Erschienen in: Algebra and Coalgebra in Computer Science : 4th International Conference, CALCO 2011, Winchester, UK, August 30-September 2, 2011, proceedings / Corradini, Andrea; Klin, Bartek; Cîrstea, Corina (Hrsg.). - Berlin : Springer, 2011. - (Lecture Notes in Computer Science ; 6859). - S. 378-385. - ISBN 978-3-642-22943-5
https://dx.doi.org/10.1007/978-3-642-22944-2_27

PREG Axiomatizer – A Ground Bisimilarity Checker for GSOS with Predicates*

Luca Aceto, Georgiana Caltais, Eugen-Ioan Goriac, and Anna Ingolfsdottir

ICE-TCS, School of Computer Science, Reykjavik University, Iceland
{luca,gcaltais10,egoriac10,annai}@ru.is

Abstract. PREG Axiomatizer is a tool used for proving strong bisimilarity between ground terms consisting of operations in the GSOS format extended with predicates. It automatically derives sound and ground-complete axiomatizations using a technique proposed by the authors of this paper. These axiomatizations are provided as input to the Maude system, which, in turn, is used as a reduction engine for provided ground terms. These terms are bisimilar if and only if their normal forms obtained in this fashion are equal. The motivation of this tool is the optimized handling of equivalence checking between complex ground terms within automated provers and checkers.

Keywords: Structural operational semantics, GSOS rule format, bisimilarity, equational axiomatizations, Maude.

1 Introduction

Proving that two process terms are related by some notion of behavioural equivalence is at the heart of the equivalence-checking approach to verification. In this paper we introduce a tool named PREG Axiomatizer[1] that tackles this problem focusing on ground (*i.e.*, fully specified) terms built using operations defined using the *preg* format, a *pr*edicates *e*xtension of the *G*SOS format presented in [3]. GSOS [8] is a restricted, yet powerful, way of defining Structural Operational Semantics (SOS) for programming and specification languages in the style introduced by Plotkin in [14]. We refer the reader to [3] for the detailed description and intuition behind the *preg* rule format and the considered notion of behavioural equivalence, which is a natural extension to predicates of the classic strong bisimulation equivalence.

Building on the techniques in [2,7], we proposed in [3] a procedure to construct a finite collection of sound equations that can be used to bring any ground term to a normal form. We showed that the normal forms of two terms are equal if and

* The authors have been been partially supported by the projects "New Developments in Operational Semantics" (nr. 080039021) and "Meta-theory of Algebraic Process Theories" (nr. 100014021) of the Icelandic Fund for Research. Eugen-Ioan Goriac is supported by the doctoral grant "Extending and Axiomatizing Structural Operational Semantics: Theory and Tools" (nr. 110294-0061) of the Icelandic Fund for Research.

[1] The tool is downloadable from http://goriac.info/tools/preg-axiomatizer/

only if the terms are bisimilar. Given a set of actions $\mathcal{A}$ and a set of predicates $\mathcal{P}$, the normal forms we refer to are terms built according to the grammar for finite trees with predicates, namely

$$s ::= \delta \mid \kappa_P \ (\forall P \in \mathcal{P}) \mid a.s \ (\forall a \in \mathcal{A}) \mid s + s,$$

that are of the shape $t = \sum_{i \in I} a_i.t_i + \sum_{j \in J} \kappa_{P_j}$. Here the P_j's are all the predicates satisfied by t, and the t_i's are terms in normal form. The empty sum ($I = \emptyset, J = \emptyset$) is denoted by the constant δ.

Intuitively, δ represents the process exhibiting no behaviour, $s + t$ is the nondeterministic choice between the behaviours of s and t, while $a.t$ is a process that first performs action a and behaves like t afterwards. For each predicate P we consider a constant κ_P, which denotes a process with no transitions. This process only satisfies P. A finite tree satisfies predicate P if and only if it has κ_P as a summand. We refer to predicates in $\mathcal{P}$ as *explicit predicates*. The operational semantics that captures this intuition is given by the rules of BCCSP extended with predicates. The SOS specification for this language consists of rules parameterized over all actions a and explicit predicates P:

$$\frac{}{a.x \xrightarrow{a} x} \ , \ \frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \ , \ \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \ , \ \frac{}{P\kappa_P} \ , \ \frac{Px}{P(x + y)} \ , \ \frac{Py}{P(x + y)}.$$

In [3] we showed that, for the above language, the following set of axioms [12] is sound and ground-complete for bisimilarity on the set of ground finite trees with predicates:

$$\begin{array}{cc} x + y = y + x & (x + y) + z = x + (y + z) \\ x + x = x & x + \delta = x \end{array}$$

Recall that our purpose is to find ground-complete axiomatizations like the one above for all the languages given in the *preg* format. In order to achieve this goal for operators whose rules involve negative premises, we use the *initial restriction operator* $\partial^1_{\mathcal{B},\mathcal{Q}}$ (where $\mathcal{B} \subseteq \mathcal{A}$ and $\mathcal{Q} \subseteq \mathcal{P}$ are the sets of initially forbidden actions and predicates, respectively). The semantics of $\partial^1_{\mathcal{B},\mathcal{Q}}$ is given by the following two types of transition rules:

$$\frac{x \xrightarrow{a} x'}{\partial^1_{\mathcal{B},\mathcal{Q}}(x) \xrightarrow{a} \partial^1_{\emptyset,\mathcal{Q}}(x')} \ \text{if } a \notin \mathcal{B}, \ \frac{Px}{P(\partial^1_{\mathcal{B},\mathcal{Q}}(x))} \ \text{if } P \notin \mathcal{Q}.$$

The axiomatization of the operators $\partial^1_{\mathcal{B},\mathcal{Q}}$ is provided in [3].

Internally, PREG Axiomatizer brings the provided rule system to a "manageable" format, introducing auxiliary operators as described in [3], and afterwards performs the axiomatization itself. The tool is implemented in the Maude language [11], which has been already proven to be very useful for analyzing SOS rule formats in [13,10]. Not only did we use Maude as a programming language, but also as an equational reduction system for the generated sets of axioms.

PREG Axiomatizer is, to our knowledge, the first public tool that automatically derives sound and ground-complete axiomatizations modulo bisimilarity

for GSOS-like languages. Prior to using the techniques presented in [2,3,7], one had to use ingenuity and dedicate a considerable amount of time in order to obtain axiomatizations for a language with even a limited number of operators.

The tool is generic, in the sense that the SOS specification defining the labelled transition system semantics of the process calculus is provided by the user. He or she does this in terms of *well-founded* GSOS systems, which means that these systems can be used to derive only finite labelled transition systems for the given terms (see [2] for more details). As presented in [9], the generated axiomatizations are guaranteed to be confluent, but, as a downside of our approach, only weakly normalizing. This downside is diminished by the fact that there exists a substantial decidable subclass of systems, namely the linear and syntactically well-founded ones [9], for which the generated axiomatizations are strongly normalizing. This subclass includes important languages such as CCS, CSP, and ACP.

2 Case Studies

In this section we present two scenarios involving several classic operations with their semantics extended with certain explicit predicates. Conventionally, the tool language accepts process term variables such as `X`, `X1`, `Y'`, actions like `a`, `b`, `c`, `a[0]`, `b[2]`, `c["name"]`, and predicates like `P`, `Q`, `P[1]`, `Q["prop"]`.

Example 1. Let us describe how PREG Axiomatizer is used in order to prove that "$a.(a.\kappa_{\downarrow};\ b.(a.\kappa_{\downarrow}))$" and "`while` $a.b.\kappa_{\downarrow}$ `do` $a.\kappa_{\downarrow}$" are bisimilar. Here `_;_` and `while_do_` are, respectively, the sequential composition and the process loop operators (presented in [8]) extended to the *preg* format with the immediate successful termination predicate $\downarrow$ (which we choose to denote by `P` in the specification for tool consumption). In Figure 1 we present the operational semantics for these operations with the rules given both in standard notation, as well as using the syntax supported by the tool.

$$\frac{x \xrightarrow{a} x'}{x;y \xrightarrow{a} x';y} : \begin{array}{c}\texttt{X -(a)-> X'}\\ \texttt{===}\\ \texttt{X ; Y -(a)-> (X' ; Y)}\end{array} \qquad \frac{x\downarrow\ \ y \xrightarrow{a} y'}{x;y \xrightarrow{a} y'} : \begin{array}{c}\texttt{P(X) , Y -(a)-> Y'}\\ \texttt{===}\\ \texttt{X ; Y -(a)-> Y'}\end{array}$$

$$\frac{x\downarrow\ \ y\downarrow}{(x;y)\downarrow} : \begin{array}{c}\texttt{P(X) , P(Y)}\\ \texttt{===}\\ \texttt{P(X ; Y)}\end{array} \qquad \frac{x\downarrow}{(\texttt{while}\ x\ \texttt{do}\ y)\downarrow} : \begin{array}{c}\texttt{P(X)}\\ \texttt{===}\\ \texttt{P(while X do Y)}\end{array}$$

$$\frac{x \xrightarrow{a} x'}{\texttt{while}\ x\ \texttt{do}\ y \xrightarrow{a} y;\texttt{while}\ x'\ \texttt{do}\ y} : \begin{array}{c}\texttt{X -(a)-> X'}\\ \texttt{===}\\ \texttt{while X do Y -(a)-> Y ; while X' do Y}\end{array}$$

Fig. 1. *preg* rule system for `_;_` and `while_do_`

The rules involving action a are also instantiated for b. After providing this specification, the user can press the button labelled "Axiomatize" and the tool generates a Maude specification including the axioms obtained by following the procedure described in [3]. We exemplify a small part of the output which consists of the axiomatization for the `while_do_` operator:

```
  eq while X0 + X1 do X3 = while X0 do X3 + while X1 do X3 .
 ceq while X do Y = a . (Y ; while X' do Y) if a . X' := X .
 ceq while X do Y = b . (Y ; while X' do Y) if b . X' := X .
 ceq while X do Y = k[P] if X := k[P] .
  eq while X1 do X2 = delta [owise] .
```

In order to check for the bisimilarity of the two process terms introduced at the beginning of the current example, one loads the generated specification and uses the Maude command `reduce`:

```
> reduce a . (a . k[P] ; b . (a . k[P])) ==
         while a . b . k[P] do a . k[P] .
result Bool: true

> reduce while a . b . k[P] do a . k[P] .
result PTerm: a . a . a . b . a . a . k[P]
```

We successfully used PREG Axiomatizer to further extend the operational semantics of $\_;\_$ with the predictable non-failure predicate $\neq \delta$ (which plays the role of the predicate "$\neq 0$" presented in [4]) with the rules: $\frac{x \xrightarrow{a} x' \;\; y \neq \delta}{x;y \xrightarrow{a} x';y}$ $\frac{x \neq \delta \;\; y \neq \delta}{(x;y) \neq \delta}$. We managed to test the property that $x;\delta$ and δ are bisimilar on various closed instantiations. It is worth noting that this property does not always hold for the initial version of $\_;\_$.

Example 2. In this example we show how we use our tool to obtain the execution tree of a network of communicating processes. This procedure is useful, for instance, when one needs to use an external model checker to verify if the communication protocol satisfies certain logical properties. Our example is based on a case study from [5].

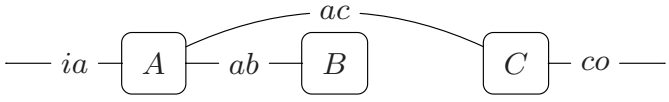

Fig. 2. Communication protocol

Consider the process network given in Figure 2 where A, B, C are the communicating processes and ia, ab, ac, co are the ports. The actions of sending, receiving, and synchronizing on the datum d over the port p are denoted by, respectively, $p!d$, $p?d$, and $p\#d$. By using these actions, the parallel composition operator $\_\|\_$, and the immediate successful termination predicate $\downarrow$, we specify the whole protocol as the term:

$$T = ia?d\,.\,(ab!d\,.\,\kappa_{\downarrow} \parallel ac!d\,.\,\kappa_{\downarrow}) \parallel ab?d\,.\,\kappa_{\downarrow} \parallel ac?d\,.\,co!d\,.\,\kappa_{\downarrow}\ .$$

We present *preg* rules for $\_\|\_$, in which $act \in \{p!d, p?d, p\#d\}$:

$$\frac{x \xrightarrow{act} x'}{x \parallel y \xrightarrow{act} x' \parallel y} \qquad \frac{y \xrightarrow{act} y'}{x \parallel y \xrightarrow{act} x \parallel y'} \qquad \frac{x\downarrow \;\; y\downarrow}{(x \parallel y)\downarrow}$$

$$\frac{x \xrightarrow{p!d} x' \;\; y \xrightarrow{p?d} y'}{x \parallel y \xrightarrow{p\#d} x' \parallel y'} \qquad \frac{x \xrightarrow{p?d} x' \;\; y \xrightarrow{p!d} y'}{x \parallel y \xrightarrow{p\#d} x' \parallel y'}$$

Fig. 3. *preg* rule system for _||_

The input for PREG Axiomatizer consists of:

- the predicate rule $\dfrac{\texttt{P(X) , P(Y)}}{\texttt{P(X || Y)}}$ `===` ,
- all the instantiations of the first two transition rules in Figure 1 for which *act* is an action from the set $\mathcal{A} = \{ia?d, ab!d, ac!d, ab?d, ac?d, ab\#d, ac\#d, co!d\}$

$$\left(\textit{e.g.,} \quad \frac{\texttt{X -(a["ia?d"])-> X'}}{\texttt{X || Y -(a["ia?d"])-> X' || Y}} \text{ `===` } \right), \text{ and}$$

- all the instantiations of the last two transition rules in Figure 1 in which p is a port from

$$\{ab, ac\} \left(\textit{e.g.,} \quad \frac{\texttt{X -(a["ab?d"])-> X' , Y -(a["ab!d"])-> Y'}}{\texttt{X || Y -(a["ab\#d"])-> X' || Y'}} \text{ `===` } \right).$$

We generate the process network execution tree (consisting of 582 states) by calling the command `reduce` on the specification term T:

```
> reduce ((a["ia?d"] . (a["ab!d"] . k[P] || a["ac!d"] . k[P])) ||
          a["ab?d"] . k[P]) || a["ac?d"] . a["co!d"] . k[P] .
result PTerm: a["ab?d"] . (...) + a["ac?d"] . (...) + a["ia?d"] . (...)
```

The parallel composition allows for arbitrary interleavings of the actions in $\mathcal{A}$, but it does not enforce the communication over the ports *ab* and *ac*. Hiding these ports so that other processes cannot interfere with the internal communications is desirable. This can be done with the help of the *restriction operator* $\partial_{\mathcal{B},\mathcal{Q}}$, a generalization of the initial restriction operator $\partial^1_{\mathcal{B},\mathcal{Q}}$ presented in Section 1, that preserves the imposed restrictions throughout the whole computation, not only for the first step. Forbidding independent send and receive actions over the ports *ab* and *ac* is denoted by the term $\partial_{\{p!d,p?d \,|\, p\in\{ab,ac\}\},\emptyset}(T)$. In PREG Axiomatizer we use `%%[`$\mathcal{B}$`;`$\mathcal{Q}$`]` as a syntactic notation for $\partial_{\mathcal{B},\mathcal{Q}}$:

```
> reduce %%[ a["ab?d"] a["ab!d"] a["ac?d"] a["ac!d"] ; empty ](
          (( a["ia?d"] . (a["ab!d"] . k[P] || a["ac!d"] . k[P])) ||
```

```
            a["ab?d"] . k[P]) || a["ac?d"] . a["co!d"] . k[P]) .
result PTerm: a["ia?d"] . (a["ab#d"] .  a["ac#d"] . a["co!d"] . k[P] +
                           a["ac#d"] . (a["ab#d"] . a["co!d"] . k[P] +
                                        a["co!d"] . a["ab#d"] . k[P]))
```

We also tested our tool by generating the normal form of $a^3.\kappa_{\downarrow} \parallel b^3.\kappa_{\downarrow} \parallel c^3.\kappa_{\downarrow}$ and obtained the same "2 page long" execution tree showed in [6], consisting of 6927 states. Maude derives this execution tree in less than 500 milliseconds on a machine with a 2.53GHz processor and 4GB of RAM.

Example 3. We now show how PREG Axiomatizer is used in order to perform equational proofs when working with predicates that have implicit behaviour. Consider, for instance, the case of the *eventual successful termination predicate* $\lightning$. It represents the extension of $\downarrow$, introduced in the previous examples, with the requirement that if $t \lightning$ holds for a term t, then $a.t \lightning$ holds for any action a.

Recall from Section 1 that our approach is based on denoting the property $\lightning$ by using the explicit process constant $\kappa_{\lightning}$ as a summand of the analyzed term. The above characterization of $\lightning$ is given by the axiom $a.(t+\kappa_{\lightning}) = a.(t+\kappa_{\lightning})+\kappa_{\lightning}$. With this in mind, one could check, for instance, if a process t "eventually terminates" by checking if it is bisimilar to $t + \kappa_{\lightning}$.

In order to prove that $a.\kappa_{\lightning} \parallel b.\kappa_{\lightning}$ is bisimilar to $(a.\kappa_{\lightning} \parallel b.\kappa_{\lightning}) + \kappa_{\lightning}$ we need to let the tool "know" that it should treat $\lightning$ (denoted by `Q` in the specification) as an implicit predicate by using the operation `expandImplicit`. This operation receives a term and the set of implicit predicate names:

```
> reduce expandImplicit(a . k[Q] || b . k[Q], Q) ==
         expandImplicit(a . k[Q] || b . k[Q] + k[Q], Q) .
result Bool: True

> reduce expandImplicit(a . k[Q] || b . k[Q], Q) .
result PTerm: k[Q] + a . (k[Q] + b . k[Q]) + b . (k[Q] + a . k[Q])
```

Predicates with implicit behaviour, like $\lightning$, can only be used during the normalization process if the operators whose definition involves these predicates are given by rules that satisfy certain sanity constraints mentioned in [3]. The tool does not currently support the automated checking for those constraints, so the user needs to do it manually before using the feature presented above. The parallel composition operator does meet those constraints.

3 Discussion and Future Work

Aside from the features mentioned in Section 2, an important part of the PREG Axiomatizer engine is dedicated to checking for the conformance of specified operations and rules to the various formats presented in [3].

There are many areas in which the tool and the theory behind it can be improved. First and foremost, an important feature would be to allow the user to specify guarded recursively defined terms in order to greatly increase the complexity of the case studies our tool can handle. The most natural way to

extend our approach in order to reason about the bisimilarity of such terms is to integrate the technique presented in [1], which is also based on generating complete axiomatizations for a class of GSOS languages generating regular behaviours. The main difficulty of this task will be the search for good strategies for applying the axioms and the unique fixed-point induction rule.

Another tool development direction is concerned with the ability to automatically check if the specification meets certain complex requirements. One of these requirements is, as presented in Section 1, the syntactic well-foundedness of the given system. Without this feature the user needs to be careful not to specify operators such as the *reentrant server* $!\_$, defined by the rule $\frac{x \xrightarrow{a} x'}{!x \xrightarrow{a} x' \parallel !x}$, for which non-normalizing axioms are derived: $!x = !'(x, x)$, $!'(a.x', x) = a.(x' \parallel !x)$. Another requirement the tool could check for consists of the sanity constraints we mentioned in Example 3.

References

1. Aceto, L.: Deriving complete inference systems for a class of GSOS languages generation regular behaviours. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 449–464. Springer, Heidelberg (1994)
2. Aceto, L., Bloom, B., Vaandrager, F.: Turning SOS rules into equations. Inf. Comput. 111, 1–52 (1994), http://portal.acm.org/citation.cfm?id=184662.184663
3. Aceto, L., Caltais, G., Goriac, E.I., Ingólfsdóttir, A.: Axiomatizing GSOS with predicates (2011), http://ru.is/faculty/luca/PAPERS/extending_GSOS_with_predicates.pdf
4. Aceto, L., Cimini, M., Ingólfsdóttir, A., Mousavi, M., Reniers, M.A.: SOS rule formats for zero and unit elements. Theoretical Computer Science (to appear, 2011)
5. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge Tracts in Theoretical Computer Science, vol. 50. Cambridge University Press, Cambridge (2010)
6. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, New York (1990)
7. Baeten, J.C.M., de Vink, E.P.: Axiomatizing GSOS with termination. J. Log. Algebr. Program. 60-61, 323–351 (2004)
8. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. J. ACM 42, 232–268 (1995), http://doi.acm.org/10.1145/200836.200876
9. Bosscher, D.J.B.: Term rewriting properties of SOS axiomatisations. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 425–439. Springer, Heidelberg (1994), http://portal.acm.org/citation.cfm?id=645868.668513
10. Chalub, F., Braga, C.: Maude MSOS Tool. Electron. Notes Theor. Comput. Sci. 176, 133–146 (2007), http://portal.acm.org/citation.cfm?id=1279349.1279455
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)

12. Milner, R.: Communication and Concurrency. Prentice-Hall International, Englewood Cliffs (1989)
13. Mousavi, M.R., Reniers, M.A.: Prototyping SOS meta-theory in Maude. Electron. Notes Theor. Comput. Sci. 156, 135–150 (2006), http://dx.doi.org/10.1016/j.entcs.2005.09.030
14. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)