Information Processing Letters 133 (2018) 21–25

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Compositional and local livelock analysis for CSP

M.S. Conserva Filho [a,*], M.V.M. Oliveira [a], A. Sampaio [b], Ana Cavalcanti [c]

[a] Universidade Federal do Rio Grande do Norte, Brazil
[b] Universidade Federal de Pernambuco, Brazil
[c] University of York, UK

ARTICLE INFO

Article history:
Received 7 December 2016
Received in revised form 10 August 2017
Accepted 30 December 2017
Available online 12 January 2018
Communicated by J.L. Fiadeiro

Keywords:
Process algebra
Divergence
Model checking
Components
Performance evaluation

ABSTRACT

The success of component-based techniques for software construction relies on trust in the emergent behaviour of the compositions. Here, we propose an efficient correct-by-construction technique for building livelock-free CSP models. Its verification conditions are based on a local analysis of the shortest event sequences (traces) that represent a recursive behaviour in the CSP model. This affords significant gains in performance in model checking. We evaluate our strategy based on models of the Milner's scheduler and the dining philosophers.

1. Introduction

Compositional modelling and verification approaches are popular [4], but rely on trust in the emergent behaviour of the compositions. Process algebras are among the adopted formalisms. CSP [6,10] is a well established process algebra to model and verify concurrent systems. CSP offers consolidated semantic models that support a wide range of verifications, including livelock freedom. A system is livelock free (divergence free) if there exists no state from which it internally computes through an infinite sequence of internal actions [10].

The main approach to prove divergence freedom requires a global analysis of the system. This strategy is automated for CSP, for instance, by FDR4 [5]. One alternative is a static analysis of the syntactic structure of a process [9]. For that, syntactic rules are proposed either to classify CSP systems as livelock-free or to report an inconclusive result. This approach is implemented in SLAP [9].

We present a technique based on a local analysis, in which we can identify livelock situations when compositions are being performed, predicting, by construction, global property based on known local properties of the components [1]. Our strategy aims at reducing complexity for verifying the absence of divergence, especially comparing with the approach in [9]. We illustrate our technique based on models of the Milner's scheduler and the dining philosophers, and show that it outperforms both FDR4 and SLAP. In cases in which livelock freedom is not ensured, we either identify the possibility of divergence or report an inconclusive result. This incompleteness is the trade-off for scalability.

The next section briefly describes our evaluation strategy. Section 3 describes our technique, whose performance is evaluated in Section 4.

2. Material and methods

The demonstration of the usefulness and efficiency of our technique consists of a comparative analysis of three different scenarios: (i) the traditional global analysis of FDR4, (ii) the static livelock-analysis of SLAP, and (iii) our

* Corresponding author.
E-mail address: madiel@ppgsc.ufrn.br (M.S. Conserva Filho).

https://doi.org/10.1016/j.ipl.2017.12.011
0020-0190/

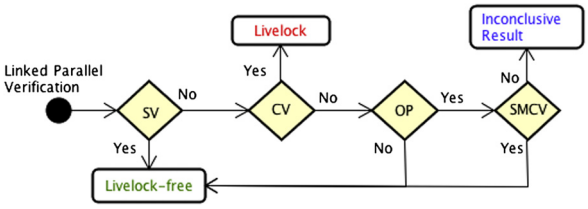

Fig. 1. BPM model of the livelock analysis for linked parallel composition.

local livelock analysis, which is presented in the next section. We have developed two case studies: the Milner's task scheduler [7], which can be modelled as a ring of cells with pairwise synchronisation, and the dining philosophers [10]. All CSP scripts used in the case studies can be found at goo.gl/mAZWXq. We have used a server with 4 core AMD Phenom II, and 8 GB of RAM in a Ubuntu system.

3. Theory

In CSP, when composing divergence-free processes, divergent behaviour can arise from the use of hiding [10]. For a given CSP process P and a set of events X, the process $P\backslash X$ converts visible occurrences of events of P in X into internal events. This transformation may yield an infinite loop of internal events. For instance, $P = (a \rightarrow P)\backslash\{a\}$ is defined in terms of the prefix operator ($\rightarrow$): it engages in event a and then recurses, but it diverges because the event a is hidden, hence, P indefinitely performs internal events without communicating with its environment. If a process can engage in an unbroken sequence of events from a set X, we must ensure that X cannot be hidden.

The hiding operator is also implicitly used in a particular kind of parallel composition: the *linked parallel composition* $P[a \leftrightarrow b]Q$, in which P and Q proceed in parallel with communications on a in P becoming hidden synchronisations with communications on b in Q. Communications on other channels are interleaved: they do not require synchronisation. In general, multiple channels may be linked as, for example, in $P[a \leftrightarrow b, c \leftrightarrow d]Q$.

We propose a constructive approach which guarantees that, for livelock-free processes that obey certain conditions and are composed pairwisely using linked parallel, the resulting composition is livelock-free. To achieve scalability, we perform an optimisation (which we refer in Fig. 1 as OP) that prunes the alternative behaviours of the resulting composition with interleaved events, choosing only one of the alternatives.

Our approach is based on three main verifications, which are systematically applied (see Fig. 1): the Simple Verification (SV) ensures livelock freedom based on an individual analysis of the processes involved in the composition. The absence of livelock is guaranteed if one of the processes is livelock-free after hiding its linking events locally. If that fails, the Complex Verification (CV) checks if the linked processes are able to communicate in an infinite loop via the linked (internal) events. If they are, we have a livelock. Otherwise, if the optimisation (OP) has not been applied, the composition is livelock-free. If, however, the optimisation has been applied, our strategy guarantees livelock freedom only if we have a Safe Multiple Composition ($SMCV$), which does not link events on a many-to-many fashion. Otherwise, the interleaved events pruned by our optimisation may lead the system to divergence. Our strategy is, therefore, inconclusive in such cases. In what follows, we present the basic definitions used in our technique and formally describe these local verifications.

3.1. Basic definitions

Our method considers developments that use livelock-free basic processes, which can be described using most of the CSP main operators, including conditionals, tail and mutual recursions. We also consider parameters. Further information on basic processes can be found in [3]. Parallelism (and hiding) is achieved by composing processes (either basic or resulting from previous compositions) using the linked parallel composition.

The first step of our technique is to identify the infinite behaviours of a given process. For that, we use a pair (tr, mip) of sequences (traces). Its first element is a trace that leads a given process to a recursive behaviour. The second one is a minimal interaction pattern of a given process, that is, the shortest finite sequence of events that represents the recursion itself. The set $XIP(P)$ contains all possible pairs (tr, mip) of the process P.

To exemplify our method, we introduce a classical concurrent system, the dining philosophers [10]. It consists of philosophers sitting at a round table that need to acquire a pair of shared forks before eating. The behaviour of each philosopher and each fork is modelled as a process P_i or F_i for values i from a set ID of philosopher and fork identifiers. We consider two philosophers and two forks and use $ID = \{1, 2\}$. A channel $fk : ID.ID.EV$, where $EV = \{U, D\}$ defines events $fk.i.j.e$ that indicate that the fork i is put up or down, depending on whether e is U or D, by the philosopher j. The fork processes are as follows.

$$F_1 = fk.1.1.U \rightarrow fk.1.1.D \rightarrow F_1 \,\Box\, fk.1.2.U \rightarrow fk.1.2.D \rightarrow F_1$$

$$F_2 = fk.2.2.U \rightarrow fk.2.2.D \rightarrow F_2 \,\Box\, fk.2.1.U \rightarrow fk.2.1.D \rightarrow F_2$$

Initially, a fork can be picked up by either philosopher. Once it is picked up, it can only be put down by the same philosopher. Accordingly, the process F_1 offers a deterministic choice ($\Box$): it engages either on the events $fk.1.1.U$ or $fk.1.2.U$. The prefix operator ($\rightarrow$) states that the corresponding down event (D) is offered afterwards. The process recurses after the down event. Hence, $XIP(F_1) = \{(\langle\rangle, \langle fk.1.1.U, fk.1.1.D\rangle), (\langle\rangle, \langle fk.1.2.U, fk.1.2.D\rangle)\}$. In this example, as F_1 returns to its initial state, tr is the empty trace ($\langle\rangle$).

Similarly, $pfk.j.i.e$ records the action e on fork j by philosopher i. The channel $wk : ID$ defines events $wk.i$, indicating that the philosopher i has just woken up. Finally, the channel $lf : ID.LF$, where $LF = \{T, E\}$ defines events $lf.i.l$, indicating that the philosopher i is either thinking (T) or eating (E).