Advances in Engineering Software 50 (2012) 110–130

Contents lists available at SciVerse ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Generating a Petri net from a CSP specification: A semantics-based method

M. Llorens, J. Oliver *, J. Silva, S. Tamarit

Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Valencia, Spain

ARTICLE INFO

Article history:
Received 7 February 2012
Accepted 13 February 2012
Available online 24 March 2012

Keywords:
Concurrent programming
CSP
Petri nets
Semantics
Traces
Specification and Simulation

ABSTRACT

The specification and simulation of complex concurrent systems is a difficult task due to the intricate combinations of message passing and synchronizations that can occur between the components of the system. Two of the most extended formalisms used to specify, verify and simulate such kind of systems are CSP and the Petri nets. This work introduces a new technique that allows us to automatically transform a CSP specification into an equivalent Petri net. The transformation is formally defined by instrumenting the operational semantics of CSP. Because the technique uses a semantics-directed transformation, it produces Petri nets that are closer to the CSP specification and thus easier to understand. This result is interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri net, but it also allows them to use all the tools and analysis techniques developed for Petri nets.

1. Introduction

Nowadays, few computers are based on a single processor architecture. Contrarily, modern architectures are based on multiprocessor systems such as the dual-core or the quad-core; and a challenge of manufacturer companies is to increase the number of processors integrated in the same motherboard. In order to take advantage of these new hardware systems, software must be prepared to work with parallel and heterogeneous components that work concurrently. This is also a necessity of the widely generalized distributed systems, and it is the reason why the industry invests millions of dollars in the research and development of concurrent languages that can produce efficient programs for these systems, and that can be automatically verified thanks to the development of modern techniques for the analysis and verification of such languages.

In this work we focus on two of the most important concurrent formalisms: the Communicating Sequential Processes (CSPs) [10,24] and the Petri nets [18,20]. CSP is an expressive process algebra with a big collection of software tools for the specification and verification of complex systems. In fact, CSP is currently one of the most extended concurrent specification languages and it is being successfully used in several industrial projects [3,8]. Complementarily, Petri nets are particularly useful for the simulation and animation of concurrent specifications. They can be used to graphically animate a specification and observe the synchronization of components step by step. For these reasons, attempts to combine both models exist (see, e.g., [1]). In this work we define a fully automatic transformation that allows us to transform a CSP specification into an equivalent Petri net (i.e., the sequences of observable events produced are exactly the same). This result is very interesting because it allows CSP developers not only to graphically animate their specifications through the use of the equivalent Petri nets, but it also allows them to use all the tools and analysis techniques developed for Petri nets. Our transformation is based on an instrumentation of the CSP's operational semantics. Roughly speaking, we define an algorithm that explores all computations of a CSP specification by using the instrumented semantics. The execution of the semantics produces as a side-effect the Petri net associated with each computation, and thus the final Petri net is produced incrementally.

In summary, the steps performed by the transformation are the following: firstly, the algorithm takes a CSP specification and executes the extended semantics with an empty store. The execution of the semantics produces a Petri net that represents the performed computation. When the computation is finished, the extended semantics returns to the algorithm a new store with the information about the choices that have been executed. Then, the algorithm determines with this information whether new computations not explored yet exist. If this is the case, the semantics is executed again with an updated store. This is repeated until all possible computations have been explored. This sequence of steps gradually augments the Petri net produced. When the algorithm detects that no more computations are possible (i.e., the store is empty), it outputs the current Petri net as the final result.

This work extends a previous work by the same authors presented at the *7th International Conference on Engineering Computational Technology* [14]. In this new version we provide additional

* Corresponding author.
E-mail addresses: mllorens@dsic.upv.es (M. Llorens), fjoliver@dsic.upv.es (J. Oliver), jsilva@dsic.upv.es (J. Silva), stamarit@dsic.upv.es (S. Tamarit).

0965-9978/$ - see front matter © 2012 Civil-Comp Ltd and Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.advengsoft.2012.02.006

explanations and examples, and new important original material. The new material includes:

1. An instrumentation of the standard CSP operational semantics that produces as a side-effect a Petri net associated to the computations performed with the semantics.
2. New simplification algorithms that significantly reduce the size of the Petri nets generated while keeping the equivalence properties.
3. An improved implementation that has been made public (both the source code and an online version).
4. The correctness results. They prove the termination of the transformation algorithm; and the equivalence between the produced Petri net and the original CSP specification.

The rest of the paper has been organized as follows. Section 2 overviews related work and previous approaches to the transformation of CSP into Petri nets. In Section 3 we briefly recall the syntax and semantics of CSP and Petri nets. Section 4 presents an algorithm able to generate a Petri net equivalent to a given CSP specification. To obtain the Petri net, the algorithm uses an instrumentation of the standard operational semantics of CSP which is also introduced in this section. Then, in Section 5 we introduce some algorithms to further transform the generated Petri nets. The transformation simplifies the final Petri net producing a reduced version that is still equivalent to the original CSP specification. The correctness of the technique presented is proved in Section 6. In Section 7, we describe the *CSP2PN* tool, our implementation of the proposed technique. Finally, Section 8 concludes.

2. Related work

Transforming CSP to Petri nets is known to be useful since almost their origins, because it not only has a clear practical utility, but it also has a wide theoretical interest because both concurrent models are very different, and establishing relations between them allows us to extend results from one model to the other. In fact, the problem of transforming a CSP specification into an equivalent Petri net is complex due to the big differences that exist between both formalisms. For this reason, some previous approaches aiming to transform CSP to Petri nets have been criticized because, even though they are proved equivalent, it is hardly possible to see a relation between the generated Petri net and the initial CSP specification (i.e., when a transition of the Petri net is fired, it is not even clear to what CSP process corresponds this transition). In this respect, the transformation presented here is particularly interesting because the Petri net is generated directly from the operational semantics in such a way that each syntactic element of the CSP specification has a representation in the Petri net. And, moreover, the sequences of steps performed by the CSP semantics are directly represented in the Petri net. Hence, it is not difficult to map the animation of the Petri net to the CSP specification.

We can group all previous approaches aimed at transforming CSP to Petri nets into two major research lines. The first line is based on traces describing the behavior of the system. In [16], starting from a trace-based representation of the behavior of the system, according to a subset of the Hoare's theory where no sequential composition with recursion is allowed, a stochastic Petri net model is built in a modular and systematic way. The overall model is built by modeling the system's components individually, and then putting them together by means of superposition. The second line of research includes all methodologies that translate CSP specifications into Petri nets directly from the CSP syntax. One of the first works translating CSP to Petri nets was [4], where distributed termination is assumed but nesting of parallel commands is not allowed. In [6], a CSP-like language is considered and translated into a subclass of Pr/T nets with individual tokens, where neither nesting of parallel commands is allowed nor distributed termination is taken into account. Other papers in this area are [19] that considers a subset of CCSP (the union of Milner's CCS [17] and Hoare's CSP [10]), and [5] which provides full CSP with a truly concurrent and distributed operational semantics based on Condition/Event Systems. There are also some works that translate process algebras into stochastic or timed Petri nets in order to perform real-time analyses and performance evaluation. Notable examples are [25,15] that translate CSP specifications and [23] that define a compositional stochastic Petri net semantics for the stochastic process algebra PEPA [9]. Even though this work is essentially different from ours because it is based on different formalisms, its implementation [2] is somehow similar to ours because the translation from PEPA to stochastic Petri nets is completely automatic. As in our work, all these papers do not allow recursion of nested parallel processes because the set of places of the generated Petri net would be infinite. In some way, our new semantics-based approach opens a third line of research where the transformation is directed by the semantics.

3. CSP and Petri nets

3.1. The syntax and semantics of CSP

This section recalls CSP's syntax and operational semantics. For concretion, and to facilitate the understanding of the following definitions and algorithms, we have selected a subset of CSP that is sufficiently expressive to illustrate the method, and it contains the most important operators that produce the challenging problems such as deadlocks, non-determinism and parallel execution.

Fig. 1 summarizes the syntax constructions used in CSP [10] specifications. A *specification* is a finite collection of process definitions. The left-hand side of each definition is the name of a process, which is defined in the right-hand side (abbrev. *rhs*) by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing ($a \rightarrow P$): Event a must happen before process P.
Internal choice ($P \sqcap Q$): The system non-deterministically chooses to execute one of the two processes P or Q.
External choice ($P \Box Q$): It is identical to internal choice but the choice comes from outside the system (e.g., the user).
Synchronized parallelism ($P \underset{X \subseteq \Sigma}{\|} Q$): Both processes are executed in parallel with a set X of synchronized events. A particular case of parallel execution is *interleaving* (represented by $|||$) where no synchronizations exist (i.e., $X = \emptyset$) and thus both processes can execute in any order. Whenever a synchronized event $a \in X$ happens in one of the processes, it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that processes are synchronized in all common events.
Stop (*STOP*): Synonym of deadlock, i.e., it finishes the current process.

Example 1. Consider the Moore machine [11] in Fig. 2 to compute the remainder of a binary number divided by three. The different values for the possible remainders are 0, 1 and 2. Note that if a decimal value n written in binary is followed by a 0 then its decimal value becomes $2n$ and if n is followed by a 1 then its value becomes $2n + 1$. If the remainder of $n/3$ is r, then the remainder of $2n/3$ is $2r \ mod\ 3$. If $r = 0$, 1, or 2, then $2r \ mod\ 3$ is 0, 2, or 1, respectively. Similarly, the remainder of $(2n + 1)/3$ is 1, 0, or 2, respectively. So, this machine has 3 states: q_0 is the start state and represents a