Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 65 (2007) 14–29

Science of Computer Programming

www.elsevier.com/locate/scico

Modelling and model checking suspendible business processes via statechart diagrams and CSP

W.L. Yeung[a,*], K.R.P.H. Leung[b], Ji Wang[c], Wei Dong[c]

[a] *Department of Computing and Decision Sciences, Lingnan University, Hong Kong*
[b] *Department of Information and Communications Technology, Hong Kong Institute of Vocational Education, Hong Kong*
[c] *National Laboratory for Parallel and Distributed Processing, Changsha, Hunan, PR China*

Received 15 April 2006; received in revised form 1 July 2006; accepted 1 August 2006
Available online 1 December 2006

Abstract

When modelling object behaviour with UML statechart diagrams, the history mechanism can be useful for modelling the suspension of a "normal" business process upon certain "abnormal" events together with the subsequent resumption, as illustrated by the examples in this paper. However, previous approaches to model checking statechart diagrams often ignore the history mechanism. We enhanced such a previous approach based on Communicating Sequential Processes (CSP) and developed a support tool for it.

Keywords: Statechart diagrams; History mechanism; Object behaviour; Process modelling; Model checking

1. Introduction

An essential task of modelling business activities is to identify the different types of business transactions and the order in which they are conducted. For this there are two inter-related modelling concepts. The first one is an *object*, which reflects how business transactions are related to and distinguished from each other in an *information system*, e.g. two borrowing transactions conducted in a library may differ in the book objects being borrowed; a borrowing transaction is related to a returning transaction if they are applied to the same book object. The second modelling concept is a *process*, which is the order in which related transactions are carried out against/by a particular object throughout its lifetime, e.g. a library membership object must first be created and *then* may be renewed a number of times *before* it is finally cancelled.

Process modelling has always been a challenging task. Poorly modelled processes lead ultimately to information systems that handle business transactions incorrectly or inflexibly, and hence to inaccurate/incomplete information for managers. The problem is elevated to a larger scale in enterprise information systems which involve unprecedented numbers of objects and processes covering every aspect of business operations nowadays.

* Corresponding author. Tel.: +852 2616 8095; fax: +852 2892 2442.
E-mail address: wlyeung@ln.edu.hk (W.L. Yeung).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.08.007

Much research has been carried out on proposing, refining, extending, and integrating languages and notations for specifying processes, with the aim of ensuring correctness and making the task more efficient and manageable. The Unified Modelling Language (UML) [1] settles on *statechart diagrams*, which is an adaptation of Harel's statecharts [2]. UML also includes a kind of diagram known as activity diagrams, which are mainly used for modelling workflows but may also provide an alternative view of processes. This paper is, however, only concerned with statechart diagrams.

A statechart diagram represents a process in terms of *states* and *transitions*; transitions among states are triggered by *events* that correspond to occurrences of business transactions. A process represented in such a way is traditionally called a finite state machine (FSM). Compared with the traditional representation of FSM's (i.e. state transition diagrams), statechart diagrams are incorporated with features such as composite states and concurrent states that, on the one hand, facilitate the modelling of complex processes in a more succinct and manageable manner while, on the other, present new challenges for the verification task due to a much richer semantics.

Formal methods are widely recognised as a useful means of increasing software reliability. The formal verification of statechart diagrams has been an active research topic ever since UML was proposed. The syntax and semantics of statechart diagrams have been formalised in various ways and tools have been developed for supporting their automated verification through model checking.

The result of our research presented in this paper enhances a previous approach to formalising and model checking statechart diagrams [3,4]. This approach is based on a mapping of statechart diagrams into the formalism of Communicating Sequential Processes (CSP) [5,6] together with the associated FDR2 model checking tool [7]. This approach, together with many others (e.g. [8–10]), does not, however, support the *history mechanism*, which is often useful in modelling business activities that involve *suspension*. The main enhancement presented here is the support of the history mechanism for modelling *suspendible* business processes as well as the development of a support tool for the mapping.

The next section briefly outlines the previous approach on which this work is based. Section 3 discusses modelling of suspendible business processes in statechart diagrams. Section 4 explains the representation of suspendible processes in CSP via a mapping from statechart diagrams. Section 5 introduces some software tools that support the application of model checking. Section 6 reviews some related work. Section 7 evaluates the approach and discusses limitations of the tools. Section 8 concludes the paper with some indications for further work.

2. A CSP approach to formalising statechart diagrams

In this section, an approach to formalising statechart diagrams originally proposed by Ng and Butler [3] and subsequently improved by Yeung et al. [4] is briefly reviewed. The reader is referred to [3,4] for details.

2.1. The language of CSP

In the language of CSP, a process is described in terms of the possible interactions it can have with its environment, which may be thought of as another process or set of processes. Interactions are described in terms of instantaneous atomic synchronisations, or *events*. A process can be considered as a "black box" with an interface containing a number of events through which it interacts with other processes. The set of all events in the interface of a process P, written αP, is called its *alphabet*. It is important to note that interface events are intended as synchronisations between the participating processes and not as autonomous actions under the control of a single process.

The following paragraphs briefly introduce the CSP operators used in this paper. A comprehensive description of the language is found in [5,6]. The language of CSP used in this paper is defined by the following pseudo Backus–Naur form definition:

$$\mathrm{P} ::= \mathit{STOP} \mid a \to \mathrm{P} \mid a : A \to P_a \mid \mathrm{P} \mathbin{\Box} \mathrm{P} \mid \mathit{SKIP} \mid \mathrm{P} \mathbin{;} \mathrm{P} \mid \mathrm{P} \underset{A}{\parallel} \mathrm{P} \mid \mathrm{P} \setminus A$$

where Σ is the set of all possible events, a ranges over Σ, and $A \subseteq \Sigma$.

Let a and b be events and P, Q, and R be CSP processes. The process *STOP* is the deadlocked process, unable to engage in any events or make any progress. The prefix process $a \to P$ is ready to engage in event a (and in no other event). It will continue to wait until its environment is also ready to perform a, at which point synchronisation on this event will occur. Once the event is performed, the subsequent behaviour of $a \to P$ will be that of process P.

The prefix choice $a : A \rightarrow P_a$ remains willing to perform any event from set A until one is chosen. Its subsequent behaviour, described by P_a, is dependent on that event. A construct can be defined to allow the input on channel *in* of any item x in a set M, and the value x determines the subsequent behaviour:

$$in?x : M \rightarrow Q(x) \mathrel{\widehat{=}} a : in.M \rightarrow P_a$$

where the set $in.M = \{in.m \mid m \in M\}$ and $P_{in.m} = Q(m)$ for every $m \in M$. The atomic synchronisation events here are of the form $in.m$. The complement is the output prefix which has the form $out!x \rightarrow P$ and this is simply a shorthand for $out.x \rightarrow P$. Furthermore, given channels $in_1, in_2, \dots$ defined over message types $M_1, M_2, \dots$, $\{\!|in_1, in_2, \dots|\!\}$ is short for the union of the sets $in_1.M_1, in_2.M_2, \dots$.

Given two processes P and Q, an external choice $P \mathbin{\Box} Q$ is initially ready to engage in events that either P or Q is ready to engage in. The first event performed resolves the choice in favour of the component that was able to perform it, and the subsequent behaviour is given by this component. *SKIP* is the process that does nothing but terminates successfully. In the sequential composition $P; Q$, the combined process first behaves as P and Q becomes active immediately after the successful termination of P. $P \mathbin{\underset{A}{\|}} Q$ is the parallel composition of P and Q in which the two processes must synchronise on events in the set A. For instance,

$$(a \rightarrow P \mathbin{\Box} b \rightarrow Q) \mathbin{\underset{\{a\}}{\|}} (a \rightarrow R) = a \rightarrow \left(P \mathbin{\underset{\{a\}}{\|}} R\right).$$

Finally, in $P \setminus A$, P's ability to synchronise with the environment on any event $a \in A$ is disabled, with all such events taking place internally (hidden) as soon as they are ready.

With the parallel and hiding operators, we can "simulate" the execution of coroutines with concurrent processes that synchronise with each other at the points of coroutine resumption. In Section 3, we shall explain the history mechanism of UML statechart diagrams in terms of coroutine resumption.

The following processes simulate two coroutines that take turn to output messages on channel *out*:

$$\begin{aligned} L &= out!\text{“LEFT”} \rightarrow \textcircled{r}.R.L \rightarrow \textcircled{r}.L.R \rightarrow L \\ R &= \textcircled{r}.R.L \rightarrow out!\text{“RIGHT”} \rightarrow \textcircled{r}.L.R \rightarrow R \\ COROUTINES &= \left(L \mathbin{\underset{\{\!|\textcircled{r}|\!\}}{\|}} R\right) \setminus \{\!|\textcircled{r}|\!\} \end{aligned}$$

which generate a (unbounded) sequence of "LEFT" and "RIGHT" alternately. The $\textcircled{r}$-prefix events act as synchronisations that ensure alternate execution of coroutines.

Given two CSP processes P and Q, P is a traces-refinement of Q, written as $Q \sqsubseteq_t P$, if any trace of events that happen in P can also happen in Q. For instance, if $Q = a \rightarrow Q \mathbin{\Box} b \rightarrow Q$ and $P = a \rightarrow P$, we have:

$$Q \sqsubseteq_t P$$

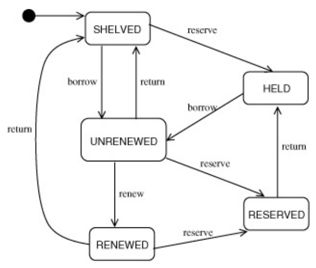
since Q does allow for an indefinite sequence of a's, as happens in P (although Q also allows other sequences involving both a's and b's).

2.2. UML statechart diagrams

Fig. 1 shows a statechart diagram for a book object in a library circulation record system. For now, assume that there are only four types of transactions, namely, borrow, return, renew, and reserve, with corresponding events as seen in the diagram. The reader may notice the following properties about the order of events from the diagram:

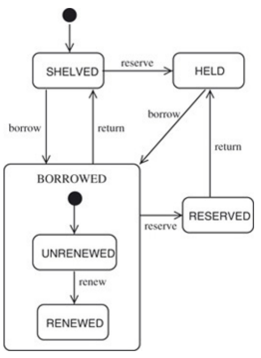
(i) A borrowed book may be renewed at most once.
(ii) Once reserved, a borrowed book can no longer be renewed.
(iii) Once a book is reserved, it cannot be reserved again until after it has been borrowed.
(vi) A book on shelf can also be reserved (once) before it is borrowed.

Fig. 1 also shows the CSP representation of the same process. The overall process is defined in CSP as *BOOK*, which is in turn defined by a set of mutual-recursively defined CSP processes, each of which corresponds to an individual state in the statechart diagram and represents the behaviour of the book object starting from that particular state. For



$$\begin{aligned}
HELD &\mathrel{\widehat{=}} borrow \rightarrow UNRENEWED \\
RESERVED &\mathrel{\widehat{=}} return \rightarrow HELD \\
RENEWED &\mathrel{\widehat{=}} return \rightarrow SHELVED \,\square\, reserve \rightarrow RESERVED \\
UNRENEWED &\mathrel{\widehat{=}} return \rightarrow SHELVED \,\square\, renew \rightarrow RENEWED \\
&\quad \square\, reserve \rightarrow RESERVED \\
SHELVED &\mathrel{\widehat{=}} borrow \rightarrow UNRENEWED \,\square\, reserve \rightarrow HELD \\
BOOK &\mathrel{\widehat{=}} SHELVED
\end{aligned}$$

Fig. 1. Statechart diagram for a book object.



$$\begin{aligned}
HELD &\mathrel{\widehat{=}} borrow \rightarrow BORROWED && (1) \\
RESERVED &\mathrel{\widehat{=}} return \rightarrow HELD \\
RENEWED &\mathrel{\widehat{=}} return \rightarrow SHELVED \,\square\, reserve \rightarrow RESERVED \\
UNRENEWED &\mathrel{\widehat{=}} return \rightarrow SHELVED \,\square\, renew \rightarrow RENEWED \\
&\quad \square\, reserve \rightarrow RESERVED \\
BORROWED &\mathrel{\widehat{=}} UNRENEWED && (2) \\
SHELVED &\mathrel{\widehat{=}} borrow \rightarrow BORROWED \,\square\, reserve \rightarrow HELD && (3) \\
BOOK &\mathrel{\widehat{=}} SHELVED
\end{aligned}$$

Fig. 2. Statechart diagram featuring a composite state for a book object.

instance, since SHELVED is the initial state, the overall behaviour of a book (i.e. *BOOK*) is defined by its behaviour starting from the SHELVED state (i.e. *SHELVED*).

Fig. 2 shows an alternative statechart diagram for the book object, together with a corresponding CSP representation. The new diagram takes advantage of using a composite state to represent exactly the same process in a slightly more compact manner. The CSP representation has also been revised, with a new process *BORROWED* and the definitions of *HELD* and *SHELVED* changed, to reflect the new diagram's structure. As an assurance that the new and old diagrams both represent the same process, substituting (2) into (1) and (3) gives the original CSP representation.

2.3. Mapping statechart diagrams into CSP

The CSP processes defined in Figs. 1 and 2 can be systematically derived from the corresponding statechart diagrams through a set of mapping functions [3,4]. For the sake of clarity and limitation of space, we present in this section a simplified version of the mapping functions which support the main features of interest here, namely composite states and inter-level transitions. The full versions (see [3,4]) support other major statechart features including final states, unlabelled transitions, entry and exit actions, do-activities, and choice states.

A UML statechart diagram describes a state machine M consisting of:

– a finite set of states, M_s
– a finite set of transitions, T_M
– a finite set of events, E_M

where M_s is partitioned into three disjoint sets: simple states M_{ss}, composite states M_{cs}, and initial states M_{is}. In this paper, we use X, Y, Z as variables of states and t, u, v as variables of transitions.

Hierarchical relationships among states are captured in a strictly anti-symmetric relation $IMM_M : M_s \leftrightarrow M_s$. Given any two states $X, Y \in M_s$, we have $Y \mapsto X \in IMM_M$ if and only if Y is directly (immediately) enclosed in X. Furthermore,

– IMM_M^+, the transitive non-reflexive closure of IMM_M, captures all direct and indirect enclosure relationships among M_s.

- IMM_M^*, the transitive reflexive closure of IMM_M, captures all direct and indirect enclosure relationships among M_s, as well as the identity function.
- We can define a function $ENCL_M : M_s \to \mathbb{P}\, M_s$ that maps a state to the set of all its immediately enclosing states, i.e. $ENCL_M(Y) \mathrel{\widehat{=}} \{X \mid Y \mapsto X \in IMM_M\}$.
- Similarly, we can define the functions $ENCL_M^+$ and $ENCL_M^*$ based on the closure versions of IMM_M, i.e. $ENCL_M^+(Y) \mathrel{\widehat{=}} \{X \mid Y \mapsto X \in IMM_M^+\}$ and $ENCL_M^*(Y) \mathrel{\widehat{=}} \{X | Y \mapsto X \in IMM_M^*\}$.

The function $label_M : M_s \rightarrowtail \mathcal{L}_M$ associates each state to its label, where $\mathcal{L}_M$ is the set of state labels (names). The following functions associate each transition $t \in T_M$ to its source state, target state, and trigger event: (There are four kinds of event in UML: signal event, call event, change event, and time event. Here, we restrict our attention to only the first two kinds of event.)

- $source_M : T_M \to M_s$
- $target_M : T_M \to M_s$
- $event_M : T_M \nrightarrow E_M$.

Note that elements of $\mathcal{L}_M$ and E_M are included in the set of events Σ for defining CSP processes. Also note that a transition may have no trigger event if it originates from an initial state. Furthermore, we define

- $O_M : M_s \nrightarrow \mathbb{P}\, T_M$ as a function which maps a state $X \in M_s$ to the set of outgoing transitions emanating from X, i.e. $O_M(X) \mathrel{\widehat{=}} \{t \mid source_M(t) = X\}$
- $O_M^*(X) \mathrel{\widehat{=}} \bigcup_{Y \in ENCL_M^*(X)} \left\{ t \,\middle|\, \begin{array}{l} t \in O_M(Y) \wedge \\ \forall u \in \bigcup_{Z \mapsto Y \in IMM_M^+} O_M(Z) \bullet event_M(t) \neq event_M(u) \end{array} \right\}$ as the set of outgoing transitions emanating from X as well as all (immediately and transitively) enclosing states of X. In case a outer transition has the same trigger as an inner transition, *it is the inner transition that has the priority* according to the semantics of UML [1].

Given a finite state machine M represented in a statechart diagram, we can define a function

$$\mathcal{H}_M : M_s \nrightarrow \text{CSP}$$

where M_s is the set of states of machine M. For a state $X \in M_s$, $\mathcal{H}_M(X)$ is the CSP process that represents the (subsequent) behaviour of machine M starting at state X. $\mathcal{H}_M$ is defined in the following based on the structure of the set M_s.

Definition 1 (*Initial State*). Given any initial state $X \in M_{\text{is}}$ and $t \in O_M(X)$,

$$\mathcal{H}_M(X) = \mathcal{H}_M(target_M(t)).$$

An initial state has a single outgoing transition. For simplicity, we assume that every composite state has at most one initial state in its immediate enclosure and does not have a trigger event. The behaviour of the state machine following the transition is defined as that of the transition's target.

Definition 2 (*Simple State*). Given $X \in M_{\text{ss}}$

$$\mathcal{H}_M(X) = \mathop{\Box}_{t \in O_M^*(X)} event_M(t) \to \mathcal{H}_M(target_M(t)).$$

A simple state $X \in M_{\text{ss}}$ may have one or more outgoing transitions, each of which has a trigger event.[1] In addition, any outgoing event-triggered transition of any (immediately or transitively) enclosing composite state also applies to the simple state.

Definition 3 (*Composite State*). Given $X \in M_{\text{cs}}$ such that $\exists t \in T_M \bullet target_M(t) = X$ and $Y \in M_{\text{is}}$ such that $IMM_M(Y) = X$

$$\mathcal{H}_M(X) = \mathcal{H}_M(Y).$$

[1] Only call events and signal events are assumed here.

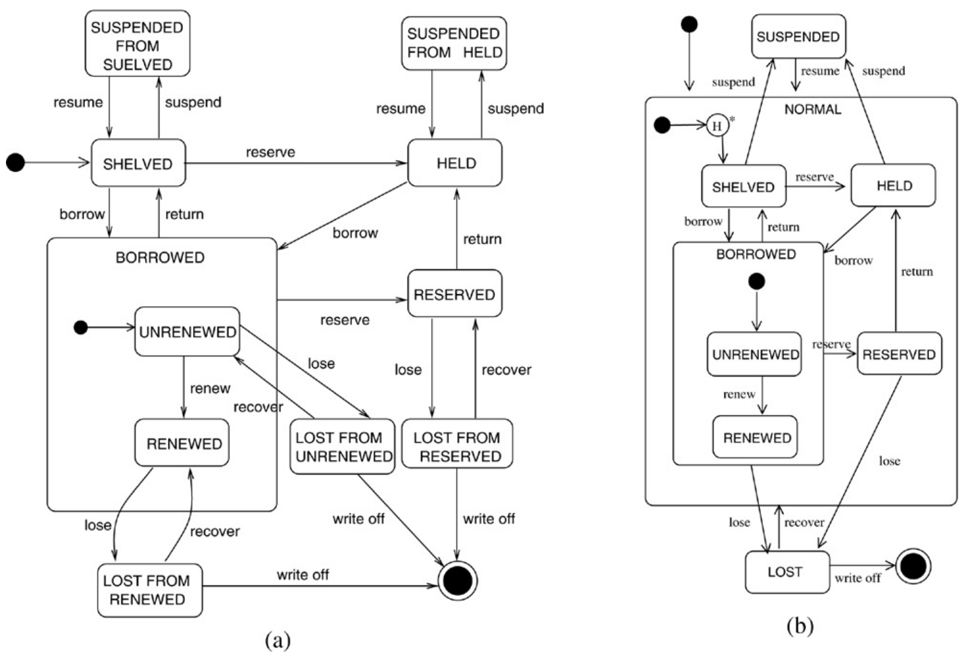


Fig. 3. Statechart diagrams for a suspendible book process.

If a composite state is the target of any incoming transition(s), it must have an initial state in its immediate enclosure. In such cases, the composite state is equivalent to the initial state. Note that if a composite state is not the target of any incoming transition, it does not correspond to any CSP process.

As an example, let M be the FSM represented by the statechart diagram in Fig. 2, we have
$M_s = \{SHELVED, BORROWED, UNRENEWED, RENEWED, RESERVED, HELD\}$

$$\begin{aligned}
\mathcal{H}_M(HELD) &\mathrel{\widehat{=}} borrow \rightarrow \mathcal{H}_M(BORROWED)\\
\mathcal{H}_M(RESERVED) &\mathrel{\widehat{=}} return \rightarrow \mathcal{H}_M(HELD)\\
\mathcal{H}_M(RENEWED) &\mathrel{\widehat{=}} return \rightarrow \mathcal{H}_M(SHELVED) \mathrel{\Box} reserve \rightarrow \mathcal{H}_M(RESERVED)\\
\mathcal{H}_M(UNRENEWED) &\mathrel{\widehat{=}} return \rightarrow \mathcal{H}_M(SHELVED) \mathrel{\Box} renew \rightarrow \mathcal{H}_M(RENEWED)\\
&\qquad \Box\ reserve \rightarrow \mathcal{H}_M(RESERVED)\\
\mathcal{H}_M(BORROWED) &\mathrel{\widehat{=}} \mathcal{H}_M(UNRENEWED)\\
\mathcal{H}_M(SHELVED) &\mathrel{\widehat{=}} borrow \rightarrow \mathcal{H}_M(BORROWED) \mathrel{\Box} reserve \rightarrow \mathcal{H}_M(HELD).
\end{aligned}$$

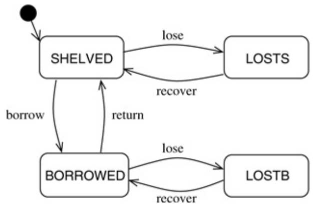
As SHELVED is the initial of the entire FSM, $\mathcal{H}_M(SHELVED)$ as defined above represents the process of a book object, which is equivalent to the *BOOK* process defined in Fig. 2.

3. Suspendible business processes

Fig. 3(a) shows yet another statechart diagram for the book object. The new diagram represents a process involving five additional types of transaction, namely, suspend, resume, lose, recover, and write-off. The library may suspend a book from circulation (for maintenance purposes) when a book is either being SHELVED or HELD and subsequently resume it to its last state when suspended. On the other hand, a book on loan could be reported as lost by the borrower and be effectively suspended from circulation. If a lost book is subsequently reported as recovered, it will be resumed to the last state when reported as lost. A lost book may never be recovered and eventually written off.

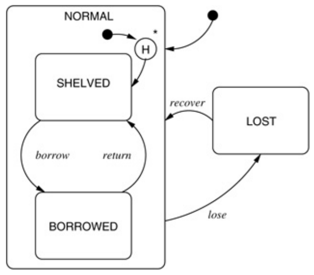
For the sake of discussion, we refer to the additional states (SUSPENDED FROM SHELVED/HELD, LOST FROM UNRENEWED/RENEWED/RESERVED) as "abnormal" states whereas the pre-existing states are "normal" states.

Fig. 3(b) shows an alternative statechart diagram for the *same* suspendible book process. The new diagram takes advantage of using the *history* mechanism (Ⓗ*), which is best explained by comparing the alternative statechart



$$\begin{aligned}
LOSTB &\mathrel{\widehat=} recover \to BORROWED\\
LOSTS &\mathrel{\widehat=} recover \to SHELVED\\
BORROWED &\mathrel{\widehat=} return \to SHELVED \mathbin{\Box} lose \to LOSTB\\
SHELVED &\mathrel{\widehat=} borrow \to BORROWED \mathbin{\Box} lose \to LOSTS\\
BOOK &\mathrel{\widehat=} SHELVED
\end{aligned}$$

Fig. 4. Statechart diagram for a simplified book object (non-history version).



$$\begin{aligned}
LOST(x) &\mathrel{\widehat=} recover \to \text{if } x = s \text{ then } SHELVED\\
&\qquad\qquad\qquad \text{else } BORROWED\\
BORROWED &\mathrel{\widehat=} return \to SHELVED \mathbin{\Box} lose \to LOST(b)\\
SHELVED &\mathrel{\widehat=} borrow \to BORROWED \mathbin{\Box} lose \to LOST(s)\\
NORMAL &\mathrel{\widehat=} SHELVED\\
BOOK &\mathrel{\widehat=} NORMAL
\end{aligned}$$

Fig. 5. Statechart diagram for a simplified book object (history version).

diagram against the original statechart diagram in Fig. 3(a). Whenever a transition from one of the normal states (which are now altogether enclosed within the NORMAL composite state) to an abnormal state is triggered, the state machine "remembers" the particular normal state and then when the NORMAL composite state is activated again through its history ($\bullet\!\!\longrightarrow\!\text{Ⓗ}^*$), the last remembered state becomes the active state.

Observe that the NORMAL state is only entered through its history. While this is not compulsory, we shall take advantage of such cases in deriving the CSP representation as explained in the following section.

Comparing the two statechart diagrams in Fig. 3, we can see that:

(i) The number of abnormal states is reduced from five to two.
(ii) The number of transitions from abnormal states (back) to normal states is also reduced from five to two.
(iii) All the normal states are now enclosed in a single composite state (NORMAL).
(iv) The new statechart diagram is visually more compact.
(v) The new statechart diagram describes the same suspendible business process.

The advantages of using the history mechanism come at the expense of a more complicated semantic model. This will become evident when we extend our mapping of statechart diagrams into CSP to handle the history mechanism.

4. Modelling suspendible processes in CSP

To help explain the modelling of suspendible processes in CSP, we first consider a simplified version of the book object. Figs. 4 and 5 show two equivalent statechart diagrams for the book object with only two "normal" states, SHELVED and BORROWED. Fig. 4 does not use any history whereas Fig. 5 does. Transitions between the two NORMAL states are triggered by the *borrow* and *return* events. Apart from the "normal" process, the book may be reported as lost at any time and then later recovered again. The non-history version (Fig. 4) shows explicitly how *lose* and *recover* affect the state of a book object. In the history version (Fig. 5), a *lose* event would trigger a transition to the LOST state, from which a *recover* event would trigger a transition back to the last active substate within the NORMAL state through its history.

Figs. 4 and 5 also show the corresponding CSP representations. We are interested in the history version in Fig. 5. *LOST* involves a parameter in its definition. It responds to the *recover* event and then chooses between *BORROWED* and *SHELVED* based on the value of the parameter, which is set by *BORROWED* and *SHELVED* each time when they "call" *LOST*. While the mapping in this example seems to work well and does maintain a correspondence between states and processes, there is a serious drawback: history information about BORROWED and SHELVED are not only handled by the corresponding CSP processes (*BORROWED* and *SHELVED*) only, but also *LOST* which makes use of such history information—this can be considered as contrary to the principle of *information hiding* [11].

The implication is that any state with a transition directly or indirectly leading to the history indicator of another (composite) state has to carry history information about the latter.

An alternative approach to handling history information that respects the principle of information hiding is illustrated by the following CSP description of the same diagram in Fig. 4:

$$\begin{aligned}
\mathit{LOST} &= ⓡ.\mathit{LOST}.\mathit{NORMAL} \rightarrow \mathit{recover} \rightarrow ⓡ.\mathit{NORMAL}.\mathit{LOST} \rightarrow \mathit{LOST}\\
\mathit{BORROWED} &= \mathit{return} \rightarrow \mathit{SHELVED}\\
&\quad \Box\ \mathit{lose} \rightarrow ⓡ.\mathit{LOST}.\mathit{NORMAL} \rightarrow ⓡ.\mathit{NORMAL}.\mathit{LOST} \rightarrow \mathit{BORROWED}\\
\mathit{SHELVED} &= \mathit{borrow} \rightarrow \mathit{BORROWED}\\
&\quad \Box\ \mathit{lose} \rightarrow ⓡ.\mathit{LOST}.\mathit{NORMAL} \rightarrow ⓡ.\mathit{NORMAL}.\mathit{LOST} \rightarrow \mathit{SHELVED}\\
\mathit{NORMAL} &= \mathit{SHELVED}\\
\mathit{BOOK} &= \left(\mathit{LOST} \underset{\{\!|ⓡ|\!\}}{\parallel} \mathit{NORMAL}\right) \setminus \{\!|ⓡ|\!\}.
\end{aligned}$$

There are two concurrent processes, *LOST* and *NORMAL*, whose behaviour resemble two coroutines as discussed in Section 2.1. The two concurrent processes synchronise with each other on two special events, ⓡ.*LOST.NORMAL* and ⓡ.*NORMAL.LOST*, which correspond to transitions to the LOST and NORMAL states, respectively. For instance, after an *recover* event, *LOST* is ready for a ⓡ.*NORMAL.LOST* event, which corresponds to a transition to the NORMAL state. Since the transition is meant to take place automatically following the trigger event, the ⓡ.*NORMAL.LOST* event, together with the ⓡ.*LOST.NORMAL*, are designated as internal and hidden from the environment using the "\" operator.

4.1. Mapping functions

The mapping functions defined in Section 2.3 need to be revised to handle the use of history. First of all, we impose the following the restrictions on the use of history:

(i) Incoming transitions do not "penetrate" inside a history-bearing composite state—they stop at the boundary. In other words, a history-bearing composite state must always be entered through its history, i.e. "no history override".
(ii) A history-bearing composite state remembers its last active substate at any level of its enclosure, i.e. "no shallow history".
(iii) A history-bearing composite state may not contain any other history-bearing composite states at any level of its enclosure, i.e. "no nested history".

The above restrictions on the use of history-bearing composite states render them behaviourally analogous to *coroutines*: an initial transition to a history-bearing composite state corresponds to a call to a coroutine; transitions among substates of the history-bearing composite state correspond to the coroutine's internal state changes; transitions out of the history-bearing composite state correspond to suspending the coroutine and transferring control away to other coroutines or the main program; incoming transitions resume the coroutine.

In this paper, a suspendible business process is considered as having a "normal" life with some "normal" events, together with some "abnormal" events that could suspend the normal life temporarily. The stated restrictions are least relevant when the "normal" life is always resumed *at the point of last suspension* as in our book example. In those cases where the "normal" life is *not always* resumed at the point of last suspension, restrictions (i) and (ii) would undermine the use of deep history. For instance, in the book example, if a suspended book may also be replaced (apart from being resumed) and we assume that a replaced book is always returned to the shelf, we may modify the statechart diagram in Fig. 3(b) by adding a transition from SUSPENDED to SHELVED, triggered by a *replace* event. However, this violates restriction (i). To get round this, one has to give up (or reduce) the use of history and resort to the less compact style exemplified in Fig. 3(a).

The full definition of the revised mapping functions that handles history can be found in [12]; the following is a simplified version.

Given a state machine M, we further define M_{hcs} as the set of history-bearing composite states (HCS's) such that $M_{\text{hcs}} \subseteq M_{\text{cs}} \subset M_s$. We also define the set M_{hs} which contains all the HCS's and their substates (excluding initial states), i.e.

$$M_{\text{hs}} \mathrel{\widehat{=}} \{X \mid \exists\, Y \in M_{\text{hcs}} \bullet X \mapsto Y \in \mathit{IMM}^*_M\} \setminus M_{\text{is}}.$$

For the convenience of our definition, the entire state machine is regarded as being enclosed in a *top-level* composite state $S0_M$ such that $\forall\, X \in M_s \bullet X \mapsto S0_M \in \mathit{IMM}^*_M$. For any state within a history-bearing composite state (HCS), the following (recursive) function returns the enclosing HCS:

$$H_M(X) \mathrel{\widehat{=}} \text{if } X \in M_{\text{hcs}} \text{ then } X \text{ else } H_M(\mathit{IMM}_M(X)).$$

Note that if $X \in M_{\text{hcs}}$, we have $H_M(X) = X$.

The special $\textcircled{r}$-prefix events that we have seen in the example in the preceding section take the form:

$$\textcircled{r}.\langle \mathit{target}\rangle.\langle \mathit{source}\rangle$$

where $\langle \mathit{target}\rangle$ and $\langle \mathit{source}\rangle$ can each be the label of a HCS. In addition, whenever the target of an $\textcircled{r}$-event is a history-free state (state neither being nor within a HCS), $\langle \mathit{target}\rangle$ would be the label of the history-free state; on the other hand, whenever the source of an $\textcircled{r}$-event is a history-free state, $\langle \mathit{source}\rangle$ would be the label of $S0_M$. Finally, given a set of states $A \subseteq M_s$, we define

$$\mathit{label}^*_M(A) \mathrel{\widehat{=}} \{\mathit{label}_M(X) | X \in A\}.$$

Revised Definition 1 (*Initial State*). Given an initial state $X \in M_{\text{is}}$ and $t \in O_M(X)$, $\mathcal{H}_M(X) = \mathcal{H}_M(\mathit{target}_M(t))$ if and only if

– there are no history-bearing composite states, or
– t targets a history-free state (state neither being or within a history-bearing composite state), or
– X is within, *but not immediately within*, a history-bearing composite state.

If X is not within a history-bearing composite state but t targets a history-bearing composite state $Y \in M_{\text{hcs}}$, then

$$\begin{aligned}
\mathcal{H}_M(X) &= \textcircled{r}.\mathit{label}_M(Y)!\mathit{label}_M(S0_M) \to \mathcal{M}_M \\
\mathcal{M}_M &= \mathop{\Box}\limits_{Z \in M_s \setminus \{S0_M\} \setminus M_{\text{hs}} \setminus M_{\text{is}}} \textcircled{r}.\mathit{label}_M(Z)?x : \mathit{label}^*_M(M_{\text{hcs}}) \to \mathcal{H}_M(Z).
\end{aligned}$$

If X is immediately within a history-bearing composite state $Y \in M_{\text{hcs}}$, then

$$\mathcal{H}_M(X) = \textcircled{r}.\mathit{label}_M(Y)?x : \mathit{label}^*_M(M_{\text{hcs}} \cup \{S0_M\} \setminus \{H_M(X)\}) \to \mathcal{H}_M(\mathit{target}_M(t)).$$

Note that $\mathcal{M}_M$ corresponds to the main program when suspended, waiting for control to be returned from one of the history-bearing composite states (i.e. M_{hcs}) directly to one of history-free states (i.e. $M_s \setminus \{S0_M\} \setminus M_{\text{hs}} \setminus M_{\text{is}}$).

Revised Definition 2 (*Simple State*). Given $X \in M_{\text{ss}}$

$$\begin{aligned}
\mathcal{H}_M(X) &= \mathop{\Box}\limits_{t \in O^*_M(X)} \mathit{event}_M(t) \to \mathcal{R}_M(X, t) \\
\mathcal{R}_M(X, t) &= \text{if } X \notin M_{\text{hs}} \wedge \mathit{target}_M(t) \in M_{\text{hcs}} \text{ then} \\
&\qquad \textcircled{r}.\mathit{label}_M(\mathit{target}_M(t))!\mathit{label}_M(S0_M) \to \mathcal{M}_M \\
&\quad \text{else if } X \in M_{\text{hs}} \wedge \mathit{target}_M(t) \in M_s \setminus \{S0_M\} \setminus M_{\text{hs}} \setminus M_{\text{is}} \text{ then} \\
&\qquad \textcircled{r}.\mathit{label}_M(\mathit{target}_M(t))!\mathit{label}_M(H_M(X)) \to \\
&\qquad\quad \textcircled{r}.\mathit{label}_M(H_M(X))?x : \mathit{label}^*_M(M_{\text{hcs}} \cup \{S0_M\} \setminus \{H_M(X)\}) \to \mathcal{H}_M(X) \\
&\quad \text{else if } X \in M_{\text{hs}} \wedge \mathit{target}_M(t) \in M_{\text{hcs}} \text{ then} \\
&\qquad \text{if } H_M(X) \neq H_M(\mathit{target}_M(t)) \text{ then} \\
&\qquad\quad \textcircled{r}.\mathit{label}_M(\mathit{target}_M(t))!\mathit{label}_M(H_M(X)) \to \\
&\qquad\qquad \textcircled{r}.\mathit{label}_M(H_M(X))?x : \mathit{label}^*_M(M_{\text{hcs}} \cup \{S0_M\} \setminus \{H_M(X)\}) \to \mathcal{H}_M(X)
\end{aligned}$$

```
        else
          $\mathcal{H}_M(X)$
      else
        $\mathcal{H}_M(target_M(t))$
```

For each applicable outgoing transition t of simple state X, $\mathcal{R}_M(X, t)$ corresponds to the resumption mechanism of the coroutine model. Consider the example in Fig. 5, we have:

$$\begin{aligned}
\mathcal{R}_M(\mathit{SHELVED}, \mathit{borrow}) &= \mathcal{H}_M(\mathit{BORROWED}) && (4)\\
\mathcal{R}_M(\mathit{BORROWED}, \mathit{lose}) &= \textcircled{r}.\mathit{LOST}!\mathit{NORMAL} \to \textcircled{r}.\mathit{NORMAL}?x : \{\mathit{BOOK}\}\\
&\quad \to \mathcal{H}_M(\mathit{BORROWED}) && (5)\\
\mathcal{R}_M(\mathit{LOST}, \mathit{recover}) &= \textcircled{r}.\mathit{NORMAL}!\mathit{BOOK} \to \mathcal{M}_M && (6)\\
\mathcal{M}_M &= \underset{Z \in \{\mathit{LOST}\}}{\Box} \textcircled{r}.Z?x : \{\mathit{NORMAL}\} \to \mathcal{H}_M(Z).
\end{aligned}$$

The first case (4) is the simplest one in which the source and target are within the same (history-bearing) composite state. The second case (5) involves a transition from (within) a historical composite state to a history-free state. In terms of coroutine execution, control is transferred from NORMAL to the main program ($\textcircled{r}.\mathit{LOST}!\mathit{NORMAL}$) and then NORMAL waits for control to be returned again ($\textcircled{r}.\mathit{NORMAL}?x : \{\mathit{BOOK}\}$) at which time it is resumed at the state before the last exit. Note that we have assumed the label of the top-level composite state ($S0_M$) as "BOOK".

Case (6) involves a transition from a history-free state to a history-bearing composite state. In terms of coroutine execution, control is transferred from the main program to NORMAL ($\textcircled{r}.\mathit{NORMAL}!\mathit{BOOK}$) and then the main program has to wait.

Revised Definition 3 (*Composite State*). Given $Y \in \{X \mid X \in M_{\text{hcs}} \vee (X \in M_{\text{cs}} \setminus M_{\text{hcs}} \wedge \exists\, t \in T_M \bullet \mathit{target}_M(t) = X)\}$ and $Z \in M_{\text{is}}$ such that $\mathit{IMM}_M(Z) = Y$, we have

$$\mathcal{H}_M(X) = \mathcal{H}_M(Z).$$

A history-bearing composite state must be the target of at least one transition. A non-history-bearing composite state may or may not be the target of any transitions.

Definition 4 (*State Machine*). Given a state machine M with at least one history-bearing composite state,

$$\left(\mathcal{H}_M(S0_M) \underset{\{\!|\textcircled{r}|\!\}}{\|} \prod_{X \in M_{\text{hcs}}}^{M} \mathcal{H}_M(X)\right) \setminus \{\!|\textcircled{r}|\!\}$$

where

$$\prod_{X \in \{X_1, X_2, \ldots, X_n\}}^{M} \mathcal{H}_M(X) =$$

$$\left(\mathcal{H}_M(X_1) \underset{B_1}{\|} \left(\mathcal{H}_M(X_2) \underset{B_2}{\|} (\cdots (\mathcal{H}_M(X_{n-1}) \underset{B_{n-1}}{\|} \mathcal{H}_M(X_n) \setminus B_{n-1}) \cdots)\right) \setminus B_2\right) \setminus B_1$$

and

$$\begin{aligned}
B_1 &= \bigcup_{Z \in \{X_2, \ldots, X_n\}} \{\textcircled{r}.\mathit{label}_M(Z).\mathit{label}_M(X_1), \textcircled{r}.\mathit{label}_M(X_1).\mathit{label}_M(Z)\}\\
B_2 &= \bigcup_{Z \in \{X_3, \ldots, X_n\}} \{\textcircled{r}.\mathit{label}_M(Z).\mathit{label}_M(X_2), \textcircled{r}.\mathit{label}_M(X_2).\mathit{label}_M(Z)\}\\
&\ \ \vdots \qquad\qquad \vdots\\
B_{n-1} &= \bigcup_{Z \in \{X_n\}} \{\textcircled{r}.\mathit{label}_M(Z).\mathit{label}_M(X_{n-1}), \textcircled{r}.\mathit{label}_M(X_{n-1}).\mathit{label}_M(Z)\}.
\end{aligned}$$

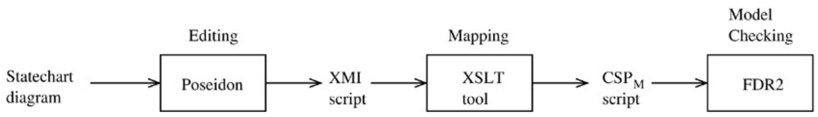


Fig. 6. Software tools for editing, mapping, and model checking statechart diagrams.

A state machine with historical composite state(s) consists of a number of concurrent processes, one for each history-bearing composite state together with one for the overall statechart diagram. Note that the set $\{\!|\textcircled{r}|\!\}$ contains all the $\textcircled{r}$-prefix events.

5. Software tools for model checking

The mapping functions described in the preceding section have been implemented as a software tool for supporting the model checking of suspendible processes represented in UML statechart diagrams. Fig. 6 shows how the tool interfaces with the Poseidon for UML CASE tool (http://www.gentleware.com) and the FDR2 model checker [7], which is the standard model checker for CSP. We use Poseidon as a graphical tool for editing statechart diagrams. For each UML statechart diagram edited, Poseidon saves a textual representation according to the OMG-XML Metadata Interchange (XMI) version 1.2 [13]. Fig. 7 shows fragments of the XMI representation of the statechart diagram in Fig. 3(b).

The software tool itself is written as a set of XSLT templates [14], with each mapping function corresponding to one or more of these templates. The templates (3 files) are available for download at http://cptra.ln.edu.hk/~wlyeung/XSLT.

The XSLT templates are interpreted by the SAXON processor (available for download at http://saxon.sourceforge.net) which takes as input the XMI representation of statechart diagrams from Poseidon and generates CSP$_M$, a machine-readable representation of the CSP language for model checking with FDR2. Fig. 8 shows the CSP$_M$ representation of the statechart diagram in Fig. 3(b) as generated by the software tool.

Model checking has been implemented with symbolic algorithms that can handle systems with more than 10^{20} states [15]. As far as we are concerned here, FDR2 is mainly used for verifying the traces-refinement relationship "$\sqsubseteq_t$" between two CSP processes (see Section 2.1).

Using the aforementioned tools, we can verify suspendible processes expressed using the history mechanism in statechart diagrams through model checking in a number of ways. First, using the book object again as our example, having "simplified" the statechart diagram in Fig. 3(a) to become the one in 3(b) using the history mechanism, we can verify the *equivalence* of the two diagrams by obtaining their CSP$_M$ representation with the tools and checking in FDR2 that the two corresponding CSP processes refine each other. This is shown in the screen-shot in Fig. 9(a) together with some analysis data.

In case they are not equivalent, FDR2 will enter its debugger mode and show a trace of events leading to a behavioural discrepancy between the two processes. Fig. 9(b) shows another screen-shot of using FDR2 in the debugger mode.

Secondly, we can verify the suspendible book process against its non-suspendible version as shown in Fig. 2. There are two methods to do so in FDR2. One method is to check for traces-refinement: every trace of events that can happen in the non-suspendible version may also happen in the suspendible version, but *not vice-versa*. The second method is to check the equivalence between the suspendible and non-suspendible versions with all the "abnormal" events hidden from the former. Both ways of checking try ensure that the suspendible process can at least handle all "normal" transactions in the normal way.

Finally, we can verify a suspendible process, using the two methods just mentioned, against not just its non-suspendible version but also any "abstract" process that *specifies* certain (normal or abnormal) aspect of the business process being modelled. For instance, we may want to verify the suspendible book process against the abstract process represented by the statechart diagram in Fig. 10 regarding only the "suspend", "resume" and "reserve" transactions.

6. Related work

Von der Beeck [16] gave a survey of several variants of statecharts, prior to the arrival of UML statechart diagrams. UML statechart diagrams have been formalised in numerous other formalisms such as pi-calculus [17], LOTOS [18],

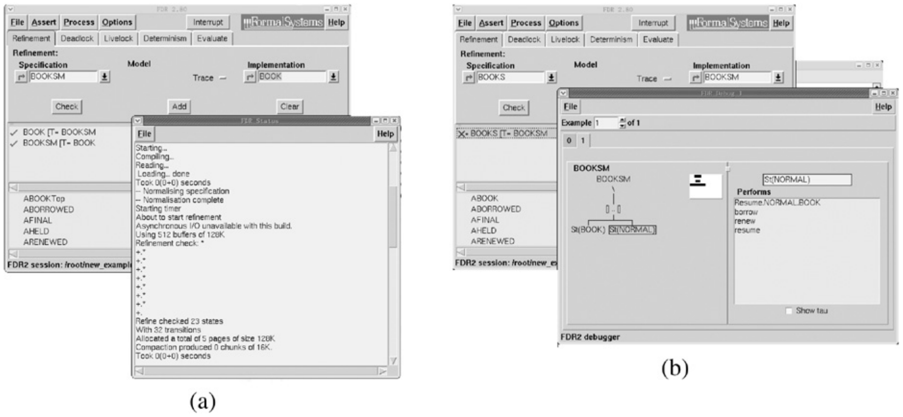
```
<?xml version = '1.0' ...>
<XMI xmi.version = '1.2' ... >
  <XMI.content>
    <UML:Model ...>
      ...
      <UML2:StateMachine xmi.id = '...' name = 'BookSM' ...>
        ...
        <UML2:Region.subvertex>
            Definition of states and state hierarchy, e.g.
            <UML2:Pseudostate xmi.id = '...' name = 'START0'
                kind = 'initial'>
              <UML2:Vertex.outgoing>
                <UML2:Transition xmi.idref = '...'/>
              </UML2:Vertex.outgoing>
            </UML2:Pseudostate>
            <UML2:State xmi.id = '...' name = 'SUSPENDED' ... >
              <UML2:Vertex.outgoing>
                <UML2:Transition xmi.idref = '...'/>
              </UML2:Vertex.outgoing>
              <UML2:Vertex.incoming>
                <UML2:Transition xmi.idref = '...'/>
                <UML2:Transition xmi.idref = '...'/>
              </UML2:Vertex.incoming>
            </UML2:State>
            ...
        </UML2:Region.subvertex>
        <UML2:Region.transition>
          Definition of transitions, e.g.
          <UML2:Transition xmi.id = '...'... >
            <UML2:Transition.source>
              <UML2:Pseudostate xmi.idref = '...'/>
            </UML2:Transition.source>
            <UML2:Transition.target>
              <UML2:State xmi.idref = '...'/>
            </UML2:Transition.target>
          </UML2:Transition>
          ...
        </UML2:Region.transition>
        ...
      </UML2:StateMachine>
      ...
      <UML2:BehavioredClassifier.ownedTrigger>
          Definition of event triggers, e.g.
          <UML2:CallTrigger xmi.id = '...'name = 'resume'.../>
      </UML2:BehavioredClassifier.ownedTrigger>
      ...
    </UML:Model>
  </XMI.content>
</XMI>
```

Fig. 7. Fragments of XMI representation of the statechart diagram in Fig. 3(b).

ASM [19], and EHA [20]. Closest to our work is the work by Ng and Butler [3] in which a formalisation of UML statechart diagrams in CSP is presented in a style that is followed in this paper. Yeung et al. [4] improved this formalisation on inter-level transitions. Most of the previous formalisations do not tackle the history mechanism.

```
channel borrow, lose, recover, renew, reserve, return, suspend, writeoff
datatype States = BOOKTop'| BOOKTop| SUSPENDED | LOST | Final_State_1 | NORMAL |
SHELVED | HELD | RESERVED | BORROWED | UNRENEWED | RENEWED
subtype Process = BOOKTop | NORMAL
subtype THstates = NORMAL channel Resume: States.Process
St(BOOKTop') = Resume?x:diff(States,THstates)?y:THstates -> St(x)
St(BOOKTop) = Resume.NORMAL!BOOKTop -> St(BOOKTop')
St(SUSPENDED) = resume -> Resume.NORMAL!BOOKTop -> St(BOOKTop')
St(NORMAL) = Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(SHELVED)
St(SHELVED) = suspend -> Resume.SUSPENDED!NORMAL ->
Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(SHELVED) [] reserve -> St(HELD) []
borrow -> St(BORROWED)
St(HELD) = suspend -> Resume.SUSPENDED!NORMAL ->
Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(HELD) [] borrow -> St(BORROWED)
St(BORROWED) = St(UNRENEWED)
St(UNRENEWED) = renew -> St(RENEWED) [] return -> St(SHELVED) [] reserve ->
St(RESERVED) [] lose -> Resume.LOST!NORMAL ->
Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(UNRENEWED)
St(RENEWED) = return -> St(SHELVED) [] reserve -> St(RESERVED) [] lose ->
Resume.LOST!NORMAL -> Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(RENEWED)
St(RESERVED) = return -> St(HELD) [] lose -> Resume.LOST!NORMAL ->
Resume.NORMAL?x:diff(Process,{NORMAL}) -> St(RESERVED)
St(LOST) = write_off -> St(Final_State_1) [] recover -> Resume.NORMAL!BOOKTop
-> St(BOOKTop')
St(Final_State_1) = STOP
BookSM = (St(BOOKTop) [|union( {Resume.x.BOOKTop|x<-THstates},
{Resume.x.y|x<-diff(States,THstates),y<-THstates} )|] (St(NORMAL))) \ {|Resume|}
```

Fig. 8. CSP$_M$ representation of the book process.



(a)

Specification: BOOKSM Model: Trace Implementation: BOOK

✓ BOOK [T= BOOKSM
✓ BOOKSM [T= BOOK

ABOOKTop
ABORROWED
AFINAL
AHELD
ARENEWED

FDR2 session: /root/new_exampl

```
Starting...
Compiling...
Reading...
 Loading... done
Took 0(0+0) seconds
-- Normalising specification
-- Normalisation complete
Starting timer
About to start refinement
Asynchronous I/O unavailable with this build.
Using 512 buffers of 128K
Refinement check: *
+.*
+.*
+.*
+.*
+.*
+.*
+.*
+.
Refine checked 23 states
With 32 transitions
Allocated a total of 5 pages of size 128K
Compaction produced 0 chunks of 16K.
Took 0(0+0) seconds
```

(b)

Specification: BOOKS Model: Trace Implementation: BOOKSM

X- BOOKS [T= BOOKSM

ABOOK
ABORROWED
AFINAL
AHELD
ARENEWED

FDR2 session: /root/new_e

BOOKSM
BOOKSM
St(BOOK) St(NORMAL)

St(NORMAL)
Performs
Resume.NORMAL.BOOK
borrow
renew
resume

FDR2 debugger

Fig. 9. Screenshots of FDR2.

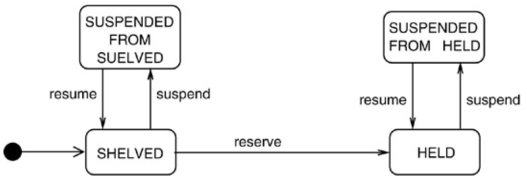


Fig. 10. An abstract specification process for the book object.

Von der Beeck [21] gave a structural operational semantics to statechart diagrams which covers shallow and deep history states as well as composite AND-states; his work, however, has not been implemented in any model checking tools/environments as far as we are aware.

A recent survey of model checkers for statecharts can be found in [22]. Well-known commercial model checkers for statecharts include STATEMATE [23]. Other statecharts model checking tools/environments include [24,25].

We managed to obtain and evaluate three publicly available support tools and environments specifically for model checking UML statechart diagrams, namely, JACK [26], UMLAUT [8], and UMC [10]. None of them supports the history mechanism. On the other hand, Rhapsody [27,28], a prominent commercial model checker for UML 2.0, supports the history mechanism but only for deep history; shallow history is not supported as in the case of our tools. Roscoe [29] also developed a compiler (translator) for translating a textual representation of statecharts into CSP for model checking with FDR2 [7] but it does not interface with any CASE tools as far as we know.

Closely related to using CSP to describe business processes is the StAC (Structured Activity Compensation) language which employs CSP-like operators for describing processes and supports the notion of compensation [30]. Attempts have also been made to translate StAC into Promela, SPL, and XTL predicates for model checking business processes using SPIN, STeP [31], and XTL [32], respectively.

7. Evaluation and discussion

In terms of scalability, our approach to model checking statechart diagrams is obviously limited by the ability of FDR2. The early versions of FDR explicitly and fully expand the state-space of a process and "can deal with processes with approximately 10^7 states in about 4 h on a typical workstation" [33]. The new version, FDR2, incorporates "the ability to build up a system gradually, at each stage compressing the subsystems to produce an equivalent process with (hopefully) many fewer states" [7]. This allows FDR2 to check exponentially larger systems, such as a network of 10^{20} dining philosophers [33].

The examples that we have seen in this paper are small—the book process as described in Fig. 3(b) has only 23 states as it involves only one history-bearing composite state (and hence two concurrent processes). The number of states is affected by:

(i) The total number of normal and abnormal states.
(ii) The use of discrete variables in operations and guards in actions and transitions.
(iii) The use of more than one history-bearing composite state in defining a suspendible business process.

Among these three factors, the last one would cause the most dramatic impact as each history-bearing composite state translates into a concurrent process which increases the number of states exponentially. At present, our toolset has no provision for analysing or reducing the number of states in a statechart diagram; the user could only find out information about the state space from FDR2.

In terms of usability, we have found the Poseidon diagram editor to be both intuitive to use and reasonably flexible for editing UML statechart diagrams according to the UML 2.0 standard. By default, it generates and saves an OMG-XMI representation of every complete and incomplete statechart diagram. As for the XSLT translation tool, it is simply a straightforward implementation of the mapping functions with no provision for error reporting. As such, it is the responsibility of the user to ensure that only valid and complete XMI scripts are input for the translation. For the output CSP_M scripts, there is a tool called checker (available from http://www.fsel.com) that does syntax- and type-checking (with error reporting) of the CSP_M scripts before we use them in FDR2.

The mapping functions defined in Section 4.1 assume the use of the history mechanism under certain restrictions, which represent a major limitation of our approach. Removing these restrictions would allow even more flexibility and expressiveness in using statechart diagrams. Nevertheless, as already pointed out in Section 4.1, one could always scale down the use of history and resort to a less compact statechart diagram for the same business process.

Compared with other approaches to model checking statechart diagrams (e.g. JACK [26], UMLAUT [8], and UMC [10]), ours involves a mapping that translates a statechart diagram into a CSP representation in a way that maintains a strong "structural correspondence" between the former and the latter. From our experience, this helps relate any errors revealed by the FDR2 model checker (in its debugger mode) to the original statechart diagram.

8. Conclusion and further work

An approach to modelling and verifying suspendible business processes has been presented. It involves using the history mechanism of UML statechart diagrams in modelling suspendible processes and also a mapping that translates statechart diagrams into CSP for model checking. To support this approach, a set of tools have been developed and integrated together.

The mapping functions used in defining the semantics of statechart diagrams are readily implementable and allows for a prototyping approach to the semantic definition, i.e. we experimented with the software tool that implemented the mapping functions while developing and refining the mapping itself. Further work includes removing the restrictions on the use of the history mechanism and extending the mapping to cater for more features of statechart diagrams including concurrent AND-states. Further development of the support tools and integrating them in the model driven architecture are also desirable.

Acknowledgements

The first author is supported by Research Grant DR04A7 from Lingnan University. The third author is supported by Program for New Century Excellent Talents in University under Grant No. NCET-04-0996. The third and fourth authors are supported by National Natural Science Foundation of China Grants No. 60233020, 60303013. The authors would like to thank the anonymous referees for their helpful comments and suggestions.

References

[1] Object Management Group, OMG Unified Modeling Language Specification Version 1.5, March 2003.
[2] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.
[3] M.Y. Ng, M. Butler, Towards formalizing UML state diagrams in CSP, in: Proc. 1st IEEE International Conference on Software Engineering and Formal Methods, Brisbane, Australia, IEEE, 2003, pp. 138–147.
[4] W.L. Yeung, K.R.P.H. Leung, J. Wang, W. Dong, Improvements towards formalizing UML state diagrams in CSP, in: Proc. 12th Asia Pacific Software Engineering Conference, Taipei, December, 2005, pp. 176–184.
[5] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[6] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.
[7] Formal Systems (Europe) Ltd., Failures-Divergence Refinement: FDR2 User Manual, 2003.
[8] W.M. Ho, J.-M. Jquel, A.L. Guennec, F. Pennaneac'h, UMLAUT: An extendible UML transformation framework, in: Proc. Automated Software Engineering 1999, 1999, pp. 275–278.
[9] T. Schafer, A. Knapp, S. Merz, Model checking UML state machines and collaborations, Electronic Notes in Theoretical Computer Science 47 (2001) 1–13.
[10] S. Gnesi, F. Mazzanti, On the fly model checking of communicating UML State Machines, in: Proc. Second ACIS International Conference on Software Engineering Research Management and Applications, SERA2004, Los Angeles, USA, May, 2004.
[11] D. Parnas, On the criteria to be used in decomposing systems into modules, Comm. ACM 15 (12) (1972) 1053–1058.
[12] W.L. Yeung, Towards formalizing UML state diagrams with history in CSP, Tech. rep., Lingnan University. http://cptra.ln.edu.hk/~wlyeung/history.ps, 2005.
[13] Object Management Group, OMG-XML Metadata Interchange (XMI) Specification Version 1.2, June 2002.
[14] W3C, XSL Transformations (XSLT) Version 1.0, November 1999.
[15] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer Academic, 1993.
[16] M.V. der Beeck, A comparison of statecharts variants, in: Proc. Formal Techniques in Real Time and Fault Tolerant Systems, in: LNCS, vol. 863, Springer, 1994, pp. 128–148.
[17] V.S.W. Lam, J. Padget, Formalization of UML statechart diagrams in the pi-calculus, in: Proc. 13th Australian Software Engineering Conference, 2001, pp. 213–223.
[18] B. Cheng, L. Campbell, E. Wang, Enabling automated analysis through the formalization of object-oriented modeling diagrams, in: Proceedings of IEEE International Conference on Dependable Systems and Networks, IEEE, 2000, pp. 305–314.
[19] E. Börger, A. Cavarra, E. Riccobene, Modeling the dynamics of UML state machines, in: Proc. Abstract State Machines: Theory and Applications, ASM 2000, in: LNCS, vol. 1912, Springer-Verlag, 2000, pp. 223–241.
[20] W. Dong, J. Wang, X. Qi, Z.-C. Qi, Model checking UML statecharts, in: Proc. 8th Asia-Pacific Software Engineering Conference APSEC'01, IEEE, 2001.
[21] M. von der Beeck, Formalization of UML-Statecharts, in: UML 2001, in: LNCS, vol. 2185, Springer, 2001, pp. 406–421.
[22] P. Bhaduri, S. Ramesh, Model checking of statechart models: Survey and research directions. URL http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0407038, 2004.
[23] T. Bienmulller, W. Damm, H. Wittke, The STATEMATE verification environment—making it real, in: E.A. Emerson, A.P. Sistla (Eds.), CAV 2000, in: Lecture Notes in Computer Science, vol. 1855, Springer, 2000, pp. 561–567.
[24] E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, Implementing statecharts in PROMELA/SPIN, in: Proc. Second IEEE Workshop on Industrial Strength Formal Specification Techniques, IEEE, 1998, pp. 90–101.
[25] P.J. Pingree, E. Mikk, The hivy tool set, in: CAV 2004, in: LNCS, vol. 3114, Springer, 2004, pp. 466–469.
[26] S. Gnesi, D. Latella, M. Massink, Model checking UML statechart diagrams using JACK, in: HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering, IEEE, 1999, pp. 46–55.
[27] I. Schinz, T. Toben, C. Mrugalla, B. Westphal, The Rhapsody UML verification environment, in: Proc. 2nd International Conference on Software Engineering and Formal Methods, SEFM 2004, IEEE, Bejing, China, 2004, pp. 174–183.
[28] D. Harel, H. Kugler, The rhapsody semantics of statecharts (or, on the executable core of the UML) (preliminary version), in: Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG) Final Report, in: LNCS, vol. 3147, Springer, 2004, pp. 325–354.

[29] B. Roscoe, Compiling statemate statecharts into CSP and verifying them using FDR—abstract. http://web.comlab.ox.ac.uk/-oucl/work/bill.roscoe/publications/94ab.ps, January 2003.

[30] M. Butler, C. Ferreira, A process compensation language, in: IFM'2000—Integrated Formal Methods, in: LNCS, vol. 1945, 2000, pp. 61–76.

[31] J.C. Augusto, M. Butler, C. Ferreira, S. Craig, Using SPIN and STeP to verify StAC specifications, in: Proceedings of 5th International A.P. Ershov Conference on Perspectives of System Informatics, PSI'03, in: LNCS, vol. 2890, Novosibirsk, Russia, 2003, pp. 207–213.

[32] J.C. Augusto, M. Leuschel, M. Butler, C. Ferreira, Using the extensible model checker XTL to verify StAC business specifications, in: Pre-Proceedings of 3rd Workshop on Automated Verification of Critical Systems, AVoCS 2003, Southampton, UK, 2003, pp. 253–266.

[33] A.W. Roscoe, et al., Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock, in: Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 1019, Springer, 1995, pp. 133–152.