also necessary to provide guaranties about the correction of the solution.

Formal verification methods have recently become usable by industry and achieved success in different levels of use: compilers [1], operating systems [2], air traffic control [3], [4], smart cards [5], etc. This increasing utilization naturally produces a demand for professionals able to apply these methods [6].

In this work, we use formal verification techniques based on higher-order logic and tools (Hoare Logic, JML, Krakatoa, Why, Coq) on an imperative case study (assisted ambient living software) developed with Java programming language.

II. Specification Tools

An imperative language is characterized by a well-defined sequence of commands that can modify the execution state of the program, i.e., commands can generate side-effects [7].

A Hoare triple is consonant to this model of specification by means of preconditions and postconditions [6, p. 224].

Floyd [8] proposed a modeling technique for imperative applications via flowcharts (directed graphs) and the generation of conditions to be checked through the flowchart. Influenced by this work, Hoare [9] presented a deductive system composed by axioms and deduction rules that allows us to reason about a subset of instructions present in all imperative languages.

The deductive system proposed by Hoare is the foundation of several specification languages for imperative programs, such as JML, and is based on the notion of Hoare triples used to establish the connection between a precondition ϕ, an algorithm P, and a postcondition ψ:

$$\phi \ \{P\} \ \psi \tag{1}$$

The semantic of (1) is that "if the assertion ϕ holds before the execution of the program P then ψ holds after the execution of P".

In the case of a JML specification, annotations have to be added in the source code as comments, following the syntax `/*@ annotation */` or `//@ annotation`. Some available constructions are:

- `\result`: denotes the result of a function.
- `\old()`: denotes the value of the given expression at the function entry.
- `p != null`: proposition specifying that the given pointer (`p`) can be safely dereferenced.

A detailed explanation about the specification language JML can be found at [10].

Krakatoa [11] is a system that accepts Java code with JML annotations. It receives annotated code as input and validate them w.r.t syntactic errors. The output is given as input to the intermediary software Why [12] that allows the linkage with proof assistants and automatic provers. Proof obligations are generated by this system that invokes the provers to prove them. The number of generated proof obligations is usually huge, but a big amount of them is trivial in this sense that automatic provers can solve them. This approach, that uses automatic provers to solve trivial proof obligations, and leave to proof assistants only the non-trivial ones, usually speed up the formalization process.

A general picture of the formalization process can be seen in Figure 1.

III. Case study

Assisted ambient living proposes hardware and software solutions that provide an intelligent house environment, as well as a set of integrated services furnishing assistance to people. For instance, this kind of system is useful to detect abnormal situation in a house with an elderly person, such as a fall, and send an alert to a remote help center. The project EMERGE [14] aims to detect and prevent emergencies through

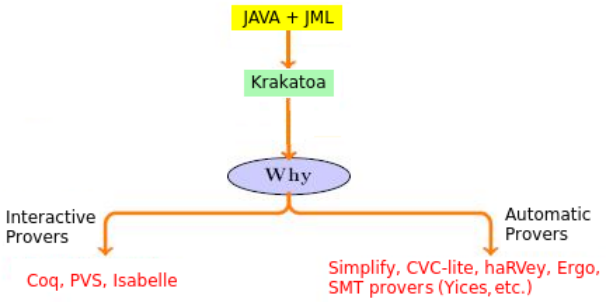

Figure 1. Generation of proof obligations (based on [13]).

an environment supplied with sensors that capture data from the environment, process it via software that is able to detect emergencies, and take adequate actions. The project EMERGE proposes a specification of the architecture and requisites of the system involving a combination of multiagents, but it does not implement the system itself.

The project presented in [15] partially implements the emergency system. The Java programming language with the GAIA [15, p. 59] methodology were used for the design and analysis of the agents. A semiformal method was used to design some properties of liveness, i.e., properties that need to be fulfilled if a specific event happens. In addition, the content of the messages was expressed by a pattern that combines terms and predicates. In this work, we present a formalization of this partial implementation of the emergency agent (`EmergencyAgent`) that constitutes the critical part of the system.

A. Emergency Agent

The emergency agent is formed by three Java classes:

- `RequestConfirmationBehaviour`;
- `MonitorLongTermDeviationBehaviour`;
- `MonitorAcuteEmergencyBehaviour`.

The implementation of these classes is based on a notion of liveness that is represented by the following expression:

$$(\underline{InvokeDetectAlarm}|\underline{InvokeDetectEmergency})^{\omega} .AskConfirmation.[\underline{InvokeSendDataDispatcher}. SendNotification] \ ||(\underline{InvokeDetectDeviation})^{\omega}. \underline{InvokeSendDataSSM}.SendNotification$$

which is detailed in [15, p. 62 *et seq.*].

In order to verify this behavior, one needs to add JML annotations concerning the intended semantics of the agents and prove the generated proof obligations. We did the proofs in the Coq Proof Assistant, which is a robust and well established formalization tool. In total, 19 Java classes were annotated.

B. Non-verified Components

Our formalization of the emergency behavior subsumes that the implementation of the frameworks used by the system are in accordance to the official documentation. For instance, the project under verification uses the JADE (Java Agent DEvelopment) framework which provides middleware functionalities for agent communication, and its correctness is assumed true.

IV. JML SPECIFICATIONS

The annotations were created in two steps:

1) Annotation of the behavior of the agent in the main class (`MonitorAcuteEmergencyBehaviour`).
2) Annotation of the peripheral classes restricted to the part of the code that is used by the main class.

In addition, the JML specification corresponding to the logical abstraction of the intended behavior was based on its formal documentation [15], [16], [17].

A. Main Class

The main class `MonitorAcuteEmergencyBehaviour` controls the emergency events, and the agent (`ElderlyAgent`) must confirm (or not) an emergency, if such event happens. Note that a timeout is possible, which also confirms the emergency. The interaction among `EmergencyAgent` and `ElderlyAgent` is represented by the behavior `RequestConfirmationBehaviour` that is in the list of the executable behaviors of `EmergencyAgent`.

The class `MonitorAcuteEmergencyBehaviour` is derived from the class `CyclicBehaviour` of the API JADE [16] that implements the interface `Observer` of the API Java SDK [17]. It has three methods:

- `onStart()`: this method is executed during the initialization of the behavior, and is executed only once, just before the first call to the method `action()` of the behavior.
- `action()`: this method is used for classes derived from `CyclicBehaviour` (such as `MonitorAcuteEmergencyBehaviour`); it is executed continuously.
- `update()`: this method notifies the classes that implement the interface `Observer` about occurrences of external events.

Finally, the agent has to deal with multiple emergencies, in such a way that it is not enough to model this information as a boolean, but in fact one needs to quantify them.

1) Annotations: We created the class `states` to store the amount of detected emergencies. This class is not used by the original code, but only by the annotations that contain the attribute `qty_emergencies`.

Receiving emergency notifications is an activity of the method `update()` and is annotated:

- considering the type of its parameter `arg` (task dealt with the clause `instanceof`);
- and expressing the possibility of an update to the variable `states.qty_emergencies`.

```
/*@
  @ assigns states.qty_emergencies;
  @ behavior push_emergency:
  @   assumes arg instanceof Emergency;
  @   ensures
  @     states.qty_emergencies ==
  @     \old(states.qty_emergencies) + 1;
  @ behavior not_push_emergency:
  @   assumes !(arg instanceof Emergency);
  @   ensures
  @     states.qty_emergencies ==
  @     \old(states.qty_emergencies);
  @*/
public void update(Observable o,
                   Object arg) {
```

In the above annotation one can observe the specification of two possible behaviors for this method: `push_emergency` and `not_push_emergency`. The former (resp. the latter) occurs if the type of the parameter of the method is (resp. is not) `Emergency`, as represented by the clause `assumes`. The clause `ensures` specify the behavior that schedules more than one emergency, if it is the case.

Once the emergencies are detected (method `update()`), the inclusion of a new object of the type `RequestConfirmationBehaviour` is handled by `action()` method. This method, roughly speaking, consumes detected emergencies creating new behaviors. Here we show just an excerpt of `action()` method annotations, emphasizing the decrement of `qty_emergencies` and the corresponding increment of `qty_behaviours`.

```
  ...
  @ behavior emergency:
  @   assumes states.qty_emergencies >= 1;
  @   ensures
  @     states.qty_emergencies ==
  @       \old(states.qty_emergencies) - 1
  @     && myAgent.qty_behaviours ==
  @       \old(myAgent.qty_behaviours) + 1
  ...
public void action() {
```

Finally, the method `onStart()`, responsible for the initialization of the behavior, is annotated accordingly:

```
/*@ ensures threadFactory != null
  @         && myAgent != null; @*/
public void onStart() {
```

B. *The* `Agent` *Class*

The `Agent` class is a component of the API JADE [16, p. 10 *et seq.*] and is a class common to every agent definable by the user. The classes `EmergencyAgent` and `ElderlyAgent` derive from `Agent` and represent the two main agents belonging to the system. These agents are instantiated and registered in `StartAgent`. Since `Agent` is used (by inheritance) in the project, it had to be annotated.

The behavior `MonitorAcuteEmergencyBehaviour` needs to interact with the agent `EmergencyAgent` to indicate the necessity of the execution of a new behavior after the detection of a new emergency. This is possible due to the object that `MonitorAcuteEmergencyBehaviour` has internally (`myAgent`), and due to the method `addBehaviour()` of the class `Agent`. The object `myAgent` is accessible by inheritance.

1) Annotations: The representation of the possible behaviors of the agent is modeled by a vector that uses JML model fields [18, p. 11]. It is important to notice that this construction belongs just to the annotations, and not to the code itself:

```
//@ model integer qty_behaviours = 0;
//@ model Behaviour [] behaviours;
```

The vector `behaviours` has type `Behaviour []` and is used to host the information of each new behavior. The variable `qty_behaviours` is used to count this vector. The method `addBehaviour()` is specified in such a way that it represents (in postcondition, i.e., `ensures` clause) the inclusion of an element to this vector:

```
/*@
  @ requires b instanceof
  @     ThreadedBehaviourWrapper;
  @ ensures qty_behaviours ==
  @     \old(qty_behaviours) + 1 &&
  @     behaviours[\old(qty_behaviours)]
  @     == b;
  @*/
public void addBehaviour(Behaviour b);
```

The system requires the release of a new behavior with parallel execution. According to the JADE documentation, this can be obtained by nested behaviors of an object of type `ThreadedBehaviourWrapper`, and hence it is required for the added behavior to have this type (precondition above, i.e., `requires` clause).

Here we have shown only some annotations. The complete set can be found at the following URL:

https://lambda.cic.unb.br/svn/mestrado/jp/vida-assistida/codigo-anotado

V. Proof obligations

Proof obligations are generated by the softwares Krakatoa (version 2.30) and Why3 (version 0.71) by invoking the former

to the main class:

```
krakatoa MonitorAcuteEmergencyBehaviour.java
```

In this case, 71 proof obligations were generated and `Why3ide` interface is launched. (See Figure 2).

In this section we will focus on some proof obligations. Firstly, it is important to notice that most of the proof obligations are trivial in the sense that they could be proved by (automatic) theorem provers. The Coq (version 8.3pl4) files containing proof obligations have an axiomatization concerning the memory model for Java programs as preamble [19, p. 12 *et seq.*].

Some generated proof obligations are presented in Table I.

#	Proof obligations	CVC3	Coq	Eprover	Spass
01	*Method action, default behavior*	0.22	1.51		
	Method action, Behavior 'emergency'				
	normal postcondition				
02	*1*	0.18	1.44		
03	*2*	0.17	1.69		
04	*3*	0.17	1.74		
05	*4*	0.18	1.73		
	normal postcondition				
06	*1*	0.14	1.76		
07	*2*	5.11	1.83	⏲	⏲
08	*3*	5.34	1.98	⏲	⏲
09	*4*	5.23	1.70	⏲	⏲
10	*Method action, Behavior 'none_emergency'*	0.15	2.08		
11	*normal postcondition*	⏲	1.63	⏲	⏲
12	*Method onStart, Safety*	0.11			
13	*Method update, default behavior*	0.11			
14	*Method update, Behavior 'push_emergency'*	0.12	1.49		
15	*normal postcondition*	0.11	1.51		
16	*downcast*	0.09	1.54		

Table I. PROOF OBLIGATIONS

An example of a proof obligation that could not be automatically proved is partially given bellow[1]:

```
... (preamble)
Theorem
  WP_parameter__MonitorAcuteEmergency
  Behaviour_onStart_ensures_default :
...
H1 : strict_valid_struct_Object
    result 0 (1 - 1)
    usObject_alloc_table2
...
H3 : usMonitorAcuteEmergency
    Behaviour_threadFactory2 =
    store usMonitorAcuteEmergency
    Behaviour_threadFactory1
    this_16 result
_____________________________________(1/1)
usNon_null_Object
  (select usMonitorAcuteEmergency
  Behaviour_threadFactory2 this_16)
  usObject_alloc_table2
```

[1]This goal is shown after an application of `intuiton` Coq tactic. In this Coq file excerpt, also note that all the axiomatization of Java memory model (the preamble) is suppressed as well as some theorem assumptions and type declarations.

This goal concerns the sole postcondition of the method `onStart()`:

```
ensures threadFactory != null
```

In this case, it is important to notice a fact included by the axiomatization previously mentioned concerning storage and retrieval of dynamic information (pointers): If two pointers `p1` and `p2` are the same, and if one stores the value `a` in position `p1` of the memory region `m`, afterwards if the stored value is selected in position `p2` of the same memory region then the original value `a` is the retrieved value.

```
Axiom select_store_eq : forall (t:Type)
     (v:Type), forall (m:(memory t v)),
     forall (p1:(pointer t)),
     forall (p2:(pointer t)),
     forall (a:v), (p1 = p2) ->
     ((select (store m p1 a) p2) = a).
```

In addition, observe these facts:

- hypothesis `H3` has structure `m' = store m p1 a;`
- the goal it is at the form `P (select m' p2) z;`
- H3 could be rewritten at the goal leading to goal structure `P (select (store m p1 a) p2) z;`
- `p1 = p2 = this_16.`

Putting in other words, one could rewrite hypothesis `H3` (which represents a modification of the state of `threadFactory`) at the goal, and since the storage occurs in the very same position (`this_16`), it is possible to applicate (above) axiom `select_store_eq` to rewrite the goal to the following:

```
usNon_null_Object result usObject_alloc_table2
```

The proof is completed after unfolding the definitions of `strict_valid_struct_Object` in hypothesis `H1`.

VI. ADAPTATION OF PROOF OBLIGATIONS

While building JML specifications and working with proof obligations, a few attention points emerged, sometimes requiring an adequacy of annotation or proof strategy. Here we show the main points.

A. Workspace *Java*

Java projects containing a large number of classes and files make use of directory hierarchy, dividing the project in "groups" according to development modeling. This structure of folders and files are usually named *workspace*.

The system of our case study has 109 ".java" files, and 61 directories in the workspace just for holding the source files. Unfortunately this directory structure does not work on krakatoa due to a malfunction in the processing of `package` and `import` directives[2].

[2]The `package` and `import` directives are used to indicate to Java compiler (`javac`), respectively where class files will be found, and which are the dependency between classes [20].

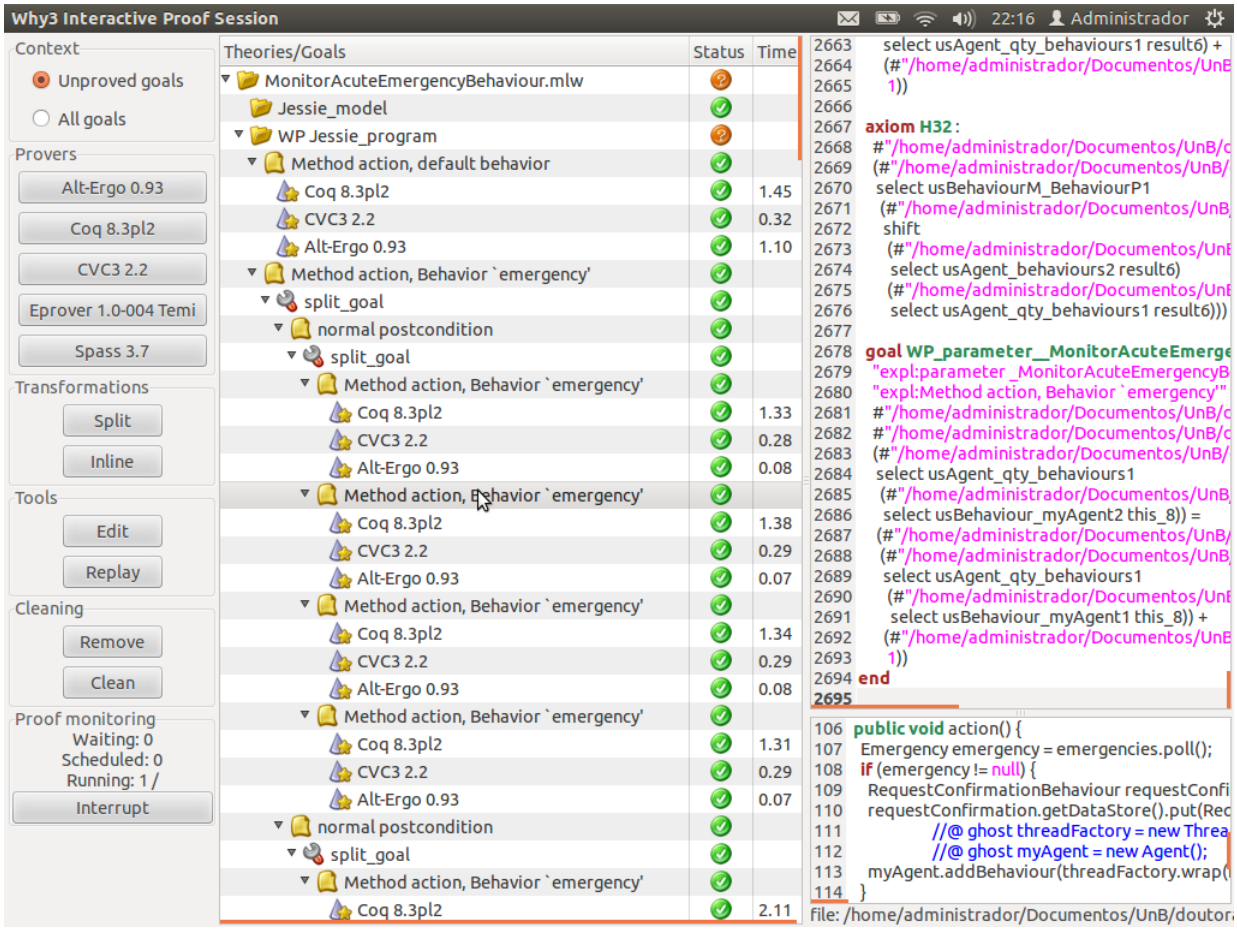

Figure 2. Krakatoa proof obligations

The system has two interdependent classes: `Behaviour` and `Agent`. Hence, one "imports" the other. Regardless the fact that this is a valid construction in Java, Krakatoa cycles infinity processing this dependency. In fact, a simple test case like bellow reproduces this situation:

1) create directories "p1" and "p2";
2) inside "p1", create file "Class1.java" like this:

```
package p1;
import p2.Class2;
public class Class1 {}
```

3) inside "p2", create file "Class2.java":

```
package p2;
import p1.Class1;
public class Class2 {}
```

4) run "`krakatoa p1/Class1.java`";
5) stop (infinite) execution and inspect log file (`krakatoa.log`):

```
...
adding java file Class1
  (fullname ./p1/Class1.java)
importing package java.lang
importing p2.Class2
importing package java.lang
importing p1.Class1
importing package java.lang
importing p2.Class2
importing package java.lang
importing p1.Class1
...
```

The workaround was to deactivate `package` and `import` directives (i.e. "commenting" them) and leaving all the project files in a single directory. Since there was no name clashes between classes, this succeeded.

B. *Generic programming*

One feature supported by Java language is called generics[3] or "generic programming" [20]. Basically the construction allows inclusion of a parameter representing a type, while defining a class. The intention is that one can declare a class as like `MyClass<T>`, where `T` is a parameter used in class body in places normally used for types. Further, instantiations as `MyClass<bool> object` or `MyClass<int> object` would make `object` behaves exactly as in `MyClass<T>` declaration, except for appropriate substitutions[4].

This kind of polymorphic behavior is formally described by Pierce [21, p. 361 *et seq.*], presenting the so called *universal*

[3]Generics was only supported on Java language since version 5.0.

[4]In these examples, substitutions are respectively `T == bool` (in `MyClass<bool> object`) and `T == int` (in `MyClass<int> object`).

types, which leads to a **parametric polymorphism** Ã la System F.

The studied case uses the interface `Queue` and the class `ConcurrentLinkedQueue`, declared respectively as `Queue<E>` and `ConcurrentLinkedQueue<E>`, therefore employing Java generics.

Krakatoa does not support generic programming [19, p. 16], and hence the solution was to deal with polymorphism manually, i.e., creating files with the original code replicated with a manual replacement of the parameter `E`.

The main class requires substitution of `Emergency` for `E`. This way, the classes `Queue_Emergency` and `ConcurrentLinkedQueue_Emergency` were created with this manual substitution[5].

C. Model fields

Sometimes the annotation process requires modeling (abstractly) some specific data structure. JML provides an specification mechanism called "model fields", which lets you create variables that exist only in the annotation scope, so these fields cannot be accessed by Java code [18, p. 11 *et seq.*].

This feature can be used to provide abstraction of object concrete states, where each abstraction can denotes different aspects of the same object [22, p. 10].

This kind of construction was used at two points of annotations:

- in `Agent` class, used for modeling a behavior vector and for maintaining information about vector size:
  ```
  //@ model integer qty_behaviours = 0;
  //@ model Behaviour [] behaviours;
  ```
- in `HashMap` class, employed for specifying a vector holding objects of any type (class `Object`[6]):
  ```
  //@ model Object [] vetor;
  ```

We also tried to use model fields to abstract a data queue, by building a generic annotation to the interface `Queue_Emergency`, like those ones in `Agent` and `HashMap`. Nevertheless, when we tried to generate proof obligations, the tool presented a malfunction:

```
File "java/java_typing.ml", line 1109,
    characters 8-8:
Fatal error: exception Assert_failure
    ("java/java_typing.ml", 1109, 8)
```

This exception is inserted in Krakatoa/Why source code (file `java_typing.ml`), and analyzing it we verified that model fields are not yet supported for interfaces[7].

Since we are not allowed to change the original source code, we decided to create a new class that was merely used to host state variables only referenced by annotations, but not by the system. The new class `states` was then created with the solely purpose of simulating model fields feature for interfaces.

D. Inner classes

Java language allows definition of a class inside another ("inner" or "nested" classes), used to logically group classes in a single file, increasing encapsulation[8] [20].

The class `ThreadedBehaviourFactory` (which the system depends on) defines a inner class named `ThreadedBehaviourWrapper`, but Krakatoa does not recognize inner class declaration:

```
File "./ThreadedBehaviourFactory.java",
    line 95, characters 39-63:
    typing error: unknown identifier
    ThreadedBehaviourWrapper
```

Since this inner class were necessary for annotations, this situation was circumvented by moving it to a dedicated file (`ThreadedBehaviourWrapper.java`), and then transforming it on a regular class. The inner class mechanism has no additional semantics but to provide scope control, so this workaround had no semantic impact.

E. (Partially) Valid objects

Another situation which needed some sort of annotation adaptation was the use of methods belonging to dynamically allocated objects [20].

In annotation process of `action()` method, two objects (`threadFactory` and `myAgent`) were required by specification (as a precondition, i.e., `requires` clause) as not being null. This way, these objects must be valid as a precondition for `action()` method call.

Despite this one of the proof obligations initially could not be proved because the tools did not introduce this precondition (validity) at the assumptions of the theorem to be proven.

To cope with this problem, we had to include extra (ghost[9]) code in annotations to redundantly show these objects were pre-allocated, i.e., allocated before `action()` method being called.

```
/*@
  @ requires threadFactory != null &&
  @          myAgent != null;
  ...
public void action() {
  ...
  /*@
```

[5] `Queue_Emergency` is the manual substitution semantically equivalent to `Queue<Emergency>`, and `ConcurrentLinkedQueue_Emergency` to `ConcurrentLinkedQueue<Emergency>`.

[6] In Java, the class `Object` is the superclass of all classes [20]. Pierce formalized this notion by means of `Top` constant as the maximum element in a subtype relation [21, p. 185].

[7] "In the Java programming language, an interface is a reference type, similar to a class, that can contain only constants, method signatures, and nested types." [20] This way interfaces cannot contain method bodies. Note that model fields are supported on regular classes. The problem only arises when one is using them over interfaces.

[8] Encapsulation is a concept present in object oriented languages; it hides internal states (variables) of objects (class instances) and requires that all interaction among objects has to be made using methods (functions). These methods are responsible for updating internal states [20].

[9] Ghost code is JML specification only code which, differently from model fields, can represent an assignment of a value. In this case, it represents pointer assignments (instantiation of objects) [18, p. 14].

```
    @ ghost threadFactory =
    @   new ThreadedBehaviourFactory();
    @ ghost myAgent = new Agent();
    @*/
  myAgent.addBehaviour(
    threadFactory.wrap(
    requestConfirmation));
  ...
}
```

Observe that in the above code, `ghost` clauses bear legal Java code which would normally be used for object initialization (allocation). This ghost code leads to a proof obligation automatically proved.

Instead, suppressing the inserted ghost code, would produce an unprovable Coq context:

```
...
H2 : usNon_null_Object
  (select usA_b1 this_0)
  usObject_alloc_table1
...
H : alloc_extends
  usObject_alloc_table1
  usObject_alloc_table2
...
_____________________________________(1/1)
0 <= offset_max usObject_alloc_table2
  (select usA_b1 this_0)
```

The axiomatization preamble defines the function `usNon_null_Object` (used in hypothesis `H2`) in a structure similar to the one presented in the above goal:

```
Definition usNon_null_Object
 (x_5:(pointer usObject))
 (usObject_alloc_table:
   (alloc_table usObject)): Prop :=
 (0%Z <=
 (offset_max usObject_alloc_table x_5))%Z.
```

The only difference between `H2` and the goal is that `H2` refers `usObject_alloc_table`**1** while the goal mentions `usObject_alloc_table`**2**. Hypothesis `H` guarantees the required transitivity[10], but to use this fact involving `alloc_extends`, one is required to furnish a `alloc_extends` premise stronger than the one represented by `H2`[11].

The theorem was proved after introduction of the previously shown `//@ ghost` directives.

VII. Conclusion and Future Work

We presented an ongoing formalization of an assisted ambient living developed in Java. The used tools include an annotation language (JML), programs for converting annotated programs into proof obligations (Krakatoa, Why3) and provers (both automatic and interactive). The lack of important features in the krakatoa system is a limitation for its usage in real applications.

In addition, JML is a specification of sequential Java, which will require additional work to treat concurrency. The formalization of the concurrency involved in our case study is, in fact, a major task that needs be done in the near future. For doing so, we aim to profit from a specification of the π-calculus in Coq [23].

Apart from that, krakatoa can be employed to analyse formally not exclusively academic applications, and in fact, there exists some industry initiatives in this direction. Although it was understood that our goal has been achieved, it was also observed that the formalization and verification of mass-produced commercial code is a challenge as big as or higher than the construction of the application.

Two factors showed significant contribution to this scenario: the mechanical translation of imperative code to an appropriate axiomatization, and the use of automatic provers to eliminate most of proof obligations.

The first factor has its importance enforced by the semantic complexity of commercial programming languages such as Java and C. Despite the fact that Hoare Logic is a formalism, which captures basic imperative constructions, these languages have many extensions that poses difficulty to the complete comprehension of code. In this case, manually translating source code to provers like Coq or PVS is a task that can limit large scale use of the involved solutions, leading to error prone repetitive work. Mechanized translation helps focusing efforts in the appropriate representation of semantical aspects of proof environments.

The second factor (the use of automatic provers), is significant because even small portions of code generate a high number of obligations to be proven. At the studied case, just two goals had necessarily to be solved by interactive provers. The majority of goals could be discharged automatically.

In addition, there are initiatives, like Moy [24, p. 119 *et seq.*], for modular and automatic generation of annotations containing pre- and postconditions, and loop invariants.

Finally, all constructions of specification languages presented here are first order, which shorten expressivity about programs. In spite of that, significant parts of commercial applications can be covered with such annotations. Moreover, the specification language presented here (JML) has been evolved towards supporting high order constructions, but they are still experimental.

[10] To see that, one can look at definition of `alloc_extends` at Coq file preamble.

[11] The fact to be proved relates to `offset_max` and `offset_min` function, and `H2` only mentions `offset_max` function.

References

[1] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, pp. 363–446, December 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1666192.1666216

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct 2009, pp. 207–220.

[3] S. Boldo and T. M. T. Nguyen, "Hardware-independent proofs of numerical programs," in *Second NASA Formal Methods Symposium (NFM 2010)*, César Muñoz, Ed., vol. NASA/CP-2010-216215. Washington D.C. United States: NASA, 04 2010, pp. 14–23, Hisseo project, funded by Digiteo. [Online]. Available: http://hal.inria.fr/inria-00534410/en/

[4] L. Lensink, C. Muñoz, and A. Goodloe, "From verified models to verifiable code," NASA, Langley Research Center, Hampton VA 23681-2199, USA, Technical Memorandum NASA/TM-2009-215943, June 2009.

[5] C. Marché, C. Paulin-Mohring, and X. Urbain, "The Krakatoa tool for certication of Java/JavaCard programs annotated in JML," 2003. [Online]. Available: http://www.lri.fr/~urbain/textes/jlap.ps.gz

[6] M. R. A. Huth and M. D. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, England: Cambridge University Press, 2000.

[7] M. Marcotty and H. F. Ledgard, *Programming language landscape: syntax, semantics, and implementation*. USA: SRA School Group, 1986.

[8] R. W. Floyd, "Assigning meanings to programs," in *Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, vol. 19, 1967, pp. 19–32. [Online]. Available: http://www.cs.ucdavis.edu/~su/teaching/ecs240-s09/readings/FloydMeaning.pdf

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580,583, 1969.

[10] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: a behavioral interface specification language for Java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1127878.1127884

[11] "The Krakatoa verification tool for Java programs," 2009, versão: 2.18, URL: http://krakatoa.lri.fr/.

[12] "The Why verification tool," 2009, versão: 2.23, URL: http://why.lri.fr/.

[13] J. C. Filliâtre and C. Marché, "The why/krakatoa/caduceus platform for deductive program verification," in *Proceedings of the 19th international conference on Computer Aided verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 173–177. [Online]. Available: http://portal.acm.org/citation.cfm?id=1770351.1770379

[14] "EMERGE," 2009, uRL: http://www.emerge-project.eu/.

[15] V. U. C. Nunes, "Orquestração de serviços por meio de agentes de software no domínio de vida ambiente-assistida," URL: http://monografias.cic.unb.br, Dezembro 2009, monografia de conclusão de bacharelado em Ciência da Computação, UnB.

[16] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, "JADE Programmer's Guide," April 2010.

[17] O. and/or its affiliates, "Java Platform, Standard Edition 6, API Specification," 2011. [Online]. Available: http://download.oracle.com/javase/6/docs/api/

[18] G. T. Leavens and Y. Cheon, "Design by contract with jml," 2006. [Online]. Available: http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf

[19] C. Marché, "The Krakatoa tool for deductive verification of Java programs," Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009, http://krakatoa.lri.fr/ws/. [Online]. Available: http://krakatoa.lri.fr/ws/

[20] O. and/or its affiliates, "The Java Tutorials," 2012. [Online]. Available: http://docs.oracle.com/javase/tutorial/index.html

[21] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[22] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, ser. LNCS, vol. 4111. SV, 2006, pp. 342–363. [Online]. Available: http://dx.doi.org/10.1007/11804192_16

[23] R. Affeldt and N. Kobayashi, "A Coq library for verification of concurrent programs," *Electronic Notes in Theoretical Computer Science*, vol. 199, pp. 17–32, 02 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1346363.1346642

[24] Y. Moy, "Automatic modular static safety checking for c programs," Ph.D. dissertation, Université Paris-Sud, Jan. 2009. [Online]. Available: http://www.lri.fr/~marche/moy09phd.pdf