Static Analysis Techniques for Robotics Software Verification

Agostino Cortesi
Università Cà Foscari
Venice, Italy
cortesi@unive.it

Pietro Ferrara
ETH Zurich
Switzerland
pietro.ferrara@inf.ethz.ch

Nabendu Chaki
University of Calcutta
Kolkata, India
nchaki@gmail.com

Abstract-- **We overview the main semantics-based static analysis techniques for software verification: Data-flow analysis, Control-flow Analysis, Model Checking, and Abstract Interpretation. The complexity of control software, lying at the core of robotic systems, and the intensive use of numeric values pose several challenges for the formal verification of either functional or non-functional properties.**

Index Terms—**Static Analysis, Formal Verification, Robotics Software, Abstract Interpretation.**

I. Introduction

Programming robotics software involves reasoning through complex system interactions among sensors, actuators, intelligence and control processors. This challenging and error-prone process requires strong collaboration between engineers and software programmers, and the resulting code sometimes lacks reliability due to the presence of bugs or improper response to hardware failures.

The two main current practices to face this issue are: (1) adopt a (possibly object oriented) model-based programming approach, and use (possibly interactive) robotics simulation environments since the design phase, and (2) systematically apply testing techniques incorporating automated tests, online and offline analysis and software-in-the-loop tests in combination with real robot hardware.

However, this is not often sufficient to guarantee the required behavior, and this is a crucial issue in particular when dealing with robots systems where safety of operations is crucial. Here, formal automatic verification techniques become necessary. The adoption of semantics-based static analysis techniques may in fact certify the reliability of the resulting software, and it may also dramatically reduce the testing effort.

In this paper, we discuss the main features of the main static analysis techniques, namely data-flow analysis, control-flow analysis, model-checking and abstract interpretation. This might provide robotics software developers useful hints about which is the most appropriate approach to follow depending on the kind of analyzed property and software system.

These techniques has to be automatic, provably sound (i.e., semantically correct), though not necessarily complete. This makes them different from other static verification approaches like code surfing or manual source code review can be summarized as follows. Incompleteness means that given a program P and a property p, the result of the analysis must be either "every execution of P satisfies p" or "I don't know". In the second case, there could be actually an execution of P that does not satisfy p, or we could have a false positive because of a loss of accuracy in the analysis.

II. Verification Issues in Robotics Software

Writing software for robots is a difficult task, as it is often structured as a deep stack starting from driver-level layers (managing sensor/actuator hardware components) and continuing up through abstract reasoning, and beyond [29]. Moreover, most manufacturers of robot hardware also provide their own software, leading to the lack of standardization of programming methods for robot software. This is why robotics software architectures often support large-scale software integration efforts, where the layers may deeply differ with respect to programming paradigm, programming language, and program development platform.

As a first consequence, any classical program verification approach based on a programming environment supporting pre/post condition constraints may be not an effective solution in this scenario (and we will not consider it in the rest of the paper). In fact, it assumes full control over the whole programming development, whereas the software to be integrated comes too often from different and non-standardized programming development environments.

The presence of different layers of software leads to the need of a suite of different verification tools, where the most difficult task remains the verification of the so called emerging properties of the system, the ones that do not immediately rely on components' features but emerge only after their integration.

III. Semantics-Based Static Analysis Techniques

In this Section, for each of the analysis techniques mentioned in the Introduction, we will briefly describe

how the program is represented, how the property to be analyzed is represented, how the analysis process is expressed, how the analysis result is computed, and a simple applicative example.

A. Data-Flow Analysis

In Data-Flow Analysis [23,28,30,31], the program is seen as a *graph*, whose nodes are elementary basic-blocks of statements and edges depict how control may pass from one node to another. Each node is associated to an *in-set* and an *out-set* of elements, describing respectively the state of the program variables when entering or exiting the node with respect to the analyzed property. So, the property is expressed by elements of a suitable set ranging over syntactic program elements (variables, expressions, etc.) and node labels. The analysis is computed as the least solution of a pair of mutually recursive equations stating, for each node, (1) the effect of the statement on the in/out-sets, and (2) how the in/out-set information associated to a node propagates to the adjacent nodes (predecessor/successor nodes in the control graph). For the termination of the analysis, it is sufficient that the transfer functions are monotone and closed under composition, and that the values in the in-/out-sets range into a partial ordered structure satisfying the ascending chain condition [28].

Consider, for instance, the following slice of code:

```
timer = 0; arm_status=0; step=D;
 while (timer < N){
   timer = timer +1;
   arm_status= arm_status + step;
 }
```

The data-flow analysis needs to work on its associated control-flow graph:

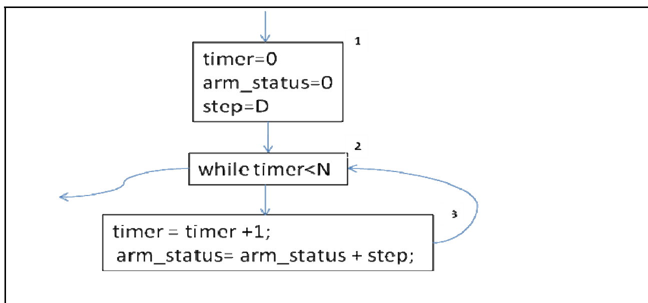

Fig.1 : Graph representation of a program for Data-Flow Analysis

We are interested to establish, for each program point, which assignments may have been made and not overwritten when program execution reaches this point along some execution path (reaching definition analysis). This property is represented by means of sets of pairs (x,l) where x is a variable and l is a label, associated to each nodes' entrance/exit. The mutually recursive equations are:

$$\textit{out-set}(n) = (\textit{in-set}(n) - \{(x,m) \mid x \text{ is defined in } n\}) \cup \{(x,n) \mid x \text{ is defined in } n\}$$

$$\textit{in-set}(n) = \cup \{\textit{out-set}(m) : n \text{ is a successor of } m\}$$

The iterative computation of the least solution of these equations, starting from the empty-set, leads to a fix-point which says, for instance, that the only definition that affects variable `step` at node 3 is the one at note 1, and so we can optimize the code getting rid of the variable `step`, using D in node 3 in place of `step`.

Typical examples of properties properly computed by data-flow analysis include constant propagation, variable liveness, reaching definition, available expressions, just to name a few.

B. Control-Flow Analysis

We have just seen that Data-Flow Analysis takes as a starting point a graph representation of the program, capturing the control flow in its executions. Depicting the control flow graph is straightforward for programs written in imperative or object oriented programs. Instead, this might not be the case when dealing with higher-order or concurrent languages, and in particular when there is no static control-flow graph at compile-time, as in the case of communication protocols [3,5,6,8,17, 28,32].

For example, in a programming language with higher-order functions like Scheme, the target of a function call may not be explicit: in the isolated expression `lambda(f)(f x)` it is unclear which procedure f may refer to. In order to collect the possible targets, we must consider where this expression can be invoked, and what argument it may get as an input. Hence, the aim of a control-flow analysis is to compute information about the *behavior of a process* and to store it in some data structures, called analysis components. The result of a control-flow analysis can be expressed by a pair (C,r). The first component C serves as a cache associating abstract values with each labeled program point, whereas the second component r serves as an environment associating abstract values with each variable. The technique consists in four steps: (i) acceptability conditions for pairs (C,r) are stated w.r.t. expressions, (ii) a syntax-directed specification of the analysis is expressed by a set of constraints that capture the impact of each statement on the pairs (C,r), (iii) a fixpoint iterative algorithm is applied to find a minimal solution of this set of constraints, and (iv) the solution is proven to satisfy the acceptability condition above, and it may be depicted graphically.

Figure 2 depicts a higher order functional language expression, and the control graph generated by the analysis [32].

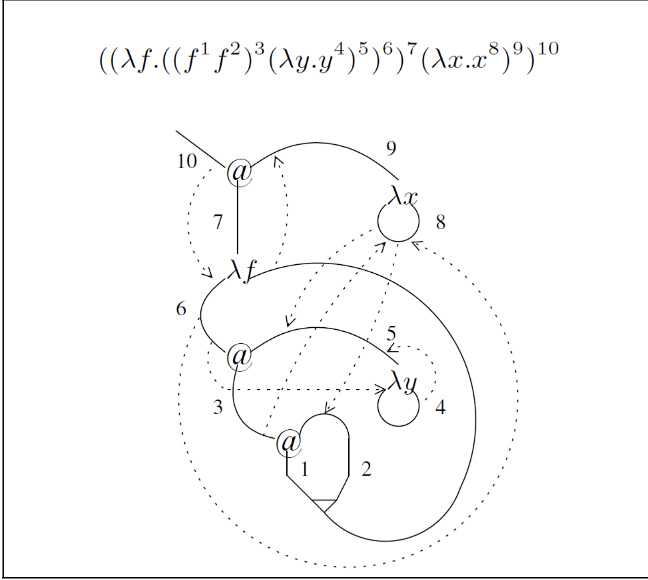

Fig.2 : Control-Flow Analysis of a functional program

C. Model-Checking

A model checker checks whether a system, interpreted as an automaton, is a Kripke model of a property expressed as a temporal logic formula [1,7,15,18,22,33]. The core idea of the model checking technique is to model the behavior of real-time systems over time by following this procedure: (1) the system is represented by a finite state labeled automaton, (2) the property to be verified is expressed by a temporal logic formula in a linear-time or branching time temporal logics, (3) each state of the automaton is associated with elementary properties that are true when the system is in that state, and (4) by structural induction on the logic formula, a fixpoint algorithm associates to each state of the automaton all the the subformulas of the formula to be verified that are true or false in that state. At the end of the procedure, if the initial formula is true in the initial state of the automaton, then the property is verified.

The main advantage of model checking is that it provides information even in the case when the property to be analyzed cannot be formally proven by the analysis. In fact, in this case, a counterexample is generated by the analysis itself, giving evidence to at least an execution path that deserves to be deeply analyzed in order to fix the program. However, the price to pay for this additional feature is the very strong finiteness constraint on the initial software system to be analyzed, and a serious scalability issue due to state explosion in the construction of the automaton starting from the source code.

As an example of application of the model checking technique, consider for instance the hybrid automaton depicted in Fig.3. It models a robot that works on a conveyor belt with two boxes [36]. Variable *d* represents the clock. At the beginning, the robot is looking to the belt (*d_stay*). Then, when there are two boxes in the belt (*s_ready*), it picks them up (*d_pick*), it turns right (*d_turnright*), it puts them down (*d_putdown*), and it turns left (*d_turnleft*), waiting for other boxes (*d_stay*). All these actions have some timing represented by bounds on *d*. Each component of the system is modeled by a particular automaton. For instance, each box is modeled by an automaton aimed at checking if the box stays in the belt, it is picked up by a robot (*d_pick*), the robot put it down (*d_put*), or it falls. On these automata one may want to prove that the boxes never fall.

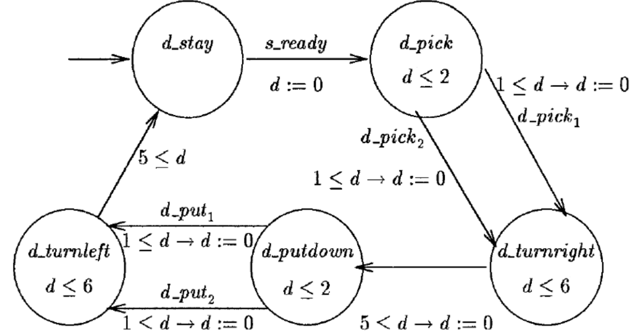

Fig. 3: A robot automaton

Model checking applies on the robotic system at high level, and usually the model of the system is manually written by an expert user. Therefore, model checking does not deal with the implementation of the system.

D. Abstract Interpretation

Abstract interpretation can be seen as the most general setting to express and compute static analysis [4,9,10,11,12,19,25].

In order to verify a behavioral property of a program, one needs to build up an abstraction on all the possible executions of the program, i.e., on the set of all its (possibly infinite) traces focusing on that property. Therefore, a verification process can be expressed in terms of a sound and computable abstraction of the concrete semantics (the latter is called "abstract semantics"). If the concrete semantics is expressed in terms of sequences of memory states, then data-flow analysis can be seen as an abstraction that just focuses on properties of variables values. When considering instead control sequences in the concrete program traces, we may get a control-flow analysis. Finally, when the abstraction is expressed in terms of temporal logic formulas, model checking may be seen as an instance of the abstract interpretation framework too.

The two main key-concepts of abstract interpretation are (1) the correspondence between concrete and abstract semantics through Galois connections, and (2) the feasibility of a fixed point computation of the abstract semantics, through the fast convergence of widening operators.

As a classical example of application of abstract interpretation, consider the imperative code depicted in Fig. 4.

```
1: int x = 1;
2: while(x<=10000)
3:   x++;
4: println(x);
```

Fig 4: a loop counter

The semantics of this procedure can be computed by ranging the values of variable `x` on the domain of intervals instead of the domain of integer numbers. In particular, it can be computed as a fixpoint of the system of equations depicted in Fig.5, where x_i represents the possible values of variable `x` after program point `i`, and $\oplus$ is the operation on intervals that soundly approximates the binary sum operation on integers.

$$\begin{cases} x_1 = [1,1] \\ x_2 = (x_1 \cup x_3) \cap [-\infty, 9999] \\ x_3 = x_2 \oplus [1,1] \\ x_4 = (x_1 \cup x_3) \cap [10000, +\infty] \end{cases}$$

Fig 5: Abstract Program

By using both a threshold widening operator analysis, and by narrowing the solution through a chaotic decreasing iterative fixpoint computation, in about 15 steps we get to the solution depicted in Fig 6, which provides a very accurate over-approximation of the values that variable x may be assigned in any actual program execution.

$$\begin{cases} x_1 = [1,1] \\ x_2 = [1, 9999] \\ x_3 = [2, 10000] \\ x_4 = [10000, 10000] \end{cases}$$

Fig 6: Result of AI interval analysis

The results of abstract interpretation-based static analyses are "sound by construction". The accuracy of the analysis can be easily tuned at different levels of precision and efficiency by adopting various abstract domains. For instance, in the case of numerical properties, we can range from very efficient though not so informative domain like *Sign* or *Parity*, to infinite though still non-relational domains like *Intervals*, up to infinite relational domains like *Octagons* and *Polyhedra*, whose computational cost may anyway be acceptable for critical code. Usually, the static analyses based on abstract interpretation are completely automatic (e.g., the widening operator allows one to infer information on loops adopting infinite height domains like Intervals) and work directly on the code of the program.

	DFA	CFA	MC	AI
Program	Control-flow graph	Labeled source	Automaton	Source code
Property	Set of abstract representations	Set of abstract representations	Temporal logic formula	Lattice (Galois connect.)
Analysis	Two mutually recursive equations	Set of const-raints on cache + environment	Formulas propagation associated to state transitions	Abstract semantics
Result	Fixpoint	Fixpoint	Fixpoint	Fixpoint

Table 1: Features of Static analysis techniques

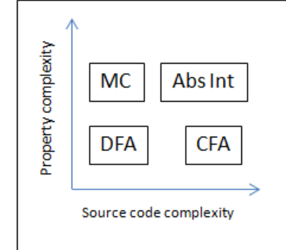

Fig 7: Application fields of Static Analysis Techniques

	DFA	CFA	MC	AI
Automatic	✓	✓		✓
Precise			✓	✓
Scalable	✓	✓		✓
Soundness	✓	✓		✓

Table 2: Evaluation of Static analysis techniques

IV. A Synopsys of Static Analysis Techniques

Table 1 summarizes the main features of the techniques described in the previous Section, by considering how the program and the analyzed property are represented, and how the analysis is defined. Observe that in all the cases the result is obtained by applying a fix-point algorithm, whose termination must be guaranteed in order to ensure the effectiveness of the analysis.

When focusing instead on the application target of the techniques above, we may classify them with respect to the complexity of the source program and of the property to be verified, respectively, as depicted in Fig. 7.

Finally, Table 2 reports a tentative evaluation of the various static analyses techniques. We follow four axes taken from [4]: automation, precision, scalability, and soundness.

On the one hand, data and control flow analyses are automatic, scalable, and sound, but they cannot deal with complex properties. On the other hand, model checking has been already applied to rather complex properties, but it is neither automatic (the model has to be manually specified), nor scalable (because of the state explosion problem), nor formally sound (there is usually no relation between the model and its actual implementation). The only approach covering all the four axes of evaluation is abstract interpretation. Nevertheless, the price to pay is higher in terms of user competence required to develop the analysis. In fact, while it is a relatively simple task, for an average skilled engineer, to draw an automata representation out of a robot model and verify it by a

model checker (like the one in Figure 3), it is a much more complex task the specialization of a generic abstract analyzer. The latter requires in fact the combination of strong and specific background both on programming language semantics and on algebraic structures.

The picture above is confirmed also by the use of the mentioned techniques as reported by the recent literature in the robotics area. In fact, model checking has been mainly applied for high- level verification of the robot model, and not much to the final implementation code [29]. Instead, abstract interpretation has been successfully applied to safety critical source code to automatically detect absence of run-time errors in standard programming languages like C [4], and it may cover also the interaction with database query languages [19]. Finally, data and control flow analyses are used inside of the compilers of domain specific programming languages mainly as a support for program transformation and generated code optimization [28].

V. Challenges of Robotics Software Verification

There are still various research challenges that have to be solved in robotics software verification. In particular, it is nowadays clear that static analysis techniques will be fundamental to produce robust robotics software. If on the one hand the problem has been already studied (and sometimes successfully solved) on particular systems and software, on the other hand a general solution has not yet been identified. This is mainly due to the fact that "most manufacturers of robot hardware also provide their own software. While this is not unusual in other automated control systems, the lack of standardization of programming methods for robots does pose certain challenges. For example, there are over 30 different manufacturers of industrial robots, so there are also 30 different robot programming languages required." [36]

The development of automatic static analyses of complex properties is a rather expensive and time consuming process. The wide spectrum of programming languages and robotics system often limits the benefits of static analysis techniques, since these could be only applied to particular sub-systems.

A unifying scenario would surely push to apply static analysis techniques to robotics software verification, since these static analyses could be applied to a broad range of software.

On the semantic level, the main open problem is how to cover the gap between the complex high level properties proved by model checking, and the source code level properties proved automatically by abstract interpretation. In particular, it would be quite relevant to prove that the actual implementation of a system corresponds to its high-level model.

Finally, it would be very important to further develop, both at theoretical and practical level, methods and tools for the static analysis of non-functional properties (like portability, usability, robustness, etc.) that play a crucial role in robotics software, but received very limited attention in the formal verification literature, with the notable exception of [10].

Acknowledgments

Work partiallly supported by PRIN "Security Horizons" project.

References

[1] Alur, R., C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. -H Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. 1995. "The Algorithmic Analysis of Hybrid Systems." Theoretical Computer Science 138 (1): 3-34.

[2] Bauer, N., S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. 2004. Verification of PLC Programs Given as Sequential Function Charts. Lecture Notes in Computer Science. Vol. 3147.

[3] Bertolino, Antonia and Martina Marre. 1994. "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs." IEEE Transactions on Software Engineering 20 (12): 885-899.

[4] Blanchet B, Mauborgne L, Cousot P, Miné A, Cousot R, Monniaux D, Feret J, Rival X. A static analyzer for large safety-critical software. ACM SIGPLAN Notices. 2003;38(5):196-207

[5] Bodei C, Buchholtz M, Degano P, Nielson F, Nielson HR. Static validation of security protocols. Journal of Computer Security. 2005;13(3):347-90.

[6] Braghin, C., Cortesi, A., Focardi, R. "Information flow security in Boundary Ambients". 2008. Information and Computation, 206 (2-4), pp. 460-489.

[7] Clarke, Edmund M., Orna Grumberg, and David E. Long. 1994. "Model Checking and Abstraction." ACM Transactions on Programming Languages and Systems 16 (5): 1512-1542.

[8] Cook B, Podelski A, Rybalchenko A. Termination proofs for systems code. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI); 2006. p. 415-26.

[9] Cortesi A, Filé G, Giacobazzi R, Palamidessi C, Ranzato F. Complementation in abstract interpretation. ACM Transactions on Programming Languages and Systems. 1997;19(1):7-47.

[10] Cortesi, A., Logozzo, F., "Abstract interpretation-based verification of non-functional requirements". 2005. Lecture Notes in Computer Science, vol. 3454, pp. 49-62.

[11] Cortesi, A., Zanioli, M. "Widening and narrowing operators for abstract interpretation". 2011. Computer Languages, Systems and Structures, 37 (1), pp. 24-42.

[12] Cousot P, Cousot R. Abstract interpretation and application to logic programs. The Journal of Logic Programming. 1992;13(2-3):103-79.

[13] Currie, David W., Alan J. Hu, Sreeranga Rajan, and Masahiro Fujita. 2000. "Automatic Formal Verification of DSP Software." Proceedings of the 37th Design Automation Conference , pp. 130-135.

[14] DeFouw, Greg, David Grove, and Craig Chambers. 1998. "Fast Interprocedural Class Analysis." Proc. ACM Symposium on Principles of Programming Languages , pp. 222-236.

[15] Dierks, H. 2004. "Comparing Model Checking and Logical Reasoning for Real-Time Systems." Formal Aspects of Computing 16 (2): 104-120.

[16] Dudek, Gregory, Michael Jenkin. 2000. "Computational principles of mobile robotics". Cambridge Univ Press.

[17] Dwyer, M. B., L. A. Clarke, J. M. Cobleigh, and G. Naumovich. 2004. "Flow Analysis for Verifying Properties of Concurrent Software Systems." ACM Transactions on Software Engineering and Methodology 13 (4): 359-430.
[18] Godefroid, Patrice. 1997. "Model Checking for Programming Languages using VeriSoft.". Conference Record of the Annual ACM Symposium on Principles of Programming Languages, pp. 174-186.
[19] Halder, R., Cortesi, A. "Abstract interpretation of database query languages". 2012. Computer Languages, Systems and Structures, 38 (2), pp. 123-157.
[20] Halder, R., Cortesi, A. "Abstract program slicing on dependence condition graphs". 2013. Science of Computer Programming. Article in Press.
[21] Thomas A. Henzinger, Pei-Hsin Ho: HYTECH: The Cornell HYbrid TECHnology Tool. Hybrid Systems 1994: 265-293, 1995.
[22] Henzinger, T. A., X. Nicollin, J. Sifakis, and S. Yovine. 1994. "Symbolic Model Checking for Real-Time Systems." Information and Computation 111 (2): 193-244.
[23] Horwitz, Susan, Thomas Reps, and Mooly Sagiv. 1995. "Demand Interprocedural Dataflow Analysis." Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Pages 104-115.
[24] Kramer J. and Scheutz M. 2007. "Development environments for autonomous mobile robots: A survey," Autonomous Robots, vol. 22, no. 2, pp. 101–132.
[25] Logozzo, F., Cortesi, A. "Semantic hierarchy refactoring by abstract interpretation". 2006. Lecture Notes in Computer Science, vol. 3855, pp. 313-331.
[26] Mantovani, Jacopo. 2008. "Automatic Software Verification for Robotics". AI Commun. 21(4): 263-264.
[27] Mertke, T. and G. Frey. 2001. "Formal Verification of PLC-Programs Generated from Signal Interpreted Petri Nets". Proceedings of the IEEE International Conference on Systems, Man and Cybernetics 2001, pp. 2700-2705.
[28] Nielson, Flemming, Hanne R Nielson, Chris Hankin. 1999. "Principles of program analysis". Springer-Verlag
[29] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, Andrew Y. "ROS: an open-source Robot Operating System", ICRA Workshop on Open Source Software, 2009
[30] Reps, Thomas, Susan Horwitz, and Mooly Sagiv. 1995. "Precise Interprocedural Dataflow Analysis Via Graph Reachability." Proc.22nd ACM Symposium on Principles of Programming Languages, pp.49-61.
[31] Sagiv, M., T. Reps, and S. Horwitz. 1996. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation." Theoretical Computer Science 167 (1-2): 131-170.
[32] Van Horn, D., Mairson, H.G. 2007. "Relating complexity and precision in control flow analysis." Proc. of the ACM SIGPLAN International Conference on Functional Programming, pp 85-96.
[33] Visser, W., K. Havelund, G. Brat, S. Park, and F. Lerda. 2003. "Model Checking Programs." Automated Software Engineering 10 (2): 203-232.
[34] Völker, N. and B. J. Krämer. 2002. "Automated Verification of Function Block-Based Industrial Control Systems." Science of Computer Programming 42 (1): 101-113.
[35] Yoo, J., Cha, S., & Jee, E. (2008). A verification framework for FBD based software in nuclear power plants. Paper presented at the Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 385-392.
[36] Wikipedia: "Robot Software". http://en.wikipedia.org/wiki/Robot_software