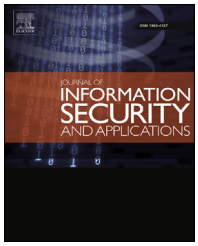
JOURNAL OF INFORMATION SECURITY AND APPLICATIONS ■■ (2016) ■■–■■

Available online at www.sciencedirect.com

ScienceDirect

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



Formal modeling and analysis of time- and resource-sensitive simple business processes ☆

Kazuhiro Ogata [a,*], *Thapana Chaimanont* [a,†], *Min Zhang* [b]

[a] *Japan Advanced Institute of Science and Technology (JAIST), Japan*
[b] *East China Normal University (ECNU), China*

ARTICLE INFO

Article history:
Available online

Keywords:
Alloy
Business process
Formal analysis
Resource
Round-based model
Maude
Time

ABSTRACT

A time- and resource-sensitive simple business process (TR-SBP) consists of a finite set of finite series of activities that have timing and resource constraints. A TR-SBP seems simple, but its analysis needs to consider what are not explicitly mentioned as activities and may introduce a non-negligible number of intermediate states. In this sense, the analysis has similarities with security protocol analysis that needs to consider intruders. We formalize TR-SBPs as a round-based model called Formal TR-SBPs in this paper. We describe how to specify Formal TR-SBPs in Maude, a specification language based on rewriting logic and Alloy, a specification language based on first-order relational logic, and how to analyze Formal TR-SBPs based on those specifications with Maude and Alloy. A nursing problem is used as an example of TR-SBPs in this paper.

1. Introduction

A time- and resource-sensitive simple business process (TR-SBP) consists of a finite set of finite sequences of activities with timing and resource constraints. In the nursing domain, an example of activities is "for a patient to do a rehabilitation" (doRehab); doRehab requires one nurse and it takes between 30 and 40 minutes to perform doRehab. TR-SBP seems simple in that it does not explicitly have any complex control structures, such as loops. It is necessary, however, to take into account what are not explicitly mentioned as activities and may introduce a non-negligible number of intermediate states between two consecutive time units so as to analyze a TR-SBP because we need to consider situations in which enough resources are not available for activities. In this sense, analysis of TR-SBPs has some similarities with analysis of security protocols (Lowe, 1996) because the latter needs to think about intruders that are not explicitly mentioned in the informal description of security protocols and may introduce a non-negligible number of states.

This paper formalizes TR-SBPs as a round-based model (Charron-Bost and Schiper, 2009; Dwork et al., 1988), and the formalized TR-SBPs are called Formal TR-SBPs. Round-based models are semi-synchronous state machines in which it takes one unit of time to complete one round and each active entity performs some designated atomic actions in one round but the order in which those active entities perform atomic actions is not relevant. Round-based models are used to formalize distributed algorithms, such as consensus algorithms. We describe how to specify Formal TR-SBPs in Maude (Clavel et al., 2007) and Alloy (Jackson, 2012). Maude (maude.cs.illinois.edu) is a specification language based on rewriting logic, and Alloy (alloy.mit.edu) is a specification language based on first-order relational logic. Maude is equipped with several tools with which specifications written in Maude can be analyzed. One

☆ This work was partially supported by JSPS Kakenhi 23220002, 26240008 and 26540024, and is an extended and revised version of the paper (Ogata et al., 2015) published at DCIT 2015.

* *Corresponding author.* Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Nomi, Ishikawa, 923-1292, Japan. Tel.: +81-761-51-1211; fax: +81-761-1149.
E-mail address: ogata@jaist.ac.jp (K. Ogata).
† He is currently with Fujitsu, Ltd.

http://dx.doi.org/10.1016/j.jisa.2016.05.001
2214-2126/

of them is a reachability analysis tool of state machines that is provided as the search command. Specifications written in Alloy can be analyzed with state-of-the-art SAT solvers, such as MiniSAT (minisat.se). A nursing problem (Watahiki, 2013) is used as an example of TR-SBPs in this paper. All experiments reported in the paper were conducted on a computer with 3.4 GHz processor and 32 GB memory.

The rest of the paper is organized as follows. §2 uses a mutual exclusion protocol as an example to outline Maude and Alloy. §3 uses the nursing problem to describe TR-SBPs. §4 formalizes TR-SBPs and §5 describes how to analyze Formal TR-SBPs. §6 describes how to specify and analyze Formal TR-SBPs using Maude and Alloy. §7 mentions related work. §8 concludes the paper.

2. Preliminaries

Let us consider a flawed mutual exclusion protocol (FMUTEX) in which two threads are involved as an example. The pseudo code executed by each thread i (where $i = 1, 2$) is as follows:

$$
\begin{aligned}
&\mathrm{L}i1: \text{ "Remainder Section"}\\
&\mathrm{L}i2: \textbf{ if } \mathit{locked} = \mathrm{True}\\
&\mathrm{L}i3: \textbf{ then goto } \mathrm{L}i2;\\
&\mathrm{L}i4: \textbf{ else } \mathit{locked} := \mathrm{True}; \textbf{ fi}\\
&\mathrm{CS}i: \text{ "Critical Section"}\\
&\mathrm{L}i5: \mathit{locked} := \mathrm{False};\\
&\mathrm{L}i6: \textbf{ goto } \mathrm{L}i1;
\end{aligned}
$$

Each thread does not need any shared resources in Remainder Section, while it needs some shared resources that should be used mutually exclusively in Critical Section. The two threads share the variable *locked* whose value is either True or False. Initially, each thread i is in Remainder Section (or at the label Li1), and *locked* is False. When each thread i wants to use some shared resources, it moves to Li2, and if *locked* is False, then the thread i moves to Li4 and sets *locked* to True,entering Critical Section. Otherwise, the thread i moves to Li3 and goes back to Li2, checking *locked* again. When the thread i leaves Critical Section, it sets *locked* back to True and goes back to Remainder Section. Note that *locked* is used in neither Remainder Section nor Critical Section.

FMUTEX is used as an example to briefly describe Maude and Alloy.

2.1. *Maude*

A specification written in Maude consists of declarations of sorts, sub-sort relations, variables, operators, equations and rewrite rules. Sorts, sub-sort relations and operators are interpreted as sets, sub-set relations and functions over those sets, respectively. Operators and variables are used to construct terms. Operators may have no arguments, and such operators are called constants and those themselves are terms. Variables are expressed as symbols that only consists of capital letters and written in italics in this paper. Operators may be constructors that constitute data values. Equations specify that two terms are equal. Rewrite rules specify state transitions of state machines whose states are expressed as terms.

For FMUTEX, we first prepare the two sorts Label and Value, and the 16 constants L11, … CS1, L15, L16, L21, …, CS2, L25, L26 of Label and the two constants True and False of Value. All of the constants are constructors. We then prepare the sort OComp and the following three constructors of OComp:

$$
\begin{aligned}
&\textbf{op } (\mathrm{locked} : \_) : \mathrm{Value} \to \mathrm{OComp}\ [\textbf{ctor}] .\\
&\textbf{op } (\mathrm{pc1} : \_) : \mathrm{Label} \to \mathrm{OComp}\ [\textbf{ctor}] .\\
&\textbf{op } (\mathrm{pc2} : \_) : \mathrm{Label} \to \mathrm{OComp}\ [\textbf{ctor}] .
\end{aligned}
$$

where **op** is used to declare operators, **ctor** specifies that the operator is a constructor, and an underscore _ is the place where an argument is put. The term $(\mathrm{locked} : v)$, where v is one of True and False, is used to specify that the value of the variable *locked* is v, and the term $(\mathrm{pc}i : l)$, where l is one of Li1, …, CSi, Li5, Li6, is used to specify that the location of the thread i is l.

States of the state machine formalizing FMUTEX are expressed as soups (associative and commutative collections) of $(\mathrm{locked} : v)$, $(\mathrm{pc1} : l_1)$ and $(\mathrm{pc2} : l_2)$. The sort, sub-sort relation and constructors for such soups are as follows:

$$
\begin{aligned}
&\textbf{sort } \mathrm{OCSoup} .\\
&\textbf{subsort } \mathrm{OComp} < \mathrm{OCSoup} .\\
&\textbf{op } \mathrm{void} : \to \mathrm{OCSoup}\ [\textbf{ctor}] .\\
&\textbf{op } \_\_ : \mathrm{OCSoup}\ \mathrm{OCSoup} \to \mathrm{OCSoup}\ [\textbf{ctor assoc comm id} : \mathrm{void}] .
\end{aligned}
$$

where **sort** is used to specify sorts, **subsort** is used to specify sub-sort relations, **assoc** is used to specify that the operator is associative, **comm** is used to specify that the operator is commutative, and **id** : void is used to specify that void is an identity of the operator. The sub-sort declaration specifies that OComp

Please cite this article in press as: Kazuhiro Ogata, Thapana Chaimanont, Min Zhang, Formal modeling and analysis of time- and resource-sensitive simple business processes, Journal of Information Security and Applications (2016), doi: 10.1016/j.jisa.2016.05.001