A Model Checker for Verifying ConGolog Programs

Leila Kalantari and **Eugenia Ternovska**
Department of Computing Science
Simon Fraser University, Burnaby, BC, Canada, V5A 1S6
{lkalanta,ter}@cs.sfu.ca

Abstract

We describe our work in progress on a model checker for verifying ConGolog programs. ConGolog is a novel high-level programming language for robot control which incorporates a rich account of concurrency, prioritized execution, interrupts, and changes in the world that are beyond robot's control. The novelty of this language requires new methods of proving correctness. We apply the techniques from XSB tabling and the μ-calculus, to overcome the challenge of verifying complex non-terminating programs, in a terminating time.

This note describes our work on a model checker (Clarke Jr., Grumberg, & Peled 1999) for verifying ConGolog programs. ConGolog is a programming language for high-level control of robots (De Giacomo, Lespérance, & Levesque 2000). The language is based on the situation calculus, a formal language for representing effects of actions (cf. (Reiter 2001)). ConGolog includes facilities for executing prioritized non-terminating concurrent processes, as well as facilities for dealing with interrupts and exogenous actions. This language differs from other concurrent languages in that the initial state does not have to be specified completely; also it allows the user to define primitive actions using axioms of the situation calculus. Due to the complex features of ConGolog that we stated above, manual error checking for ConGolog programs becomes almost an impossible task. This fact necessitates the automatic verification of ConGolog programs and also shows the difficulty of implementing it.

Let us consider two simple examples. Imagine an elevator controller program, called $Ctrl_1$, which picks a random floor, tests whether there is an outstanding request from that floor; and if so, it proceeds to serve that request. Another simple elevator controller program is called $Ctrl_2$. The elevator goes up as long as there is any unserved requests for the upper floors; and if not, it goes down as long as there is an outstanding request from a lower floor. On its way, the elevator tests whether there is an unserved request from the current floor; and if so, it serves it. For $Ctrl_1$, it is possible that some requests will be ignored forever since the floor to serve is picked randomly. If this happens, we say that *starvation* occurs. Notice that starvation does not happen for $Ctrl_2$, and any request will be served eventually. Our model checker should be able to discover the behavior of these two controllers. For $Ctrl_1$, it should detect the starvation property; and for $Ctrl_2$, it should verify that starvation will never occur.

The biggest challenge of verification of ConGolog programs is how to verify properties of non-terminating processes in terminating time. As pointed out in (De Giacomo, Lespérance, & Levesque 2000), this task is especially difficult in the presence of non-deterministic choice.

There are several approaches to formal verification. One of the most common ones uses branching time temporal modal logics for writing specifications. For the verification of ConGolog programs, we also use the branching time approach, which we need in order to deal with non-determinism of ConGolog programs. We use greatest and least least fixed points constructions, similar to those used in the propositional μ-calculus (cf. (Emerson 1990)) to express properties of non-terminating and non-deterministic processes. For example, the starvation property mentioned above can be formulated in the μ-calculus as

$$\mu Z_2\ (\exists n \exists s'\ (now = do(reqElevator(n), s') \wedge \phi) \vee \Diamond Z_2),$$

where

$$\phi := \nu Z_1\ (ButtonOn(n, now) \wedge \Diamond Z_1),$$

and now represents the current situation. This property expresses negation of the fairness requirement that every request is eventually served.

We were inspired by the ideas of (Ramakrishna *et al.* 1997) for dealing with non-termination because the abstract language for describing computational processes in (Ramakrishna *et al.* 1997) and ConGolog are both based on process algebra (Bergstra, Ponse, & Smolka 2001). We have exploited the tabling power of XSB logic programming system (cf. (XSB)) to deal with this challenge. We define a special program for exogenous (i.e., outside of robot's control) actions to model the environment. The program nondeterministicly chooses an action a, tests whether it is an exogenous action, and if so, it will execute it. To perform verification, we run each controller concurrently with a program which models the environment.

While designing the model checker, we faced the following difficulty. Along the computational tree, which represents all possible executions of a ConGolog program, the

term representing the current situation grows and never repeats itself. This fact makes it difficult to use XSB tabling to handle model checking for non-terminating programs.

Let us describe our approach to how to deal with this problem. All possible executions of a non-terminating program compose an infinite computational tree. We observed that, when trying to verify a property of a non-terminating program, it is enough to verify that property for an initial finite fragment of this tree. If the property holds in that fragment, we can conclude that it will hold for the whole infinite computational tree. Our justification of this observation relies on a repetition pattern which occurs in the computational tree — the state of the system, and the state of the program. The first repetition factor is the repletion of the state of the system along a computational path. Clearly, when the number of states is finite, the only way to obtain infinite computations is by looping through the transition system. So, there must be at least one state that repeats itself. The second repetition factor is related to the execution of ConGolog programs. There are two main constructs in the language of ConGolog by which we can obtain infinite computations. The first one is δ^*, which means that program δ is being repeated zero or more times. Clearly, after executing δ once, what is left to execute is, again, δ^*. So, the program repeats itself at some point in the tree of situations. This fact is reflected in the axiomatization of the predicates $trans$ and $trans^*$ (cf. (De Giacomo, Lespérance, & Levesque 2000)). The same argument applies to the $while$ construct, which is evident from the same axiomatization.

Our model checker works on the computational tree by using recursion on the subtrees. XSB tabling is used to terminate this recursion. When the same state of the system and state of the program is encountered, instead of going on with computation, XSB will make use of the results stored in its table. This way, we verify properties of non-terminating ConGolog programs in terminating time. At the moment, our model checker is restricted to non-terminating programs.

At the time of writing this note, we have not completed testing of the model checker on large programs. Occasionally, we obtain unexpected and contradictory results, and some of them seem to coincide with the known bugs of XSB system (Refer to the section "Restrictions and Current Known Bugs" of "The XSB Programmers' Manual"). We are trying to obtain a better understanding of the XSB system to find the origin of the occasional misbehavior. As soon as we get around these problems, we will proceed to prove the correctness of our implementation. We will also study to what extend we can eliminate any of our restrictions, most notably, the requirement of finite number of fluents (i.e., properties of the world which change with performing actions).

Acknowledgments

The first author is grateful to Dr. David Warren and Luis Fernando Pias de Castro for their help with the XSB system.

References

Bergstra, J.; Ponse, A.; and Smolka, S. 2001. *Handbook of Process Algebra*. Elsevier Science.

Clarke Jr., E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model Cheking*. MIT Press.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121:109–169.

Emerson, I. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*. 996–1072.

Ramakrishna, Y.; Ramakrishnan, C.; Ramakrishnan, I.; Smolka, S.; Swift, T.; and Warren, D. S. 1997. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, Lecture Notes in Computer Science. Haifa, Israel: Springer-Verlag.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press.

The XSB, a logic programming system developed at the State University of New York at Stony Brook. `http://xsb.sourceforge.net`.