Journal of Applied Mathematics & Bioinformatics, vol.3, no.1, 2013, 43-56
ISSN: 1792-6602 (print), 1792-6939 (online)
Scienpress Ltd, 2013

Proof Carrying Code using Algebraic Specifications

Nikolaos Triantafyllou[1], Katerina Ksystra[2], Petros Stefaneas[3] and Panayiotis Frangos[4]

Abstract

Proof Carrying Code is a methodology developed to establish trust between code consumer and producer. The latter formally proves that the code he sents to the former satisfies some safety properties. That proof is received by the consumer together with the code, that is under inspection. Next the consumer verifies the proof before authorizing the execution of the code. While PCC is a powerful approach, some issues like reusability and the difficulty of the producer to produce the formal proofs, hinder its wide use. In this paper we propose an alternative schema for proof carrying code using tools from the fields of algebraic specifications and design by contract to counter some of these problems.

Mathematics Subject Classification: 68U01

Keywords: CafeOBJ; JML; Observational Transition Systems; Design by Contract; Behavioral Specifications; Hidden Algebra; Proof Carrying Code

[1] National Technical University of Athens, e-mail: nitriant@central.ntua.gr
[2] National Technical University of Athens, e-mail: katksy@central.ntua.gr
[3] National Technical University of Athens, e-mail: petros@math.ntua.gr
[4] National Technical University of Athens, e-mail: pfrangos@central.ntua.gr

Article Info: *Received* : November 1, 2012. *Revised* : January 30, 2013
Published online : March 30, 2013

1 Introduction

Modern day systems are no longer statically configured. Instead, they are built in ways that facilitate extensibility as much as possible. This becomes obvious if we consider web applications where mobile code is being used on computers daily (JavaScript applications, Java Applets, and so on). The benefits of using mobile code are decreased however due to the lack of trust that usually exists between the mobile code producer and consumer. The provided code can harm the safety of the system either intentionally or unintentionally. A first measure to establish the trust between the two parties is to digitally sign the code. But this says nothing about the code itself except from the fact that it was created by the signing source. The code could still breach the safety of the system.

In [1] G. C. Necula introduces the concept of Proof Carrying Code (PCC) to counter this problem. PCC is a technique in which the code consumer can statically verify that the code sent by the producer satisfies a set of safety properties. This is achieved by certifying the compilation process using a logic defined by the consumer. The producer is responsible for providing a formal proof that the under inspection code adheres with the safety properties, using the safety policy.

There are two main disadvantages in using PCC. The first is the difficulty of creating the proof rules. These rules are designed to allow the proof of a specific set of safety properties. As a result, the slightest change in the policy can result in new proof rules and new proofs for the safety and soundness of the rules as well. The second is the generation of the proofs themselves. The burden of this task falls on the shoulders of the mobile code producer, a role he is not usually accustomed to. While this task is automated to a high degree by a certifying compiler, there are usually proof obligations that require manual intervention [2].

In this paper we will propose a framework for statically verifying mobile code, based on PCC. This approach will overcome some of the disadvantages we mentioned earlier. The key component of the proposed framework is a translation from hidden algebra specifications to design by contract specifications. This will allow the use of the hidden algebra specification as the safety policy. As a result, there will be no need for the code producer to provide a

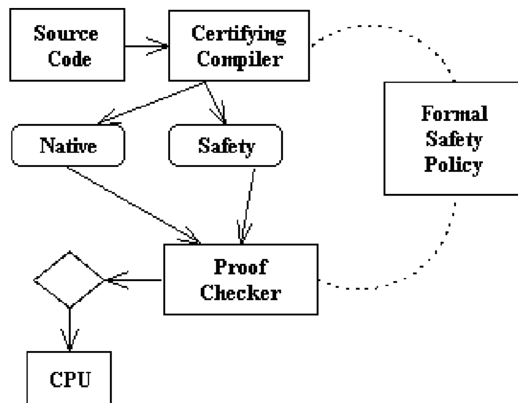

Figure 1: basic PPC architecture

proof for the compliance of the provided code. The rest of the paper is organized as follows: section 2 gives an overview of the PCC technique and section 3 presents the algebraic specification methodology of hidden algebra. Section 4 introduces the reader to the specification paradigm of design by contract (DBC). In section 5 we describe the proposed framework and finally section 6 concludes the paper.

2 Proof Carrying Code

In order to define a PCC implementation, four components are mandatory [2]. A formal specification language is required for expressing the safety policy. Next, a formal semantics of the mobile code language (these are usually defined as a logic) that relates the code with the specification. In addition we need a language in which the proofs will be expressed and finally an algorithm to validate the proofs.

The safety policy is the corner stone of the approach and defines a set of rules that express which actions a program is allowed to perform and under which circumstances these are allowed (in Hoare logic style).

A PCC binary undergoes three phases [2]: certification, validation and consumption. In the first phase, certification, the mobile code is generated and the producer ensures that it adheres to the safety policy. Next the source

code is compiled and a proof that it complies with the set of safety properties is generated from the rules of the safety policy. This is encoded to avoid tampering and it is transferred to the code consumer. The receipt of the code initiates the second stage, the validation. The consumer validates the proof. If this step is successful then the code is loaded for execution. This can be carried out off-line. After a successful validation, the code is safe to be executed, i.e. it enters the consumption stage.

Research has been conducted on overcoming the disadvantages of PCC and especially making it less reliant upon the explicit proof rules of the safety policy. A. W. Appel in [3], develops a variant of PCC called fundamental proof carrying code (FPCC) that uses higher order logic and removes the requirement of individual proof rules for each instruction of the compiler. However, this does not solve the problem of forcing the code producer to create the proofs of compliance.

3 Hidden Algebra

Hidden algebra (HA) is an approach for giving semantics particularly for concurrent distributed object oriented systems, as well as for software engineering in general [4]. This approach is based on equations in contrast with formalisms like higher order logic for example. The reason behind this decision is that the proofs supported by this calculus allow maximal simplicity of mechanization and at the same time this approach allows adequate expressiveness [4]. Hidden algebra is a generalization of process algebra and transition systems which by including non-monadic operations, allows the use of equations that contain methods or attributes that are parameterized by data [4]. The key of this approach is the concept of behavioral satisfaction. This means that we focus on how systems behave in response to a given set of experiments and not on the implementation details of those systems. This type of specification is referred to as behavioral specification.

For a set of sorts S, we say that the set A is S-sorted if it can be regarded as a family of sub-sets as $A = \cup\{A_s\}_{s \in S}$. Using this set of sorts, we can define a signature Σ as the pair <S, F> where F is a set of function symbols. Such that F is equipped with a mapping $F \rightarrow S^* \times S$ meaning that each

$f \in F$, $f : s_1 \times \cdots \times s_n \rightarrow s$. Then the type (or rank) of f is defined as $rank(f) = s_1 \ldots s_n s \in S^*$. Given a signature as above, a Σ-Algebra A consists of a non-empty family of carrier sets $\{A_s\}_{s \in S}$ and a total function $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ for each function symbol $f : s_1 \times \cdots \times s_n \rightarrow s \in F$. A Σ-homomorphism between two Σ-algebras A and B, denoted by $h : A \rightarrow B$, is a family of mappings $\{h_s : A_s \rightarrow B_s\}_{s \in S}$ that preserves the operators. By imposing a partial ordering on the sorts we get an Order Sorted Σ-Algebra (OSA).

An *order sorted signature* is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature, $(S, \leq)$ is a poset, and the operators satisfy the following monotonicity condition; $\Sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ implies $s1 \leq s2$. Given a many-sorted signature, a (S, Σ)-algebra A is a family of sets $\{A_s \mid s \in S\}$ called the carriers of A, together with a function $A_\Sigma : A_w \rightarrow A_s$ for each Σ in $\Sigma_{w,s}$. Where $A_w = A_{s1} \times \cdots \times A_{sn}$ *and* w=s$_1$...s$_n$. Let $(S, \leq, \Sigma)$ be an order sorted signature. A $(S, \leq, \Sigma)$-*algebra* is a (S, Σ)-algebra A such that, $s \leq s'$ in S implies $A_s \subseteq A_{s'}$ and $\Sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ implies $A_\Sigma : A_{w1} \rightarrow A_{s1}$ equals $A_\Sigma : A_{w2} \rightarrow A_{s2}$ *on* A_{w1} [5]. The purpose of the formalization of order sorted signatures is to define sorts (similar to classes in OO), functions (similar to methods in OO) and inheritance between the sorts.

By partitioning the set of sorts to visible and hidden sorts (where visible sorts represent the data part of a specification while hidden sorts denote the state of an abstract machine) we get the following definition.

Definition 3.1 (Hidden Algebra). *Given a signature* $(S, \leq, \Sigma)$ *and a subset*
$H \subset S$ *the hidden sorts, a hidden algebra (or a hidden model in general) A interprets the visible sorts V and the operations* Ψ *of the visible sorts as a fixed model D (the data model, say an order sorted algebra) such that* $A\restriction_{V,\Psi} = D$ *(where* $\restriction$ *is the model reduct). Given two signatures* $(S, \leq, \Sigma)$ *and* $(S', \leq, \Sigma')$ *a signature morphism* $\phi : (S, \leq, \Sigma) \rightarrow (S', \leq, \Sigma')$ *consists of a mapping z on sorts that preserves the partial ordering, i.e. for* $s \leq s'$ *then* $z(s) \leq z(s')$ *and an indexed mapping on operators g, such that* $\{g_{s_1 \ldots s_n} : \Sigma_{s_1 \ldots s_n s} \rightarrow {\Sigma'}_{z(s_1) \ldots z(s_n) f(s)}\}_{s_1 \ldots s_n s \in S, n \geq 0}$.

Given a HA signature Σ, the class of its models consists of all Σ-algebras A. In order to define behavioral equivalence we need the following definitions.

Definition 3.2 (Behavioral operators). *The set of behavioral operators of a Σ-algebra A, is defined as Σ^b, where $\Sigma^b \subseteq \Sigma$, such that each $w \in \Sigma^b_{w,s}$ has exactly one hidden sort in w.*

Behavioral operators have object-oriented meaning. $\sigma \in \Sigma^b_{w,s}$ denotes a method on the state space if s is hidden, and an attribute if s is visible.

Definition 3.3 (Behavioral context). *A Σ-context c[z] is a Σ-term c with a marked variable z occurring only once in c. A behavioral context is any Σ-context c[z] such that all operations above z in c are behavioral.*

Definition 3.4 (Behavioral equivalence). *Given a Σ algebra A, two elements (of the same sort s) a and a' are called behaviorally equivalent, denoted as $a \sim_s a'$ iff $A_c(a) = A_c(a')$ for all visible behavioral contexts c.*

On the visible sorts the behavioral equivalence coincides with the (strict) equality relation. Behavioral equivalence allows the creation of elegant and small specifications since we are only concerned with how a state behaves under the attributes and not how it is constructed. This has led to the development of Observational Transition Systems (OTS), which are a proper sub-class of behavioral specifications. The OTS formalization is naturally implemented in algebraic specification languages like CafeOBJ [6] and thus OTSs have been used for the specification and verification of many real life systems.

Formally an OTS is a transition system that can be written in terms of equations. Assuming there exists a universal state space Y, and that each data type we wish to use is already defined in terms of initial algebra, an OTS S is defined as the triplet $S =< O, I, T >$, where [7]:

- O is a finite set of observers. Each $o \in O$ is a function $o : Y \to D$, where D is a data type and may differ from observer to observer. Given an OTS S and two states $u_1, u_2 \in Y$ the equivalence between them wrt. S is defined as $\forall o \in O, o(u_1) =_S o(u_2)$.
- I is the set of initial states for the system, i.e. $I \subseteq Y$.

- T is finite set of transition functions. Each $\tau \in ?$ is a function $\tau : Y \to Y$, such that $\tau(u_1) = \tau(u_2)$ for each $[u] \in Y/_{=_S}$ and $u_1, u_2 \in [u]$. $\tau(u)$ is called the successor state of u with respect to S. Also with each τ, comes a condition $c - \tau$, called the effective condition of τ, such that $\tau(u) =_S u$ if $\neg(c - \tau(u))$.

Observers and transitions are usually parameterized, generally they are denoted as $o_{i_1 \ldots i_m}$ and $\tau_{j_1 \ldots j_n}$ provided that there exist data types D_k and $k \in \{i_1, \ldots i_m, j_1, \ldots j_n\}$ with $m, n \geq 0$.

Design by contract (DBC) is a method for developing object oriented software which has been first proposed in [8] in 1992 . The basic idea of this approach is that the class' methods and the programs that invoke them have a so call contract between them. On the one hand the invoker of a method guarantees certain pre-conditions before calling the method and on the other hand the method is able to guarantee that certain post-conditions will hold after its execution. The main contribution of DBC is that these contracts are executable, meaning that they are defined in the program code in the programming language and are translated into executable code by the compiler. This makes possible that any violation of the contracts may occur while the program is running, will be detected immediately.

Various compilers/libraries have been developed for programming languages to enable DBC functionality. For example, modernJass[9], cofoja[10] and JML[11] for Java, spec#[12] for C# and jsContract[13] for Javascript.

In JML, a specification is created using special annotated comments, which start with the sign `@`. The specification of the obligations of the calling program is denoted by using the key words `//@ requires`. The methods' post-conditions, that the implementer has to meet, are denoted using `//@ensures`. A contract in JML is a set of pre and post conditions for a method. Such contracts can be used as an abstraction of a methods' behavior. JML uses Java's expression syntax to write the predicates that are used to express the assertions, and the pre and post conditions. The lack of expressiveness of the Java language to write formal specifications is solved by extending Java expressions with the required specification contracts (for example quantifiers [11]). The semantics of a JML method's specification is that when the method is invoked in a state where its pre-conditions are met, then one of the following two things will happen. Either the method terminates normally, in which

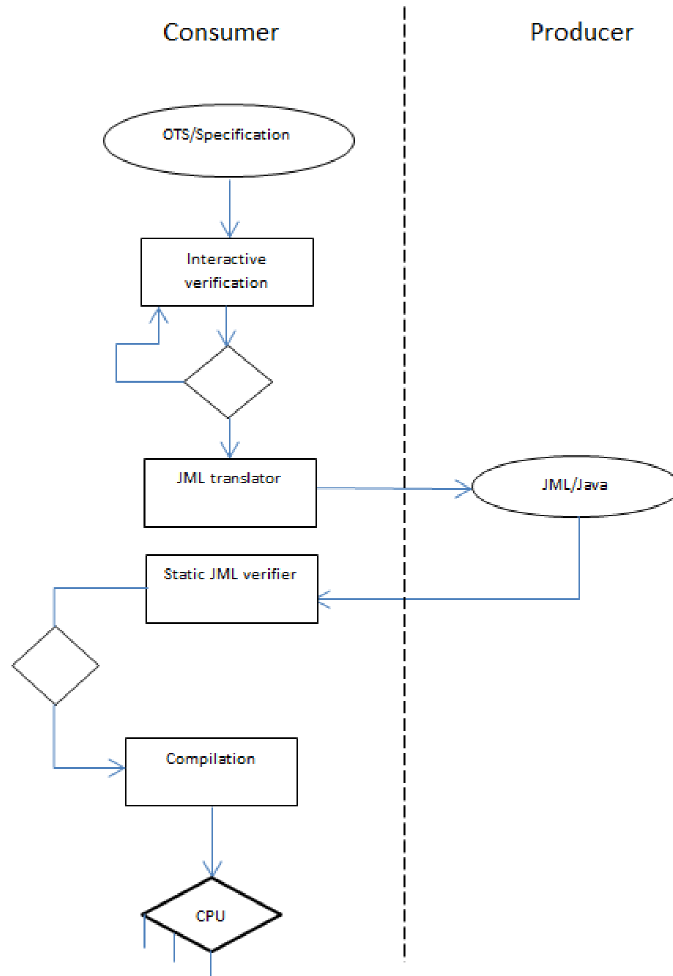

Figure 2: architecture of the proposed system

case the post-conditions of the specification are guaranteed, or the method throws an exception, in which case the exception thrown must be permitted by the specification (either by default or explicitly) and the exceptional post-conditions must be satisfied [11] (these conditions are denoted by the signals clause).

In addition, JML is designed to be used with a wide number of different tools [14, 15]. The range of these tools is from DBC support to runtime assertion checking (RAC) and to static verification (SV) of specifications using theorem provers [1]. Here, we are interested in SV tools since we wish to be able to verify the code before it is executed. Several SV tools have been developed for JML, like ESC/JAVA2[16], TACO[17], KeY[18] and so on. These permit to the user to check for inconstancies between the code and the JML annotations.

4 Proof Carrying Algebraic Specification Framework

One of the main road blocks for the full scale adoption of PCC is the obligation of the code producer to provide a formal proof that the code satisfies a set of safety properties. This is a role that the code producers are not comfortable with, particullarly since the verification process requires human interaction for the verification of some proof obligations. Another road block is the lack of reusability of the safety properties. An improvement to the PCC schema would be a framework where the producer is not oblidged to prove the compliance of the code and at the same time the compliance is easily verified by the consumer. Also, a degree of reusability of the safety policies is desired.

Figure 2, shows the process of generating and using proof carrying code with the proposed framework (JML could be replaced by any other DBC language). We distinguish the following steps:

- Specification. A specification of the system and the behaviour of an arbitrary program is created by the consumer, using hidden algebra.
- Verification of safety. A proof is conducted by the consumer, that the specification meets the desired safety properties.
- Translation and Transfer. The specification gets translated into a language that is easy to check against implementation and that the code producer can understand. For this we used DBC languages. This is the safety policy sent to the producer.
- Implementation. The code producer sends the program to the consumer.
- Verification of compliance. The consumer statically verifies that the recieved code is compliant to the specification.
- Authorization. If the previous step is successful the code is authorized for execution.

Initially the code consumer creates an abstract OTS specification of the system that will execute the mobile code, we will refer to this OTS as S_{sys}. Next a very abstract specification of arbitrary mobile code is created, we will

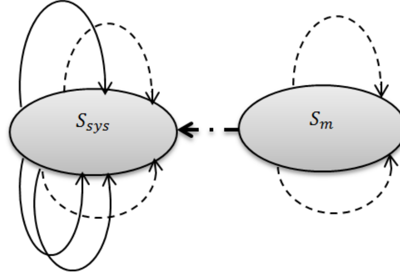

Figure 3: a safety policy

refer to this as S_m. One way to specify arbitrary code is by having S_m import S_{sys}. The transitions of S_m are then defined as a mapping of a subset of the transitions of S_{sys}. They correspond to the system functions the mobile code can invoke. Their effective conditions define the preconditions for using them and the equations defining declare their post-conditions. So in this way the specification of the mobile code defines the interface for the communication with system. Note that only the specification of the arbitary mobile code has to be made public, thus ensuring information hidding on the part of the consumer if required.

They only way for the foreign code to interact with (change the state of) the system is through those transitions. Thus, if we define the way that these transitions are allowed to be used and verify that under these rules the safety properties of interest hold, we can be certain that any instatiation of this loose specification of mobile code will not violate the safety of the system. Such a specification can be seen in figure 3. The ovals represent the state space of the OTSs and the arrows the behavioral operators. The dotted arrows in the two OTSs denote the subset of transitions from S_{sys} that are mapped to transitions of S_m. The dotted arrow from S_m to S_{sys}, denotes a special kind of behavioral operator called projection. The projection given a state of S_m returns the corresponding state of S_{sys} [19].

Next the consumer verifies that this specification satisfies the desired safety properties and makes adjustments if necessery. This part of the verification is computer human interactive. Experience has shown that behavioral specifications can be used to verify the most complext of liveness or safety properties. In addition in the OTS framework proofs can be reused just like objects to

create more complex proofs. Thus, it is possible to built libraries of proofs about specifications, that the consumer can use to verify properties in minimum time. For each system the verification of the safety properties needs only be done once and it is not carried out by the producer. Instead it is carried out by the consumer, who is much more confortable in this role since he wants to ensure that his system will not be compromised.

The following step is to translate this mathematical specification to a form that is easy to verify against an implementation. Also, we wish for the specification to be easily understood by the code producer. Design by contract languages (DBC), such as JML, fit this role prefectly. Firstly, they are expressed in a programming language like syntax, so the programmers can understand the specification without much trouble. Secondly, there are numerous tools that allow the automatic verification that an implementation complies with a DBC specification. Thirdly the contracts can be discarded (treated as comments), if that is required. So the specification can have no impact on the execution of the program. This translation is transfered to the producer.

Upon receiving the translated specification the producer can proceed to the implementation of the mobile programm. Also he can (optionally) statically verify if his implementation is compliant with the received specification. Once the code is finished it can be sent to the consumer.

The consumer receives the mobile code and statically verifies it complies with the specification using the plethora of available such DBC tools. If the verification is successful the code is compiled if necessery (compilation isn't required for scripting languages) and is authorized for execution.

4.1 OTS to DBC

A key component to the proposed framework is the existance of a correct translation from OTS specifications to the various DBC specifications. We are already working in that direction and have produced some first results for a translation of OTS/CafeOBJ to JML specifications in the report [20]. A proof that such translations are correct can be conducted using the definition of equivallence for behavioral specification []. A behavioral specification (like an OTS) is defined by a set of equations E created using a hidden subsignature Γ of Σ. Formally $\mathfrak{B}(\Sigma, \Gamma, E)$. Two behavoral specifications over the same

hidden signature (like an OTS and its translation to JML) $\mathfrak{B}(\Sigma, \Gamma, E_1)$ and $\mathfrak{B}'(\Sigma, \Gamma', E_2)$ are equal iff all operations in Γ' are behaviorally congruent for $\mathfrak{B}$ [21]. We intend to provide such a proof for our translation in the near future.

5 Conclusions

He have proposed a framework for PCC in which the producer is not required to conduct tedious proofs about the safety of the mobile code. In addition the proposed definition of safety policies allows their reuse. These two points have been considered as roadblocks to the adoption of PCC so far. These results can be achieved through the combination of two independent methodologies. Hidden algebra and design by contract. We use the former to specify the behavior of the system and of mobile code. This permits safe and sound proofs of the properties of interest. Such specifications however can not be easily linked to the implementation of the mobile code. In addition, programmers usually are not accustomed to such formalisms. We overcome these difficulties by proposing the translation of hidden algebra specifications to design by contract specifications. After such a translation the specification can be sent to the producer. Once the implementation is ready the consumer can statically and automatically verify its compliance with the spefication. We believe that the shifting of roles (moving the proof of the properties) from the producer to the consumer better reflects reality, where the consumer is better motivated to invest extra effort for the protection of his system. Also, because the specifications and proofs can be reused we believe that the burden will be dimished to some degree. Fundamental issues for the proposed framework, remain the correct translation of the specifications. We have already taken steps in that direction.

Acknowledgements. This paper was supported by the THALIS project "Algebraic Modeling of Topological and Computational Structures and Applications". The Project THALIS is implemented under the Operational Project Education and Life Long Learning and is co-funded by the European Union (European Social Fund) and National Resources (ESPA).

References

[1] G.C. Necula, Proof-Carrying Code, *Proceedings The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, (January, 1997), 106–119.

[2] M. Mahajan, Proof-Carrying Code, *INFOCOMP Journal of Computer Science*, **6**(4), (2007), 1–6.

[3] A.W. Appel, Foundational Proof-Carrying Code, *Proceeding 16th Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, USA, (2001), 247–257.

[4] J. Goguen and G. Malcolm, A hidden agenda, *Theoretical Computer Science*, **245**(1), (August, 1996), 55–101.

[5] J. Goguen and J. Meseguer, Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations, *Theoretical Computer Science*, **105**(2), (November, 1992), 217-273.

[6] CafeOBJ homepage, http://www.ldl.jaist.ac.jp/cafeobj/

[7] K. Ogata and K. Futatsugi, Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method, Algebra Meaning and Computation, *Lecture Notes in Computer Science*, **4060**, (2006), 596–615.

[8] B. Meyer, Applying "Design by Contract", *Journal Computer archive*, **25**(10), (1992), 40–51.

[9] ModernJass homepage, http://modernjass.sourceforge.net/

[10] Cofoja homepage, code.google.com/p/cofoja/

[11] G.T. Leavens and Y. Cheon, *Design by Contract with JML*, 2004.

[12] M. Barnett, K. Rustan, M. Leino and W. Schulte, In CASSIS 2004, LNCS 3362, Springer, 2004.

[13] Jscontract, http://kinsey.no/projects/jsContract/

[14] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby and D.R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, *Science of Computer Programming - Formal methods for components and objects pragmatic aspects and applications*, **55** (1-3), (March, 2005), 185 - 208.

[15] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe and R. Stata, Extended static checking for Java, *Proceedings of the Conference on Programming Language Design and Implementation*, (2002), 234-245.

[16] P. Chalin, J.R. Kiniry, G.T. Leavens, and E. Poll, Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2, *Proceedings of FMCO*, (2005), 342–363.

[17] Taco home page, http://www.dc.uba.ar/inv/grupos/rfm_folder/TACO

[18] KeY home page, http://www.key-project.org/

[19] R. Diaconescu, Behavioural Specification for Hierarchical Object Composition, *Lecture Notes in Computer Science*, **3188**, (2004), 134–156.

[20] N. Triantafyllou, P. Stefaneas and P. Frangos, OTS/CafeOBJ2JML: An attempt to combine Design By Contract with Behavioral Specifications, *CORR*, **abs/1205.5106**, (2012).

[21] J.A. Goguen and G. Rosu, Hidding more of Hidden Algebra, *Proceeding FM '99 of the Wold Congress on Formal Methods in the Development of Computing Systems*, **II**, (1999), 1704–1719.