Information and Computation ••• (••••) •••–•••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Deadlock analysis of unbounded process networks

Naoki Kobayashi [a,*], Cosimo Laneve [b,c,*]

[a] Dept. of Computer Science, University of Tokyo, Japan
[b] Dept. of Computer Science and Engineering, University of Bologna, Italy
[c] INRIA FOCUS, France

ARTICLE INFO

Article history:
Received 22 December 2014
Available online xxxx

Keywords:
Deadlocks
Process calculi
Type systems
Behavioural types
Lam programs
Fixpoints

ABSTRACT

Deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes is extremely complex and solutions either give imprecise answers or do not scale. To enable the analysis of such programs, (1) we define an algorithm for detecting deadlocks of a basic model featuring recursion and fresh name generation: the *lam programs*, and (2) we design a type system for value-passing CCS that returns lam programs. We show the soundness of the type system, and develop a type inference algorithm for it. The resulting algorithm is able to check deadlock-freedom of programs that cannot be handled by previous analyses, such as those that build unbounded networks.

1. Introduction

Deadlock-freedom of concurrent programs has been largely investigated in the literature [1–6]. The proposed algorithms automatically detect deadlocks by building graphs of dependencies (a, b) between resources, meaning that the release of a resource referenced by a depends on the release of the resource referenced by b. The absence of cycles in the graphs entails deadlock freedom. When programs have infinite states, in order to ensure termination, current algorithms use finite approximate models that are excerpted from the dependency graphs. The cases that are particularly critical are those of programs that create networks with an arbitrary number of nodes.

To illustrate the issue, consider the following *value-passing CCS* [7] process that computes the factorial:

$$Fact(n, r, s) = \textbf{if } n = 0 \textbf{ then } r?m.s!m \textbf{ else } (\nu t)(r?m.t!(m * n) \mid Fact(n - 1, t, s))$$

Here, $r?m$ waits to receive a value for m on r, and $s!m$ sends the value m on s. The expression $(\nu t)P$ creates a fresh communication channel t and executes P. If the above code is invoked with $r!1 \mid Fact(n, r, s)$, then there will be a synchronisation between $r!1$ and the input $r?m$ in the body of $Fact(n, r, s)$. In turn, this may delegate the computation of the factorial to another process in parallel by means of a subsequent synchronisation on a new channel t. That is, in order to compute the factorial of n, *Fact* builds a network of $n + 1$ nodes, where node i takes as input a value m and outputs $m * i$. Due to the inability of statically reasoning about unbounded structures, the current analysers usually return false positives when fed with `Fact`. For example, this is the case of `TyPiCal` [8,4], a tool developed for pi-calculus [9] – an extension of value-

* Corresponding authors.
E-mail addresses: koba@is.s.u-tokyo.ac.jp (N. Kobayashi), cosimo.laneve@unibo.it (C. Laneve).

http://dx.doi.org/10.1016/j.ic.2016.03.004
0890-5401/

passing CCS with channel data types – as well as of a recent type system of Padovani [10]. (In particular, `TyPiCal` fails to recognise that there is no circularity in the dependencies among r, s, and t.)

In this paper we develop a technique to enable the deadlock analysis of processes with arbitrary networks of nodes. Instead of reasoning on finite approximations of such processes, we associate them with terms of a basic recursive model, called *lam* – for *deadLock Analysis Model* – which collects dependencies and features recursion and dynamic name creation [11,12]. For example, a (simplified) lam function corresponding to `Fact` is[1]

$$\mathtt{fact}(a_r, a_s) = (a_r, a_s) + (\nu\, a_t)\big((a_r, a_t) \,\&\, \mathtt{fact}(a_t, a_s)\big)$$

where a_r, a_s, a_t are "level" names associated to r, s, t, respectively, and (a_r, a_s) displays the dependency between the actions $r?m$ and $s!m$ and (a_r, a_t) the one between $r?m$ and $t!(m * n)$. The function `fact` is defined operationally by unfolding the recursive invocations; see Section 3. The unfolding of $\mathtt{fact}(a_r, a_s)$ yields the following sequence of abstract states (bound names in the definition of `fact` are replaced by fresh ones in the unfoldings):

$$\begin{aligned}
\mathtt{fact}(a_r, a_s) &\longrightarrow (a_r, a_s) + \big((a_r, a_t) \,\&\, \mathtt{fact}(a_t, a_s)\big)\\
&\longrightarrow (a_r, a_s) + (a_r, a_t) \,\&\, (a_t, a_s) + (a_r, a_t) \,\&\, (a_t, a_u) \,\&\, \mathtt{fact}(a_u, a_s)\\
&\longrightarrow (a_r, a_s) + (a_r, a_t) \,\&\, (a_t, a_s) + (a_r, a_t) \,\&\, (a_t, a_u) \,\&\, (a_u, a_s)\\
&\qquad + (a_r, a_t) \,\&\, (a_t, a_u) \,\&\, (a_u, a_v) \,\&\, \mathtt{fact}(a_v, a_s)\\
&\longrightarrow \cdots .
\end{aligned}$$

While the model of `fact` is not finite-state, in Section 4 we demonstrate that it is decidable whether the computations of a lam program will ever produce a circular dependency. In our previous work [11,12], the decidability was established only for the restricted subset of lams, called *linear recursive*, where recursive invocations may occur at most once. The algorithm in [11,12] uses a technique that is a generalisation of permutation theory and, therefore, it is different from the one in this contribution. In addition, the technique of [11,12] was imprecise for nonlinear recursive lams, such as those corresponding to the Fibonacci process:

$$\begin{aligned}
\mathit{Fib}(n, r) \;=\; &\textbf{if } n < 2 \textbf{ then } r?n.0 \textbf{ else}\\
&(\nu t)(\nu s)\,(\mathit{Fib}(n-1, s) \mid s?x.(\mathit{Fib}(n-2, t) \mid t?y.r!x + y\,)
\end{aligned}$$

We also define a type system that associates lams to processes. Using the type system, for example, the lam program `fact` can be extracted from the factorial process `Fact`. The syntax, semantics, and examples of value-passing CCS are in Section 5. The type system is defined in Section 6, where we also discuss the extension of our technique to pi-calculus (actually we decided to target value-passing CCS for the sake of simplicity because it is simpler than pi-calculus and it is already adequate to demonstrate the power of our lam-based approach). As a byproduct of the above techniques, our system is powerful enough to detect deadlocks of programs that create networks with arbitrary numbers of processes. The algorithm to infer a lam program from a process is detailed in Section 7. We discuss the differences of our techniques with respect to the other ones in the literature in Section 8 and we deliver some concluding remark in Section 9.

This article is a revised and enhanced version of [13] that includes the full proofs of all the results and the type inference algorithm for value-passing CCS.

2. Preliminaries

We use an infinite set $\mathcal{A}$ of (*level*) *names*, ranged over by a, b, c, $\cdots$. A relation on a set A of names, denoted $\mathtt{R}, \mathtt{R}', \cdots$, is an element of $\mathcal{P}(A \times A)$, where $\mathcal{P}(\cdot)$ is the standard powerset operator and $\cdot \times \cdot$ is the Cartesian product. Let

- $\mathtt{R}^+$ be the *transitive closure* of $\mathtt{R}$.
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that $\mathtt{R}_i \subseteq {\mathtt{R}'_j}^+$.
- $(a_0, a_1), \cdots, (a_{n-1}, a_n) \Subset \{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ if and only if there is $\mathtt{R}_i$ such that $(a_0, a_1), \cdots, (a_{n-1}, a_n) \in \mathtt{R}_i$.
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \,\&\, \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \le i \le m \text{ and } 1 \le j \le n\}$.

We use $\mathcal{R}, \mathcal{R}', \cdots$ to range over *sets of relations*, such as $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ and $\{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$, which are elements of $\mathcal{P}(\mathcal{P}(A \times A))$.

[1] Actually, the lam function associated to `Fact` by the type system in Section 6 is more verbose because every channel has two corresponding level names:

$$\mathtt{fact}(a_r, a'_r, a_s, a'_s) = (a'_r, a_s) + (\nu\, a_t, a'_t)\big((a'_r, a'_t) \,\&\, \mathtt{fact}(a_t, a'_t, a_s, a'_s)\big)$$

These pairs of names are the capability and obligation levels in [4] and we refer to Section 6 for a detailed discussion.

Please cite this article in press as: N. Kobayashi, C. Laneve, Deadlock analysis of unbounded process networks, Inf. Comput. (2016), http://dx.doi.org/10.1016/j.ic.2016.03.004