TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Proceedings of INFINITY '98

Javier Esparza

TUM-I9825
SFB-Bericht Nr. 342/09/98 A
Juli 98

TUM–INFO–07-I9825-150/1.–FI

Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1998 SFB 342 Methoden und Werkzeuge für
die Nutzung paralleler Architekturen

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

INFINITY '98

3rd International Workshop on
Verification of Infinite State Systems

Aalborg, Denmark
July 18, 1998
(A Satellite Workshop to ICALP'98)

Supported by the Sonderforschungsbereich 342
„Tools and methods for parallel computer architectures“

PROGRAM

Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays (Extended Abstract)

Ranko Lazić*† Bill Roscoe‡

To be presented at INFINITY '98
(Revised version. July 7, 1998.)

Abstract

Our main result says that determinism of a concurrent system which uses unbounded arrays (i.e. memories) can be verified by considering an appropriate finite array size.

That is made possible by restricting the ways in which array indices and values can be used within the system. The restrictions are those of data independence: the system must not perform any operations on the indices and values, but it is only allowed to input them, store them, and output them. Equality tests between indices are also allowed.

The restrictions are satisfied by many concurrent systems which use arrays to model memories or databases. As a case study, we have verified that a database system which allows users to lock, read and write records at multiple security levels is secure.

1 The Parameterised Verification Problem

Concurrent systems are frequently infinite-state because they have parameters that can vary unboundedly. For example, a memory cache is likely to be parameterised by the data types of addresses and values, a protocol for fault-tolerance is likely to be parameterised by the number of nodes in the network, etc. Given such a system, we typically want to be able to consider all possible instantiations of its parameters, rather than having to restrict our attention to one instantiation at a time. Thus we come to the Parameterised Verification Problem:

PVP. Given a parameterised concurrent system `P` and a condition `C`, does `P` satisfy `C` for all instantiations of the parameters? ■

There has recently been much research on the PVP. It is undecidable in general [AK86], and so most effort has been put into either finding decision procedures for restricted versions of it (see e.g. [Wol86, JP93, ID96a, HB95, HDB97, GS92, YJL96, EN96]), or providing automated techniques whose termination is unpredictable or which require user involvement (see e.g. [HL95, HGD95, CZ+97, HIB97, CGJ95, ID96b, KM+97]).

The main result of this paper, Theorem 1, says that in order to verify that a concurrent system `P` which uses unbounded arrays (i.e. memories) is deterministic for all array sizes (finite or infinite), it suffices to consider a single appropriate finite size. In other words, the theorem provides a decision procedure for the following restricted version of PVP: the parameters are data types of array indices and values, and the

*Oxford University Computing Laboratory, U.K. and Mathematical Institute, Belgrade University, Yugoslavia. Supported by a Junior Research Fellowship at Christ Church (Oxford), and previously by a Domus and Harmsworth Senior Scholarship at Merton College (Oxford) and by a scholarship from Hajrija & Boris Vukobrat and Copechim France S.A.

†The contact author. Address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, U.K. Tel: +44 1865 273 838. Fax: +44 1865 273 839. E-mail: `rsl@comlab.ox.ac.uk`.

‡Oxford University Computing Laboratory, U.K. Supported by grants from the U.S. Office of Naval Research.

condition `C` is the property of being deterministic. The procedure is guaranteed to terminate whenever the data types sizes which it suffices to consider can be computed in a finite time and the resulting instantiation of `P` is finite-state (which are both issues that are orthogonal to the parameterisation).

Since checking whether the appropriate instantiation of `P` is deterministic can be done by a single run of a model checker,[1] the paper contributes to turning model checkers into practical tools for *verification*, rather than primarily *refutation* [Rus97].

Before we present Theorem 1 in Section 4, we shall explain what we mean by arrays whose index and value types are parameters, and review determinism, a condition with very important applications in the field of computer security. In the remaining two sections, we shall outline the database system case study, and point towards future work.

In the space available, we could not afford to go into technical details. The technical report [LR98a] contains the full paper, including a sketch of the proof of Theorem 1.

2 Unbounded Arrays

The main novelty in the paper is that we consider concurrent systems which can use arrays whose indices and values come from variable types, i.e. data types that are parameters of the system. Such an array is unbounded because its size is the size of the type of its indices, and we allow type parameters to vary unboundedly.

Our main result, Theorem 1, is made possible by restricting the ways in which array indices and values can be used within the system. More precisely, the system is required to be *data independent* with respect to the types of indices and values: it must not apply any operations to elements of those types, but it is only allowed to input them, store them, and output them. Equality tests between indices are also allowed. From another point of view, we have shown that it is possible to relax the data independence conditions by allowing arrays and still obtain finite instantiations which suffice for verification. (A precise definition of data independence can be found in [Laz98b], and approximate definitions in [Ros98, LR98a].)

Many systems which use arrays to model memories or databases satisfy the data independence conditions on indices and values. A typical example is a memory cache which uses an array to model the main memory that it is interacting with, and whose replacement policy does not involve calculation. Indeed, the only operation such a cache needs to perform on indices or values is equality tests between indices.

Theorem 1 is also made possible by having the indices and values belong to *different* data types, so that for example nested indexing is not allowed.

For simplicity, we shall be considering a single variable type `X` of indices and a single variable type `Y` of values. Let `Array(X, Y)` denote the type of all arrays with indices from `X` and values from `Y`. Such arrays can also be thought of as maps from `X` into `Y`. For creating and using them, we allow only the following operations, whose meanings are self-explanatory:

```
init(X) :: Y -> Array(X, Y)
getval  :: (Array(X, Y), X) -> Y
update  :: (Array(X, Y), X, Y) -> Array(X, Y)
```

In addition to allowing arrays whose indices and values come from variable types, Theorem 1 allows uninterpreted many-valued predicates on variable types which are not types of array values, and uninterpreted constants of variable types. Here by "many-valued predicates" we mean functions into fixed finite types, and by "uninterpreted" we mean that the predicates and constants are just symbols and that the verification should establish that the condition (in this case determinism) is satisfied for all possible interpretations. Such predicates and constants will be needed in the database system case study (see Section 5).

[1] Determinism checking is supported directly in the model checker FDR2 [Ros98, FS97]. In other model checkers, some preliminary transformations may be necessary: for example, the condition of being deterministic can be expressed in a linear temporal logic on a product of the labelled transition system of `P` with itself.

3 Determinism

A concurrent system P is said to be *deterministic* if, from the point of view of its environment, it cannot exhibit any nondeterminism. More precisely, there must not exist an execution of a sequence t of communications after which P can accept some communication a and a possibly different execution of t after which P can refuse a. In the process algebra CSP [Hoa85, Ros98], it is appropriate to also require livelock (i.e. divergence) freedom, so that the condition of being deterministic can be formalised as:

- $(t^\frown\langle a\rangle, \{\}) \in \mathit{failures}_\perp(\mathtt{P}) \Rightarrow (t, \{a\}) \notin \mathit{failures}_\perp(\mathtt{P})$, and
- $\mathit{divergences}(\mathtt{P}) = \{\}$.

Determinism has been studied most extensively in CSP. Apart from its great significance in the theory, it has found very important applications in the field of computer security. Namely, it has been established [RWW96, Ros95, Wul97, Ros98] that the key to specifying that there is no information flow across a concurrent system from a high security user to a low security user is the determinism of a suitably formed abstraction: what the system looks like to the low user when the actions of the high user are turned into internal nondeterminism.

4 The Theorem

For concreteness and practical applicability, we have stated and proved Theorem 1 in terms of CSP_M [Sca98, Ros98], the machine-readable version of CSP which is the language used in the model checker FDR2 [Ros98, FS97].

The following quantities, defined for a concurrent system P and a variable data type X, will be used in the theorem to compute the sufficient size for the type of indices:

$W^{\mathtt{P}}_{\mathtt{X}}$ is the maximum number of elements of type X that P ever has to store for future use. Here it is crucial that we do not need to count elements which are stored only because they are indices of array components that have previously been assigned to or read from. This enables $W^{\mathtt{P}}_{\mathtt{X}}$, and hence the sufficient size for X in the theorem, to be finite even when P may during its execution access an unbounded number of different array components.

$L^{\mathtt{P}}_{?,\mathtt{X}}$ is the maximum number of elements of type X that can be input in a *single* communication of P.

$L^{\mathtt{P}}_{\overline{?},\mathtt{X}}$ is the maximum number of elements of type X that can be input in a *single* communication of P without being recorded in it. (This quantity can be made nonzero only by renamings which omit inputs from communications.)

$L^{\mathtt{P}}_{\sqcap,\mathtt{X}}$ is the maximum number of elements of type X that can be chosen in a *single* internal nondeterministic choice in P.

The quantity $W^{\mathtt{P}}_{\mathtt{X}}$ virtually always dominates the calculations. In fact, in most practical examples, including the case study in the next section, the remaining three quantities have values 1, 0, 0 or 1, 0, 1 respectively.

For simplicity, we state the theorem with only one uninterpreted many-valued predicate and only one uninterpreted constant.

Theorem 1 *Suppose* P *is a concurrent system with two variable types* X *and* Y *with respect to which it is data independent, except that it can use:*

- *arrays of type* `Array(X, Y)`*, i.e. arrays with indices from* X *and values from* Y*,*
- *an uninterpreted K-valued predicate* r *on* X*, and*
- *an uninterpreted constant* c *of type* Y*.*

Suppose also that `P` *does not use any equality tests between elements of type* `Y`*, and that no state of* `P` *has two possible communications that differ only in outputs of type* `Y`*. (Two mild regularity conditions are also needed: see [LR98a].)*

Let $B = 2 \times W^{\mathtt{P}}_{\mathtt{X}} + \max(L^{\mathtt{P}}_{?,\mathtt{X}} + L^{\mathtt{P}}_{\overline{?},\mathtt{X}}, L^{\mathtt{P}}_{\sqcap,\mathtt{X}})$.

Then `P` *is deterministic for all instantiations of* `X`*,* `Y`*,* `r` *and* `c` *(with finite or infinite* `X` *and* `Y`*), provided it is deterministic for the following instantiation:*

- `X` *is defined to be any type with exactly* $(B+2) \times K$ *elements,*
- `Y` *is defined to be any type with exactly* 2 *elements, say* `{0, 1}`*,*
- `r` *is defined to map exactly* $B+2$ *elements of* `X` *to each of the* K *elements in its range, and*
- `c` *is defined to have any fixed value from* `Y`*, say* `0`*.*

Proof. The proof uses symbolic labelled transition systems [HL95, LR98b] and factoring out of symmetry [CE+96, ES96, ID96a, Jen96], and involves much additional development to be able to deal with arrays and with the stated definition of $W^{\mathtt{P}}_{\mathtt{X}}$.

A sketch can be found in [LR98a]; the full proof will appear in [Laz98a]. ■

It is not impossible for B to be infinite, in which case the theorem is unlikely to be useful in practice. (As we have said, the question of whether B is infinite is orthogonal to the parameterisation by `X` and `Y`.)

When `P` does not use an uninterpreted predicate, the theorem can be applied with $K = 1$.

4.1 Decision Procedures

The theorem immediately gives us a decision procedure for our restricted version of the PVP: compute B and then check whether the resulting instantiation of `P` is deterministic. As we have remarked, this procedure terminates provided B is computable in a finite time and the instantiation of `P` is finite-state.

The efficiency of the determinism checking phase can be considerably improved by automatically factoring out the symmetry associated with the type `X` and by allowing at most one input of type `Y` to have the value `1` in any execution. Moreover, by performing a suitable *symbolic check*, the computation of B can effectively be done lazily during the check itself, resulting in greater accuracy.

It is also possible to automatically check whether the assumptions of the theorem, or at least their suitable stronger versions, are satisfied.

At the time of writing, extensions to the model checker FDR2 to fully support the obtained decision procedures are being planned.

4.2 Related Work

We are not aware of any general results in the literature enabling systems that can access unboundedly many different array components (i.e. memory locations) during their execution to be verified by reduction to finite instantiations (as required by most model checkers). Such systems were typically verified by one of the following methods:

- applying existing data independence theorems together with special-case observations or arguments, as in [HM+95, ID96a, LR96, Ros98];
- symbolic execution which is not guaranteed to always terminate, or which relies on the number of array accesses being bounded, as in [VBJ97, HIB97];
- special-case abstractions, as in [Gra94].

We also cannot point to previous results facilitating reduction to finite instantiations when uninterpreted predicates on variable types are present, although it is substantially easier to extend the known methods to accomodate them than it is for unbounded arrays.

5 A Case Study

With the help of Theorem 1, we have verified that a database system which allows users to lock, read and write records at multiple security levels is secure. The system has similar functionality to one of the case studies in [Wul97].[2]

As usual in this sort of analysis, it suffices to look at only two security levels of user, on the grounds that more complex security policies can be partitioned into multiple binary analyses: see [RWW96, Ros95, Wul97, Ros98] for details. The system thus works with two user identities: a high security user `Hugh` and a low security user `Lois`.

Without going into details, the system consists of a manager process `DBM` and a disk process `Disk` which stores the current values of all records. It has two variable types, `RECORDS` and `DATA`, and it uses an array with indices from `RECORDS` and values from `DATA` to model the contents of the disk. It also uses an uninterpreted 2-valued predicate `rlevel :: RECORDS -> {0, 1}` which provides the security level associated with each record, and an uninterpreted constant `inval :: DATA` for the initial value of all the records.

That the system is secure is understood to mean that there can be no information flow from `Hugh` to `Lois` across it. As we have remarked in Section 3, this can be verified by verifying that a suitable abstraction of the system, namely what it looks like to `Lois` when `Hugh`'s actions are turned into internal nondeterminism, is deterministic [RWW96, Ros95, Wul97, Ros98].

Let us call this abstraction `P`. It turns out that `P` satisfies all the assumptions of Theorem 1 with `X`, `Y`, `r` and `c` now being called `RECORDS`, `DATA`, `rlevel` and `inval` respectively, and that

$$B = 2 \times (\texttt{limit(Hugh)} + \texttt{limit(Lois)}) + 1$$

where `limit(Hugh)` and `limit(Lois)` are limits of how many records the users can have open or locked at any one time. Thus Theorem 1 has successfully reduced the problem of verifying that the system is secure for each instantiation of `RECORDS`, `DATA`, `rlevel` and `inval` to verifying that it is secure for a single finite instantiation.

For a few small values of `limit(Hugh)` and `limit(Lois)`, we have verified on the model checker FDR2 that the reduced instantiation of `P` is indeed deterministic. It remains an open problem, outside the scope of the present paper, to by an automated verification establish determinism for any `limit(Hugh)` and `limit(Lois)` and when one or both of those limits are removed.

A complete CSP_M script with the case study can be found at:

`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security/`

6 Further Work

We should stress that the techniques we have developed are not restricted to verifying determinism. Indeed, we hope to obtain similar theorems for verifying arbitrary conditions. In CSP, that means for verifying that a concurrent system which uses unbounded arrays refines [Hoa85, Ros98] a given specification.[3]

Acknowledgements

We are very grateful to Stephen Brookes, Michael Goldsmith, Ramin Hojati, Dominic Hughes, Martin Hyland, Norris Ip, Gavin Lowe, Oege de Moor, Kedar Namjoshi, Peter O'Hearn, Luke Ong, Irfan Zakiuddin and others for helpful discussions, and to Lars Wulf for basing the functionality of our case study on a case study in his thesis.

[2] [Wul97] is concerned with abstractions and using them to reason about information flow and computer security; it does not deal with the Parameterised Verification Problem.

[3] Some such refinement checks have been included in the script for the case study obtainable from the Web.

References

[AK86] Apt, K.R. and D.C. Kozen, *Limits for Automatic Verification of Finite State Concurrent Systems*, 307–309, Information Processing Letters 22 (6), 1986.

[CE+96] Clarke, E.M., R. Enders, T. Filkorn and S. Jha, *Exploiting Symmetry in Temporal Logic Model Checking*, 77–104, in [Eme96].

[CGJ95] Clarke, E.M., O. Grumberg and S. Jha, *Verifying Parameterized Networks Using Abstraction and Regular Languages*, 395–407, Proc. of the 6$^{\text{th}}$ CONCUR, Springer LNCS 962, 1995.

[CZ+97] Corella, F., Z. Zhou, X. Song, M. Langevin and E. Cerny, *Multiway Decision Graphs for Automated Hardware Verification*, 7–46, Formal Methods in System Design 10 (1), Kluwer, 1997.

[Eme96] Emerson, E.A. (ed.), Formal Methods in System Design 9 (1–2) (Special Issue on Symmetry in Automatic Verification), Kluwer, 1996.

[EN96] Emerson, E.A. and K.S. Namjoshi, *Automatic Verification of Parameterized Synchronous Systems*, 87–99, Proc. of the 8$^{\text{th}}$ CAV, Springer LNCS 1102, 1996.

[ES96] Emerson, E.A. and A.P. Sistla, *Symmetry and Model Checking*, 105–131, in [Eme96].

[FS97] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[Gra94] Graf, S., *Verification of a Distributed Cache Memory by Using Abstractions*, 207–219, Proc. of the 6$^{\text{th}}$ CAV, Springer LNCS 818, 1994.

[GS92] German, S.M. and A.P. Sistla, *Reasoning About Systems with Many Processes*, 675–735, Journal of the ACM 39 (3), 1992.

[HB95] Hojati, R. and R.K. Brayton, *Automatic Datapath Abstraction in Hardware Systems*, 98–113, Proc. of the 7$^{\text{th}}$ CAV, Springer LNCS 939, 1995.

[HDB97] Hojati, R., D.L. Dill and R.K. Brayton, *Verifying Linear Temporal Properties of Data Insensitive Controllers Using Finite Instantiations*, Proc. of the 13$^{\text{th}}$ CHDL, 1997.

[HGD95] Hungar, H., O. Grumberg and W. Damm, *What if Model Checking Must be Truly Symbolic?*, 230–244, Proc. of the 1$^{\text{st}}$ TACAS, BRICS Notes Series NS-95-2, Department of Computer Science, University of Aarhus, 1995. (Also in Springer LNCS 1019.)

[HIB97] Hojati, R., A.J. Isles and R.K. Brayton, *Automatic State Reduction Technique for Hardware Systems Modelled Using Uninterpreted Functions and Infinite Memory*, Proc. of the 2$^{\text{nd}}$ IEEE HLDVT, 1997.

[HL95] Hennessy, M. and H. Lin, *Symbolic Bisimulations*, 353–389, Theoretical Computer Science 138 (2), 1995.

[HM+95] Hojati, R., R. Mueller-Thuns, P. Lowenstein and R.K. Brayton, *Automatic Verification of Memory Systems which Execute Their Instructions Out of Order*, Proc. of CHDL, 1995.

[Hoa85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.

[ID96a] Ip, C.N. and D.L. Dill, *Better Verification Through Symmetry*, 41–75, in [Eme96].

[ID96b] Ip, C.N. and D.L. Dill, *Verifying Systems with Replicated Components in Murφ*, 147–158, Proc. of the 8$^{\text{th}}$ CAV, Springer LNCS 1102, 1996.

[Jen96] Jensen, K., *Condensed State Spaces for Symmetrical Coloured Petri Nets*, 7–40, in [Eme96].

[JP93] Jonsson, B. and J. Parrow, *Deciding Bisimulation Equivalences for a Class of Non-finite-state Programs*, 272–302, Information and Computation 107 (2), 1993.

[KM+97] Kesten, Y., O. Maler, M. Marcus, A. Pnueli and E. Shahar, *Symbolic Model Checking with Rich Assertional Languages*, 424–435, Proc. of the 9th CAV, Springer LNCS 1254, 1997.

[Laz98a] Lazić, R.S., D.Phil. Thesis, Oxford University Computing Laboratory, to appear in 1998.

[Laz98b] Lazić, R.S., *Theorems for Model Checking Data Independent CSP*, Technical Report, Oxford University Computing Laboratory, to appear in 1998.

`http://www.comlab.ox.ac.uk/oucl/publications/tr.html`

[LR96] Lazić, R.S. and A.W. Roscoe, *Using Logical Relations for Automated Verification of Data Independent CSP*, Proc. of the Workshop on Automated Formal Methods (Oxford, U.K., June 1996), to appear in Electronic Notes in Theoretical Computer Science.

[LR98a] Lazić, R.S. and A.W. Roscoe, *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays*, Technical Report PRG-TR-2-98, Oxford University Computing Laboratory, 1998.

`http://www.comlab.ox.ac.uk/oucl/publications/tr.html`

[LR98b] Lazić, R.S. and A.W. Roscoe, *A Semantic Study of Data Independence with Applications to Model Checking*, to be submitted to Theoretical Computer Science in 1998.

[Ros95] Roscoe, A.W., *CSP and Determinism in Security Modelling*, Proc. of the IEEE Symposium on Security and Privacy, 1995.

[Ros98] Roscoe, A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

[Rus97] Rushby, J.M., *Are Formal Methods Best Used for Refutation or for Verification?*, A talk given at the 4th NASA Langley Formal Methods Workshop (LFM), 1997. (Not in the published proceedings.)

[RWW96] Roscoe, A.W., J.C.P. Woodcock and L. Wulf, *Non-interference Through Determinism*, 27–54, Journal of Computer Security 4 (1), 1996. (Revised from Proc. of ESORICS '94, Springer LNCS 875.)

[Sca98] Scattergood, J.B., *Tools for CSP and Timed CSP*, D.Phil. Thesis, Oxford University Computing Laboratory, to appear in 1998.

[VBJ97] Velev, M., R.E. Bryant and A. Jain, *Efficient Modelling of Memory Arrays in Symbolic Simulation*, 388–399, Proc. of the 9th CAV, Springer LNCS 1254, 1997.

[Wol86] Wolper, P., *Expressing Interesting Properties of Programs in Propositional Temporal Logic*, 184–193, Proc. of the 13th ACM POPL, 1986.

[Wul97] Wulf, L., *Interaction and Security in Distributed Computing*, D.Phil. Thesis, Oxford University Computing Laboratory, 1997.

[YJL96] Yen, H.-C., S.-T. Jian and T.-P. Lao, *Deciding Bisimulation and Trace Equivalences for Systems With Many Identical Processes*, 445–464, Theoretical Computer Science 170 (1–2), 1996.

Modelling and Verification of Unbounded Length Systolic Arrays in M2L(Str)

Tiziana Margaria, Michael Mendler, Claudia Gsottberger

Lehrstuhl für Programmiersysteme
Universität Passau
Innstr. 33, D-94032 Passau (Germany)

tel: +49 851 509.3096 fax: +49 851 509.3092
{tiziana,mendler,gsottber}@fmi.uni-passau.de

Abstract

Formal verification of hardware circuits requires both a reliable formal description system and an adequate verification tool. This paper proposes a new method for the automatic verification of a class of systolic systems, based on the use of monadic second order logic (over strings) as a modeling language. The method is applied to a case study from the literature, which illustrates the modelling, synthesis, and verification features on a class of iterative, parametric, linear systolic arrays. It also presents a first performance comparison between the verification tools Mona and MOSEL for this logic.

Keywords: Modelling Languages, Programming and Verification Tools, Design of Embedded Environments, Monadic Second-Order Logic, (Synchronous) Hardware Description Languages, Systolic Systems.

Corresponding Author: Tiziana Margaria (tiziana@fmi.uni-passau.de)

1 Introduction and Background

Systolic arrays [SuMe77, KuLe79] are intensively used e.g. in the implementation of *hardware accelerators*[1] and of encoders/decoders for security purposes. Systolic systems can be seen as synchronous networks of parallel processors. They have nice properties from both an engineering and a mathematical point of view, by combining multiprocessing and pipelining techniques with the more theoretical concepts of cellular automata and algorithms. Such systems exhibit a regular behavior both over *time* and over *structure*. Traditionally, this behavior has been formalized for verification purposes by means of recursion, and it has led to induction-based proofs. Unfortunately neither recursive modeling nor induction-based reasoning are familiar to hardware designers, which accounts in part for the scarce role played so far by formal methods in the common practice of circuit design. The modeling language

[1] I.e., devices capable of performing the same functions of, for example, typical Abstract Data Type objects, like e.g. the priority queue of [CaMP89].

M2L(Str) (**M**onadic **S**econd-Order **L**ogic over **Str**ings) proposed in this paper avoids recursion and thus is much closer to the modeling practice of hardware designers. As usual for hardware description languages, parameterization is used to capture the structure contained in a systolic array. In particular,

- Parameterization over *time* permits capturing the sequential behavior of a single processor in the form of difference equations.
- Parameterization over *structure* is particularly suited for VLSI implementations where the same basic cell is often instantiated many times to yield a regular structure.

Together with the fully automated tool support for M2L(Str), this yields modular definition of circuits and hierarchical verification, as required in industrial practice.

M2L(Str)is equivalent to Büchi's weak successor arithmetic WS1S [Büch60] (see [Thom90, Thom97]). Though the logic and its decision procedure are known since long they have not received much attention for practical applications, mainly because of their staggering non-elementary complexity. Yet, hardware technology has made some progress since the 60ies and relevant practical problems are usually far better behaved than the worst-case complexity would suggest.
Recently, second-order monadic theories such as M2L(Str) have been rediscovered for applications showing their practical potential as natural high-level description languages that combine the *full automation* of the model-oriented with the *expressiveness* of the logic-oriented methods [HJJK95, ABBS95]. In the software area M2L(Str) has been used for the RPC-Memory specification case study [KNS96], a variant of the problem of the dining philosophers [HJJK95], or in a controller case study for distributed systems [MaMe97] with attention to codesign aspects. In the hardware area M2L(Str) has been applied to gate level circuits [BaKl95], hardware controllers [MaMe96], and sequential circuits with parametric data-paths [Marg96].
To our knowledge only a few implementations of monadic second-order theories are available or under construction at the moment. In Århus the Mona [HJJK95] and Mona++ packages implement interpretations over strings and trees respectively, and in Kiel the AMoRE system [Matz95] offers a decision procedure for the logic over trees. At Passau we developed the MOSEL [KMMG97, KMMG97a] synthesis and verification toolset. It is available also as part of METAFrame [SMCB96], an environment for the analysis, verification and construction of complex systems, which provides e.g. the graphic facilities for automata display. Decision procedures for monadic second-order theories are also soon to be integrated into STeP [BBCC96]. A decision procedure for a restricted fragment of S1S, the first-order formulas with a single outermost second-order quantifier, has recently been presented in [SchWe97].

Though tools are still at the stage of prototypes, concrete experiences and performance results are already available for the Mona [HJJK95] and MOSEL [KMMG97] tools. In this paper we use *linear*, i.e., one-dimensional, systolic arrays as a case study to give a first comparison of the two systems. The structure of such iterative systems, called *iterative networks*, is illustrated in Fig. 1: arrays are constructed as a linear chain of interconnected instances C_1 to C_n of the same *basic cell* C. Each cell C_i communicates with the environment via *local inputs* y_i and x_i^1 to x_i^p (which are the primary inputs and outputs of the systolic array and which are externally controllable). Additionally, there are uni- or bidirectional *communication channels* between adjacent cells. The inputs coming from the extreme left and right model the communication with the host, and are called *boundary conditions*. An *iterative system* is the class of all iterative networks having the same basic cell and boundary conditions, and a different number of cells. The iterative system is also characterizable as an iterative network with unknown (parameterized) number of cells.

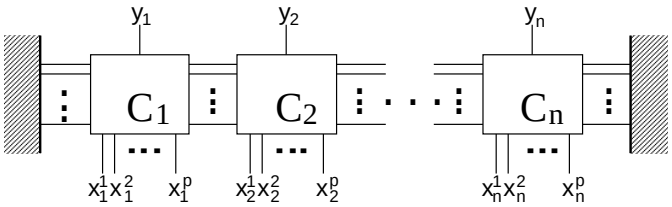

Figure 1: Structure of Iterative Linear Systolic Arrays

```
t ::=  0  |  $  |  p  |  t  +o  i | 0 + i
T ::=  all |  P  |  compl(T)  |  T1  inter  T2 |  T + i
F ::=  true  |  t1  =  t2  |  t1 < t2  |  T1  =  T2  |  t in T  |
       ~F  |  F1 & F2  |  All p: F  |  All P: F
```

Figure 2: A basic syntax for M2L(Str)

We deal with the general case, namely *sequential*[2] *iterative systems* of the unilateral, bilateral, or circular kind[3]. The problems we are going to attack involve, therefore, parameterization over time, due to the sequential behavior of the basic cells, and parameterization over structure, since we consider arrays with arbitrary number of cells.
The internal structure of the basic cells of such systems is usually specified as a finite state machine (FSM) and implemented in hardware at the gate-level. In particular, at this granularity we will regard each cell as a "slice" of the overall array.
After presenting our specification language in Section 2 we describe the two considered verification setups in Section 3. Section 4 presents fully automatic verification of pipeline properties for an example from the literature [RhSo93]. Finally, Section 5 summarizes the lessons learned so far concerning the practice of designing tools for M2L(Str) and presents first performance results.

2 The Specification Language

The M2L(Str) syntax described here is common to the Mona and the MOSEL tools (see Sect. 3). The basic operators are reported in Figure 2, where we distinguish first-order terms `t` denoting positions, string expressions `T`, and formulas `F`. To understand the modeling of iterative systolic arrays, we need to be familiar with the interpretation of the logic over *time* on the one, and over *structure* on the other hand.

M2L(Str) for waveforms The interpretation of the logic over waveforms is needed to model and reason about the sequential circuit implementing the basic cell. Here,

- First-order terms `t` describe *discrete time points* or *clock cycles*. `0` is the start cycle and `$` the final cycle of the considered time interval. The operators `+` and `+o` denote (with slight differences)

[2] *Sequential* (resp. combinational) networks have sequential (resp. combinational) circuits as basic cells. Sequential iterative networks correspond to two-dimensional combinational iterative networks.

[3] An iterative network is *unilateral* if the communication channels carry information only in one direction, otherwise it is *bilateral*. It is *circular* if its cells are connected in a ring.

addition modulo the interval length, where `i` ranges over natural numbers. Finally, `p` ranges over first-order variables.

- Second-order terms `T` denote Boolean *signals* over the considered time interval, represented as the set of cycle times in which the signal is set to 1. `all` is the constant 1 signal, `inter` is the pointwise 'and' of two signals, `compl(T)` denotes the pointwise complement of `T`, and `+` is the operator which shifts `T` right by `i` steps, thus corresponding to the i-clock delay operator on signals. Finally, `P` ranges over signals.

- Formulas `F` specify the behavior of a circuit over all observation intervals. The atomic formulas are equations `t1 = t2` and inequations `t1 < t2` of clock cycles, equations on signals `T1 = T2`, and the construct `t in T`, which is true if signal `T` has value 1 in cycle `t`. Negation `~` and conjunction `&` are as usual. Finally, we can quantify over cycles and signals, `All p: F`, `All P: F`.

M2L(Str) for bit-sliced structures Here first-order terms are taken to represent bit-slice indices, and second-order terms represent bit-vectors of generic length. The formula `t in T` then means "the bit with index `t` is set to 1 in vector `T`." Interpreted in the domain of iterative systolic systems, single bit-slices correspond to single cells, while a bit-vector ranges over the whole iterative system. In this setting,

- First-order expressions `t` describe positions of cells in an array. Since the operator `+o` which denotes position shift modulo the string length, the logic is adequate for modeling circular systems too.

- Second-order expressions `T` range over entire linear arrays.

Derived M2L(Str) operators Many of the connectives given here, like e.g. position variables and their connectives, are only included for convenience since they may be encoded within the logic using second-order variables (see e.g. [Thom90]). Similarly, dual connectives like, e.g., `false`, the empty string `empty`, bitwise `union`, implication `=>`, equivalence `<=>`, existential quantification `Ex` on strings are available, as well as a short form for Boolean (propositional) variables, represented as `@p`, over which quantification is possible too. Predicate definitions are equalities terminated by semicolons, and comments are introduced by the symbol `#`.
Semantic models are constructed by converting formulas to automata as sketched in [HJJK95] and [KMMG97]. For any formula `F` that is not a tautology, a minimal length counter-example can be extracted from the corresponding automaton. This feature is exploited for fault detection, diagnosis and testing (see [Marg96, MaMe96]).

3 The Analysis, Synthesis, and Verification Environment

We tested two quite different system setups:

- one based on Mona [HJJK95], a tool for a second-order monadic logic implemented in ML, illustrated in Figure 3, and

- one based on MOSEL [KMMG97] (see Figure 4), a new toolset for the same logic, realized within METAFRAME [SMCB96] and implemented in C/C++.

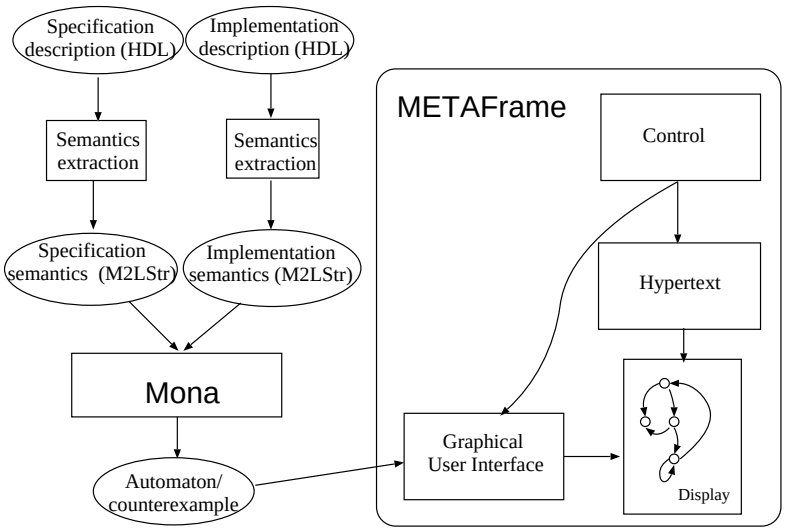

Figure 3: The Scenario with Mona

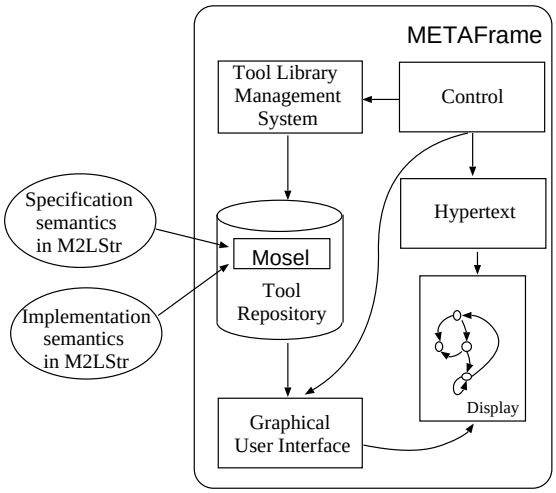

Figure 4: The Scenario with MOSEL

In both cases, circuit descriptions given in a hardware description language have to be first translated into the target logic, along the lines introduced in [Evek90, Marg93][4]. Synthesis and verification proofs can then be carried out fully automatically at the *logic* level by means of Mona or MOSEL. Results, in form of an automaton or of a minimal counter-example, can be visualized as a graph within the METAFrame environment.

Though differing for many important design aspects, as will be explained in detail in Sect. 5, in both systems predicates are defined as logic formulas and automatically transformed into minimal automata. BDD techniques are used in order to store the automata's transitions. Semantic models are automata, visualized as graphs: Figures 8 and 9 show some automata generated by both tools for the case study of Sect. 4.

[4] In these approaches the semantics of Register-Transfer and gate level descriptions was expressed in terms of first-order logic formulas.

Users can also investigate properties of the graphs by means of hypertext inspectors for nodes and edges: in addition to their label, nodes have in fact attributes like e.g. *start*, *accepting*, *non-accepting*. Some of the properties shown in the inspector windows are also indicated by coloring of the nodes/edges in the graph.
Currently, METAFrame provides graphic and hypertext facilities for the display of the results delivered by Mona, whose shallow integration level restricts its use to an input/output compatible external tool. In contrast, MOSEL is part of the Tool Repository, thus additionally the entire tool management (verification and input/output format conversions) happens within METAFrame.

4 Case Study: the Example of [RhSo93] Revisited

This section illustrates, by way of an example, the use of M2L(Str) as a powerful, elegant, and concise description language, which allows an easy description of VLSI devices with regular structure in a very compact form. Entire classes of systolic circuits are captured by a single parameterized description. The parameter can be interpreted as indicating the "*length*" of the device in terms of the number of cells. Specific instances can be modeled by simply specifying the actual value of the structural parameter.

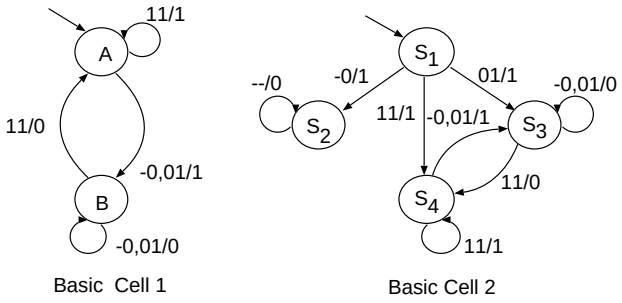

Figure 5: Automata of the Two Different Basic Cells

4.1 The Problem

The example chosen by Somenzi and Rho investigates a case where two *iterative* systems are equivalent while their *basic cells* are not. Fig. 5 shows two simple FSMs that are not equivalent. Nevertheless, we want to verify that two iterative systems with those FSMs as basic cells are equivalent. Note that this is not a case of different encodings of the same machine, since we start with two minimal deterministic FSMs which have different numbers of states. Fig. 6 shows the gate-level implementation of the basic cell reported in [RhSo93]. Here the `X` input is considered to be local, and `Y0` a communication input. There are no local outputs, and a single communication output `Y1`. For the verification of the sequential iterative systems generated by these basic cells, the boundary condition of the first cell is set to `0`. Accordingly, the following two verification problems will be addressed:

- **Problem 1:** *Verify that the sequential circuits implementing the basic cells are not equivalent.*
- **Problem 2:** *Verify that the iterative systems generated by the basic cells and under the given boundary condition are equivalent.*

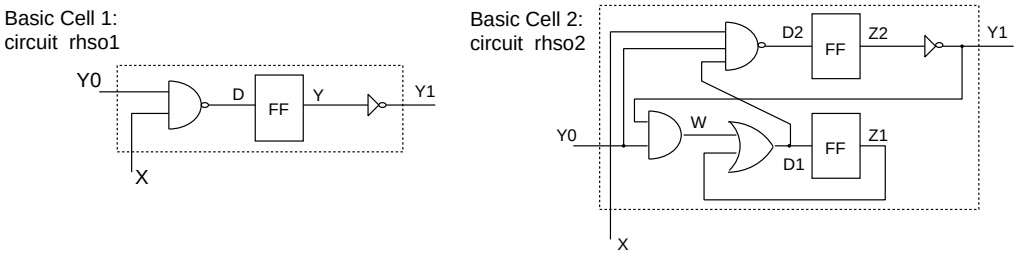

Figure 6: Original Implementation of the Basic Cells: Circuits `rhso1` and `rhso2`

4.2 The Gate-level Library

The following relations model the behavior of the elementary gates in our logic.

```
# circuit constructors as relations
  not(@a,@b) = (~(@a) <=> @b);
  and(@a,@b,@c) = ((@a & @b) <=> @c);
  or(@a,@b,@c) = ((@a | @b) <=> @c);  ...

# D-type flip-flop
   dff(D,Q) = (All t: (t < $) => (t +o 1 in Q <=> t in D) & (0 notin Q));
```

Note that we use relations instead of functions. This means that we are not primarily interested in modelling the signal flow in a circuit, distinguishing causes (variations on the inputs) and effects (the induced variations on the outputs), but rather *consistency conditions*. This more abstract view pays off when dealing with bidirectional signal flows. This way it is immediately possible to model bilateral communication in systolic systems with no additional cost.

4.3 Problem 1: Synthesis of The Basic Cells

After some reverse engineering in order to reconstruct the state assignments used in [RhSo93], we synthesize the basic cells from this state assignment by means of two-level techniques. We obtain the circuits `our-cell1` and `our-cell2` shown in Fig. 7. Since they differ from the original ones, an additional proof obligation arises:

- **Problem 1A:** *Verify that each pair of sequential circuits implementing the same basic cell is equivalent, i.e.* `rhso1` $\Longleftrightarrow$ `our-cell1` *and* `rhso2` $\Longleftrightarrow$ `our-cell2`.

The gate-level implementations of the original basic cells are given in the usual structural fashion, as netlists of gate-level components. The predicates `rhso1` and `rhso2` describe the basic cells as a collection of single gates with the appropriate connections. The body of each predicate is a conjunction of calls to predicates of the gate-level library, where the parameter-passing mechanism is used to establish the desired wiring. Internal connections (called nets) are hidden by means of existential quantification.

```
# Basic Cell 1
  rhso1(X,Y0,Y1) = (Ex D: Ex Y: All t:
                    nand(t in X,t in Y0,t in D) &
```

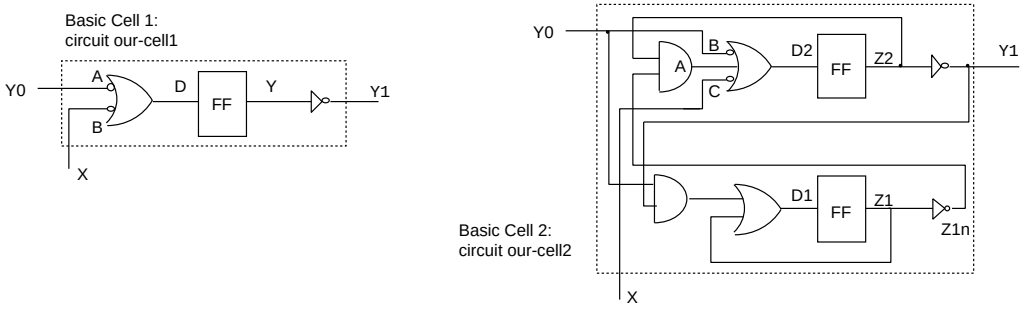

Figure 7: Re-implemented Basic Cells: Circuits `our-cell1` and `our-cell2`

```
                    dff(D,Y) &
                    not(t in Y,t in Y1));
# Basic Cell 2
  rhso2(X,Y0,Y1) = (Ex D1: Ex D2: Ex W: Ex Z1: Ex Z2: All t:
                    nand3(t in X,t in Y0,t in D1,t in D2) &
                    dff(D2,Z2) &
                    not(t in Z2,t in Y1) &
                    and(t in Y0,t in Y1,t in W) &
                    or(t in W,t in Z1,t in D1) &
                    dff(D1,Z1));
```

The circuits of Figure 7 are described similarly with netlist descriptions `our-cell1` and `our-cell2`. The combinational parts of `rhso2` and `our-cell2` will be referred to as `rhso2comb` and `our-cell2comb`, respectively. These predicates will be used to show different forms of equivalence.

4.4 Verification 1: Behaviour of the Single Cells

The Automata of the Original Cells It is possible to generate automatically the automaton corresponding to each predicate, which amounts to solving a synthesis problem. Considering the original cells, the following commands

```
rhso1(X,Y0,Y1)
rhso2(X,Y0,Y1)
```

cause the computation of the minimal automata corresponding to the predicates describing the original implementations of the basic cells.
The output offered by the tools consists of a listing of transitions presented in textual form. A visualization in MetaFrame of the same automata is shown in Fig. 8. They coincide with the automata of Fig. 5, our initial behavioral specification.

Equivalence of the two Original Basic Cells For the two original implementations of the basic cells, `rhso1` and `rhso2` to possess the same behavior over time, independently of the assignment to the free variables, the formula

```
rhso1(X,Y0,Y1) <=> rhso2(X,Y0,Y1)                    (1)
```

must be checked. This statement is refuted, as expected, since the corresponding minimal deterministic

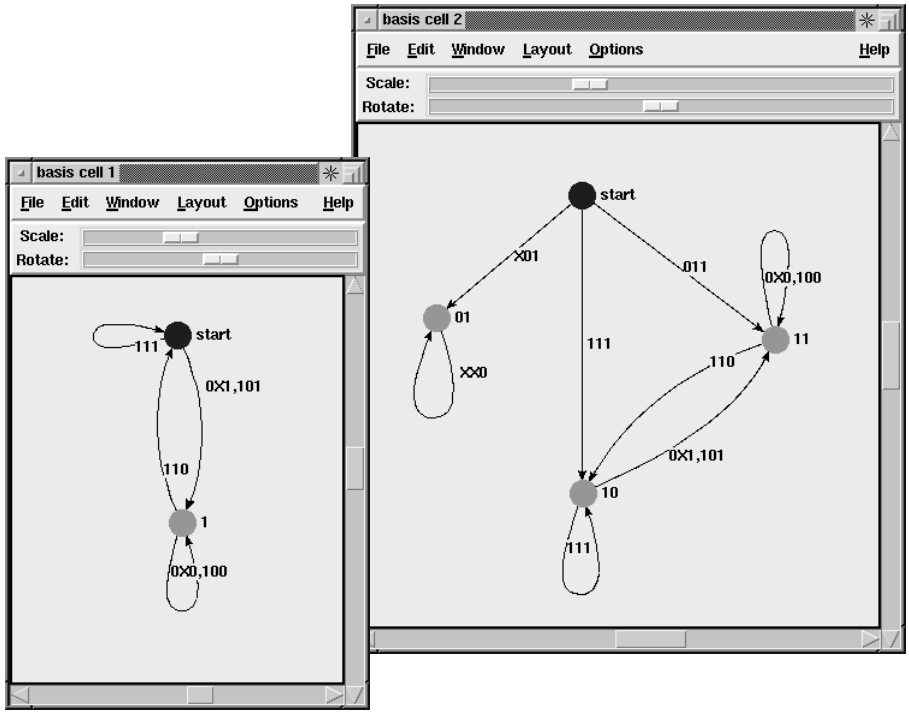

Figure 8: METAFrame's minimized automata

automata (i.e. the two automata of Figures 5, or equivalently the two of Figure 8) are different. Note that the same input `X` is fed to both circuits and that we expect to observe the same output sequences `Y0, Y1` at each pair of corresponding outputs.

4.5 Verification 1A: The Re-implemented Basic Cells

Our goal is to prove that each pair of circuits (`rhso1, our-cell1`) and (`rhso2, our-cell2`), implementing the same basic cell is equivalent. To this end, several approaches may be possible, in general.

Complete Verification. The straightforward approach is to prove the equivalence of each pair of circuits by solving a complete problem of sequential circuits verification, along the lines of the previous section. In the case of Basic Cell 2, this leads to the following formulation:

Basic Cell 2: `rhso2(X,Y0,Y1) <=> our-cell2(X,Y0,Y1)` (2)

In fact, this is a theorem. This result was expected, since both implementations have been derived from the same behavioural specification.
Note that the verification method used here is *synthesis-based*: first the automata corresponding to each of the single circuits are generated (synthesis phase), and subsequently their isomorphism is checked (verification phase).

Partial Verification Techniques. Additional information at hand can help to reduce the verification problem to its essential portion. In our case,

1. From the Karnaugh maps we see that the output functions of each pair are identical. Moreover, the output portion of each circuit pair is indeed topologically identical. The equivalence of each pair of circuits can thus be reduced to a proof of equivalence of the *state transition* portions as *sequential circuits*: if the circuits are in equivalent states, we already know that they have equivalent outputs.

2. We additionally know that the state encodings used for each pair of circuits are identical, therefore the verification of the *combinational* circuit determining the next state suffices.

This leads to the following simpler problem:

Basic Cell 2:

```
rhso2comb(X,Y1,Y2,Z1,Z2,D1,D2) <=> our-cell2comb(X,Y1,Y2,Z1,Z2,D1,D2)
```
(3)

which is a theorem too.

4.6 Problem 2: The Iterative Linear Arrays

The implementation of the arrays generated by the pairs of basic cells described is obtained by appropriately connecting neighbouring instances of the basic cells along the communication channels. We intend to characterize the behaviour of the array as experienced by a signal entering the leftmost cell at time 0, following its propagation along the array. To this aim, we define a second order variable `UY`, which describes what happens at each observation point between two adjacent cells: for both arrays, the `Y1` output of the first cell is fed into the `Y0` input of the adjacent one. The boundary condition for the first cell is enforced by stipulating that the initial value of the communication input `Y0` is low, `0 notin UY` (position `0` is not contained in the `UY` set). This connection scheme leads to the following descriptions:

```
# Systolic Array 1
  array1(X,Y) = (Ex UY: (0 notin UY) &
                        All t: ((t > 0) => (t in UY <=> t in Y)) &
                 rhso1(X,UY,Y));

# Systolic Array 2
  array2(X,Y) = (Ex UY: (0 notin UY) &
                        All t: ((t > 0) => (t in UY <=> t in Y)) &
                 rhso2(X,UY,Y));
```

This description can be interpreted as *following one wave of computation* through the array, which is the usual setting in systolic design. This means, for the initial computation we are "riding through the array" at the speed of the signals, entering the first cell at the initial time, and moving at each clock cycle to the right neighbour. The subsequent computations follow the same pattern, but the `Y0` input of the first cell is no longer `0`. Leaving the local inputs as free variables, we are able to capture all possible executions of the systolic array, under the following two operating conditions:

1. the pipeline (i.e., the set of chained storage elements) is in its initialization state when reached by the first computation front, and

2. we observe and compare outputs only when they are significant, i.e. after the latency time of the systolic arrays.

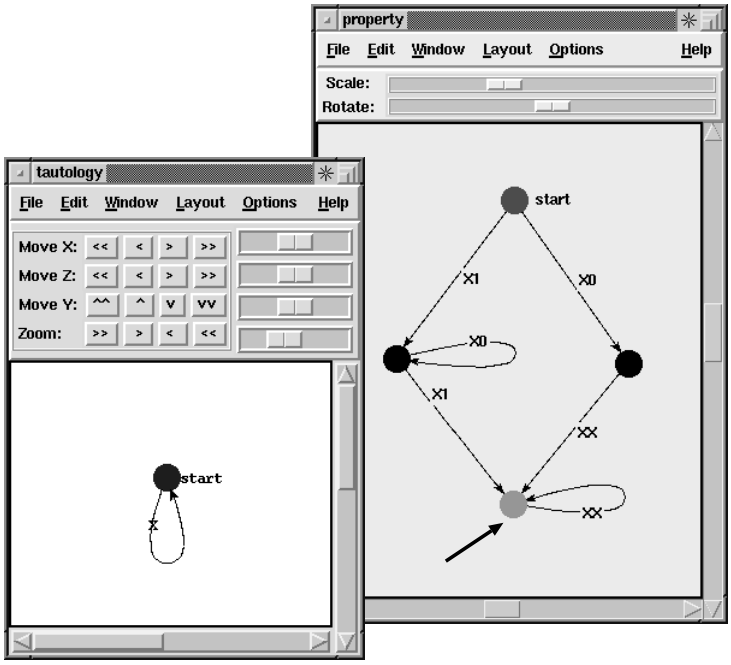

Figure 9: Verification results for the two arrays

These are the **standard conditions** for systolic systems. Note that the equivalence property considered in [RhSo93] is more restrictive than the standard conditions, since the outputs must be equivalent at any time, thus also *during* the latency time, and it disregards completely the first initialization condition. This modelling is indeed quite unusual for systolic systems.

4.7 Verification 2: Behavioural Equivalence of the Arrays

The equivalence theorem is easily stated by the following M2L(Str) formula:

```
array1(X,Y) <=> array2(X,Y)                                    (4)
```

The minimized automaton corresponding to each predicate encountered during the expansion of the formula is constructed on-the-fly, which also contains solving two problem of *automatic synthesis of iterative systems with parametric length.*
The final result is the trivial automaton reported in Fig. 9 (left). It shows that the formula is indeed a tautology, since for any input the computation stays in the only existing state, which is accepting.

4.8 Verification of Abstract Properties

In addition to the verification of alternative implementations or of a specification/ implementation relation between two circuit descriptions, we can formulate and automatically verify relevant *abstract properties* of the systems under consideration. This feature is particularly interesting when handling complex parametric systems, like the iterative sequential systems we are studying.

A relevant property of iterative systolic arrays is e.g. the satisfiability of the formulas expressing their behaviour. The property is easily formulated as follows:

```
array1(X,Y) => false                                           (5)
```

```
array2(X,Y) => false                                    (6)
```

Here we prove satisfiability of a formula by computing a counterexample for its negation, a technique largely used in theorem proving and resolution-based provers.
The result is obtained automatically by Mona in less than 1s CPU. Neither formula is true, since both lead to the minimal automaton of Fig. 9 (right): the only accepting state, marked by the arrow in the picture (which would be red on the screen), is only reachable through non-accepting states. This proves that the behaviour of the arrays is indeed satisfiable.

5 Evaluation and Performances

Modelling Hardware in this Logic. The expressive power of M2L(Str) captures only one-dimensional structures (linearly or circularly arranged). This is due to the interpretation of the logic over strings. Since strings may be taken to assume different meanings (here and in [MaMe96] sampled waveforms, in [Marg96] the bitwidth of a datapath) some degree of freedom is left to the designer. However, in general one needs to reason about behaviours of classes of circuits over time, which calls for genericity along *both* time and spatial dimensions. In some cases it is possible to some extent to 'cut' along one of the two axes: here we have renounced to model the whole array at all times, and have chosen to model one (however generic) pipeline computation. Here this choice still suffices to capture the whole behaviour, but this is not true in general.

Mona**'s Shortcomings.** In its current version Mona is still a research prototype. It suffices to demonstrate on several interesting case studies, spanning diverse application fields, that practical examples are indeed in general much better behaved than the staggering theoretical worst case complexity. However, in our experience during the last two years, in which the tool was also used actively by students in a graduate course on Formal Methods for System Design, the following weaknesses were observed:

- The intuitive definition of Mona's implemented constructs found in [HJJK95] has omissions (e.g., it misses predicate definitions), inconsistencies, and leaves unclear the correspondence between the published and the implemented versions of the logic, for instance concerning empty strings. Distinguishing between primitive and derived constructs, with explicitly documented encodings, and a proof of the correctness of the semantics, would have avoided those problems.
- The rigid *user interface* of the tool, which is in pure textual form. Mona accepts only M2L(Str) formulas and delivers automata and counter examples only as list of transitions. Not even the output format for automata descriptions can be read again by the tool.
- The shallow *integratability* of the tool in larger environments. Embedding of Mona into MetaFrame is limited by the rigid interface of the former, which forces a one-directional cooperation: since Mona has to run inside the ML interpreter, it was not possible to launch it from MetaFrame. Thus we could not use Mona as originally planned.

Design Principles for Mosel**.** The following system requirements and main design principles to Mosel arose exactly from these points, which, to our knowledge, are not addressed by any other related project.

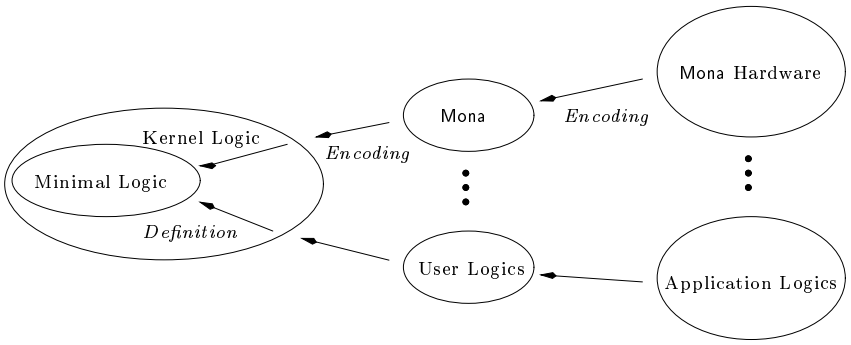

Figure 10: Layered Logics in MOSEL

1. **Definition of a formal semantics for a minimal subset of the logic** in terms of finite-state automata.

2. **Layered approach to the logic.** We introduce a hierarchy of logic layers, with increasingly powerful constructs, related by either direct embedding or more elaborate encodings as shown in Fig. 10.

   - The *minimal logic* contains a minimal set of primitives, for which the semantics is formally defined in terms of corresponding automata. This set constitutes the reference language for proofs involving semantics.
   - The *kernel logic* extends the minimal logic by additional (derived) constructs and coincides with the set of constructs actually implemented as primitives in the semantic decision procedure. The design of this extension is guided by considerations of efficiency of the computations required in the decision procedure.
   - A set of generic *user logics* correspond to an application-independent layer. They extend the kernel logic by derived operators which do not have a direct implementation, but are comfortable for generic applications. User logics may be rather different from the kernel logic and need not be a simple syntactic extension. The syntax of Mona Version 0.2 used in this paper is one of MOSEL's user logics.
   - A number of *application-specific logics*, each containing additional admissible predicates and constructs tailored to specific application domains. This paper has shown an example from the domain of hardware verification, but other application domains are possible.

   The semantic coherence of richer logics with the minimal logic is ensured by implementing outer layers of the logic through successive encodings and definitional extensions to the unique minimal logic, and by making these explicit. The coherence of the kernel wrt. the minimal logic has also been proved to some extent automatically in MOSEL, as reported in [KMMG97].

3. **Modular design**. While Mona is a single large component, MOSEL is a collection of modules which can be combined or exchanged at need. Following the concept of a repository-based library of components, MOSEL supports flexible adaptation and extension to new input or output formalisms, as well as the interchange of some of its internal components (e.g., users may replace the BDD package used in the decision procedure, or the automata minimization and

	formula	**Mona** (s)	**Mosel** (s)	**speedup**
1	rhso1(X,Y0,Y1) ⇔ rhso2(X,Y0,Y1)	8	5.05	37%
2	rhso1(X,Y0,Y1) ⇔ our-cell1(X,Y0,Y1)	4	2.02	49%
3	rhso2comb(X,Y1,Y2,Z1,Z2,D1,D2) ⇔ our-cell2comb(X,Y1,Y2,Z1,Z2,D1,D2)	44	32.79	25%
4	array1(X,Y) ⇔ array2(X,Y)	8	5.15	35%
5	array1(X,Y) ⇒ false	1	0.70	30%
6	array2(X,Y) ⇒ false	8	4.60	42%

Table 1: Performance comparison between the two M2L(Str) tools on the case study.

determinization algorithms). The aim is that the best-fitting incarnation of the tool may be put together at need, on an application-driven basis, from the collection of existing components.

4. **Integrability in a heterogeneous analysis and verification environment** like MetaFrame.

The design and the concrete architecture of Mosel have been described in [KMMG97], where we have explained in detail the realization of these system requirements, starting with the introduction of the logic layers and their semantics, followed by a description of the implementation principles and finally by the integration within MetaFrame.
Interesting is the fact that, once compared on the same level of granularity for the logic (the kernel logic), Mosel also performs 25% to 49% better than Mona, as shown in Table 1. On the other hand, we observed that Mosel is slower at the level of Mona syntax. This suggests that in the further development of Mosel the focus can shift from the basic decision procedures to improving the compilation algorithms. We are currently extending our measurements to the whole library of hardware circuits already verified by means of Mona in order to characterize better the performance profiles of both tools.

References

[ABBS95] A. Aziz, F. Balarin, R. Brayton, A. Sangiovanni-Vincentelli: "*Sequential synthesis using S1S*," Proc. ICCAD'95, pp.612–617.

[BaKl95] D. Basin, N. Klarlund: "*Hardware verification using monadic second-order logic*," Proc. CAV '95, Liège (B), July 1995, LNCS N. 939, Springer Verlag, pp. 31-41.

[BBCC96] N. Bjørner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, T. Uribe: "*STeP: Deductive-algorithmic verification of reactive and real-time systems*," Proc. CAV'96, New Brunswick, NJ (USA), Aug. 1996, LNCS 1102, Springer Verlag, pp. 415-418.

[Büch60] J.R. Büchi: "*Weak second-order arithmetic and finite automata*," Z. Math. Logik Grundl. Math., Vol. 6, 1960, pp. 66-92.

[CaMP89] P. Camurati, T. Margaria, P. Prinetto "*Systolic array description in F^2*," Microprocessing and Microprogramming, The Euromicro Journal, Vol. 27, n. 1-5, Sept. 1989, pp. 171-178.

[Evek90] H. Eveking: "*Axiomatizing hardware description languages*," Int. Journal of VLSI Design, 2(3), pp. 263-280, 1990.

[HJJK95] J. Henriksen, J. Jensen, M. Jørgensen N. Klarlund, R. Paige, T. Rauhe, A. Sandholm: "*Mona: Monadic second-order logic in practice*," Proc. TACAS'95, Aarhus (DK), May 1995, LNCS 1019, Springer Verlag, pp. 89-110.

[KMMG97] P. Kelb, T. Margaria, M. Mendler, C. Gsottberger: "MOSEL*: A flexible toolset for monadic second-order logic*," In E. Brinksma, ed., Proc. TACAS'97, Springer LNCS 1217, 1997, pp.183–202. See also `http://brahms.fmi.uni-passau.de/bs/projects/mosel/index.html`.

[KMMG97a] P. Kelb, T. Margaria, M. Mendler, C. Gsottberger: "MOSEL*: A sound and efficient tool for M2L(Str)*," Computer-Aided Verification (CAV'97), tool presentation.

[KNS96] N. Klarlund, M. Nielsen, K. Sunesen: "*A case study in verification based on trace abstraction*," In M. Broy and S. Merz and K. Spies, eds., "Formal Systems Specification, The RPC-Memory Specification Case Study", LNCS 1069, Springer Verlag, pp. 341–373.

[KuLe79] H. T. Kung, C. E. Leiserson: "*Systolic arrays (for VLSI)*," Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart eds., Society for Industrial and Applied mathematics, 1979, pp. 256-282.

[MaMe96] T. Margaria, M. Mendler: "*Automatic treatment of sequential circuits in second-order monadic logic*," 4th GI/ITG/GME Worksh. on Methoden des Entwurfs und der Verifikation digitaler Systeme, Kreischa (D), March 1996, pp. 21-30, Shaker Verlag.

[MaMe97] T. Margaria, M. Mendler: "*Model-based automatic synthesis and analysis in second-order monadic logic*," In R. Cleaveland and D. Jackson, eds., Proc. First ACM SIGPLAN Workshop on Automated Analysis of Software, Paris, January, 1997, pp.99–112. Invited for Kluwer International Journal on Automated Software Engineering.

[Marg93] T. Margaria: "*Verifica formale della correttezza del progetto di sistemi digitali*," Dissertazione di Dottorato di Ricerca in Ingegneria Informatica e dei Sistemi (in Italian), Politecnico di Torino, Turin (I), Feb. 1993.

[Marg96] T. Margaria: "*Fully automatic verification and error detection for parameterized iterative sequential circuits*," Proc. TACAS'96, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau, March 1996, LNCS 1055, Springer Verlag, pp. 258-277.

[Matz95] O. Matz, A. Miller, A. Potthoff, W. Thomas, E. Valkema: "*Report on the program AMoRE*", Techn. Rep. Nr. 9507, Inst. für Informatik und Praktische Mathematik, Universität Kiel (D), 1995.

[RhSo93] J.K Rho, F. Somenzi: "*Automatic generation of network invariants for the verification of iterative sequential systems*," Proc. CAV'93, Elounda (GR), June 1993, LNCS N. 697, pp.123-137.

[SchWe97] K. Schneider, H. Weindel: "*An efficient decision procedure for S1S*," In R. Hagelauer, M. Pfaff, eds., Proc. Workshop *Methoden des Entwurfs und der Verifikation digitaler Systeme*, Universitätsverlag Rudolf Trauner, Linz, 1997, pp.129–138.

[SMCB96] B. Steffen, T. Margaria, A.Claßen, V.Braun: "*The* METAFRAME*'95 environment*" (Experience Report for the Industry Day), Proc. CAV'96, Int. Conf. on Computer-Aided Design, Juli-Aug. 1996, New Brunswick, NJ, USA, LNCS 1102, pp. 450-453, Springer Verlag.

[SuMe77] I. E. Sutherland, C. A. Mead: "*Microelectronics and computer science*," Scientific American, Vol. 237, N. 3, September 1977, pp.210-228.

[Thom90] W. Thomas: "*Automata on infinite objects*," In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, pp. 133–191. MIT Press/Elsevier, 1990.

[Thom97] W. Thomas: "*Languages, automata, and objects*," to appear in the Handbook of Formal Language Theory (G. Rozenberg, A. Salomaa, Eds.), Vol. III, Springer Verlag.

The Regular Viewpoint on PA-Processes

D. Lugiez[1] and Ph. Schnoebelen[2]

[1] Lab. d'Informatique de Marseille, Univ. Aix-Marseille & CNRS URA 1787,
39, r. Joliot-Curie, 13453 Marseille Cedex 13 France
email: lugiez@lim.univ-mrs.fr
[2] Lab. Spécification & Vérification, ENS de Cachan & CNRS URA 2236,
61, av. Pdt. Wilson, 94235 Cachan Cedex France
email: phs@lsv.ens-cachan.fr

Abstract. PA is the process algebra allowing non-determinism, sequential and parallel compositions, and recursion. We suggest a view of PA-processes as *tree languages*.
Our main result is that the set of (iterated) predecessors of a regular set of PA-processes is a regular tree language, and similarly for (iterated) successors. Furthermore, the corresponding tree-automata can be built effectively in polynomial-time. This has many immediate applications to verification problems for PA-processes, among which a simple and general model-checking algorithm.

Introduction

Verification of Infinite State Processes is a very active field of research today in the concurrency-theory community. Of course, there has always been an active Petri-nets community, but researchers involved in process algebra and model-checking really became interested into infinite state processes after the proof that bisimulation was decidable for normed BPA-processes [BBK87]. This prompted several researchers to investigate decidability issues for BPP and BPA (with or without the normedness condition) (see [CHM94,Mol96,BE97] for a partial survey).

From BPA and BPP to PA: BPA is the "non-determinism + sequential composition + recursion" fragment of process algebra. BPP is the "non-determinism + parallel composition + recursion" fragment. PA (from [BEH95]) combines both and is much less tractable. A few years ago, while more and more decidability results for BPP and BPA were presented, PA was still beyond the reach of the current techniques. Then R. Mayr showed the decidability of reachability for PA processes [May97c], and extended this into decidability of model-checking for PA w.r.t. the EF fragment of CTL [May97b]. This was an important breakthrough, allowing Mayr to successfully attack more powerful process algebras [May97a] while other decidability results for PA were presented by him and other researchers (e.g. [Kuč96,Kuč97,JKM98,HJ98]).

A field asking for new insights: The decidability proofs from [May97b] (and the following papers) are certainly not trivial. The constructions are quite complex and hard to check. It is not easy to see in which directions the results and/or the proofs could be adapted or generalized without too much trouble. Probably, this complexity cannot be avoided with the techniques currently available in the field. We believe we are at a point where it is more important to look for new insights, concepts and techniques that will simplify the field, rather than trying to further extend already existing results.

Our contribution: In this paper, we show how **tree-automata techniques** greatly help dealing with PA. Our main results are two **Regularity Theorems**, stating that $Post^*(L)$ and $Pre^*(L)$, the set of configurations reachable from (resp. allowing to reach) a configuration in L, is a regular tree language when L is, and giving simple polynomial-time constructions for the associated automata. Many important consequences follow directly, including a simple algorithm for model-checking PA-processes.

Why does it work ? The regularity of $Post^*(L)$ and $Pre^*(L)$ could only be obtained after we had the combination of two main insights:

1. the tree-automata techniques that have been proved very powerful in several fields (see [CKSV97]) are useful for the process-algebraic community as well. After all, PA is just a simple term-rewrite system with a special context-sensitive rewriting strategy, not unlike head-rewriting, in presence of the sequential composition operator.
2. the syntactic congruences used to simplify notations in simple process algebras help one get closer to the intended semantics of processes, but they break the regularity of the behavior. The decidability results are much simpler when one only introduces syntactic congruences at a later stage. (Besides, this is a more general approach.)

Plan of the paper: We start with our definition of the PA algebra (§ 1). Then we recall what are tree automata and how sets of PA processes can be seen as tree languages (§ 2). This allows proving that $Post^*(L)$ and $Pre^*(L)$ are regular when L is a regular set of PA terms (§ 3). We then extend these results by taking labels of transitions into account (§ 4) and showing how transitions "modulo structural congruence" are handled (§ 5). Finally we consider the important applications in model-checking (§ 6). Several proofs are omited for lack of space. They can be found in the longer version of this paper at `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV`.

Related work: The set of all reachable configurations of a pushdown automaton form a regular (word) language. This was proven in [Büc64] and extended in [Cau92]. Applications to the model-checking of pushdown automata have been proposed in [FWW97,BEM97].

$\stackrel{*}{\rightarrow}$ over PA terms is similar to the transitive closure of relations defined by ground rewrite systems. Because the sequential composition operator in PA

implies a certain form of prefix rewriting, the *ground tree transducers* of Dauchet and Tison [DT90] cannot recognize $\xrightarrow{*}$. It turns out that $\xrightarrow{*}$ can be seen as a *rational tree relation* as defined by Raoult [Rao97].

Regarding the applications we develop for our regularity theorems, most have been suggested by Mayr's work on PA [May97c,May97b] and/or our earlier work on RPPS [KS97a,KS97b].

1 The PA process algebra

1.1 Syntax

$Act = \{a, b, c, \ldots\}$ is a set of *action names*.
$Var = \{X, Y, Z, \ldots\}$ is a set of *process variables*.
$E_{\mathrm{PA}} = \{t, u, \ldots\}$ is the set of PA-terms, given by the following abstract syntax

$$t, u ::= 0 \mid X \mid t.u \mid t \parallel u$$

where X is any process variable from Var. Given $t \in E_{\mathrm{PA}}$, we write $Var(t)$ the set of process variables occurring in t and $Subterms(t)$ the set of all subterms of t (t included).

A guarded PA *declaration* is a finite set $\Delta = \{X_i \xrightarrow{a_i} t_i \mid i = 1, \ldots, n\}$ of *process rewrite rules*. Note that the X_i's need not be distinct.

We write $Subterms(\Delta)$ for the union of all $Subterms(t)$ for t a right- or a left-hand side of a rule in Δ, and let $Var(\Delta)$ denotes $Var \cap Subterms(\Delta)$, the set of process variables occurring in Δ. $\Delta_a(X)$ denotes $\{t \mid \text{there is a rule } ``X \xrightarrow{a} t" \text{ in } \Delta\}$ and $\Delta(X)$ is $\bigcup_{a \in Act} \Delta_a(X)$. $Var_\varnothing \stackrel{\text{def}}{=} \{X \in Var \mid \Delta(X) = \varnothing\}$ is the set of variables for which Δ provides no rewrite.

In the following, we assume a fixed Var and Δ.

1.2 Semantics

A PA declaration Δ defines a labeled transition relation $\rightarrow_\Delta \subseteq E_{\mathrm{PA}} \times Act \times E_{\mathrm{PA}}$. We always omit the Δ subscript when no confusion is possible, and use the standard notations and abbreviations: $t \xrightarrow{w} t'$ with $w \in Act^*$, $t \xrightarrow{k} t'$ with $k \in \mathbb{N}$, $t \xrightarrow{*} t'$, $t \rightarrow, \ldots \ \rightarrow_\Delta$ is inductively defined via the following SOS rules:

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1 \parallel t_2 \xrightarrow{a} t_1' \parallel t_2} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2} \qquad \frac{}{X \xrightarrow{a} t}(X \xrightarrow{a} t) \in \Delta$$

$$\frac{t_2 \xrightarrow{a} t_2'}{t_1 \parallel t_2 \xrightarrow{a} t_1 \parallel t_2'} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1.t_2 \xrightarrow{a} t_1.t_2'} IsNil(t_1)$$

The second SOS rule for sequential composition is peculiar: it uses a syntactic predicate, "$IsNil(t_1)$", as a side condition checking that t_1 cannot evolve anymore, i.e. that t_1 is terminated. Indeed, our intention is that the t_2 part in $t_1.t_2$ only evolves once t_1 is terminated.

The $IsNil(\ldots)$ predicate is inductively defined by

$$IsNil(t_1 \parallel t_2) \stackrel{\text{def}}{=} IsNil(t_1) \wedge IsNil(t_2), \qquad IsNil(0) \stackrel{\text{def}}{=} true,$$

$$IsNil(t_1.t_2) \stackrel{\text{def}}{=} IsNil(t_1) \wedge IsNil(t_2), \quad IsNil(X) \stackrel{\text{def}}{=} \begin{cases} true \text{ if } \Delta(X) = \varnothing, \\ false \text{ otherwise.} \end{cases}$$

It is indeed a syntactic test for termination, and we have

Lemma 1. *The following three properties are equivalent:*
1. $IsNil(t) = true$,
2. $t \not\rightarrow$ *(i.e.* t *is terminated),*
3. $Var(t) \subseteq Var_{\varnothing}$.

1.3 Structural equivalence of PA terms

Several works on PA and related algebras only consider processes up-to some structural congruence. PA itself usually assumes an equivalence $\equiv$ defined by the following equations:

$$\begin{array}{lcllll} (C_{\parallel}) & t \parallel t' \equiv t' \parallel t & & & & \\ (A_{\parallel}) & (t \parallel t') \parallel t'' \equiv t \parallel (t' \parallel t'') & (N_1) & t.0 \equiv t & (N_3) & t \parallel 0 \equiv t \\ (A_{.}) & (t.t').t'' \equiv t.(t'.t'') & (N_2) & 0.t \equiv t & (N_4) & 0 \parallel t \equiv t \end{array}$$

$\equiv$ respects the behaviour of process terms. However, *we do not want to identify PA terms related by* $\equiv$ *!*

Our approach clearly separates the behavior of E_{PA} (the $\rightarrow$ relation) and structural equivalence between terms (the $\equiv$ relation). We get simple proofs of results which are hard to get in the other approach because the transition relation and the equivalence relation interact at each step.

In the following, we study first the $\rightarrow$ relation. Later (§ 5) we combine $\rightarrow$ and structural equivalence and show how it is possible to reason about "PA-terms modulo $\equiv$". In effect, this shows that our approach is also more general since we can define the "modulo $\equiv$" approach in our framework.

2 Tree languages and PA

We shall use tree automata to recognize sets of terms from E_{PA}.

2.1 Regular tree languages and tree automata

We recall some basic facts on tree automata and regular tree languages. For more details, the reader is referred to any classical source (e.g. [CDG+97,GS97]).

A *ranked alphabet* is a finite set of symbols $\mathcal{F}$ together with an arity function $\eta : \mathcal{F} \rightarrow \mathbb{N}$. This partitions $\mathcal{F}$ according to arities: $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \cdots$. We

write $\mathcal{T}(\mathcal{F})$ the set of terms over $\mathcal{F}$ and call them *finite trees* or just *trees*. A *tree language* over $\mathcal{F}$ is any subset of $\mathcal{T}(\mathcal{F})$.

A (finite, bottom-up) *tree automaton* (a "TA") is a tuple $\mathcal{A} = \langle \mathcal{F}, Q, F, R \rangle$ where $\mathcal{F}$ is a ranked alphabet, $Q = \{q, q', \ldots\}$ is a finite set of *states*, $F \subseteq Q$ is the subset of *final states*, and R is a finite set of *transition rules* of the form $f(q_1, \ldots, q_n) \longmapsto q$ where $n \geq 0$ is the arity $\eta(f)$ of symbol $f \in \mathcal{F}$. *TA's with ε-rules* also allow some transition rules of the form $q \longmapsto q'$.

The transition rules define a rewrite relation on terms built on $\mathcal{F} \cup Q$ (seeing states from Q as nullary symbols). This works bottom-up. We write $t \stackrel{\mathcal{A}}{\longmapsto} q$ when $t \in \mathcal{T}(\mathcal{F})$ can be rewritten (using any number of steps) to $q \in Q$ and say t is accepted by $\mathcal{A}$ if it can be rewritten into a final state of $\mathcal{A}$. We write $L(\mathcal{A})$ for the set of all terms accepted by $\mathcal{A}$. Any tree language which coincide with $L(\mathcal{A})$ for some $\mathcal{A}$ is a *regular tree language*. Regular tree languages are closed under complementation, union, etc.

An example: Let $\mathcal{F}$ be given by $\mathcal{F}_0 = \{a, b\}$, $\mathcal{F}_1 = \{g\}$ and $\mathcal{F}_2 = \{f\}$. There is a TA, $\mathcal{A}_{\text{even}\,g}$, accepting the set of all $t \in \mathcal{T}(\mathcal{F})$ where g occurs an even number of times in t. $\mathcal{A}_{\text{even}\,g}$ is given by $Q \stackrel{\text{def}}{=} \{q_0, q_1\}$, $R \stackrel{\text{def}}{=} \{a \longmapsto q_0, b \longmapsto q_0, g(q_0) \longmapsto q_1, g(q_1) \longmapsto q_0, f(q_0, q_0) \longmapsto q_0, f(q_0, q_1) \longmapsto q_1, f(q_1, q_0) \longmapsto q_1, f(q_1, q_1) \longmapsto q_0\}$ and $F \stackrel{\text{def}}{=} \{q_0\}$. Let t be $g(f(g(a), b))$. $\mathcal{A}_{\text{even}\,g}$ rewrites t (deterministically) as follows:

$$g(f(g(a), b)) \longmapsto g(f(g(q_0), q_0)) \longmapsto g(f(q_1, q_0)) \longmapsto g(q_1) \longmapsto q_0.$$

Hence $t \longmapsto q_0 \in F$ so that $t \in L(\mathcal{A}_{\text{even}\,g})$.

The *size* of a TA $\mathcal{A}$, denoted by $|\mathcal{A}|$, is the number of states of $\mathcal{A}$ augmented by the size of the rules of $\mathcal{A}$ where a rule $f(q_1, \ldots, q_n) \longmapsto q$ has size $n + 2$. Notice that, for a fixed $\mathcal{F}$ where the largest arity is m, $|\mathcal{A}|$ is in $O(|Q|^{m+1})$.

A TA is *deterministic* if all transition rules have distinct left-hand sides (and there are no ε-rule). Our earlier $\mathcal{A}_{\text{even}\,g}$ example was deterministic. Given a non-deterministic TA, the classical subset construction yields a deterministic TA accepting the same language (this construction involves a potential exponential blow-up in size).

Telling whether $L(\mathcal{A})$ is empty for some TA $\mathcal{A}$ can be done in time $O(|\mathcal{A}|)$. Telling whether a given tree t is accepted by a given $\mathcal{A}$ can be done in time polynomial in $|\mathcal{A}| + |t|$.

A TA is *completely specified* (also *complete*) if for each $f \in \mathcal{F}_n$ and $q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \to q$. By adding a sink state and the obvious rules, any $\mathcal{A}$ can be extended into a complete TA accepting the same language.

2.2 Some regular subsets of E_{PA}

E_{PA}, the set of PA-terms, can be seen as a set of trees, i.e. as $\mathcal{T}(\mathcal{F})$ for $\mathcal{F}$ given by $\mathcal{F}_0 = \{0, X, Y, \ldots\}$ $(= \{0\} \cup \mathit{Var})$ and $\mathcal{F}_2 = \{., \|\}$.

We begin with one of the simplest languages in E_{PA}:

Proposition 2. *For any t, the singleton tree language $\{t\}$ is regular, and a TA for $\{t\}$ needs only have $|t|$ states.*

The set of terminated processes is also a tree language. Write $L^{\varnothing}$ for $\{t \in E_{\mathrm{PA}} \mid \mathit{IsNil}(t)\}$. An immediate consequence of Lemma 1 is

Proposition 3. *$L^{\varnothing}$ is a regular tree language, and a TA for $L^{\varnothing}$ needs only have one state.*

3 Regularity of $Post^*(L)$ and $Pre^*(L)$ for a regular language L

Given a set $L \subseteq E_{\mathrm{PA}}$ of PA-terms, we let $Pre(L) \stackrel{\text{def}}{=} \{t \mid \exists t' \in L, t \to t'\}$ and $Post(L) \stackrel{\text{def}}{=} \{t \mid \exists t' \in L, t' \to t\}$ denotes the set of (immediate) *predecessors* (resp. *successors*) of terms in L. $Pre^+(L) \stackrel{\text{def}}{=} Pre(L) \cup Pre(Pre(L)) \cup \cdots$ and $Post^+(L) \stackrel{\text{def}}{=} Post(L) \cup Post(Post(L)) \cup \cdots$ contain the *iterated* predecessors (resp. successors). Similarly, $Pre^*(L)$ denotes $L \cup Pre^+(L)$ and $Post^*(L)$ is $L \cup Post^+(L)$, also called the *reachability set*.

In this section we prove the regularity of $Pre^*(L)$ and $Post^*(L)$ for a regular language L. $Pre^*(L)$ and $Post^*(L)$ do not take into account the labels accompanying PA transitions, but these will be considered in section 4.

For notational simplicity, given two states q, q' of a TA $\mathcal{A}$, we denote by $\delta_{\parallel}(q, q')$ (resp. $\delta_{\cdot}(q, q')$ any state q'' such that $q \parallel q' \stackrel{\mathcal{A}}{\longmapsto} q''$ (resp. $q.q' \stackrel{\mathcal{A}}{\longmapsto} q''$), possibly using ε-rules.

3.1 Regularity of $Post^*(L)$

First, we give some intuition which helps understanding the construction of a TA $\mathcal{A}_{Post^*}$ accepting $Post^*(L)$.

Let us assume Δ contains $X \to r_1$ and $Y \to r_2$, and that r_1 is terminated. Starting from $t_1 = X.Y$, there exists the transition sequence $t_1 \to t_2 \to t_3$ illustrated in figure 1.

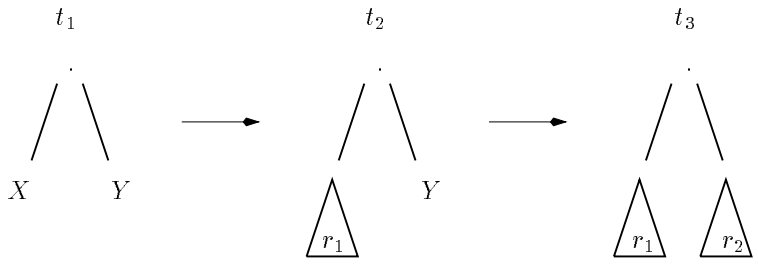

Fig. 1. An example sequence: $X.Y \to r_1.Y \to r_1.r_2$

We want to build $\mathcal{A}_{Post^*}$, a TA that reads t_3 (i.e. $r_1.r_2$) bottom-up and sees that it belongs to $Post^*(L)$. For this, the TA has to recognize that t_3 comes from t_1 (i.e. $X.Y$) and check that t_1 is in L.

1. $\mathcal{A}_{Post^*}$ must recognize that r_1 (resp. r_2) is the right-hand side of a rule $X \to r_1$ (resp. $Y \to r_2$). Therefore we need an automaton $\mathcal{A}_\Delta$ which recognizes such right-hand sides.
2. The automaton $\mathcal{A}_{Post^*}$ works on t_3 but must check that t_1 is in L. Therefore we need an automaton $\mathcal{A}_L$ accepting L. $\mathcal{A}_{Post^*}$ mimicks $\mathcal{A}_L$ but it has additional rules simulating rewrite steps: once r_1 has been recognized (by the $\mathcal{A}_\Delta$ part), the computation may continue as if X were in place of r_1. The same holds for r_2 and Y.
3. The transition between t_2 and t_3 is allowed only if r_1 is terminated. Therefore we need an automaton $\mathcal{A}_\varnothing$ to check whether a term is terminated.
4. A non-terminated term is allowed to the left of a "." when no transition has been performed to the right. Therefore we use a boolean value to indicate whether rewrite steps have been done or not.

These remarks lead to the following construction.

Ingredients for $\mathcal{A}_{Post^*}$: Assume $\mathcal{A}_L$ is an automaton recognizing $L \subseteq E_{\mathrm{PA}}$. $\mathcal{A}_{Post^*}$ is a new automaton combining several ingredients:

- $\mathcal{A}_\varnothing$ is a *completely specified* automaton accepting terminated processes (see Proposition 3).
- $\mathcal{A}_L$ is a *completely specified* automaton accepting L.
- $\mathcal{A}_\Delta$ is a completely specified automaton recognizing the subterms of Δ. It has all states q_s for $s \in \mathit{Subterms}(\Delta)$. We ensure "$t \xmapsto{\mathcal{A}_\Delta} q_s$ iff $s = t$" by taking as transition rules $0 \longmapsto q_0$ if $0 \in \mathit{Subterms}(\Delta)$, $X \longmapsto q_X$ if $X \in \mathit{Subterms}(\Delta)$, $q_s \parallel q_{s'} \longmapsto q_{s \parallel s'}$ (resp. $q_s.q_{s'} \longmapsto q_{s.s'}$) if $s \parallel s'$ (resp. $s.s'$) belongs to $\mathit{Subterms}(\Delta)$. In addition, the automaton has a sink state $q_\perp$ and the obvious transitions so that it is a completely specified automaton.
- The boolean b records whether rewrite steps have occurred.

States of $\mathcal{A}_{Post^*}$: The states of $\mathcal{A}_{Post^*}$ are 4-uples $(q_\varnothing \in Q_{\mathcal{A}_\varnothing}, q_L \in Q_{\mathcal{A}_L}, q_\Delta \in Q_{\mathcal{A}_\Delta}, b \in \{\mathit{true}, \mathit{false}\})$ where $Q_{\dots}$ denotes the set of states of the relevant automaton.

Transition rules of $\mathcal{A}_{Post^*}$: The transition rules are:

type 0: all rules of the form $0 \longmapsto (q_\varnothing, q_L, q_\Delta, \mathit{false})$ s.t. $0 \xmapsto{\mathcal{A}_\varnothing} q_\varnothing$, $0 \xmapsto{\mathcal{A}_L} q_L$ and $0 \xmapsto{\mathcal{A}_\Delta} q_\Delta$.

type 1: all rules of the form $X \longmapsto (q_\varnothing, q_L, q_\Delta, \mathit{false})$ s.t. $X \xmapsto{\mathcal{A}_\varnothing} q_\varnothing$, $X \xmapsto{\mathcal{A}_L} q_L$, and $X \xmapsto{\mathcal{A}_\Delta} q_\Delta$.

type 2: all ε-rules of the form $(q_\varnothing, q'_L, q_s, b') \longmapsto (q_\varnothing, q_L, q_X, \mathit{true})$ s.t. $X \to s$ is a rule in Δ and $X \xmapsto{\mathcal{A}_L} q_L$.

type 3: all rules of the form
$(q_\varnothing, q_L, q_\Delta, b) \parallel (q'_\varnothing, q'_L, q'_\Delta, b') \longmapsto (\delta_\parallel(q_\varnothing, q'_\varnothing), \delta_\parallel(q_L, q'_L), \delta_\parallel(q_\Delta, q'_\Delta), b \vee b')$
type 4a: all rules of the form
$(q_\varnothing, q_L, q_\Delta, b).(q'_\varnothing, q'_L, q'_\Delta, false) \longmapsto (\delta_.(q_\varnothing, q'_\varnothing), \delta_.(q_L, q'_L), \delta_.(q_\Delta, q'_\Delta), b)$.
type 4b: all rules of the form
$(q_\varnothing, q_L, q_\Delta, b).(q'_\varnothing, q'_L, q'_\Delta, b') \longmapsto (\delta_.(q_\varnothing, q'_\varnothing), \delta_.(q_L, q'_L), \delta_.(q_\Delta, q'_\Delta), b \vee b')$ s.t. $q_\varnothing$ is a final state of $\mathcal{A}_\varnothing$.

This construction ensures the following lemma, whose complete proof is given in the full version of this paper.

Lemma 4. *For any $t \in E_{\mathrm{PA}}$, $t \overset{\mathcal{A}_{Post^*}}{\longmapsto} (q_\varnothing, q_L, q_\Delta, b)$ iff there is some $u \in E_{\mathrm{PA}}$ and some $p \in \mathbb{N}$ such that $u \xrightarrow{p} t$, $u \overset{\mathcal{A}_L}{\longmapsto} q_L$, $u \overset{\mathcal{A}_\Delta}{\longmapsto} q_\Delta$, ($b = false$ iff $p = 0$) and $t \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$.*

If we now let the final states of $\mathcal{A}_{Post^*}$ be all states $(q_\varnothing, q_L, q_\Delta, b)$ s.t. q_L is a final state of $\mathcal{A}_L$, then $\mathcal{A}_{Post^*}$ accepts a term t iff $u \xrightarrow{*} t$ for some u accepted by $\mathcal{A}_L$ iff t belongs to $Post^*(L)$. We get our first main result:

Theorem 5. (Regularity of $Post^*(L)$)
(1) If L is a regular subset of E_{PA}, then $Post^*(L)$ is regular.
(2) Furthermore, from a TA $\mathcal{A}_L$ recognizing L, is it possible to construct (in polynomial time) a TA $\mathcal{A}_{Post^*}$ recognizing $Post^*(L)$. If $\mathcal{A}_L$ has k states, then $\mathcal{A}_{Post^*}$ needs only have $O(k.|\Delta|)$ states.

Notice that a TA for $Post^+(L)$ can be obtained just by requiring that the final states have $b = true$ as their fourth component.

3.2 Regularity of $Pre^*(L)$

Assume we have a TA $\mathcal{A}_{Pre^*}$ recognizing $Pre^*(L)$. If we consider the same sequence $t_1 \to t_2 \to t_3$ from Fig. 1, we want $\mathcal{A}_{Pre^*}$ to accept t_1 if t_3 is in L. The TA must then read t_1, imitating the behaviour of $\mathcal{A}_L$. When $\mathcal{A}_{Pre^*}$ sees a variable (say, X), it may move to any state q of $\mathcal{A}_L$ that could be reached by some $t \in Post^*(X)$. This accounts for transitions from X, and of course we must keep track of the actual occurences of transitions so that they do not occur in the right-hand side of a "." when the left-hand side is not terminated.

This leads to the following construction:

Ingredients for $\mathcal{A}_{Pre^*}$: Assume $\mathcal{A}_L$ is an automaton recognizing $L \subseteq E_{\mathrm{PA}}$. $\mathcal{A}_{Pre^*}$ is a new automaton combining several ingredients:

- $\mathcal{A}_\varnothing$ is a *completely specified* automaton accepting terminated processes (see Proposition 3).
- $\mathcal{A}_L$ is the automaton accepting L.
- The boolean b records whether some rewriting steps have been done.

States of $\mathcal{A}_{Pre^*}$: A state of $\mathcal{A}_{Pre^*}$ is a 3-tuple $(q_\varnothing \in Q_{\mathcal{A}_\varnothing}, q_L \in Q_{\mathcal{A}_L}, b \in \{true, false\})$ where $Q_{\dots}$ denotes the set of states of the relevant automaton.

Transition rules of $\mathcal{A}_{Pre^*}$: The transition rules of $\mathcal{A}_{Pre^*}$ are defined as follows:

type 0**:** all rules of the form $0 \longmapsto (q_\varnothing, q_L, false)$ s.t. $0 \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$ and $0 \overset{\mathcal{A}_L}{\longmapsto} q_L$.
type 1a**:** all rules of the form $X \longmapsto (q_\varnothing, q_L, true)$ s.t. there exists some $u \in Post^+(X)$ with $u \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$ and $u \overset{\mathcal{A}_L}{\longmapsto} q_L$.
type 1b**:** all rules of the form $X \longmapsto (q_\varnothing, q_L, false)$ s.t. $X \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$ and $X \overset{\mathcal{A}_L}{\longmapsto} q_L$.
type 2**:** all rules of the form $(q_\varnothing, q_L, b) \parallel (q'_\varnothing, q'_L, b') \longmapsto (\delta_\parallel(q_\varnothing, q'_\varnothing), \delta_\parallel(q_L, q'_L), b \vee b')$.
type 3a**:** all rules of the form $(q_\varnothing, q_L, b).(q'_\varnothing, q'_L, b') \longmapsto (\delta_\cdot(q_\varnothing, q'_\varnothing), \delta_\cdot(q_L, q'_L), b \vee b')$ s.t. $q_\varnothing$ is a final state of $\mathcal{A}_\varnothing$.
type 3b**:** all rules of the form $(q_\varnothing, q_L, b).(q'_\varnothing, q'_L, false) \longmapsto (\delta_\cdot(q_\varnothing, q'_\varnothing), \delta_\cdot(q_L, q'_L), b)$.

This construction allows the following lemma, whose complete proof is given in the full version of this paper.

Lemma 6. *For any $t \in E_{\mathrm{PA}}$, $t \overset{\mathcal{A}_{Pre^*}}{\longmapsto} (q_\varnothing, q_L, b)$ iff there is some $u \in E_{\mathrm{PA}}$ and some $p \in \mathbb{N}$ such that $t \xrightarrow{p} u$, $u \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$, $u \overset{\mathcal{A}_L}{\longmapsto} q_L$ and ($b = false$ iff $p = 0$).*

If we now let the final states of $\mathcal{A}_{Pre^*}$ be all states $(q_\varnothing, q_L, b)$ s.t. q_L is a final state of $\mathcal{A}_L$, then $t \xrightarrow{*} u$ for some u accepted by $\mathcal{A}_L$ iff $\mathcal{A}_{Pre^*}$ accepts t (this is where we use the assumption that $\mathcal{A}_\varnothing$ is completely specified). This is summarized by the next theorem.

Theorem 7. (Regularity of $Pre^*(L)$)
(1) If L is a regular subset of E_{PA}, then $Pre^*(L)$ is regular.
(2) Furthermore, from an automaton $\mathcal{A}_L$ recognizing L, is it possible to construct (in polynomial time) an automaton $\mathcal{A}_{Pre^*}$ recognizing $Pre^*(L)$. If $\mathcal{A}_L$ has k states, then $\mathcal{A}_{Pre^*}$ needs only have $4k$ states.

Proof. (1) is an immediate consequence of Lemma 6. Observe that the regularity result does not need the finiteness of Δ (but $Var(\Delta)$ must be finite).

(2) Building $\mathcal{A}_{Pre^*}$ effectively requires an effective way of listing the type 1a rules. This can be done by computing a product of $\mathcal{A}_X$, an automaton for $Post^+(X)$, with $\mathcal{A}_\varnothing$ and $\mathcal{A}_L$. Then there exists some $u \in Post^+(X)$ with $u \overset{\mathcal{A}_\varnothing}{\longmapsto} q_\varnothing$ and $u \overset{\mathcal{A}_L}{\longmapsto} q_L$ iff the the language accepted by the final states $\{(q_X, q_\varnothing, q_L) \mid q_X \text{ a final state of } \mathcal{A}_X\}$ is not-empty. This gives us the pairs $q_\varnothing, q_L$ we need for type 1a rules. Observe that we need the finiteness of Δ to build the $\mathcal{A}_X$'s. □

Actually, the $\xrightarrow{*}$ relation between PA-terms is a *rational tree relation* in the sense of [Rao97]. This entails that $Pre^*(L)$ and $Post^*(L)$ are regular tree languages when L is. Raoult's approach is more powerful than our elementary constructions but it relies on complex new tools (much more powerful than usual

TA's) and does not provide the straightforward complexity analysis we offer. Moreover, the extensions we discuss in section 4 would be more difficult to obtain in his framework.

3.3 Applications

Theorems 5 and 7 already give us simple solutions to verification problems over PA: the *reachability problem* asks, given t, u (and Δ), whether $t \xrightarrow{*} u$. The *boundedness problem* asks whether $Post^*(t)$ is finite. They can be solved in polynomial time just by looking at the TA for $Post^*(t)$. Variant problems such as "*can we reach terms with arbitrarily many occurences of X in parallel ?*" can be solved equally easily.

4 Reachability under constraints

In this section, we consider *reachability under constraints*, that is, reachability where the labels of transitions must respect some criterion. Let $C \subseteq Act^*$ be a (word) language over action names. We write $t \xrightarrow{C} t'$ when $t \xrightarrow{w} t'$ for some $w \in C$, and we say that t' can be reached from t under the constraint C. We extend our notations and write $Pre^*[C](L)$, $Post^*[C](L)$, ... with the obvious meaning.

Observe that, in general, the problem of telling whether $t \xrightarrow{C}$ (i.e. whether $Post^*[C](t)$ is not empty) is undecidable for the PA algebra even if we assume regularity of C [1]. In this section we give sufficient conditions over C so that the problem becomes decidable (and so that we can compute the C-constrained Pre^* and $Post^*$ of a regular tree language).

Recall that the *shuffle* $w \parallel w'$ of two finite words is the set of all words one can obtain by interleaving w and w' in an arbitary way.

Definition 8. $\{(C_1, C_1'), \ldots, (C_m, C_m')\}$ is a (finite) *seq-decomposition* of C iff for all $w, w' \in Act^*$ we have

$$w.w' \in C \quad \text{iff} \quad (w \in C_i, w' \in C_i' \text{ for some } 1 \leq i \leq m).$$

$\{(C_1, C_2'), \ldots, (C_m, C_m')\}$ is a (finite) *paral-decomposition* of C iff for all $w, w' \in Act^*$ we have

$$C \cap (w \parallel w') \neq \varnothing \quad \text{iff} \quad (w \in C_i, w' \in C_i' \text{ for some } 1 \leq i \leq m).$$

[1] E.g. by using two copies $\underline{a}, \overline{a}$ of every letter a in some Σ, and by using the regular constraint $C \stackrel{\text{def}}{=} (\underline{a_1}.\overline{a_1} + \cdots + \underline{a_n}.\overline{a_n})^* \underline{\#}.\overline{\#}$, we can state with "$(\underline{t_1} \parallel \overline{t_2}) \xrightarrow{C} ?$" that t_1 and t_2 share a common trace ending with $\#$. This can be used to encode the (undecidable) empty-intersection problem for context-free grammars.

The crucial point of the definition is that a seq-decomposition of C must apply to all possible ways of splitting any word in C. It even applies to a decomposition $w.w'$ with $w = \varepsilon$ (or $w' = \varepsilon$) so that one of the C_i's (and one of the C'_i's) contains ε. Observe that the formal difference between seq-decomposition and paral-decomposition comes from the fact that $w \parallel w'$, the set of all shuffles of w and w' may contain several elements.

Definition 9. A family $\mathbb{C} = \{C_1, \ldots, C_n\}$ of languages over Act is a *finite decomposition system* iff every $C \in \mathbb{C}$ admits a seq-decomposition and a paral-decomposition only using C_i's from $\mathbb{C}$. A language C is *decomposable* if it appears in a finite decomposition system.

Not all $C \subseteq Act^*$ are decomposable, e.g. $(ab)^*$ is not. It is known that decomposable languages are regular and that all commutative regular languages are decomposable. (Write $w \sim w'$ when w' is a permutation of w. A commutative language is a language C closed w.r.t. $\sim$). Simple examples of commutative languages are obtained by considering the number of occurrences (rather than the positions) of given letters: for any positive weight function θ given by $\theta(w) \stackrel{\text{def}}{=} \sum_i n_i |w|_{a_i}$ with $n_i \in \mathbb{N}$, the set C of all w s.t. $\theta(w) = k$ (or $\theta(w) < k$, or $\theta(w) > k$, or $\theta(w) = k \mod k'$) is a commutative regular language, hence is decomposable.

However, a decomposable language needs not be commutative: finite languages are decomposable, and decomposable languages are closed by union, concatenation and shuffle.

Theorem 10. (Regularity)
For any regular $L \subseteq E_{\mathrm{PA}}$ and any decomposable C, $Pre^*[C](L)$ and $Post^*[C](L)$ are regular tree languages.

Proof. The construction is similar to the constructions for $Pre^*(L)$ and $Post^*(L)$. See the full version of the paper. □

5 Handling structural equivalence of PA-terms

In this section we show how to take into account the axioms $(A_{\cdot})$, $(C_{\parallel})$, $(A_{\parallel})$ and (N_1) to (N_4) (from section 1.3) defining the structural equivalence on E_{PA} terms.

Some definitions of PA consider PA-terms modulo $\equiv$. This viewpoint assumes that a PA-term t really denotes an equivalence class $[t]_{\equiv}$, and that transitions are defined between such equivalence classes, coinciding with a transition relation we would define by

$$[t]_{\equiv} \xrightarrow{a} [u]_{\equiv} \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \exists t' \in [t]_{\equiv}, u' \in [u]_{\equiv} \text{ s.t. } t' \xrightarrow{a} u'. \tag{1}$$

This yields a new process algebra: $\mathrm{PA}_{\equiv}$.

In our framework, we can *define* a new transition relation between PA-terms: $t \stackrel{a}{\Rightarrow} t'$ iff $t \equiv u \stackrel{a}{\rightarrow} u' \equiv t'$ for some u, u', i.e. $[t]_{\equiv} \stackrel{a}{\rightarrow} [u]_{\equiv}$. We adopt the usual abbreviations $\stackrel{*}{\Rightarrow}$, $\stackrel{k}{\Rightarrow}$ for $k \in \mathbb{N}$, etc.

Seeing terms modulo $\equiv$ does not modify the observable behaviour because of the following standard result:

Proposition 11. *$\equiv$ has the transfer property, i.e. it is a bisimulation relation, i.e. for all $t \equiv t'$ and $t \stackrel{a}{\rightarrow} u$ there is a $t' \stackrel{a}{\rightarrow} u'$ with $u \equiv u'$ (and vice versa).*

Proof. Check this for each equation, then deal with the general case by using congruence property of $\equiv$ and structural induction over terms, transitivity of $\equiv$ and induction over the number of equational replacements needed to relate t and t'. Observe that *IsNil* is compatible with $\equiv$. □

Proposition 12. *$t \stackrel{k}{\Rightarrow} u$ iff $t \stackrel{k}{\rightarrow} u'$ for some $u' \equiv u$.*

The reachability problem solved by Mayr actually coincides with "reachability modulo $\equiv$" or "reachability through $\stackrel{*}{\Rightarrow}$". Our tree automata method can deal with this, as we now show.

5.1 Structural equivalence and regularity

$(A_.)$, $(C_\|)$ and $(A_\|)$ are the associativity-commutativity axioms satisfied by . and $\|$. We call them the *permutative axioms* and write $t =_P u$ when t and u are permutatively equivalent.

(N_1) to (N_4) are the axioms defining 0 as the neutral element of . and $\|$. We call them the *simplification axioms* and write $t \searrow u$ when u is a simplification of t, i.e. u can be obtained by applying the simplification axioms *from left to right* at some positions in t. Note that $\searrow$ is a (well-founded) partial ordering. We write $\swarrow$ for $(\searrow)^{-1}$. The *simplification normal form* of t, written $t{\downarrow}$, is the unique u one obtains by simplifying t as much as possible (no permutation allowed).

Such axioms are classical in rewriting and have been extensively studied [BN98]. $\equiv$ coincide with $(=_P \cup \searrow \cup \swarrow)^*$. Now, because the permutative axioms commute with the simplification axioms, we have

$$t \equiv t' \quad \text{iff} \quad t \searrow u =_P u' \swarrow t' \text{ for some } u, u' \quad \text{iff} \quad t{\downarrow} =_P t'{\downarrow}. \qquad (2)$$

Lemma 13. *For any t, the set $[t]_{=_P} \stackrel{def}{=} \{u \mid t =_P u\}$ is a regular tree language, and an automaton for $[t]_{=_P}$ needs only have $m.(m/2)!$ states if $|t| = m$.*

Note that for a regular L, $[L]_{=_P}$ (and $[L]_{\equiv}$) are not necessarily regular.

The simplification axioms do not have the nice property that they only allow finitely many combinations, but they behave better w.r.t. regularity. Write $[L]_{\searrow}$ for $\{u \mid t \searrow u \text{ for some } t \in L\}$, $[L]_{\swarrow}$ for $\{u \mid u \searrow t \text{ for some } t \in L\}$, and $[L]{\downarrow}$ for $\{t{\downarrow} \mid t \in L\}$.

Lemma 14. *For any regular L, the sets $[L]_{\searrow}$, $[L]_{\swarrow}$, and $[L]{\downarrow}$ are regular tree languages. From an automaton $\mathcal{A}$ recognizing L, we can build automata for these three languages in polynomial time.*

Corollary 15. *"Boundedness modulo $\equiv$" of the reachability set is decidable in polynomial-time.*

Proof. Because the permutative axioms only allow finitely many variants of any given term, $Post^*(L)$ contains a finite number of non-$\equiv$ processes iff $[Post^*(L)]{\downarrow}$ is finite. □

We can also combine (2) and lemmas 13 and 14 and have

Proposition 16. *For any t, the set $[t]_{\equiv}$ is a regular tree language, and an automaton for $[t]_{\equiv}$ needs only have $m.(m/2)!$ states if $|t| = m$.*

Now it is easy to prove decidability of the reachability problem modulo $\equiv$: $t \stackrel{*}{\Rightarrow} u$ iff $Post^*(t) \cap [u]_{\equiv} \neq \varnothing$. Recall that $[u]_{\equiv}$ and $Post^*(t)$ are regular tree-languages one can build effectively. Hence it is decidable whether they have a non-empty intersection.

This gives us a simple algorithm using exponential time (because of the size of $[u]_{\equiv}$). Actually we can have a better result [2]:

Theorem 17. *The reachability problem in* $\mathrm{PA}_{\equiv}$*, "given t and u, do we have $t \stackrel{*}{\Rightarrow} u$?", is NP-complete.*

Proof. NP-hardness of reachability for BPP's is proved in [Esp97] and the proof idea can be reused in our framework (see long version).
NP-easiness is straightforward in the automata framework. We have $t \stackrel{*}{\Rightarrow} u$ iff $t \stackrel{*}{\rightarrow} u'$ for some u' s.t. $u'{\downarrow} =_P u{\downarrow}$. Write u'' for $u'{\downarrow}$ and note that $|u''| \leq |u|$. A simple NP algorithm is to compute $u{\downarrow}$, then *guess non-deterministically* a permutation u'', then build automata $\mathcal{A}_1$ for $[u'']_{\searrow}$ and $\mathcal{A}_2$ for $Post^*(t)$. These automata have polynomial-size. There remains to checks whether $\mathcal{A}_1$ and $\mathcal{A}_2$ have a non-empty intersection to know whether the required u' exists. □

6 Model-checking PA processes

In this section we show a simple approach to the model-checking problem which is an immediate application of our main regularity theorems. We do not consider the structural equivalence $\equiv$ until section 6.3, where we show that the decidability results are a simple consequence of our previous results.

[2] First proved in [May97c]

6.1 Model-checking in E_{PA}

We consider a set $Prop = \{P_1, P_2, \ldots\}$ of *atomic propositions*. For $P \in Prop$, Let $Mod(P)$ denotes the set of PA processes for which P holds. We only consider propositions P such that $Mod(P)$ is a regular tree-language. Thus P could be "t can make an a-labeled step right now", "there is at least two occurences of X inside t", "there is exactly one occurence of X in a non-frozen position", ...

The logic EF has the following syntax:

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \mathsf{EX}\varphi \mid \mathsf{EF}\varphi$$

and semantics

$$\begin{array}{rl} t \models P & \stackrel{\text{def}}{\Leftrightarrow} t \in Mod(P), \\ t \models \neg\varphi & \stackrel{\text{def}}{\Leftrightarrow} t \not\models \varphi, \\ t \models \varphi \wedge \varphi' & \stackrel{\text{def}}{\Leftrightarrow} t \models \varphi \text{ and } t \models \varphi', \end{array} \qquad \begin{array}{l} t \models \mathsf{EX}\varphi \stackrel{\text{def}}{\Leftrightarrow} t' \models \varphi \text{ for some } t \to t', \\ t \models \mathsf{EF}\varphi \stackrel{\text{def}}{\Leftrightarrow} t' \models \varphi \text{ for some } t \stackrel{*}{\to} t'. \end{array}$$

Thus $\mathsf{EX}\varphi$ reads "it is possible to reach in one step a state s.t. φ" and $\mathsf{EF}\varphi$ reads "it is possible to reach (via some sequence of steps) a state s.t. φ".

Definition 18. The *model-checking problem* for EF over PA has as inputs: a given Δ, a given t in E_{PA}, a given φ in EF. The answer is yes iff $t \models \varphi$.

We now extend the definition of Mod to the whole of EF: $Mod(\varphi) \stackrel{\text{def}}{=} \{t \in E_{\mathrm{PA}} \mid t \models \varphi\}$, we have

$$\begin{array}{ll} Mod(\neg\varphi) = E_{\mathrm{PA}} - Mod(\varphi) & Mod(\mathsf{EX}\varphi) = Pre^{+}(Mod(\varphi) \\ Mod(\varphi \wedge \varphi') = Mod(\varphi) \cap Mod(\varphi') & Mod(\mathsf{EF}\varphi) = Pre^{*}(Mod(\varphi) \end{array} \tag{3}$$

Theorem 19. *(1) For any EF formula φ, $Mod(\varphi)$ is a regular tree language. (2) If we are given tree-automata $\mathcal{A}_P$'s recognizing the regular sets $Mod(P)$, then a tree-automaton $\mathcal{A}_\varphi$ recognizing $Mod(\varphi)$ can be built effectively.*

This gives us a decision procedure for the model-checking problem: build an automaton for $Mod(\varphi)$ and check whether it accepts t. Observe that computing a representation of $Mod(\varphi)$ is more general than just telling whether a given t belongs to it. Observe also that our results allow model-checking approches based on combinations of forward and backward methods (while Theorem 19 only relies on the standard backward approach.)

The above procedure is non-elementary since every nesting level of negations potentially induces an exponential blowup. Actually, negations in φ can be pushed towards the leaves and only stop at the EF's, so that really the tower of exponentials depend on the maximal number of alternations between negations and EF's in φ. The procedure described in [May97b] is non-elementary and today the known lower bound is PSPACE-hard.

6.2 Model-checking with constraints

We can also use the constraints introduced in section 4 to define an extended EF logic where we now allow all $\langle C \rangle \varphi$ formulas for decomposable C. The meaning is given by $Mod(\langle C \rangle \varphi) \stackrel{\text{def}}{=} Pre^*[C](Mod(\varphi))$. This is quite general and immediately include the extensions proposed in [May97b].

6.3 Model-checking modulo $\equiv$

The model-checking problem solved in [May97b] considers the EF logic over $\text{PA}_{\equiv}$.

In this framework, the semantics of EF-formulas is defined over equivalence classes, or equivalently, using the $\stackrel{a}{\Rightarrow}$ relation and only considering atomic propositions P s.t. $Mod(P)$ is closed under $\equiv$.

But if the $Mod(P)$'s are closed under $\equiv$, then $t \models \varphi$ in PA iff $t \models \varphi$ in $\text{PA}_{\equiv}$ (a consequence of Proposition refprop-equiv-transfer), so that our earlier tree-automata algorithm can be used to solve the model-checking problem for $\text{PA}_{\equiv}$. We can also easily allow constraints like in the previous section.

Conclusion

In this paper we showed how tree-automata techniques are a powerful tool for the analysis of the PA process algebra. Our main results are two general Regularity Theorems with numerous immediate applications, including model-checking of PA with an extended EF logic.

The tree-automata viewpoint has many advantages. It gives simpler and more general proofs. It helps understand why some problems can be solved in P-time, some others in NP-time, etc. It is quite versatile and we believe that many variants of PA can be attacked with the same approach.

We certainly did not list all possible applications of the tree-automata approach for verification problems in PA. Future work should aim at better understanding which problems can benefit from our TA viewpoint and techniques.

Acknowledgments We thank H. Comon and R. Mayr for their numerous suggestions, remarks and questions about this work.

References

[BBK87] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proc. Parallel Architectures and Languages Europe (PARLE'87), Eindhoven, NL, June 1987, vol. II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 94–111. Springer-Verlag, 1987.

[BE97] O. Burkart and J. Esparza. More infinite results. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 30–31, 1996*, volume 5 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.

[BEH95] A. Bouajjani, R. Echahed, and P. Habermehl. Verifying infinite state processes with sequential and parallel composition. In *Proc. 22nd ACM Symp. Principles of Programming Languages (POPL'95), San Francisco, CA, USA, Jan. 1995*, pages 95–106, 1995.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1997.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Büc64] J. R. Büchi. Regular canonical systems. *Arch. Math. Logik Grundlag.*, 6:91–111, 1964.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata and their application, 1997. A preliminary version of this (yet unpublished) book is available at `http://l3ux02.univ-lille3.fr/tata`.

[CHM94] S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994.

[CKSV97] H. Comon, D. Kozen, H. Seidl, and M. Y. Vardi, editors. *Applications of Tree Automata in Rewriting, Logic and Programming*, Dagstuhl-Seminar-Report number 193. Schloß Dagstuhl, Germany, 1997.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symp. Logic in Computer Science (LICS'90), Philadelphia, PA, USA, June 1990*, pages 242–248, 1990.

[Esp97] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31(1):13–25, 1997.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems (extended abstract). In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97), Bologna, Italy, July 11–12, 1997*, volume 9 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer-Verlag, 1997.

[HJ98] Y. Hirshfeld and M. Jerrum. Bisimulation equivalence is decidable for normed Process Algebra. Research Report ECS-LFCS-98-386, Lab. for Foundations of Computer Science, Edinburgh, May 1998.

[JKM98] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. Tech. Report TUM-I9805, Institut für Informatik, TUM, Munich, Germany, February 1998. To appear in Proc. ICALP'98, Aalborg, DK, July 1998.

[KS97a] O. Kouchnarenko and Ph. Schnoebelen. A model for recursive-parallel programs. In *Proc. 1st Int. Workshop on Verification of Infinite State Systems (INFINITY'96), Pisa, Italy, Aug. 1996*, volume 5 of *Electronic Notes in Theor. Comp. Sci.* Elsevier, 1997.

[KS97b] O. Kushnarenko and Ph. Schnoebelen. A formal framework for the analysis of recursive-parallel programs. In *Proc. 4th Int. Conf. Parallel Computing Technologies (PaCT'97), Yaroslavl, Russia, Sep. 1997*, volume 1277 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 1997.

[Kuč96] A. Kučera. Regularity is decidable for normed PA processes in polynomial time. In *Proc. 16th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'96), Hyderabad, India, Dec. 1996*, volume 1180 of *Lecture Notes in Computer Science*, pages 111–122. Springer-Verlag, 1996.

[Kuč97] A. Kučera. How to parallelize sequential processes. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 302–316. Springer-Verlag, 1997.

[May97a] R. Mayr. Combining Petri nets and PA-processes. In *Proc. 4th Int. Symp. Theoretical Aspects Computer Software (TACS'97), Sendai, Japan, Sep. 1997*, volume 1281 of *Lecture Notes in Computer Science*, pages 547–561. Springer-Verlag, 1997.

[May97b] R. Mayr. Model checking PA-processes. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 332–346. Springer-Verlag, 1997.

[May97c] R. Mayr. Tableaux methods for PA-processes. In *Proc. Int. Conf. Automated Reasoning with Analytical Tableaux and Related Methods (TABLEAUX'97), Pont-à-Mousson, France, May 1997*, volume 1227 of *Lecture Notes in Artificial Intelligence*, pages 276–290. Springer-Verlag, 1997.

[Mol96] F. Moller. Infinite results. In *Proc. 7th Int. Conf. Concurrency Theory (CONCUR'96), Pisa, Italy, Aug. 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer-Verlag, 1996.

[Rao97] J.-C. Raoult. Rational tree relations. *Bull. Belg. Math. Soc.*, 4:149–176, 1997.

A String–rewriting Characterization of Context–free Graphs

Hugues Calbrix[1] and Teodor Knapik[2]

[1] **College Jean Lecanuet**, BP 1024
76171 Rouen Cedex, France
e-mail: HugCalbrix@aol.com
[2] **IREMIA**, Université de la Réunion, BP 7151,
97715 Saint Denis Messageries Cedex 9, Réunion
e-mail: knapik@univ–reunion.fr

Abstract. This paper introduces Thue specifications, an approach for string–rewriting description of infinite graphs. It is shown that *strongly reduction–bounded* and *unitary reduction–bounded* rational Thue specifications have the same expressive power and both characterize the context–free graphs of Muller and Schupp. The problem of strong reduction–boundedness for rational Thue specifications is shown to be undecidable but the class of unitary reduction–bounded rational Thue specifications, that is a proper subclass of strongly reduction–bounded rational Thue specifications, is shown to be recursive.

Keywords: string–rewriting, infinite graphs, automata.

1 Introduction

Since countable graphs or, more precisely, transition systems can model any software or digital hardware system, the study of infinite graphs is, in authors opinion, an important task. Obviously, dealing with infinite graphs requires a finite description. With this aim in view, several formalisms have arisen in graph rewriting [8]. In this paper another approach is introduced. The idea comes from the categorist's way of expressing equations between words (of the monoid generated by the arrows of a category) by means of commutative diagrams.[1]

By orienting them, equations between words are turned into string–rewrite rules. String rewriting systems were introduced early in this century by Axel Thue [14] in his investigations about the word problem and are also known as semi–Thue systems. Later, semi–Thue systems became useful in formal languages theory (see [12] for an overview) and, as pointed out in this paper, are also of interest as finite descriptions of infinite graphs. Other approaches relating semi–Thue systems to infinite graphs may be found in [13] and [3]. In the latter paper, the class of context–free graphs is characterized by means of prefix rewriting using labeled rewrite rules.

The link between infinite graphs and semi–Thue systems introduced in this paper raises the following question. Which classes of graphs may be described by semi–Thue systems ? As a first element of the answer, a string–rewriting characterization of context–free graphs of Muller and Schupp [9] is provided as follows. Sect. 2 is devoted to basic definitions. Thue specifications and their

[1] The idea underlying the definition of the Cayley graph associated to a group presentation leads to a similar result.

graphs are defined in Sect. 3. Two classes of Thue specifications are described in Sect. 4 and the main result is established in Sect. 5. In Sect. 6 the authors investigate whether these classes are recursive. Several conclusions close the paper.

2 Preliminaries

Assuming a smattering of string–rewriting and formal languages several basic definitions and facts are reminded in this section. An introductory material on above topics may be found in e.g. [2] and [11].

Words Given a finite set A called *alphabet*, the elements of which are called *letters*, A^* stands for the *free monoid* over A. The elements of A^* are all *words* over A, including the *empty word*, written ε. A subset of A^* is a *language* over A. Each word u is mapped to its *length*, written $|u|$ via the unique monoid homomorphism from A^* onto $(\mathbb{N}, 0, +)$ that maps each letter of A to 1. When $u = xy$ for some words x and y then y is called a *suffix of* u. The set of suffixes of u is written $\mathrm{suff}(u)$. This notation is extended to sets in the usual way: $\mathrm{suff}(L) = \bigcup_{u \in L} \mathrm{suff}(u)$ for any language L.

Semi-Thue systems A semi–Thue system $\mathcal{S}$ (an *sts* for short) over A is a subset of $A^* \times A^*$. A pair (l, r) of $\mathcal{S}$ is called *(rewrite) rule*, the word l (resp. r) is its lefthand (resp. righthand) side. As any binary relation, $\mathcal{S}$ has its domain (resp. range) written $\mathcal{D}\mathrm{om}(\mathcal{S})$ (resp. $\mathcal{R}\mathrm{an}(\mathcal{S})$). Throughout this paper, only finite semi–Thue systems are considered.

The *single-step reduction relation* induced by $\mathcal{S}$ on A^*, is the binary relation $\rightarrow_{\mathcal{S}} = \{(xly, xry) \mid x, y \in A^*,\ (l, r) \in \mathcal{S}\}$. A word u *reduces* into a word v, written $u \rightarrow_{\mathcal{S}}^* v$, if there exist words $u_0, \ldots u_k$ such that $u_0 = u$, $u_k = v$ and $u_i \rightarrow_{\mathcal{S}} u_{i+1}$ for each $i = 0, \ldots, k-1$. The integer k is then the *length of the reduction* under consideration.

A word v is *irreducible* with respect to (w.r.t. for short) $\mathcal{S}$ when v does not belong to $\mathcal{D}\mathrm{om}(\rightarrow_{\mathcal{S}})$. Otherwise v is *reducible* w.r.t. $\mathcal{S}$. It is easy to see that the set of all irreducible words w.r.t. $\mathcal{S}$, written $\mathcal{I}\mathrm{rr}(\mathcal{S})$, is rational whenever $\mathcal{D}\mathrm{om}(\mathcal{S})$ is, since $\mathcal{D}\mathrm{om}(\rightarrow_{\mathcal{S}}) = A^*(\mathcal{D}\mathrm{om}(\mathcal{S}))A^*$. A word v is a *normal form* of a word u, when v is irreducible and $u \rightarrow_{\mathcal{S}}^* v$.

Graphs Given an alphabet A, a *simple directed edge–labeled graph G over A* is a set of *edges*, viz a subset of $D \times A \times D$ where D is an arbitrary set. Given $d, d' \in D$, an edge from d to d' labeled by $a \in A$ is written $d \xrightarrow{a} d'$. A (finite) *path* in G from some $d \in D$ to some $d' \in D$ is a sequence of edges of the following form: $d_0 \xrightarrow{a_1} d_1, \ldots, d_{n-1} \xrightarrow{a_n} d_n$, such that $d_0 = d$ and $d_n = d'$.

For the purpose of this paper, isolated vertices need not to be considered. Moreover, the interests lies basically in graphs, the vertices of which are all accessible from some distinguished vertex. Thus, a graph $G \subseteq D \times A \times D$ is said to be *rooted on a vertex $e \in D$* if there exists a path from e to each vertex of G. The following assumption is made for the sequel. Whenever in a definition of a graph a vertex e is distinguished as root, then the maximal subgraph rooted on e is understood.

Pushdown Machines and Context-free Graphs An important class of graphs with decidable monadic second–order theory is characterized in [9]. The graphs of this class are called *context–free* by Muller and Schupp and may be defined by means of pushdown machines.

A *pushdown machine* over A (a *pdm* for short) is a triple $\mathscr{P} = (Q, Z, T)$ where Q is the set of *states*, Z is the *stack alphabet* and T is a finite subset of $A \cup \{\varepsilon\} \times Q \times Z \times Z^* \times Q$, called the set of *transition rules*. A is the *input alphabet*. A pdm $\mathscr{P}$ is *realtime* when T is a finite subset of $A \times Q \times Z \times Z^* \times Q$.

An *internal configuration* of a pdm $\mathscr{P}$ is a pair $(q, h) \in Q \times Z^*$. To any pdm $\mathscr{P}$ together with an internal configuration ι, one may associate an edge-labeled oriented graph $\mathcal{G}(\mathscr{P}, \iota)$ defined as follows. The vertices of the graph are all internal configurations accessible from the configuration ι. The latter one is the root of the graph. There is an edge labeled by $a \in A \cup \{\varepsilon\}$ from (q_1, h_1) to (q_2, h_2) whenever there exists a letter $z \in Z$ and two words $g_1, g_2 \in Z^*$ such that $h_1 = g_1 z$, $h_2 = g_1 g_2$ and $(a, q_1, z, g_2, q_2) \in T$.

It may be useful to note that the context–free graphs are exactly all equational (in the sense of Courcelle [5]) graphs of finite degree. The equational graphs are also called regular by Caucal [3]. Finally, since any pdm over A is a realtime pdm over $A \cup \{\varepsilon\}$, realtime pdm's are as powerful as pdm's for describing graphs. In other words, the graphs of realtime pdm's form a complete set of representatives of context–free graphs.

3 Thue Specifications and Their Graphs

The key ideas of this paper are introduced in the present section.

Definition 3.1. An *(oriented) Thue specification* (an *ots* for short) over an alphabet A is a triple $\langle \mathcal{S}, L, u \rangle$ where $\mathcal{S}$ is a semi–Thue system over A, L is a subset of $\mathrm{Irr}(\mathcal{S})$ and u is a word of L. An ots is *rational* if L is so.

The reader may notice that, according to the definition below, the models of oriented Thue specifications have a flavour of the Cayley graphs.

Definition 3.2. The *model* of an ots $\langle \mathcal{S}, L, u \rangle$ is the graph, written $\mathcal{G}(\mathcal{S}, L, u)$, defined as follows. The vertices of the graph are all words of L that are accessible via edges from the root u of the graph. The edges of $\mathcal{G}(\mathcal{S}, L, u)$ are labeled by the letters of A. There is an edge labeled by a from w to v whenever v is a normal form of wa.

It should be noted that termination of $\mathcal{S}$ is not required in this definition. Thus a vertex w of $\mathcal{G}(\mathcal{S}, L, u)$ has no outgoing edge labeled by a if and only if wa has no normal form or no normal form of wa belong to L.

Example 3.3. Over the alphabet $A_1 = \{a, b\}$, consider a single-rule sts $\mathcal{S}_1 = \{(ba, ab)\}$. The set of irreducible words is a^*b^*. The graph $\mathcal{G}(\mathcal{S}_1, \mathrm{Irr}(\mathcal{S}_1), \varepsilon)$ (see Fig. 1) is isomorphic to $\omega \times \omega$.

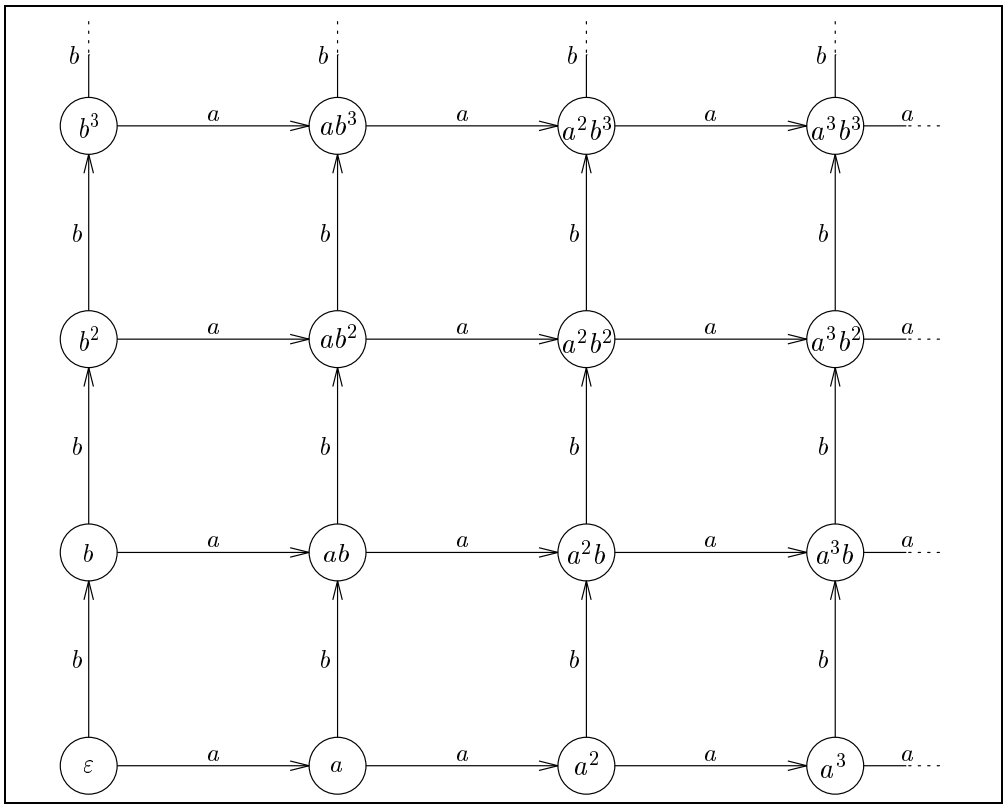

Fig. 1. Graph $\mathcal{G}(\mathcal{S}_1, \mathcal{I}\mathrm{rr}(\mathcal{S}_1), \varepsilon)$

Example 3.4. Over the alphabet $A_2 = \{a, b, \overline{a}, \overline{b}\}$ consider the following sts: $\mathcal{S}_2 = \{(a\overline{a}, \varepsilon), (\overline{a}a, \varepsilon), (b\overline{b}, \varepsilon), (\overline{b}b, \varepsilon)\}$. The set of irreducible words w.r.t. $\mathcal{S}_2$ is the set of all reduced words representing the elements of the free group generated by $\{a, b\}$. The graph[2] $\mathcal{G}(\mathcal{S}_2, \mathcal{I}\mathrm{rr}(\mathcal{S}_2), \varepsilon)$ (see Fig. 2) is isomorphic to the Cayley graph of the two-generator free group.

4 Two equivalent conditions

This section introduces two notions that help characterizing Thue specifications that generate context–free graphs.

An sts $\mathcal{S}$ over A is *strongly reduction–bounded* on a subset L of $\mathcal{I}\mathrm{rr}(\mathcal{S})$ when there exists a positive integer k such that for each word u of L and each a in A, the length of any reduction of ua is less than k. The integer k is then called a *reduction bound* of $\mathcal{S}$ (on L).

An sts $\mathcal{S}$ is *unitary reduction–bounded* on a subset L of $\mathcal{I}\mathrm{rr}(\mathcal{S})$ when it is strongly reduction–bounded on L and 1 is a reduction bound of $\mathcal{S}$.

[2] For each edge, there is the opposite edge (not depicted) corresponding to the formal inverse of the label.

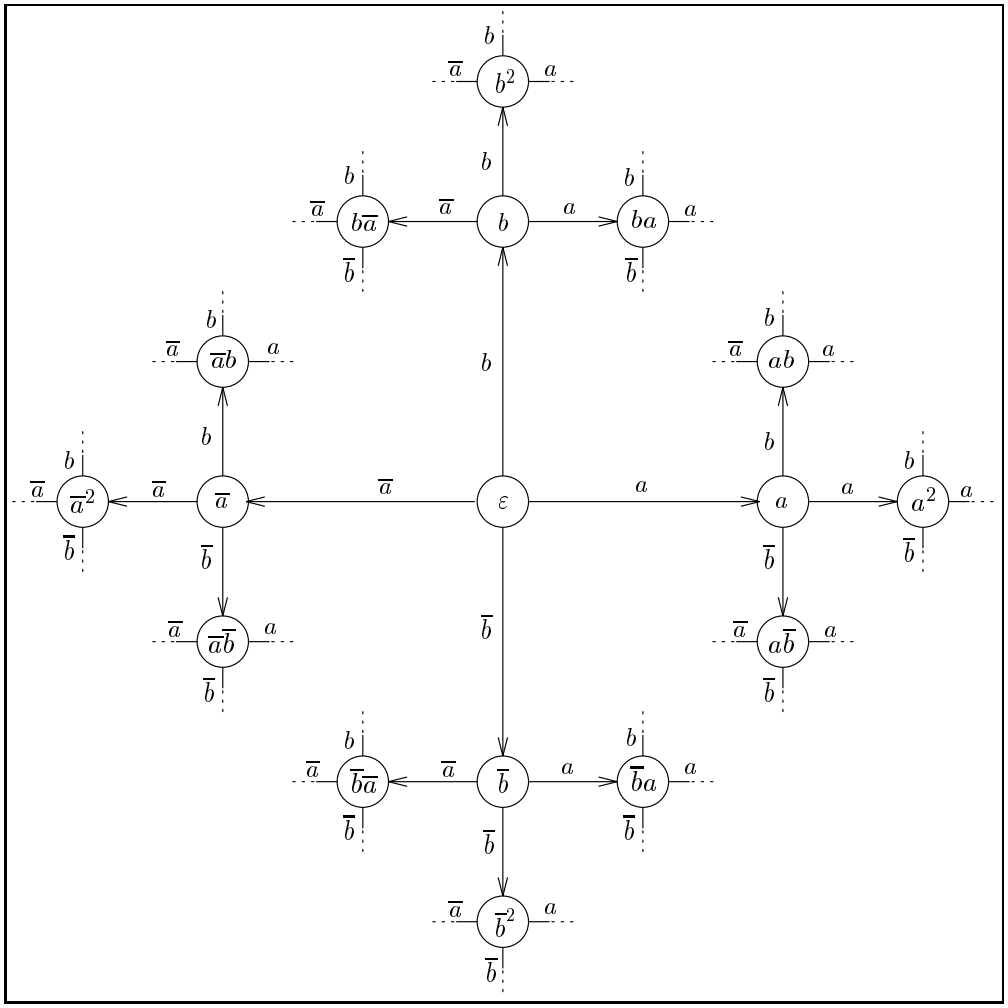

Fig. 2. Graph $\mathcal{G}(\mathcal{S}_2, \mathcal{I}\mathrm{rr}(\mathcal{S}_2), \varepsilon)$

A Thue specification $\langle \mathcal{S}, L, u \rangle$ is *strongly reduction–bounded* (resp. *unitary reduction–bounded*) is $\mathcal{S}$ is strongly reduction–bounded (resp. unitary reduction–bounded) on L.

The system $\mathcal{S}_1$ from Example 3.3 is not strongly reduction–bounded on $\mathcal{I}\mathrm{rr}(\mathcal{S})$. Indeed, the word b^n is irreducible and n is the length of the reduction of the word $b^n a$ into its normal form ab^n. Since n is an arbitrary positive integer, $\mathcal{S}_1$ has no reduction bound.

On the contrary, the sts $\mathcal{S}_2$ from Example 3.4 is unitary reduction–bounded on $\mathcal{I}\mathrm{rr}(\mathcal{S}_2)$. As a matter of fact, given a nonempty word $w = uc$ of $\mathcal{I}\mathrm{rr}(\mathcal{S}_2)$ with $c \in A_2$, for any $d \in A_2$ the normal form of wd is u if $c = \overline{d}$ or $d = \overline{c}$, and wd otherwise. In both cases the length of the corresponding reduction is not greater than 1. It may be observed that $\mathcal{G}(\mathcal{S}_2, \mathcal{I}\mathrm{rr}(\mathcal{S}_2), \varepsilon)$ is a tree.

The next proposition and the subsequent comment establish inclusions between some familiar classes of semi–Thue systems (see e.g. [2] and [12] for def-

initions of these classes) on one hand, and strongly (resp. unitary) reduction–bounded semi–Thue systems on the other hand.

Proposition 4.1. *For all finite semi–Thue systems $\mathcal{S}$ the following holds.*

1. *If $\mathcal{S}$ is special then $\mathcal{S}$ is unitary reduction–bounded.*
2. *If $\mathcal{R}\mathrm{an}(\mathcal{S}) \subseteq \mathcal{I}\mathrm{rr}(\mathcal{S})$ and no word of $\mathcal{R}\mathrm{an}(\mathcal{S})$ is strictly overlapped on left by a word of $\mathcal{D}\mathrm{om}(\mathcal{S})$ then $\mathcal{S}$ is unitary reduction–bounded.*
3. *If $\mathcal{S}$ is terminating left-basic then $\mathcal{S}$ is strongly reduction–bounded.*

Proof. Both (1) and (2) are obvious. Let then $\mathcal{S}$ be a finite terminating left-basic semi–Thue system over A. Let $w \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and $a \in A$. Consider the longest suffix v of w such that $va \in \mathcal{D}\mathrm{om}(\mathcal{S})$. Observe that, since $\mathcal{S}$ is left-basic, each reduction of wa concerns only va. Let therefore k be the maximum of the lengths of all reductions of the words of $\mathcal{D}\mathrm{om}(\mathcal{S})$. Obviously, k is a reduction bound of $\mathcal{S}$. □

It may be noted that no converse of statements (1), (2) or (3) of the above proposition holds. Indeed, the semi–Thue system $\{aa \to a\}$ over $\{a\}$ is unitary reduction–bounded but is not special; a is strictly overlapped on left by aa hence the system is not left-basic. On the other hand, Proposition 4.1 cannot be strengthened to the case of monadic semi–Thue systems. The semi–Thue system $\{ab \to b\}$ over $\{a, b\}$ is monadic without being strongly reduction–bounded.

The following result demonstrates that the strongly reduction–bounded rational ots and unitary reduction–bounded rational ots have the same expressive power for describing graphs.

Proposition 4.2. *Given any strongly reduction–bounded rational ots $\langle \mathcal{S}, R, u\rangle$, one may construct a unitary reduction–bounded rational ots $\langle \mathcal{S}', R', u'\rangle$ such that the graphs $\mathcal{G}(\mathcal{S}, R, u)$ and $\mathcal{G}(\mathcal{S}', R', u')$ are isomorphic.*

Proof. Let k be a reduction bound of $\mathcal{S}$ and $m = \max_{l \in \mathcal{D}\mathrm{om}(\mathcal{S})} |l|$ be the maximum length of the lefthand sides of $\mathcal{S}$. Consider any reduction of length n of a word wa such that $w \in R$ and $a \in A$. If $|wa| > mn$, then only a strict suffix of wa is reduced. The length of the reduced suffix cannot exceed mn. Since $n \leq k$, for any reduction, the length of the reduced suffix cannot exceed mk.

Let $\mathcal{S}' = \mathcal{S}'_{\#} \cup \mathcal{S}'_A$ be an sts over $A \cup \{\#\}$, where $\# \notin A$, defined as follows:

$$\mathcal{S}'_{\#} = \{(\#wa, \#v) \mid a \in A,\ w \in R, v \in \mathcal{I}\mathrm{rr}(\mathcal{S}),\ wa \xrightarrow[\mathcal{S}]{*} v \text{ and } |wa| < mk\},$$

$$\mathcal{S}'_A = \{(wa, v) \mid a \in A,\ w \in \mathrm{suff}(R), v \in \mathcal{I}\mathrm{rr}(\mathcal{S}),\ wa \xrightarrow[\mathcal{S}]{*} v \text{ and } |wa| = mk\}$$

and let $R' = \#R$.

Unit reduction–boundedness of $\mathcal{S}'$ on R' is established as follows. Let $w \in R'$ and $a \in A'$. If $a = \#$ then $wa \notin R'$. Assume then that $a \in A$. Observe first that, if $wa \to_{\mathcal{S}'_{\#}} x$ for some $x \in (A')^*$ then $x \in \mathcal{I}\mathrm{rr}(\mathcal{S}')$. Assume therefore by contradiction that there is a reduction $wa \to_{\mathcal{S}'_A} x \to_{\mathcal{S}'} y$ for some $x, y \in (A')^*$. Let w_1 be the longest prefix of w this reduction is not concerned with and let w_2 be the remaining suffix, viz $w = w_1 w_2$. Now, either $w_2 \in \mathrm{suff}(R)$ (when

$x \to_{\mathcal{S}'_A} y$) and $|w_2| > mk$ or $w_2 = \#w'_2$ (when $x \to_{\mathcal{S}'_\#} y$) for some $w'_2 \in R$ such that $|w'_2| > mk$. Hence, there exists a reduction of $w_2 a$ (resp. $w'_2 a$) w.r.t. $\mathcal{S}$ the length of which exceeds k. This contradicts the assumption that k is a reduction bound of $\mathcal{S}$ on R.

Observe that for all $w \in R$, $v \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and $a \in A$, v is a normal form of wa w.r.t. $\mathcal{S}$ if and only if $\#v$ is a normal form of $\#wa$ w.r.t. $\mathcal{S}'$. Hence, the mapping $w \mapsto \#w$ restricted to vertices of $\mathcal{G}(\mathcal{S}, R, u)$ extends to a graph isomorphism between $\mathcal{G}(\mathcal{S}, R, u)$ and $\mathcal{G}(\mathcal{S}', R', u')$ where $u' = \#u$. □

5 Main result

Proposition 4.2 together with the statements of this section lead to the main result of this paper.

Proposition 5.1. *Given any realtime pdm $\mathscr{P}$ over A and an internal configuration ι of $\mathscr{P}$, one may construct a unitary reduction–bounded rational ots $\langle \mathcal{S}, R, u \rangle$ such that the graphs $\mathcal{G}(\mathscr{P}, \iota)$ and $\mathcal{G}(\mathcal{S}, R, u)$ are isomorphic.*

Proof. Let $\mathscr{P} = (Q, Z, T)$ be a pdm over A and let $\iota = (q_0, h_0)$ be an internal configuration of $\mathscr{P}$. Without loss of generality A, Q and Z may be assumed pairwise disjoint. Set $A' = A \cup Q \cup Z$. Define an sts $\mathcal{S}$ over A' as follows

$$\mathcal{S} = \{(zqb, hq') \mid (b, q, z, h, q') \in T\}$$

and let $u = h_0 q_0$. It is well known that the pushdown store language of a pdm is rational. The following language is therefore rational:

$$R = \{hq \mid (q, h) \text{ is an internal configuration of } \mathscr{P} \text{ accessible from } \iota\} \ .$$

Moreover $R \subseteq \mathcal{I}\mathrm{rr}(\mathcal{S})$.

Observe that $\mathcal{S}$ is unitary reduction–bounded on R. Indeed, let $v \in R$ and $a \in A'$. For va to be reducible, there must exist $w \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and a rewrite rule (zqa, hq') such that $v = wzqa$. Consequently, $va \to_{\mathcal{S}} whq'$. But no word of $\mathcal{D}\mathrm{om}(\mathcal{S})$ may overlap hq' on left. Since w is irreducible, so is whq'.

The fact that is $\mathcal{G}(\mathscr{P}, \iota)$ and $\mathcal{G}(\mathcal{S}, R, u)$ are isomorphic is readily established by induction on the distance of vertex from the root, using the following one to one mapping of the vertices of $\mathcal{G}(\mathscr{P}, \iota)$ onto the vertices of $\mathcal{G}(\mathcal{S}, R, u)$: $(q, h) \mapsto hq$. □

In order to establish the converse, the following lemma is useful.

Lemma 5.2. *Given a pdm $\mathscr{P} = (Q, Z, T)$ over A, an internal configuration ι of $\mathscr{P}$ and a rational subset R of Z^*Q, one may construct a pdm $\mathscr{P}'$ together with ι' such that the graph $\mathcal{G}(\mathscr{P}', \iota')$ is isomorphic to the restriction of $\mathcal{G}(\mathscr{P}, \iota)$, rooted on ι, to the following set of vertices: $\{(q, h) \in Q \times Z^* \mid hq \in R\}$. Moreover $\mathscr{P}'$ is realtime if $\mathscr{P}$ is so.*

Proof. Let $\mathscr{A} = (D, d_0, \delta, F)$ be a finite deterministic and complete automaton over A that accepts R. Here D is the set of states of $\mathscr{A}$, $d_0 \in D$ is the initial state, $\delta\colon D \times A \to D$ is the transition function and $F \subseteq D$ is the set of final states of $\mathscr{A}$.

A is the input alphabet of $\mathscr{P}'$ and the stack alphabet is $Z' = D \times Z$. Consider a map $\kappa\colon D \times Z^* \to (D \times Z)^*$ defined as follows

$$\begin{aligned} \kappa(d, \varepsilon) &= \varepsilon, && \text{for all } d \in D, \\ \kappa(d, zg) &= \langle d, z\rangle \kappa(\delta(d, z), g), && \text{for all } \langle d, z\rangle \in D \times Z \text{ and } g \in Z^*. \end{aligned}$$

Let $\iota' = (d_0, \kappa(d_0, h_0))$. On the whole $\mathscr{P}' = (Q, Z', T')$ where

$$T' = \{(a, q, \langle d, z\rangle, \kappa(d, h), q') \mid (a, q, z, h, q') \in T,\ d \in D,\ \delta(d, zq) \in F, \\ \delta(d, hq') \in F\}\ .$$

Observe that an edge $(q, gz) \xrightarrow{a} (q', gh)$ is in the restriction of $\mathcal{G}(\mathscr{P}, \iota)$ to $\{(q, h) \mid hq \in R\}$ rooted on ι if and only if the vertex (q, gz) is in this restriction and

$$\begin{aligned} &\kappa(d_0, gz) = H\langle d, z\rangle \quad \text{for some } H \in (Z')^* \text{ and some } d \in D, \\ &\delta(d, zq) \in F,\ \delta(d, hq') \in F \text{ and} \\ &(a, q, z, h, q') \in T\ . \end{aligned}$$

Hence equivalently, there is an edge $(q, \kappa(d_0, g)\langle d, z\rangle) \xrightarrow{a} (q', \kappa(d_0, g)\kappa(d, h))$ in $\mathcal{G}(\mathscr{P}', \iota')$. Since $\iota' = (d_0, \kappa(d_0, h_0))$, the result follows by induction from the above. □

The converse of Proposition 5.1 is stated in the following.

Proposition 5.3. *Given any unitary reduction-bounded rational ots $\langle \mathcal{S}, R, u\rangle$ over A, one may construct a realtime pdm $\mathscr{P}$ and an internal configuration ι of $\mathscr{P}$ such that the graphs $\mathcal{G}(\mathcal{S}, R, u)$ and $\mathcal{G}(\mathscr{P}, \iota)$ are isomorphic.*

Proof. A pushdown machine $\mathscr{P}' = (Q', Z', T')$ is defined first together with an internal configuration ι' so that $\mathcal{G}(\mathscr{P}', \iota')$ is isomorphic to $\mathcal{G}(\mathcal{S}, \mathcal{I}\mathrm{rr}(\mathcal{S}), u)$. Set $m = \max_{l \in \mathcal{D}\mathrm{om}(\mathcal{S})} |l|$. Define a pdm $\mathscr{P}'$ as follows. The set Q of the states is indexed by irreducible words, the length of which is strictly less than m viz $Q = \{q_w \mid w \in \mathcal{I}\mathrm{rr}(\mathcal{S}) \text{ and } |w| < m\}$. The stack alphabet $Z' = Z'' \cup \{z_0\}$ has the bottom symbol $z_0 \notin Z''$ and Z'' is an arbitrary set that is in one to one correspondence f with the set of irreducible words of length m,

$$f\colon \{w \in \mathcal{I}\mathrm{rr}(\mathcal{S}) \mid |w| = m\} \to Z''\ .$$

The set T' of transition rules of $\mathscr{P}'$ is constructed as follows.

- For any $a \in A$, any $q_w \in Q$ and any $z \in Z''$, one has
  1. $(a, q_w, z, z, q_{wa}) \in T$ when $f^{-1}(z)wa \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and $|wa| < m$,
  2. $(a, q_w, z, zf(wa), q_\varepsilon) \in T$ when $f^{-1}(z)wa \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and $|wa| = m$,
  3. $(a, q_w, z, \varepsilon, q_{vr}) \in T$ when $f^{-1}(z)wa = vl$ for some $v \in \mathcal{I}\mathrm{rr}(\mathcal{S})$ and $(l, r) \in \mathcal{S}$ such that $|vr| < m$,

4. $(a, q_w, z, f(x_1)\dots f(x_n), q_y) \in T$ when $f^{-1}(z)wa = vl$ for some $v \in \mathfrak{Irr}(\mathcal{S})$ and $(l,r) \in \mathcal{S}$ such that $vr = x_1 \dots x_n y$, where $x_1, \dots, x_n, y \in \mathfrak{Irr}(\mathcal{S})$ are such that $|x_1| = \cdots = |x_n| = m$ and $|y| < m$.

– For any $a \in A$ and any $q_w \in Q$, one has
1. $(a, q_w, z_0, z_0, q_{wa}) \in T$ when $wa \in \mathfrak{Irr}(\mathcal{S})$ and $|wa| < m$,
2. $(a, q_w, z_0, z_0 f(wa), q_\varepsilon) \in T$ when $wa \in \mathfrak{Irr}(\mathcal{S})$ and $|wa| = m$,
3. $(a, q_w, z_0, z_0, q_{vr}) \in T$ when $wa = vl$ for some $v \in \mathfrak{Irr}(\mathcal{S})$ and $(l,r) \in \mathcal{S}$ such that $|vr| < m$,
4. $(a, q_w, z_0, z_0 f(x_1)\dots f(x_n), q_y) \in T$ when $wa = vl$ for some $v \in \mathfrak{Irr}(\mathcal{S})$ and $(l,r) \in \mathcal{S}$ such that $vr = x_1 \dots x_n y$, where $x_1, \dots, x_n, y \in \mathfrak{Irr}(\mathcal{S})$ are such that $|x_1| = \cdots = |x_n| = m$ and $|y| < m$.

Define now the internal configuration ι' of $\mathscr{P}'$ corresponding to the root of the graph $\mathcal{G}(\mathcal{S}, \mathfrak{Irr}(\mathcal{S}), u)$ as follows. If $|u| < m$, set $\iota' = (q_u, z_0)$. Otherwise one has $u = x_1 \dots x_n y$ for some $x_1, \dots, x_n, y \in \mathfrak{Irr}(\mathcal{S})$ such that $|x_1| = \cdots = |x_n| = m$ and $|y| < m$. Set then $\iota' = (q_y, z_0 f(x_1) \dots f(x_n))$.

It is easy to check that the one to one mapping $(q_w, z_0 h) \mapsto f^{-1}(h)w$ of the vertices of $\mathcal{G}(\mathscr{P}', \iota')$ onto the vertices of $\mathcal{G}(\mathcal{S}, \mathfrak{Irr}(\mathcal{S}), u)$ extends to a graph isomorphism. Moreover $\mathscr{P}'$ is realtime.

Obviously, $\mathcal{G}(\mathcal{S}, R, u)$ is a restriction (on vertices) of $\mathcal{G}(\mathcal{S}, \mathfrak{Irr}(\mathcal{S}), u)$ to R rooted on u. Define $\mathcal{C}_{R'} = \{(q_w, z_0 h) \mid h \in (Z'')^*,\ q_w \in Q',\ f(h)w \in R\}$ and $R' = \{z_0 h q_w \mid (q_w, z_0 h) \in \mathcal{C}_{R'}\}$. Observe that R' is rational. Moreover, the restriction of $\mathcal{G}(\mathscr{P}', \iota')$ to $\mathcal{C}_{R'}$ rooted on ι' is isomorphic to $\mathcal{G}(\mathcal{S}, R, u)$. Now, according Lemma 5.2, one may construct a pdm $\mathscr{P}$ and an internal configuration ι of $\mathscr{P}$ such that the graph $\mathcal{G}(\mathscr{P}, \iota)$ is isomorphic to $\mathcal{G}(\mathcal{S}, R, u)$. □

In view of the results established so far, it is straightforward to conclude this section as follows. Both strongly reduction–bounded and unitary reduction–bounded rational Thue specifications characterize the class context–free graphs.

6 Decision Problems

The criterion of strong reduction–boundedness defines a class of Thue specifications, the graphs of which have decidable monadic second–order theory due to the result of Muller and Schupp [9]. It may be asked whether the class of strongly reduction–bounded rational ots is recursive. The answer is positive for the subclass of unitary reduction–bounded rational ots.

Proposition 6.1. *There is an algorithm to solve the following problem.*

Instance: *A finite semi–Thue system $\mathcal{S}$ and a rational subset R of $\mathfrak{Irr}(\mathcal{S})$.*

Question: *Is $\mathcal{S}$ unitary reduction–bounded on R ?*

Proof. Let $\mathcal{S}$ be a finite sts over A. For each rule (r,l) and each $a \in A$ set $R_{(l,r),a} = ((Ra)l^{-1})r$. Observe that

$$\bigcup_{\substack{(l,r)\in\mathcal{S}\\ a\in A}} R_{(l,r),a} = \{v \mid \exists\, u \in R, \exists\, a \in A \text{ s.t. } ua \underset{\mathcal{S}}{\to} v\}\ .$$

Thus, $\mathcal{S}$ is unitary reduction–bounded if and only if $R_{(l,r),a} \subseteq \mathrm{Irr}(\mathcal{S})$ for each $(l,r) \in \mathcal{S}$ and $a \in A$. Since both $\mathcal{S}$ and A are finite, there is a finite number of inclusions to test, all between rational languages. □

It is not surprising that the above result may be extended as follows.

Proposition 6.1bis. *There is an algorithm to solve the following problem.*
Instance: *A finite semi–Thue system $\mathcal{S}$, a rational subset R of $\mathrm{Irr}(\mathcal{S})$ and a positive integer k.*
Question: *Is k a reduction bound of $\mathcal{S}$ on R ?*

Proof. The proof is similar to the one of Proposition 6.1. One has to test the inclusion in $\mathrm{Irr}(\mathcal{S})$ of the languages of the form $((\cdots(((Ra)l_1^{-1})r_1)\ldots l_k^{-1})r_k)$ for each sequence $(l_1,r_1)\ldots(l_k,r_k)$ over $\mathcal{S}$ of length k and each $a \in A$. □

As established above, one may decide whether an integer is a reduction bound of a semi–Thue system. However the decision procedure sketched in the proof does not allow, in general, to establish the existence of a reduction bound. The problem, whether a reduction bound exists, may be addressed in the context of the strong boundedness problem for Turing machines.

As defined in [10], a Turing machine $\mathscr{T}$ is *strongly bounded* if there exists an integer k such that, for each finite configuration C, $\mathscr{T}$ halts after at most k steps when starting in configuration C. The *strong boundedness problem* for Turing machines is the following decision problem.

Instance: A single–tape Turing machine $\mathscr{T}$.
Question: Is $\mathscr{T}$ strongly bounded ?

It is an easy exercise to effectively encode an arbitrary deterministic single–tape Turing machine $\mathscr{T}$ into a semi–Thue system $\mathcal{S}$ over an appropriate alphabet A and to define an effective encoding χ of the configurations of $\mathscr{T}$ into words of $\mathrm{Irr}(\mathcal{S})A$ that satisfy the following property.

> Starting from C, $\mathscr{T}$ halts after k steps if and only if any reduction of $\chi(C)$ into an irreducible word is of length k.

Now, the strong boundedness problem is undecidable for 2–symbol single–tape Turing machines (cf. Proposition 14 of [10]). This gives the following undecidability result.

Proposition 6.2. *There exists a rational set R for which the following problem is undecidable.*
Instance: *A finite semi–Thue system $\mathcal{S}$.*
Question: *Is $\mathcal{S}$ strongly reduction–bounded on R ?*

7 Conclusion

Thue specifications and their graphs have been introduced and two classes of Thue specifications have been defined: strongly reduction–bounded and unitary

reduction–bounded ots. It has been established that both unitary and strongly reduction–bounded rational Thue specifications characterize the context–free graphs. Moreover, the membership problem for the class of strongly reduction–bounded rational ots has been shown to be undecidable whereas, for its proper subclass of unitary reduction–bounded rational ots, this problem has been established as being decidable.

An important property of context–free graphs is the decidability of their monadic second–order theory. However the class of context–free graphs is not the only well–known class of graphs with decidable monadic second–order theory. More general classes of such graphs are described in e.g. [1,4,6,7]. How Thue specifications are linked via their graphs to these classes, is considered for further investigations.

References

1. K. Barthelmann. On equational simple graphs. Technical Report 9/97, Johannes Gutenberg Universität, Mainz, 1997.
2. R. V. Book and F. Otto. *String-Rewriting Systems*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
3. D. Caucal. On the regular structure of prefix rewriting. *Theoretical Comput. Sci.*, 106:61–86, 1992.
4. D. Caucal. On infinite transition graphs having a decidable monadic second-order theory. In F. M. auf der Heide and B. Monien, editors, *23th International Colloquium on Automata Languages and Programming*, LNCS 1099, pages 194–205, 1996.
5. B. Courcelle. The monadic second–order logic of graphs, II: Infinite graphs of bounded width. *Mathematical System Theory*, 21:187–221, 1989.
6. B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 193–242. Elsevier, 1990.
7. B. Courcelle. The monadic second-order theory of graphs IX: Machines and their behaviours. *Theoretical Comput. Sci.*, 151:125–162, 1995.
8. J. Engelfriet. Contex–free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*, pages 125–213. Springer-Verlag, 1997.
9. D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata and second-order logic. *Theoretical Comput. Sci.*, 37:51–75, 1985.
10. F. Otto. On the property of preserving regularity for string-rewriting systems. In H. Comon, editor, *Rewriting Techniques and Applications*, LNCS 1232, pages 83–97, 1987.
11. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1. Springer-Verlag, 1997.
12. G. Sénizergues. Formal languages and word rewriting. In *Term Rewriting: French Spring School of Theoretical Computer Science*, LNCS 909, pages 75–94, Font-Romeu, May 1993.
13. C. C. Squier, F. Otto, and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoretical Comput. Sci.*, 131(2):271–294, 1994.
14. A. Thue. Probleme über veränderungen von zeichenreihen nach gegebenen regeln. *Skr. Vid. Kristiania, I Mat. Natuv. Klasse*, 10:34 pp, 1914.

Reachability is Decidable for Ground AC Rewrite Systems

Richard Mayr
Institut für Informatik
Technische Universität München
Arcisstr. 21, D-80290 München
Germany
mayrri@informatik.tu-muenchen.de

Michaël Rusinowitch
LORIA and INRIA-Lorraine
615, rue du Jardin Botanique, BP 101
54602 Villers-Lès-Nancy CEDEX
France
rusi@loria.fr

Abstract

The reachability problem for ground associative-commutative (AC) rewrite systems is decidable. We show that ground AC rewrite systems are equivalent to *Process Rewrite Systems (PRS)* for which reachability is decidable [4]. However, the decidability proofs for PRS are cumbersome and thus we present a simpler and more readable proof in the framework of ground AC rewrite systems. Moreover, we show decidability of reachability of states with certain properties and decidability of the boundedness problem.

1 Introduction

Ground AC systems are obtained by combining ground rewrite systems (i.e. rewrite systems without substitution) with the associative and commutative laws for a symbol denoted by $+$. These laws are applied as *symmetric* rewrite rules, i.e. from right to left and vice-versa.

The reachability problem for ground AC systems has already been considered by A. Deruyver and R. Gilleron in [2] who proved that it is decidable under some restrictions: the $+$ symbols can occur only at the root of any side of a rule; if $+$ occurs at the root of the right-hand side then it has to occur at the root of the left-hand side too. This last condition forbids introducing new $+$ symbols in a term.

We show that ground AC rewrite systems are equivalent to *Process Rewrite Systems (PRS)* that were introduced by R. Mayr [4] as an extension of Petri nets with subroutines. This observation was surprising at first, since the two formalisms have different origins. Ground AC systems were defined as an extension of ground rewrite systems by associative and commutative operators like '+' or '$*$'. PRS were introduced as a process model that extends Petri nets by subroutines.

It has been shown in [4] that reachability for PRS is decidable. Here, by following a similar technique, we present a simpler and more readable proof of this result in the framework of ground AC rewrite systems. Previous known decidability results about ground AC systems (with several AC symbols) concern the word problem (symmetric reachability) [5] and the unifiability problem [6]. The latter was also obtained by splitting the rewrite systems in order to get *homogeneous* subsystems.

We also consider the problem if there is a reachable state that satisfies certain properties that are described by state formulae. We show that this "reachable property problem" is decidable for ground AC systems. Finally, we show that the problems of termination and boundedness are decidable, too.

2 Ground AC Systems vs. PRS

Ground AC rewrite systems are defined as ground rewrite systems (i.e. rewrite systems without substitution) that operate on the set of terms $T(F)$ over a signature F which contains an associative and commutative operator $+$. Note that F can contain function symbols with any arity, like $f(x_1, \ldots, x_k)$. For a detailed survey on term rewriting the reader may consult [1].

Process Rewrite Systems (PRS) [4] are defined as **prefix**-rewrite systems that operate on process terms. These process terms are built from atomic processes by the operators '.' for sequential- and '$\|$' for parallel composition. The operator for sequential composition is associative and the operator for parallel composition is associative and commutative.

Now we show that the two formalisms are equivalent. For both PRS and ground AC systems it is possible to construct equivalent systems where each rule contains only one operator. (These new systems can simulate the behavior of the original systems modulo a uniformly bounded number of silent actions.) We show the equivalence only for systems where rules contain only one operator. Monadic ground AC systems are ground AC systems where all function symbols have arity one. In a first step we show that PRS and monadic ground AC systems are equivalent.

From PRS to monadic ground AC: For each symbol X in the PRS we introduce a 1-ary function symbol $X(\ldots)$. We use a special symbol ϵ for the empty term. For every PRS-rule there is a corresponding ground AC rule:

PRS	**Ground AC**
$X.Y \to Z$	$Y(X(\epsilon)) \to Z(\epsilon)$
$X \to Y.Z$	$X(\epsilon) \to Z(Y(\epsilon))$
$X \to Y$	$X(\epsilon) \to Y(\epsilon)$
$X \to Y \| Z$	$X(\epsilon) \to Y(\epsilon) + Z(\epsilon)$
$X \| Y \to Z$	$X(\epsilon) + Y(\epsilon) \to Z(\epsilon)$

From ground AC to PRS: For every function or constant symbol f in the ground AC system we use a symbol f in the PRS. For every ground AC-rule there is a corresponding PRS rule:

Ground AC	**PRS**
$f(c) \to d$	$c.f \to d$
$c \to f(d)$	$c \to d.f$
$c \to d$	$c \to d$
$c \to d + e$	$c \to d \| e$
$c + d \to e$	$c \| d \to e$

This shows that PRS and monadic ground AC rewrite systems are equivalent. In a second step we show that monadic ground AC and general ground AC are equivalent. One direction is trivial. Now we show how to encode a general ground AC system in a monadic ground AC system. Let f be a k-ary function symbol. Then we introduce k new 1-ary function symbols $f_1, \ldots, f_k$. In all rewrite rules we replace every term of the form $f(c_1, \ldots, c_k)$ by

$$f(f_1(c_1) + \ldots + f_k(c_k))$$

This new system is monadic, by contains more than one operator in a rule. This can be transformed into an equivalent one (modulo silent actions) with only one operator in each rule.

Thus we have that Ground AC = monadic ground AC = PRS. In PRS the rewrite rules are labeled with atomic actions, while this is not customarily done in ground AC systems. However, it can done in ground AC systems just as well.

3 Reachability for Ground AC Systems

We consider finite sets of ground rewrite rules Δ. For terms t_1, t_2 we define that $t_1 \succ^{\Delta} t_2$ iff t_1 can be transformed into t_2 by applying the rules in Δ. The reachability problem is to decide if $t_1 \succ^{\Delta} t_2$ for given t_1, t_2 and Δ.

3.1 Splitting the Rules

Let Δ be a finite set of rules on F. A term or a rule is *homogeneous* if it contains at most one occurrence of a non-constant operator.

Lemma 1 *For every finite ground AC system Δ a finite set of homogeneous rules Δ' can be effectively constructed such that for all terms $t_1, t_2 \in T(F)$, $t_1 \succ^{\Delta} t_2$ iff $t_1 \succ^{\Delta'} t_2$*

Proof We derive Δ' by applying to Δ the following abstraction rules as much as possible:

$$\text{Abstract 1:} \quad a \to b \vdash a_{|p} \to g, a[g]_p \to b$$

where $a_{|p}$ is a maximal homogeneous non-constant strict subterm of the term a at a position p and g is a new constant.

$$\text{Abstract 2:} \quad a \to b \vdash a \to b[g]_p, g \to b_{|p}$$

where $b_{|p}$ is a maximal homogeneous non-constant strict subterm of b at a position p and g is a new constant.

$$\text{Abstract 3:} \quad a \to b \vdash a \to g,\ g \to b$$

where a, b are homogeneous, non-constants, with different root symbols and g is a new constant.

The application of these transformations terminates since: Abstract 3 decreases the number of rules with different non-constant operators at the root of each side; every application of Abstract 1 or Abstract 2 decrease the multiset of depths of the rule sides and do not create rules that can be transformed by Abstract 3. Every rewrite step by Δ can be simulated by rewrite steps with Δ'. The other direction follows from the fact that the symbols g are new constants. ■

3.2 Deciding Reachability by Completion

In this section rules of the form $c \to d$ where c and d are constants will be called *simple rules*. The following completion algorithm adds simple rules $c \to d$ to a set of rules R, whenever $c \succ^R d$.

Lemma 2 *Let R_1 and R_2 be two homogeneous rewrite systems s.t. the sets of operators used in the rules of R_1 and R_2 are disjoint and R_1 and R_2 contain the same simple rules. Let $R = R_1 \cup R_2$. If there is a pair of constants (u, v) s.t. $u \succ^R v$ and $(u \to v) \notin R$ then there is a pair of constants (u', v') s.t. $(u' \to v') \notin R$, but $u' \succ^{R_1} v'$ or $u' \succ^{R_2} v'$.*

Proof Choose a pair of constants u', v' s.t. $(u' \to v') \notin R$, but $u' \xrightarrow{\sigma} v'$ where σ is a sequence of rule applications of minimal length. More precisely the length of σ is minimal over the choice of u', v' and σ. By our preconditions this must exist.

We will assume that σ must contain non-simple rules from both R_1 and R_2 and derive a contradiction. W.r. we assume that the first non-simple rule in σ is from R_1. By our assumption there must be a non-simple rule from R_2 in σ. The first such rule must have the form $u'' \to t$, where u'' is a constant and t is not a constant, because the sets of operators in R_1 and R_2 are disjoint. The operator in t must disappear later, because the sequence σ ends with the constant v. This can only be done if the term t is rewritten to a constant later in the sequence σ. Thus there must a constant v'' and a subsequence σ' of σ s.t. $u'' \xrightarrow{\sigma'} v''$. This is a contradiction to the minimality of the length of σ.

Thus we have that the non-simple rules in σ are either all from R_1 or all from R_2. Since R_1 and R_2 contain the same simple rules we can assume that σ contains only rules from R_1 or only rules from R_2. Thus we have $u' \succ^{R_1} v'$ or $u' \succ^{R_2} v'$. ■

Lemma 3 *Let R_1 and R_2 be two homogeneous rewrite systems s.t.*

1. *The sets of operators used in the rules of R_1 and R_2 are disjoint.*
2. *The relations $\succ^{R_1}$ and $\succ^{R_2}$ are decidable.*
3. *If one adds rules of the form $u \to v$ to R_1 or R_2, where u and v are constants, then the new relations $\succ^{R_1'}$ and $\succ^{R_2'}$ are also decidable.*

Let $R = R_1 \cup R_2$. Given two constants x, y, it is decidable whether $x \succ^R y$.

Proof To prove the lemma it is sufficient to give an algorithm for generating all couples of constants u, v such that $u \succ^R v$, and check if (x, y) is generated. First we add all rules in R_1 of the form $c \to d$, where c and d are constants to R_2 and vice versa. By the third precondition this does not change the decidability. By precondition 1 we can apply Lemma 2. Thus if there is a

pair of constants (u, v) which satisfies $u \succ^R v$, but is not in R_1 or R_2, then there is another pair (u', v') that is also not in R_1 or R_2, but satisfies $u' \succ^{R_1} v'$ or $u' \succ^{R_2} v'$. By precondition 2 this pair can be found. Then we add this pair $u' \to v'$ to R_1 and R_2 and get R_1' and R_2'. By precondition 3 this does not change the decidability. We repeat this procedure with the new R_1' and R_2' until no new pair can be added. It terminates, because there are only finitely many different constants in R. By Lemma 2 we have then added all pairs (u, v) s.t. $u \succ^R v$. Let $\tilde{R}_1$ and $\tilde{R}_2$ be the final results of this process. It then holds that $x \succ^R y \iff (x \to y) \in \tilde{R}_1 \cup \tilde{R}_2$. ■

Theorem 1 *The reachability problem is decidable for ground rewrite systems with an associative-commutative operator.*

Proof Consider an instance of the reachability problem for a set of rules Δ and terms t_0, t. The question is if $t_0 \succ^\Delta t$. We introduce two new constants x, y and define $\Delta' := \Delta \cup \{x \to t_0, t \to y\}$. Thus we have $t_0 \succ^\Delta t \iff x \succ^{\Delta'} y$. By Lemma 1 a system of homogeneous rules Δ'' can be constructed s.t. $x \succ^{\Delta'} y \iff x \succ^{\Delta''} y$. Δ'' can be partitioned into R_1 and R_2 s.t. R_1 are all the rules that contain the commutative operator + and R_2 is the rest. Thus the first precondition of Lemma 3 is satisfied. The relation $\succ^{R_1}$ corresponds to Petri net reachability, which is decidable [3]. The relation $\succ^{R_2}$ is the reachability relation for a normal ground rewrite system and is also decidable [2]. Adding simple rules to R_1 and R_2 does not change this. Thus the other preconditions of Lemma 3 are satisfied. Thus it can be used to decide $x \succ^{\Delta''} y$, which is equivalent to $t_0 \succ^\Delta t$. ■

Remark: Note that the algorithm of Lemma 3 uses only polynomially many instances of Petri net reachability, each of which is smaller than the input. The same techniques can be applied to show the decidability of the reachability problem for ground rewrite systems with arbitrarily many different associative and commutative operators.

4 The Reachable Property Problem

Here we use the notations from [4], Section 6. The rewrite rules in Δ are assigned labels, which can be interpreted as atomic actions. These labels then form atomic propositions in state-formulae. For example let a_i be a label. The term t satisfies the state-formula a_i (denoted $t \models a_i$) if a rule with label a_i can be applied to t. A general state-formula is a boolean combination of atomic propositions. We give a method that decides the problem $t_0 \models \Diamond\Phi$,

i.e. the problem if there is a reachable state that satisfies the state formula Φ. It suffices to consider the case where Φ is $a_1 \wedge \ldots \wedge a_k \wedge \neg b_1 \wedge \ldots \wedge \neg b_l$. We will abbreviate $\neg b_1 \wedge \ldots \wedge \neg b_l$ by B, $a_1 \wedge \ldots \wedge a_k$ by A and $\bigwedge_{i \in I} a_i$ by A_I. ($A_\emptyset = true$). We also define $K = \{1, \ldots, k\}$.

We solve the problem for systems Δ in transitive normal form. To simplify the problem we assume that rules of the form $f(c_1, \ldots, c_k) \rightarrow d$ do not carry any label. This is no restriction since every system can be transformed to satisfy this condition in the following way: For every term $f(c_1, \ldots, c_k)$ that occurs on the left side of a rule determine the set A of actions that are enabled by $f(c_1, \ldots, c_k)$. Then add a new constant and e and the rule $f(c_1, \ldots, c_k) \rightarrow e$. Then for every action $a \in A$ add a rule $e \stackrel{a}{\rightarrow} e$. Then transform the system into transitive normal form. This new system is equivalent to the old one as far as the reachable property problem is concerned.

For simplicity we also assume that there are only two non-constant symbols: + and . where + is AC. (The generalization to arbitrary symbols is straightforward.) We can assume w.r. that the initial state t_0 is a constant t_0, since otherwise we just add another rule $c_0 \rightarrow t_0$. Δ_{par} (resp. Δ_{seq}) contains all rules without '.' (resp. '+'). The rules in Δ_{par} are called *par-rules* and the rules in Δ_{seq} are called *seq-rules*. Let $C = \{c_1, \ldots, c_m\}$ be the set of constants in the system. A *monomial* is a sum of constants. We denote by f_Ψ the formula of L_N stating that a monomial satisfies a formula Ψ. For instance f_B is true for monomials that are not reducible by rules with a label in $\{b_1, \ldots, b_l\}$. Given a subset C' of C, we denote by $h_{C'}$ the formula of L_N stating that the set of constants in a monomial is C'. For instance $h_{\{c_1, c_1, c_3\}}(c_1 + c_1 + c_3 + c_3)$ evaluates to true. Terms will always be flattened using associativity of +. Hence $(a + b) + c = a + b + c$ and the depth of this term is 1.

The following result is a direct consequence of a result for Petri nets due to P. Jančar [8]:

Lemma 4 *It is decidable whether $c_0 \models \Diamond\Phi$, if Δ only contains par-rules.*

If σ is one of the shortest sequences such that $t_0 \stackrel{\sigma}{\rightarrow} t$ and $t \models \Phi$ there is no subderivation of $t_0 \succ^\Delta t$ such that $c \succ^\Delta t' \succ^\Delta c'$. Otherwise by replacing this subderivation with $c \rightarrow c'$ (since Δ is in transitive normal form) we can obtain a shorter string σ' such that $t_0 \stackrel{\sigma'}{\rightarrow} t$. Hence we can assume that by commuting rules applications we can build another string η with the same properties as σ which additionally has a special structure: $\eta = \eta_0 \eta_1 \ldots \eta_d$ where all rules applications corresponding to η_i are applied at the same level i of terms.

4.1 A special case

In this section we solve the problem for the special case of $k = 0$. We compute the subset C_N of all $c \in C$ such that $c \models \Diamond B$. If no rule in B applies to c then obviously $c \in C_N$. Otherwise let μ be a shortest derivation from c to a t such that $t \models B$.

Using the same decomposition as above $\mu_0\mu_1$ is the maximal prefix of μ such that $c \stackrel{\mu_0\mu_1}{\rightarrow} v$ and all intermediate terms between c and v (including v) are of level ≤ 1.

i. Let us assume first that these terms are (non trivial) sums of constants. Then $v = U + U'$ where U, U' are possibly empty sums and:

1. $U \models B$
2. for all $c' \in U'$ we have $c' \succ^{\Delta} t_{c'}$ and $t_{c'} \models B$

By minimality of the derivation we can assume that $c \notin U'$.

ii. Now if the root symbol of v is '.' then either v is not reducible by a rule from B or one of its leaves is in C_N and is different from c by minimality. (recall that all rules have depth 1).

More generally by minimality of μ it will never contain a subderivation $c' \succ^{\Delta} s[c'] \succ^{\Delta} s[t']$. Otherwise pumping allows to find a shorter one. This shows also that the depth of t is bounded by $m = |C|$.

Hence a simple recursive algorithm solves the problem. Let $Nreach(c, n)$ be true iff c rewrites to a term of depth n not reducible by rules from B. $Nreach(c, 0)$ is easy to check, by Lemma 4, since Δ is in transitive normal form. We also introduce the auxiliary procedure $Ncheck(c, C')$. Given $c \in C$ and $C' \subseteq C$, $Ncheck$ tests whether there exists a monomial v such that $c \succ^{\Delta_{par}} v$, $v = U + U'$ and $f_B(U) \wedge h_{C'}(U')$ is true. Note that this test is effective thanks to the result of Jančar [8].

```
1  Nreach(c, n)
2      for C' ⊆ C do
3          if Ncheck(c, C') then
4                  if ⋀_{c'∈C'} Nreach(c', n − 1) then true; exit
5      for a, b ∈ C do
6          if c → a.b ∧ (Nreach(a, n − 1) ∧ Nreach(b, n − 1)) then true; exit
7      Nreach(c, n) = false
```

By computing $\{c \in C; Nreach(c, m) = \mathsf{true}\}$ we get the set C_N.

4.2 The general case

Now we study the general case. Let us analyze the structure of the term t_1 such that $c_0 \stackrel{\eta_0\eta_1}{\rightarrow} t_1$. Assume that the root symbol of t_1 is $+$ (the other case is simple). Hence the derivation contains only rules applications from Δ_{par}. We shall sort the constants in t_1 according to the property $\Diamond(A_P \wedge B)$ they satisfy, P ranging over the subsets of $K = \{1, \ldots, k\}$. Then $t_1 = U + \Sigma_{P \in \mathcal{P}(K)} U'_P$ where U, U'_P are possibly empty sums of constants and:

1. $U \models A_I \wedge B$
2. for all $c \in U'_P$ we have $c \succ^{\Delta} t_c$ and $t_c \models A_P \wedge B$
3. $I \cup \{P \in \mathcal{P}(K); U'_P \text{ is not empty}\} = K$

The pumping argument is now slightly more complex than in the base case: a subderivation starting from a constant c'': $c'' \succ^{\Delta} s[c''] \succ^{\Delta} s[t'']$ (where $s[t'']$ is not reduced anymore in μ) can be shortened if t'' and $s[t'']$ satisfy the same A_I's. Hence the length of any branch of t is bounded by $(k+1)m$.

Let $Reach(c, n, J)$ be true iff c rewrites to a term of depth n satisfying $A_J \wedge B$. We have that $Reach(c, 0, J)$ is decidable by Lemma 4, since Δ is in transitive normal form. Note that $Reach(c, n, \emptyset) = Nreach(c, n)$.

For the algorithm that decides $Reach(c, n, J)$ we need an auxiliary function $Check$. Let $c \in C$ be a constant, $j : 2^K \longmapsto 2^C$ a mapping and $I \subseteq K$. Then $Check(c, j, I)$ is true iff there exists a v such that: $c \succ^{\Delta_{par}} v$ and

$$(v = U + \Sigma_{P \in 2^K} U'_P) \wedge f_{I \wedge B}(U) \wedge (\bigwedge_{P \in 2^K} h_{j(P)}(U'_P))$$

Check is effective by the result of Jančar [8]. The algorithm for *Reach* is as follows:

```
1   Reach(c, n, K)
2       for every mapping j : 2^K ⟼ 2^C and every I ⊆ K
3         if I ∪ {P ∈ P(K); j(P) is not empty} = K then
4           if Check(c, j, I) then
5             if ⋀_{P∈P(K)} ( ⋀_{c'∈j(P)} Reach(c', n − 1, P)) then true; exit
6       for a, b ∈ C do
7         if c → a.b then
```

8 for everyK_1, K_2 s.t. $K_1 \cup K_2 = K$

9 if $(Reach(a, n-1, K_1) \wedge Reach(b, n-1, K_2))$ then true; exit

10 $Reach(c, n, K) =$ false

Now the *Reachable property* problem can be tested by computing

$$Reach(c_0, (k+1)m, K)$$

5 Termination Problems

In this section we investigate termination properties of ground AC systems. We consider the case where the systems are in transitive normal form.

Lemma 5 *It is decidable whether there exists an infinite derivation from a constant c.*

Proof For every term t we define $depth(t)$ as the maximal nesting-depth of function symbols in t. For example $depth(a + f(b + c + g(a, b))) = 2$.
Assume that there is an infinite run that starts at c s.t. in this run a constant c' is reduced to a larger non-constant term t and t is later again contracted to another constant c''. Since we consider systems in transitive normal form we have the rule $c' \to c''$ and can do this in one step. Thus if there is any infinite run then there is also an infinite run in which the depth of the terms never decreases. In other words, no seq-rules of the form $f(c_1, \ldots, c_k) \to d$ are used, but only seq-rules of the form $c \to f(d_1, \ldots, d_k)$.
Hence only the following situations can occur:

1. In the infinite sequence the depth of the terms is bounded. Then there must be a constant c' that occurs in some term in this sequence s.t. there is an infinite derivation from c' using only $\succ^{\Delta_{par}}$.

2. If the depth of the terms is not bounded then after some steps by the pigeon hole principle there is a long internal path in some term such that two identical symbols on it have been generated by the same rule $c' \to t$ where c' is a constant.

Hence there is an infinite derivation from c iff c rewrites to a term containing a constant c' such that

1. either there is an infinite derivation from c' using par-rules only.

2. c' rewrites to a term containing c'.

Case 1. amounts to check the existence of an infinite path for a Petri net, which can be done by constructing the coverability tree.
Case 2. can be decided using the results of the previous section. For any constant c' consider the system $\Delta_d = \Delta \cup \{c' \xrightarrow{\tau} c'\}$. Let $t_1, \ldots, t_k$ be the terms that can be reached from c' in one step. Checking 2. amounts to checking whether $t_k \models \Diamond\tau$ for some t_k. Once we have collected the set of constants c' for which the test is positive, we can test whether c can reach a term containing one of them using the same technique.

Thus the existence of an infinite run from some constant c is decidable. To decide this for a general term t we use a new constant c, add a rule $c \to t$ to the system, transform this new system into transitive normal form and check the existence of an infinite run from c.

A system terminates if there is no infinite run. We get the following theorem.

Theorem 2 *The termination property is decidable for ground AC systems.*

In a similar way we can decide the boundedness problem. There is a run from a constant c where the sizes of the terms are unbounded if from c one can reach a term containing c' s.t.

1. either there is an unbounded derivation from c' using par-rules only. This can be decided by the coverability graph.

2. c' rewrites to a term t containing c', but $t \neq c'$. This can be checked in the following way. Every rule whose right side contains c', but is not equal to c' is labeled with the action τ. Then it suffices to check whether $c' \models \Diamond\tau$.

Theorem 3 *Boundedness is decidable for ground AC systems.*

References

[1] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1990.

[2] A. Deruyver and R. Gilleron. The Reachability Problem for Ground TRS and some Extensions, In Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 1. Volume 351 of LNCS, Springer Verlag, pp 227–243, 1989.

[3] E. W. Mayr. An algorithm for the general Petri net reachability problem. SIAM Journal of Computing, 13:441-460,1984.

[4] R. Mayr. Process Rewrite Systems. Electronic Notes in Theoretical Computer Science, volume 7. Proceedings of the Workshop "Expressiveness in Concurrency" (EXPRESS'97), 1997.

[5] P. Narendran and M. Rusinowitch. Any ground associative-commutative theory has a finite canonical system. Proceedings of RTA, LNCS 488, Springer-Verlag, 1991.

[6] P. Narendran and M. Rusinowitch. The unifiability problem in ground AC Theories. IEEE Symposium on Logic in Computer Science. Montreal (Canada), 1993.

[7] M. Oyamaguchi The Church-Rosser Property for Ground Term Rewriting Systems is Decidable", 1987, Theoretical Computer Science, Volume 49(1), pp 43–79.

[8] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Comuter Science*, 74:71–93, 1990.

Proving the Bounded Retransmission Protocol in the π-calculus

Thérèse Hardin and Brahim Mammass

Laboratoire d'Informatique de Paris 6
Tour 45-55, 4 Place Jussieu 75252 Paris Cedex 05, France
e-mail: Therese.Hardin@lip6.fr, Brahim.Mammass@lip6.fr

Abstract. The aim of this paper is twofold. We first present a correctness proof of the Bounded Retransmission Protocol (BRP) done quite straightforwardly by bisimulation in the π-calculus. To our knowledge, it is the biggest proof realized in this framework. Then, we compare in depth several works on this protocol, focusing on how the used formalism influences implementation choices and proof techniques.

1 Introduction

The development of communication networks requires more and more sophisticated communication protocols which must be reliable [13]. Traditional verification methods use model checking techniques, but they cannot deal with infinite state systems and more generally with mobility.

Our aim is to elaborate some methodologic guides for designing and proving communication protocols using theorem provers. We choose the BRP as a case study because it is simple but, since it is parameterized, model checking cannot be directly applied. We choose the π-calculus [24] as a formal framework. It is an extension of the process algebra CCS [22] with mobility while keeping its algebraic properties. It is more expressive than CCS because it provides possibilities for coding data types, λ-calculus and higher order processes and possibilities for expressing mobility between processes by means of name passing.

So, on one hand, we present a proof of the BRP using π-calculus bisimulations. On the other hand, we study some related works [1, 7, 9, 10, 14, 15] in order to compare different approaches. Essentially, in [1, 7, 10, 15] the BRP is designed and proved using a top-down approach, that is the system specification is refined until an implementation is met. In opposition, it is the bottom-up approach which is adopted in [9, 14]: starting from the system implementation, a system abstraction is deduced. Moreover, the used formalism in [7] provides explicit time whereas in [1, 9, 10, 14, 15] it does not. The comparison between these works focuses on how the used formalism influences implementation choices and proof techniques.

The paper is organized as follows: section 2 presents the informal description of the BRP. Before formalizing this description, we complete it by making some choices. Section 3 gives an abstract view of the BRP in the π-calculus. Section 4 gives the protocol implementation in the π-calculus. Section 5 presents our correctness proof method which proceeds by bisimulation and its application to the BRP. Finally, in section 6, we present the studied papers and compare them in detail to our work.

2 The Bounded Retransmission Protocol

The BRP [14] is a communication protocol, developed at Philips Research Laboratory, that communicates messages from a producer to a consumer over an unreliable physical medium that can lose messages.

2.1 The usual description

The protocol (figure 1) consists of a sender program at the producer side, a receiver program at the consumer side, and two channels (one-place buffers): a message channel K and an acknowledgment channel L. Both channels are unreliable in that they can lose messages or acknowledgments; but, messages are neither garbled, nor received out of order. Two timers are used.

The sender sends each message over the channel K, sets $timer_1$, and then waits for an acknowledgment over the channel L. The $timer_1$ is used to detect the loss of a message or an acknowledgment. If an acknowledgment

comes back within this time, the timer is cleared, and the next message is sent. If the transmission has been completed, the sender transmits a confirmation OK to the producer to signal a successful transmission. If there is no acknowledgment, a timeout occurs whereupon the message is retransmitted, and the timer set again. There is a fixed upper bound on the number of such retransmissions (MAX). When this retransmission bound has been reached, the sender aborts transmission and confirms that the transmission failed. Either it confirms *Conf*(NOTOK) if the abort occurred during the transmission of an intermediate message or it confirms *Conf*(DTKW) if the last message in the file was not acknowledged but might have been received by the consumer.

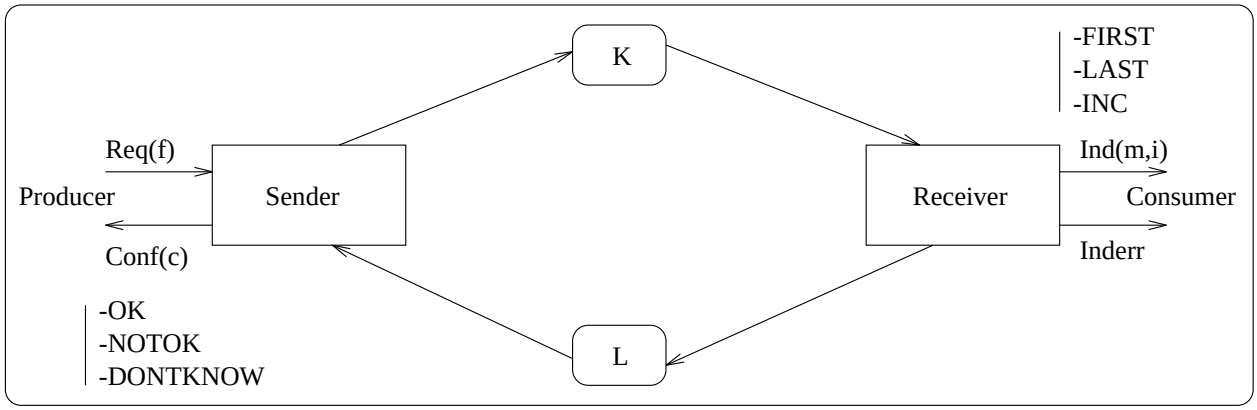

Fig. 1. The BRP protocol

The receiver waits for messages over the channel K. If the alternating bit of the received message is equal to that of the previous message, the receiver retransmits an acknowledgment over the channel L. Otherwise, it delivers the message to the consumer, changes its alternating bit, sets $timer_2$, and sends an acknowledgment over the channel L. In both cases, it waits for the subsequent message. The time associated with $timer_2$ must exceed the required time to transmit MAX times a message (i.e, $timer_2 \succ \text{MAX} \times timer_1$). If $timer_2$ expires, i.e., no new message is received, the receiver sends an *Inderr* signal to the consumer.

2.2 Completing the description

The above description has an important lack of precision. So, before formalizing the protocol, some choices have to be stated. First, we assume that the file may contain zero, one or more messages. But if it contains one message, *this first message must be considered as the last one*. Moreover, if the transfer is aborted during the transmission of this message, *a confirmation* DTKW *must be sent to the producer*.

As $timer_2 \succ \text{MAX} \times timer_1$, the sender is the first to detect a transmission abort. The sender may then receive a new request to transfer a file and send its first message while the receiver does not yet detect the abort (i.e., $timer_2$ has not expired). The receiver may consider this message as the next message (or as a duplication of the current message) of the previous file, that is incorrect. So, it the sender must wait until the receiver detects the abort. Moreover, *both the sender and the receiver must reinitialize their alternating bit before the beginning of the next transfer*. For uniformity, we decide to do this reinitialization also when the transfer has been completed.

After a successful transmission, either the sender receives no new request before expiration of $timer_2$, so the receiver may do a misleading abort. Or it receives a new request and sends the first message. The receiver may consider this first message as a duplication of the last message of the previous file because it cannot know the new alternating bit value. So, *the sender must signal the end of a transfer to the receiver before it begins the next one*. The receiver can then anticipate the expiration of $timer_2$, then *both the sender and the receiver reinitialize their alternating bit*.

Finally, note that an *Inderr* signal is sent by the receiver to the consumer if $timer_2$ expires and the last message in the file is not yet received.

3 The BRP abstract view in the π-calculus

Starting from the descritpion of 2.1, we consider the system as a black box. We define its abstract view as the observable behaviour on the external channels *Req*, *Ind*, *Inderr*, and *Conf*. We formalize this abstract view in the polyadic π-calculus [23] which is our formal framework. Its syntax and informal semantics are recalled below.

3.1 Syntax and informal semantics of the polyadic π-calculus

Let $x, y, z, u, v, \ldots$ range over $\mathcal{N}$, a set of channel names. Let $A, B, \ldots$ range over a set of agent identifiers; each identifier has a nonnegative arity. We note by $\tilde{x}$ the tuple $< x_1, x_2, \ldots, x_n >$. Let $P, Q, \ldots$ range over agents (i.e. processes) which are defined as follows:

- 0, an agent which can do nothing.
- $\overline{y}\tilde{x}.P$, an agent which outputs the tuple $\tilde{x}$ on channel y; thereafter it behaves as P. In this action, y is the *subject*, $\tilde{x}$ is the *object*, and both $\tilde{x}$ and y are *free*.
- $y(\tilde{x}).P$, an agent which receives a tuple on channel y; thereafter it behaves as P but with the newly received names in place of x_i. In this action, y is the *subject*, $\tilde{x}$ is *bound*, and y is free.
- $\tau.P$, an agent which performs the silent action τ; thereafter it behaves as P.
- $P + Q$, an agent which behaves like either P or Q.
- $P \mid Q$, an agent representing the parallel composition of P and Q. This agent can do anything that P or Q can do, and moreover if $P = \overline{y}\tilde{u}.P'$ and $Q = y(\tilde{x}).Q'$, then $P \mid Q \xrightarrow{\tau} .(P' \mid Q'\{\tilde{u}/\tilde{x}\})$ where $Q'\{\tilde{u}/\tilde{x}\}$ is the substitution of each occurrence of x_i by u_i in Q'.
- $(\nu\ x)P$, an agent which behaves like P where the name x is local but P can export x.
- $[x = y]P$, an agent which behaves like P if x and y are the same name; otherwise it does nothing.
- $A(y_1, \ldots, y_n)$ is an agent if A is an identifier of arity n; for any such identifier there is a defining equation written $A(x_1, \ldots, x_n) \stackrel{def}{=} P$, where the names $x_1, \ldots, x_n$ are distinct and are the only names which may occur free in P. The agent $A(y_1, \ldots, y_n)$ behaves like P where y_i is substituted for x_i for all $i = 1, \ldots, n$. Agent identifiers provide recursion since the defining equation of A may contain A itself.

We note $(\nu\ x_1 \ldots x_n)P$ instead of $(\nu\ x_1) \ldots (\nu\ x_n)P$.

3.2 The abstract view

The BRP abstract view is pictured in figure 2 and is expressed by three recursive equations. The file is modeled by a list of messages and we use the usual functions *cons*, *hd* and *tl* on lists.

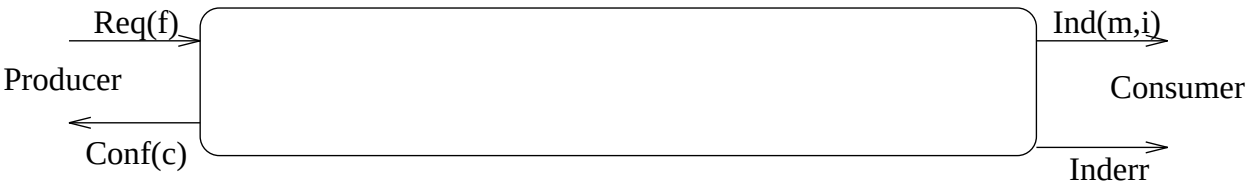

Fig. 2. The BRP abstract view

In the initial state S_0, the system waits for a file f on the channel *Req*. If f is empty ($[f = Nil]$) it confirms the transfer and returns to S_0, else it processes the first message h in the state S_1.

$S_0 \stackrel{def}{=} \mathtt{Req}(f).([f = Nil].\overline{\mathtt{Conf}}\ \mathtt{OK}.S_0 + [f = cons(h,t)].S_1(f)$

In the states S_1 and S_2, the τ actions indicate that the choice between the delivery or loss of a message or an acknowledgment is decided by the internal actions. The first two lines correspond to the case where the message is always lost. The two following lines correspond to the case where the message is received but the acknowledgment is always lost. The last ones correspond to the case of a successful message transmission.

For each of these cases, we distinguish in S_1 two subcases: either the current message is the last one in the file, or it is the first one.

$$\begin{aligned} S_1(f) \stackrel{def}{=}\ & \tau.\overline{\mathtt{Conf}}\ \mathtt{DTKW}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Conf}}\ \mathtt{NOTOK}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{LAST} > .\overline{\mathtt{Conf}}\ \mathtt{DTKW}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{FIRST} > .\overline{\mathtt{Conf}}\ \mathtt{NOTOK}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{LAST} > .\overline{\mathtt{Conf}}\ \mathtt{OK}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{FIRST} > .S_2(tl(f)) \end{aligned}$$

In the state S_2, the system handles the remaining messages of the list. The same cases as in S_1 are analysed but for each of these cases, we distinguish two subcases: either the current message is the last one in the file, or it is an intermediate one.

$$\begin{aligned} S_2(f) \stackrel{def}{=}\ & \tau.\overline{\mathtt{Conf}}\ \mathtt{DTKW}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Conf}}\ \mathtt{NOTOK}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{LAST} > .\overline{\mathtt{Conf}}\ \mathtt{DTKW}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{INC} > .\overline{\mathtt{Conf}}\ \mathtt{NOTOK}.\overline{\mathtt{Inderr}}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{LAST} > .\overline{\mathtt{Conf}}\ \mathtt{OK}.S_0 \\ & + \tau.\overline{\mathtt{Ind}} < hd(f), \mathtt{INC} > .S_2(tl(f)) \end{aligned}$$

4 The BRP implementation in the π-calculus

We start from the complete specification of section 2. To encode the protocol, we need the types integer, boolean and list which are encoded in the π-calculus [24].

We model the external channels *Req*, *Conf*, *Ind*, and *Inderr* as constant names because they are never bound during the execution of the protocol. We model $timer_1$ by the agent $T1$ which repeatedly waits for a signal over the channel *time*1, then sends a signal over the channel *timeout*1. The $timer_2$ is modeled in the same way.

$T1 \stackrel{def}{=} \mathtt{time}1.\overline{\mathtt{timeout}1}.T1$

$T2 \stackrel{def}{=} \mathtt{time}2.\overline{\mathtt{timeout}2}.T2$

We use the channel *abort* (resp. *restart*) to solve the synchronization problems between the sender and the receiver after a transmission abort (resp. after a successful transfer). *These channels are not physical ones* and should be implemented by means of timers. Introducing these two channels allows us to separate cleanly the two situations.

Every message transmitted by the sender S is a tuple $< first, last, tag, data >$. If $first$ (resp. $last$) equals $True$, then the current message is the first (resp. last) one. The variable tag contains the alternating bit, and $data$ is a file data. In the initial state, the sender initializes its variables, waits for a request on the channel *Req*, then starts the file transfer. The variable rn contains the retransmissions number.

$S(\mathtt{K}, \mathtt{L}, \mathtt{abort}, \mathtt{restart}) \stackrel{def}{=} \mathtt{Req}(f).Transfer(\mathtt{K}, \mathtt{L}, \mathtt{abort}, \mathtt{restart}, f, True, False, True, 0)$

If the file is empty, the sender sends a confirmation OK to the producer, makes a rendez-vous over the channel *restart* with the receiver and then returns to its initial state. Otherwise, it transmits the first message, sets $timer_1$, increments rn, and waits for an acknowledgment over the channel L.

$$
\begin{array}{l}
Transfer(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, f, first, last, tag, rn) \stackrel{def}{=} \\
\quad [f = Nil]\ \overline{\texttt{Conf}}\ \texttt{OK}.\overline{\texttt{restart}}.\texttt{S}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}) \\
\quad + \ [f = cons(head, tail)]\ \ last \leftarrow [tail = Nil].\ \overline{\texttt{K}} < first, last, tag, head > .\overline{\texttt{time1}}. \\
\qquad\quad Wait\_ack(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, head, tail, first, last, tag, rn + 1)
\end{array}
$$

If an acknowledgment is received, the sender resets $timer_1$, reinitializes rn, complements tag, puts $False$ in $first$ and transmits the next message in the file. If no acknowledgment is received, $timer_1$ expires and the sender retransmits the message.

$$
\begin{array}{l}
Wait\_ack(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, head, tail, first, last, tag, rn) \stackrel{def}{=} \\
\quad \texttt{L}.\texttt{timeout1}.Transfer(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, tail, False, last, Not(tag), 0) \\
\quad + \ \texttt{timeout1}.Retrans(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, cons(head, tail), first, last, tag, rn)
\end{array}
$$

If the retransmissions bound is not exceeded, the message is retransmitted. Otherwise, the transfer is aborted. The sender sends a confirmation DTKW (for the last message) or NOTOK (for an intermediate message) to the producer. Then, it makes a rendez-vous with the receiver over the channel *abort* before it begins a new transfer.

$$
\begin{array}{l}
Retrans(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, f, first, last, tag, rn) \stackrel{def}{=} \\
\quad If\ \ rn = \texttt{MAX}\ \ then \\
\qquad ([last = True]\ \ \overline{\texttt{Conf}}\ \texttt{DTKW}.\overline{\texttt{abort}}.\texttt{S}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}) \\
\qquad +[last = False]\ \ \overline{\texttt{Conf}}\ \texttt{NOTOK}.\overline{\texttt{abort}}.\texttt{S}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart})) \\
\quad else\ \ Transfer(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, f, first, last, tag, rn)
\end{array}
$$

The receiver R is described in the same way. The variable *rtag* contains the alternating bit of the previous message. The variable *end* is set to $True$ when the last message in the file is received. The variable *t2on* is set to $True$ when $timer_2$ is enabled.

$$
\texttt{R}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}) \stackrel{def}{=} Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, False, False, False)
$$

$$
\begin{array}{l}
Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, rtag, end, t2on) \stackrel{def}{=} \\
\quad \texttt{K}(first\ \ last\ \ tag\ \ m).Treat(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, first, last, tag, m, rtag, end, t2on) \\
\quad + \ \texttt{abort}.If\ \ t2on = True\ \ then\ \ \texttt{timeout2}.Abort(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, end) \\
\qquad\qquad\qquad else\ \ Abort(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, end) \\
\quad + \ \texttt{restart}.If\ \ t2on = True\ \ then\ \ \texttt{timeout2}.\texttt{R}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}) \\
\qquad\qquad\qquad else\ \ \texttt{R}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart})
\end{array}
$$

$$
\begin{array}{l}
Treat(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, first, last, tag, m, rtag, end, t2on) \stackrel{def}{=} \\
\quad If\ \ tag = rtag\ \ then \\
\qquad \overline{\texttt{L}}.Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, rtag, end, t2on) \\
\quad else\ \ If\ \ first = True\ \ then\ \ Indicate(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, first, last, m, tag, end, t2on) \\
\qquad\quad else\ \ \texttt{timeout2}.Indicate(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, first, last, m, tag, end, t2on)
\end{array}
$$

$$
\begin{array}{l}
Indicate(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, first, last, m, rtag, end, t2on) \stackrel{def}{=} \\
\quad If\ \ last = True\ \ then \\
\qquad \overline{\texttt{Ind}} < m, \texttt{LAST} > .\overline{\texttt{L}}.\overline{\texttt{time2}}.Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, rtag, True, True) \\
\quad else\ \ If\ \ first = True\ \ then \\
\qquad\quad \overline{\texttt{Ind}} < m, \texttt{FIRST} > .\overline{\texttt{L}}.\overline{\texttt{time2}}.Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, rtag, end, True) \\
\qquad else\ \ \overline{\texttt{Ind}} < m, \texttt{INC} > .\overline{\texttt{L}}.\overline{\texttt{time2}}.Wait\_msg(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, rtag, end, True)
\end{array}
$$

$$
\begin{array}{l}
Abort(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}, end) \stackrel{def}{=} If\ \ end = True\ \ then\ \ \texttt{R}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad else\ \ \overline{\texttt{Inderr}}.\texttt{R}(\texttt{K}, \texttt{L}, \texttt{abort}, \texttt{restart})
\end{array}
$$

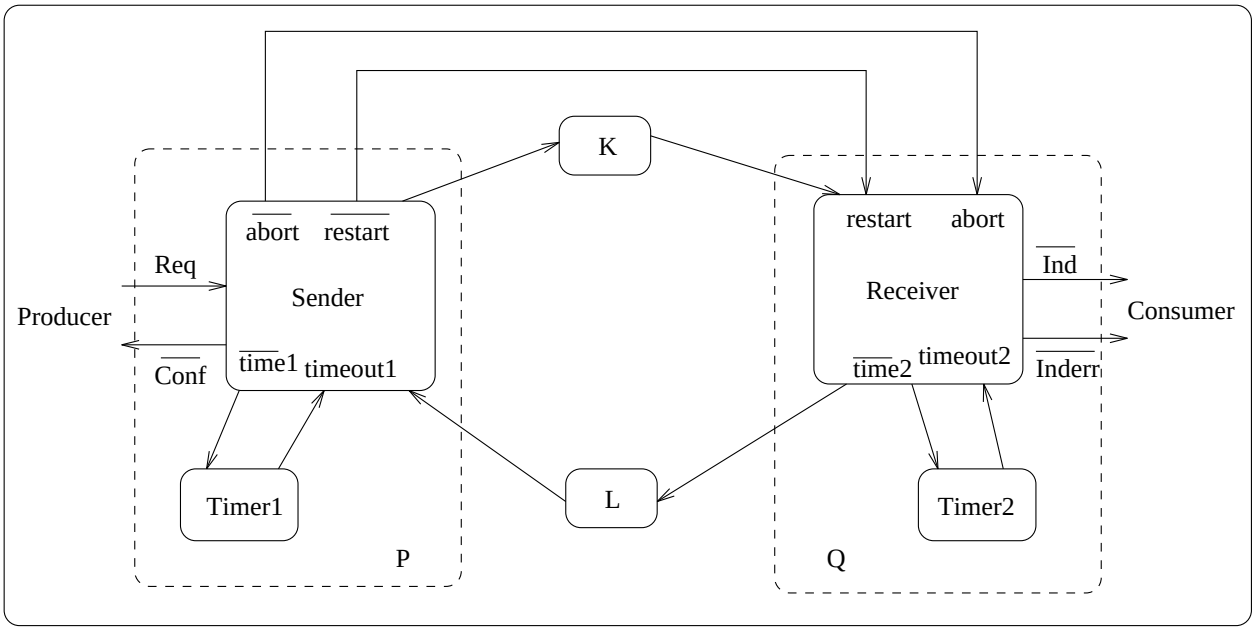

Fig. 3. The BRP implementation in the π-calculus

The sender and its timer constitute the component P of the system. They communicate via their private channels *time*1 and *timeout*1. The receiver and its timer constitute the component Q of the system. They communicate via their private channels *time*2 and *timeout*2. These two components communicate via the sender and the receiver which are linked by the channels K, L, *abort*, and *restart*. These channels are modeled by restricted names since they are private to the protocol. This global view is pictured in figure 3.

$$P(K, L, \mathtt{abort}, \mathtt{restart}) \stackrel{def}{=} (\nu\ \mathtt{time1}\ \mathtt{timeout1})\ (S(K, L, \mathtt{abort}, \mathtt{restart}) \mid T1)$$

$$Q(K, L, \mathtt{abort}, \mathtt{restart}) \stackrel{def}{=} (\nu\ \mathtt{time2}\ \mathtt{timeout2})\ (R(K, L, \mathtt{abort}, \mathtt{restart}) \mid T2)$$

The external event corresponding to the loss of a message (resp. loss of an acknowledgment) is modeled by the agent *loss_msg* (resp. *loss_ack*) which can intercept the message (resp. the acknowledgment) and return to its initial state. These two events can happen at any moment.

$$\mathtt{loss\_msg}(K) \stackrel{def}{=} K(first\ last\ tag\ m).\mathtt{loss\_msg}(K)$$

$$\mathtt{loss\_ack}(L) \stackrel{def}{=} L.\mathtt{loss\_ack}(L)$$

Hence, the system is completely described by the parallel composition of the components P and Q, and the external events *loss_msg* and *loss_ack*.

$$\mathtt{System} \stackrel{def}{=} (\nu\ K\ L\ \mathtt{abort}\ \mathtt{restart})$$
$$(P(K, L, \mathtt{abort}, \mathtt{restart}) \mid Q(K, L, \mathtt{abort}, \mathtt{restart}) \mid \mathtt{loss\_msg}(K) \mid \mathtt{loss\_ack}(L))$$

Note that the configuration of the system does not change during the execution of the protocol: the links are static. However, the mobility would be easily expressed in the π-calculus.

5 The correctness proof of the BRP

The procedure is to prove formally that the system implementation and its abstract view have equivalent behaviours. This allows us to deduce some system properties, for example the *deadlock-freeness*. In the π-calculus, the notion of behavioural equivalence is made mathematically precise by using *bisimulations* [24]. In our proof, we use some algebraic laws of these bisimulations, so we recall them in appendix A.

5.1 The proof method

Our method is inspired by Orava's and Parrow's method [25]. The proof follows these steps:

1. Analyze the system implementation by applying repeatedly the expansion law (E) in order to determine its intermediate states by using strong ground equivalence $\dot{\sim}$. For example, the system $(\overline{x}y.P \mid x(u).Q)$ is expanded to $(\overline{x}y.P + x(u).Q + \tau.(P \mid Q\{y/u\}))$, then we iterate the expansion on the new states $\overline{x}y.P$, $x(u).Q$ and $\tau.(P \mid Q\{y/u\})$. This step leads to a set R_0 of mutual recursive equations.
2. Build the fix-point of R_0. This results in a new set R_1 of mutual recursive equations.
3. Simplify R_1 by using the τ-laws, by identifying and substituting in the equations equivalent expressions up to weak bisimulation $\simeq$, and by eliminating τ-loops from equations using the law (L) in order to obtain guarded equations. This step leads to a new reduced set R_2 of guarded and mutual recursive equations.
4. Build the fix-point ABS of the equations defining the abstract view.
5. Finally, prove that R_2 is a solution of ABS. Then, by applying the law (U1), conclude that R_2 and ABS are equivalent.

5.2 Applying the method to the BRP

Starting from the BRP implementation (*System*), the step 1 is first applied separately to the components P and Q, then it is applied to the parallel composition of their expansion with the external events *loss_msg* and *loss_ack*. This technique has a great advantage: it is *modular* in that we never have to analyze the whole system implementation at once.

Because of lack of space, we cannot give the complete proof. The step 1 results in twenty four equations parameterized by the file to be transferred. The step 3 leads to a system of three equations which is proved equivalent to the abstract view. The complete proof [19] is done manually and requires about three man-months.

As a first conclusion, our method provides a clear distinction between the implementation and the abstraction of the system, proving their equivalence. Someone who wants to use the protocol as a component of a more complex system has just to use its abstract view which is simple and provides exactly the same observable behaviour.

6 Related works

We study in depth some related papers [1, 7, 9, 10, 14, 15] in order to compare different approaches. While in [1, 7, 10, 15], the BRP is developed and proved using a top-down approach, in [9, 14] it is the bottom-up approach which is adopted. The comparison tackles the following questions. Do the authors start from the same description? How are the protocol entities modeled? What is exactly proved? What are the difficulties encountered in doing the proofs? Are they due to the used formalism or to implementation choices? Is the π-calculus a well-suited framework?

The first important remark is that apart from some little and irrelevant variations, all the papers start from the same protocol description.

6.1 Groote and Van De Pol

In [10], Groote and Van De Pol use as a formal support μCRL [11], a combination of process algebra and abstract data types, to prove the correctness of the BRP. Like us, the formalism does not provide explicit time. So, the timers just have to expire, and the authors only care about scheduling of actions.

In this work, the BRP abstract view is defined by four recursive equations written in μCRL.

The BRP implementation is defined in μCRL as the parallel composition of its components, as we have done. The synchronization between the sender and the receiver, done via the channels *abort* and *restart* in our case, is enforced by two extra signals *lost* and *ready*. To avoid that a message arrives after $timer_1$ expires, the channels K and L send a signal *lost* to $timer_1$ indicating that a timeout may occur. When an abort occurs, the sender sends a signal *ready* to the receiver asking it to stop $timer_2$. Then, the receiver returns a signal *ready* to the sender allowing it to transfer a new list. Since there is a strong connection between the sender, the receiver, $timer_1$ and $timer_2$, the resulting implementation is not modular.

The authors use the *branching bisimulation*, a strong variant of weak bisimulation, which is a model of μCRL theory [12]. They prove manually the equivalence between the protocol and its abstract view. But in their case, the protocol can start transmission of a list in two distinct modes: either the receiver knows the next alternating bit (after a successful transmission), or it does not know (after a transmission abort). For this reason, an intermediate system is defined by eight equations considering these two modes. In our case, this system is simply the abstract view and is defined by three equations, so our proof is facilitated.

Their proof is mechanized in the proof-assistant Coq [6]. The authors encode the syntax, axioms and rules of μCRL in Coq. Their BRP implementation in μCRL is compact and formal, but the proof in Coq required a detailed encoding so that the resulting Coq specification is fairly large.

6.2 Helmik, Sellink and Vandraager

In [15], Helmink, Sellink and Vaandrager analyze the BRP in the setting of I/O automata [18]. The timers are represented by timer events. For example, the loss of a message or an acknowledgment causes a timeout action of $timer_1$.

The authors specify the abstract view by an I/O automaton which has the same input and output actions as the protocol but no internal actions. As channels are modeled by shared variables, their access managing is part of the abstract view and is described by means of preconditions.

The authors specify each component of the protocol (the sender, the receiver, and channels K and L) by an I/O automaton. Then, they define the full protocol as the parallel composition of these I/O automata. After an abort, the sender starts a new timer called $timer_3$. When $timer_2$ expires, the receiver generates a timeout action for $timer_3$ so that the sender can proceed and handle the next request. This solution requires that $timer_3 \succ timer_2$ and can be hardly reused if time constraints have to be changed. Moreover, the model forces them to specify, for all possible states, what happens if an input action occurs. This leads to the explosion of the I/O automata.

The main advantage of this work is the correctness criteria of the protocol that is a refinement argument showing that the BRP I/O automata implement the abstract view I/O automaton. The authors prove that the BRP is deadlock-free. Moreover, a number of protocol invariants is presented. However, the most difficulties with I/O automata verifications is finding the appropriate automata, the refinement relation and the invariants.

The safety part of the proofs is mechanically checked using Coq. But, the notions from I/O automata theory are encoded directly for the BRP. So, it is difficult to reuse this encoding for other applications.

6.3 Abrial

In [1], Abrial designs the BRP by successive refinements in the proof-assistant B [2]. The timers are represented by timer events.

The abstract view states in the B language that the file received by the consumer is a prefix of the file transmitted by the producer. The file is supposed to be transmitted instantaneously.

The author builds formally the protocol by extending gradually the implicit time in the abstract view to obtain the implementation. Each refinement step is proved to satisfy the properties expressed in the previous

one. This construction approach required seven refinements which deal with gradual distribution of various aspects of the protocol that are global in the abstract view. However, a loss of the last acknowledgment causes a misleading abort of the sender. In fact, the receiver considers that the transfer is already completed, so any retransmission done by the sender will not be acknowledged. Furthermore, the retransmissions number is still shared by the sender and the receiver in the last refinement.

The deadlock-freeness property is proved provided the protocol is performed in a fully sequential way. Moreover, the termination of the protocol is proved by determining a sequence of natural number expressions that decrease lexicographically after each protocol action. But, the most difficulty of this work is to find the appropriate refinements; there is no systematic method.

6.4 D'Argenio, Katoen, Ruys and Tretmans

In [7], D'Argenio, Katoen, Ruys and Tretmans analyze the BRP in the setting of timed automata [3]. A timed automaton is a classical finite state automaton equipped with clock variables and state invariants which constrain the amount of time the system may idle in a state.

The abstract view is provided as a file transfer service (FTS) described by logical relations between inputs and outputs.

The BRP is modeled by a network of timed automata. Channels K and L are modeled as queues of unbounded capacity. After an abort, an additional delay $SYNC$ (equivalent to $timer_3$ of [15] is set to the sender to ensure that it does not start transmitting a new file before the receiver has properly reacted to the abort.

The authors verify the protocol correctness in UPPAAL [5] which reduces the verification problem to solving a set of constraints on clock variables. The great advantage is that they obtain tight constraints on the amount of the timers. However, as UPPAAL is sensible to the number of states and transitions, data is restricted to clocks and integers. Moreover, value passing at synchronization is not supported. For these reasons, the data was removed from the transmitted message. So, properties of the FTS concerning the transmitted data are not checked. Value passing was modeled by shared variables assignments, this required to split some transitions. Channels K and L were reduced from unbounded queues to one-place buffers. So, to avoid explosion of states and transitions, the protocol is only checked for small values of the file length and the number of retransmissions.

6.5 Havelund and Shankar

Now, we discuss the opposed approach taken in [14] which starts from an implementation to deduce an abstract view. Havelund and Shankar combine model checking and theorem proving techniques to prove the correctness of the BRP. The modeling of time and synchronization between the sender and the receiver is the same as [15]. So, the previous remarks apply here.

The authors first analyze a scaled-down version (i.e, finite state system) of the BRP using Murϕ [21], a state exploration tool, as a debugging aid. Then, they translate the Murϕ description into PVS [26] and modify manually a few of the PVS declarations to obtain the infinite state implementation. This yields two PVS theories. The first one contains the protocol itself. It is modeled by a predicate that holds for a sequence of reachable states. The implementation in PVS is too detailed and not so formal. The second theory contains the correctness criteria which is defined by an invariant. This invariant needs to be greatly strengthened in order to be provable, and this invariant strengthening is the real challenge of the proof.

Finally, from the complete implementation in PVS, they deduce a finite state abstraction which bound the resources of unboundedness in the state space that are the message data, the number of retransmissions and the file length. They show that the mapping between the implementation and the abstract view preserves the initialization predicate, the next-state relation and the properties. They used the model checkers SMV [20], Murϕ and an extension of PVS with the modal μ-calculus [16] for the final model checking. However, the most difficulty of Havelund's and Shanker's approach is to find the protocol abstraction: no technique is provided to mechanize the abstraction research. For example, to find the abstraction of the sliding window protocol is a real challenge.

6.6 Graf and Saidi

In [9], the BRP is first described in terms of guarded assignments. Then, by using abstract interpretation techniques (i.e, by giving a partition of the state space induced by a set of predicates on the system variables; 19 predicates for the BRP), the authors generate automatically an abstract state graph using PVS. The obtained graph for the BRP has 475 states and 685 transitions.

The correctness of the BRP is expressed by two temporal logic formulas. The first one indicate that the sequences of received messages and of sent messages are consistent. The second one indicate that for each file, the indication delivered to the consumer and the confirmation delivered to the producer are consistent.

The main advantage of this work is that the system properties are verified automatically on the abstract graph using the Aldébaran tool [8]. However, the verification is sensible to states explosion. For this reason, the first property of the BRP is verified on a weaker abstraction where only predicates concerning the transmission of a single message are considered relevant.

7 Conclusions

Having compared with other works, the π-calculus appears as a really convenient framework for encoding and analyzing communication protocols. The major advantages of our approach are the following. The description of the protocol is compact and entirely formal. Moreover, the exhaustive analysis of all possible cases gives a good understanding of the protocol; it allows us to detect several implementation errors. The approach is *modular* since we never have to handle the whole protocol description at once. So, the implementation can be reused easily if specification changes occur. Finally, the π-calculus laws are simple and the proof by bisimulation is purely procedural. So, large parts of the proof can be mechanized.

However, without the help of a prover, the exhaustive analysis of all possible cases is a tedious work. So, one objective is to mechanize at least parts of the proof in a theorem prover. Actually, we are formalizing the π-calculus in the theorem prover PVS. Another objective is to reduce the proof effort. Since our implementation is modular, we are investigating a compositional proof of the BRP by using the relativized bisimulation [17]. As a future work, we want to extend the methodology in order to prove mobile protocols and liveness properties.

References

1. Abrial, J-R.: Specification and Design of a Transmission Protocol by Successive Refinements using B, unpublished note, 1997.
2. Abrial, J-R.: The B-Book. Cambridge University Press, 1996.
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science, **126** (1994) p183–235.
4. Alur, R., Henzinger, T., Sontag, E.D.: Hybrid Systems III. LNCS 1066, Springer-Verlag, 1996.
5. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL – A tool suite for the automatic verification of real-time systems. In [4], p232–243.
6. Cornes, C., Courant, J., Filliatre, J.C., Huet, G., Manoury, P., Paulin-Mohring, C., Munoz, C., Murthy, C., Parent, C., Saibi, A., Werner, B.: The Coq Proof Assistant Reference Manual version 5.10. Technical Report, INRIA Rocquencourt, France, February 1995.
7. D'Argenio, P.R., Katoen, J.P., Ruys, T.C., Tretmans, J.: The Bounded Retransmission Protocol must be on Time!. TACAS'97.
8. Fernandez, J.C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP (Caesar/Aldébaran Development Package): A protocol validation and verification tool box, CAV'96, LNCS 1102, 1996. TACAS'97.
9. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. CAV'97, LNCS 1254, Haifa, Israel, 1997.
10. Groote, J.F., Van de Pol, J.: A Bounded Retransmission Protocol for Large Data Packets. CAV'96, LNCS 1101, 1996.
11. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. Technical report CS-R9076, CWI, Amsterdam, December 1990.
12. Groote, J.F., Ponse, A.: Proof theory for μCRL: a language for processes with data. In Andrews, D.J., Groote, J.F., and Middelburg, C.A., editors, Proc. of the Int. Workshop on Semantics of Specification Languages, p232–251. Workshops in Computing, Springer Verlag, 1994.

13. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall, 1991.
14. Havelund, K., Shankar, N.: Experiments in Theorem Proving and Model Checking for Protocol Verification. In Proceeding of FME, March 1996, Oxford.
15. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof checking a data link protocol. In Barandregt, H., and Nipkow, T., editors, Types for proofs and programs, LNCS 806, p127–165, Springer-Verlag, 1994.
16. Janssen, G.: ROBDD Software. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
17. Larsen, K., Milner, R.: A Complete Protocol Verification Using Relativized Bisimulation. In Proceeding 14th Colloquium on Automata, Languages and Programming, LNCS 267, Springer-Verlag, 1987.
18. Lynch, N.A., Tuttle, M.R.: Hierarchical Correctness Proofs for Distributed Algorithms. In Proceeding of the 6th Annual Symposium on Principles of Distributed Computing, New York, p137–151, ACM Press, 1987.
19. Mammass, B.: Une Preuve Formelle du Bounded Retransmission Protocol dans le π-calcul. Technical report, LIP6-98-009, université de Paris VI, 1998.
20. McMillan, K.L.: Symbolic Model ChecKing. Kluwer Academic Publishers, Boston, 1993.
21. Melton, R., Dill, D.L., Norris Ip., C.: Murphi Annotated Reference Manual, version 2.6. Technical Report, Stanford University, Palo Alto, California, USA, November 1993.
22. Milner, R.: Communication and Concurrency. Prentice-Hall, 1989.
23. Milner, R.: The polyadic π-calculus: a tutorial. LFCS, technical report ECS-LFCS-91-180, October 1991.
24. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part 1 and Part 2. LFCS, technical report ECS-LFCS-89-85 and 86, June 1989.
25. Orava, F., Parrow, J.: An Algebraic Verification of a Mobile Network. Formal Aspects of Computing, 4(6), p497–543, 1992.
26. Owre, S., Rushby, J., Shankar, N., Henke, F.von.: Formal Verification For Fault-tolerant Architectures: Prolegomena to the Design of PVS. IEEE Transactions on Software Engineering, 21(2), p107–125, February 1995.

A π-calculus algebraic theory

The *strong ground equivalence* $\dot{\sim}$ corresponds to behavioural a equivalence where the precise amount of internal actions τ is significant. For example, we distinguish the agent $\tau.\tau.0$ from the agent $\tau.0$. In contrast, the *weak ground equivalence* $\dot{\simeq}$ identifies this two agents; the internal actions τ are significant only insofar as they preempt other actions.

The algebraic laws for strong ground equivalence $\dot{\sim}$, as stated in [24], are described below. To state them in a compact way, we define the derived prefix $\overline{x}(\widetilde{y}).P$ to mean $(\nu\ \widetilde{y})\overline{x}\widetilde{y}.P$ when $x \neq y$, and let α, β range over ordinary and derived prefixes. Let $fn(P)$ (resp. $bn(P)$) be the set of free (resp. bound) names in P. Hereafter, $=$ is used instead of $\dot{\sim}$ to allow different interpretations of the laws.

- (A) $P \equiv Q \vdash P = Q$ (α-conversion)
- (C0) $P = Q \vdash \tau.P = \tau.Q$, $P + R = Q + R$, $(\nu\ x)P = (\nu\ x)Q$
  $\overline{x}y.P = \overline{x}y.Q$, $P \mid R = Q \mid R$, $[x = y]P = [x = y]Q$
- (C1) $x(y).P = x(y).Q$ iff $P\{z/y\} = Q\{z/y\}$, $\forall\ z$
- 0 is a zero for $+$, and $+$ is idempotent, commutative and associative.
- (R0) $(\nu\ x)P = P$ (if $x \notin$ fn(P))
  (R1) $(\nu\ x)(\nu\ y)P = (\nu\ y)(\nu\ x)P$
  (R2) $(\nu\ x)(P + Q) = (\nu\ x)P + (\nu\ x)Q$
  (R3) $(\nu\ x)\alpha.P = \alpha.(\nu\ x)P$ (if x is not in α)
  (R4) $(\nu\ x)\alpha.P = 0$ (if x is the subject of α)
- (M0) $[x = y]P = 0$ if $x \neq y$, (M1) $[x = x]P = P$
- (I) $A(\widetilde{y}) = P\{\widetilde{y}/\widetilde{x}\}$ if $A(\widetilde{x}) \stackrel{def}{=} P$
- 0 is a zero for $\mid$, and $\mid$ is commutative and associative.
  (P3) $(\nu\ x)(P \mid Q) = P \mid (\nu\ x)Q$ (if $x \notin fn(P)$)
- (E) Let $P = \sum_i \alpha_i.P_i$, $Q = \sum_j \beta_j.Q_j$ where $bn(\alpha_i) \cap fn(Q) = \emptyset\ \forall\ i$ and $bn(\beta_j) \cap fn(P) = \emptyset\ \forall\ j$. Then $P \mid Q = \sum_i \alpha_i.(P_i \mid Q) + \sum_j \beta_j.(P \mid Q_j) + \sum_{\alpha_i\, comp\, \beta_j} \tau.R_{ij}$
  where the relation $\alpha_i\ comp\ \beta_j$ (α_i complements β_j) holds in the following four cases, which also define R_{ij}:

1. α_i is $\overline{x}u$ and β_j is $x(v)$; then R_{ij} is $P_i \mid Q_j\{u/v\}$
2. α_i is $\overline{x}(u)$ and β_j is $x(v)$; then R_{ij} is $(\nu\ w)(P_i\{w/u\} \mid Q_j\{w/v\})$ (where w is not free in $(\nu\ u)P_i$ or in $(\nu\ v)Q_j$)
3. the two others are the converse.

The weak ground equivalence $\dot{\simeq}$ is strictly weaker that strong ground equivalence $\dot{\sim}$ and also satisfies the laws described above. In addition, it satisfies the well known τ-laws [22], these are:

- (T0) $\alpha.\tau.P \dot{\simeq} \alpha.P$
- (T1) $P + \tau.P \dot{\simeq} \tau.P$
- (T2) $\alpha.(P + \tau.Q) + \alpha.Q \dot{\simeq} \alpha.(P + \tau.Q)$.

In order to eliminate τ-loops from recursively defined agents (see [22]):

- (L) If $A = P + \tau.A$ and $B = \tau.P$ then $A \dot{\simeq} B$

We define *strong (non-ground) equivalence* $\sim$ as strong ground equivalence under all substitutions σ of non-constant names, i.e., $P \sim Q$ iff $P\sigma \dot{\sim} Q\sigma$, for all substitutions σ from non-constant names to names. We define *weak (non-ground) equivalence* $\simeq$ in a similar way.

The main use of the non-ground equivalences is in the laws for recursively defined agents which we adopt from [24]. To formulate them, we need some additional notations. Let E, F, ... represent agent expressions; these are like agents with "holes" where agents or agent identifiers can be inserted. Let $E(P_1, \ldots, P_n)$ be the agent which is the result of inserting $P_1, \ldots, P_n$ into E. Two agent expressions E and F are (strongly/weakly) equivalent if $E(\widetilde{P})$ is (strongly/weakly) equivalent to $F(\widetilde{P})$ for all $P_1, \ldots, P_n$.

The first law for recursion (U0) means that if the right hand sides of definitions are transformed, respecting equivalence, then the agent defined is the same up to equivalence. This law holds for strong and weak non-ground equivalence (but fails for the ground equivalences).

(U0) Suppose that $E_1, \ldots, E_n$ and $F_1, \ldots, F_n$ are expressions and $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$ identifiers such that for all i: $E_i = F_i$ and $A_i(\widetilde{x_i}) = E_i(A_1, \ldots, A_n)$ and $B_i(\widetilde{x_i}) = F_i(B_1, \ldots, B_n)$ Then $A_i(\widetilde{x_i}) = B_i(\widetilde{x_i})$ for all i.

The second law (U1) means that if two agents satisfy the same set of recursive equations, then the agents are equivalent. This law holds for strong non-ground equivalence provided $E_1, \ldots, E_n$ are *weakly guarded* (i.e., all occurrences of P_j in $E_i(P_1, \ldots, P_n)$ are within a prefix operator). Furthermore, it holds for weak non-ground equivalence provided $E_1, \ldots, E_n$ are *guarded* (i.e., all occurrences of P_j in $E_i(P_1, \ldots, P_n)$ are within an output or input prefix operator), and *sequential* (i.e., no E_i contains a parallel composition).

(U1) Suppose that $E_1, \ldots, E_n$ are expressions and $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ are agents such that for all i: $P_i = E_i(P_1, \ldots, P_n)$ and $Q_i = E_i(Q_1, \ldots, Q_n)$ Then $P_i = Q_i$ for all i.

This article was processed using the LaTeX macro package with LLNCS style