# Patterns for Modelling Time and Consistency in Business Information Systems

J. W. Bryans*, J. S. Fitzgerald*, A. Romanovsky* and A. Roth†
*School of Computing Science
Newcastle University
United Kingdom
Email: {Jeremy.Bryans | John.Fitzgerald | Alexander.Romanovsky}@ncl.ac.uk
†SAP Research CEC Darmstadt
SAP AG, Bleichstr. 8,
64283 Darmstadt, Germany
Email: Andreas.Roth@sap.com

***Abstract*— Maintaining semantic consistency of data is a significant problem in distributed information systems, particularly those on which a business may depend. Our current work aims to use Event-B and the Rodin tools to support the specification and design of such systems in a way that integrates well into existing development processes. This paper presents Event-B patterns that may be used to represent recovery from time-bounded inconsistency and illustrates their use in a model derived from industrial applications.**

**KEYWORDS: Real-time, Patterns, Formal Verification, Event-B, Error Recovery**

## I. Introduction

Computer-based business information systems are often critical to the successful functioning of enterprises, but are challenging to develop because of their large scale and distributed character. Our work in Deploy[1] aims to provide formal modelling and analysis technology that helps to reduce development risk by allowing early-stage comparison of design alternatives and identification of potential sources of defects. In order to promote take-up, we aim for a high degree of automation in the analysis of formal models that are derived, where possible automatically, from designs expressed in domain-specific, often diagrammatic, notations [1].

We focus on applications of the kind developed using SAP technology which support companies' business processes. These can be best practice customisable processes such as "order-to-cash" or "procure-to-pay". They are assembled from components describing parts of processes, such as buying, selling, planning, site logistics and accounting. On the other hand, business processes can also be very specific to customers and can be modelled with the help of specific business process management (BPM) tools such as SAP NetWeaver BPM [2].

With the help of service-oriented architectures (SOA), business processes can be closely linked to their technical realisation using independent business application components. For a typical business process, dozens of independent business components form a complex network where (mostly asynchronous) messaging is used to satisfy the components' communication needs without giving up their loose coupling. Complexity arises from the large-scale composition of relatively simple protocols, making it a challenge to determine application-level properties such as inconsistency and race conditions. This complexity suggests that there is an argument for machine-assisted verification of these applications.

In a typical business process, the consistency of data across the components involved in the process is of considerable importance. For example, in a scenario in which a customer places an order, the quantity of a product in the sales order data should be the same in the final invoice and the dispatch note. The exact definition of consistency is application-dependent. For example, the goods in a delivery should be a subset of those ordered. We refer to this application-dependent consistency as *semantic consistency*.

Although semantic consistency is important in business information systems, inconsistency is also a fact of life. For example, orders or parts of orders can often be changed or cancelled, even though subsequent process steps have already reached an advanced state. This is problematic if, for example, the delivery of cancelled orders has already started. A sub-process may be delayed pending a manual approval or other manual processing such as moving or packaging goods, and this delay may give rise to temporary inconsistency. Finally, messages which are intended to update other process components may be blocked in transmission through lower layers, causing errors to be propagated to the higher layers. For these reasons, processes are often in temporarily inconsistent states. After a certain delay caused by late changes, manual steps, or recovery from errors, they must however assume a consistent state.

Modelling consistency and error recovery may help the developers to integrate late change or (partial) cancellation of business objects into business logics, and to mechanically analyse earlier in the development cycle the ways the business scenarios are constructed to recover from such errors and to adapt them if necessary before implementing them. Since reasoning about semantic consistency involves the consideration of delays, handling consistency in models entails understanding and modelling time.

[1]`www.deploy-project.eu`

Our objective is to support modelling and analysis of business processes that can accommodate time-bounded semantic inconsistency. We use the Event-B modelling formalism [3] and the Rodin tools platform [4], because they have several features that make them appropriate to our application area. The modelling language allows description of both structured data and functionality. The tools are open-source and based on the Eclipse framework, allowing the integration of specialised provers, model checkers, editors, pretty printers and other new features. The method and tools have a significant and growing community of practice.

Experience in Event-B modelling suggests [1] that the choices of abstractions and modelling/refinement patterns during development have a significant bearing on the level of automation achievable in analysis. We therefore focus first on abstractions and patterns that can form the basis of reasoning about bounded inconsistency. The contribution of this paper is to identify Event-B patterns that can be used to model and analyse time-bounded semantic inconsistency in distributed business information systems and to demonstrate their use in a realistic scenario.

In a typical SOA scenario, a business process developer is not responsible for implementing the services or the middleware, but will rather use existing service components and configure middleware. It is therefore important for such a developer to explore several options of how to realize a business process early on in the design process. Formalizing, simulating and automatically analyzing processes and consistency properties is thus an important aspect of the efficiency and quality of business process design.

We first briefly describe salient aspects of Event-B and patterns (Section II-A) and give an abstract machine that forms the basis of the patterns presented (Section II-B). We consider the Time Constraint Pattern (TCP) in Section III-A and describe an adaptation that separates time and consistency in Section III-B. We give a simple pattern for modelling error detection and recovery at the level of abstraction of the business information systems models that we deal with (Section IV). An example of the application of this pattern to an realistic business process is given in Section VI. This is integrated with the adapted TCP to form a single pattern modelling both time and consistency in Section V. Related work is discussed in Section VII. Conclusions and further work are described in Section VIII.

## II. Background

As indicated in Section I, inconsistency can arise through the normal operation of a business process which includes the possibility of cancellations and updates. It may also arise as a result of faults in lower levels, such as middleware or message transport layers. In normal operation, each process attempts to maintain semantic consistency by informing other processes of changes that need to be made. The parts of the system related to this change are necessarily inconsistent while these messages are created, transmitted, received and processed. This normal operational inconsistency is time-bounded. Further, the system may include a recovery mechanism to allow it to recover from the occurrence of faults at lower levels in the system. This will involve the distribution of error recovery messages. In both normal operation and error recovery message transmission, latency will be a significant factor in bounding the resulting inconsistency. We therefore begin to model time-bounded recovery from inconsistency by considering message transfer.

### A. *Event-B and Event-B Patterns*

The basic modelling unit in Event-B is the *machine*. Each machine may contain *variables* modelling persistent state data, *invariants* that restrict the valid content of variables, and guarded *events* that describe functionality in terms of actions defined over the state variables. Definitions of the carrier sets and constants may be defined in units called contexts that are visible to machines. A system model typically consists of a chain of Event-B machines, each of which (apart from the first) is linked to its predecessor by a refinement relation expressed in terms of a linking invariant. A typical Event-B model has an extremely simple initial machine, with detail added in a controlled way through refinement steps.

Machines and refinement steps give rise to proof obligations that ensure internal consistency of machines and behaviour preservation across refinement steps. Types of proof obligations include, among others, guard strengthening and invariant preservation. Guard strengthening requires that, in a machine refinement, the guards of an event in the abstract machine are refined by the guards of the corresponding event in the concrete machine. Invariant preservation requires that within a machine, each event preserve each invariant.

Patterns in Event-B [5], [6], [7], [8] are a means of expressing reusable modelling structures and managing effort by promoting proof re-use. Several forms of Event-B pattern have been proposed, differing in their levels of generality. Iliasov [7] treats a pattern as a general model transformation method. Hoang, Fürst and Abrial [6], by contrast, treats a pattern as a fragment of Event-B designed to be directly substituted into a development. The first and last machine of the pattern are referred to as the abstracta and concrete pattern respectively. Cansell et al. [8] regard a pattern as being less general than Iliasov, but still requiring some specialisation before it is applied to a development.

In the approach taken by Hoang et al. [6] the application of a pattern to an Event-B development requires that each variable and event in the abstract pattern must be matched with a variable or event in the current machine in the development. Variables match if a subset of the variables in the development have the same type as variables in the abstract pattern. Events match if a subset of the events in the development have the same behaviour as the events in the abstract pattern. The development can then be extended by replacing matched events and variables (and relevant invariants) with the events, variables and invariants of the concrete pattern. The application of the pattern presented by Cansell et al. in [8] requires both data refinement (of variables) and event refinement.

Beginning with the pattern presented in [8] (the Time Constraint Pattern), we develop a pattern to add timing information to a model of a business information system. We then develop a pattern for adding error recovery behaviour, and combine the two patterns developed. The patterns that we develop follow the approach of Hoang et al., since our goal is to automatically apply these patterns.

## *B. The Abstract Channel*

The duration of message transmission between components is a major source of time-bounded inconsistency. We therefore need to be able to describe real-time requirements, and in particular to articulate and prove properties relating to time-bounded inconsistency, as well as the means of recovery from inconsistency. In this section we develop an abstract machine corresponding to a small fragment of a business information system which contains no representation of time or error recovery. We then investigate patterns for refining this machine by adding time and error recovery. In Section III-A we apply the TCP to this abstract channel. In Section III-B we adapt the TCP to separate the modelling of time and consistency. In Section IV we propose a pattern to describe error recovery, and in Section V we combine the modified TCP with the error recovery pattern to generate the Timed Error Recovery Pattern. An outline of these developments is given in Fig. 1.

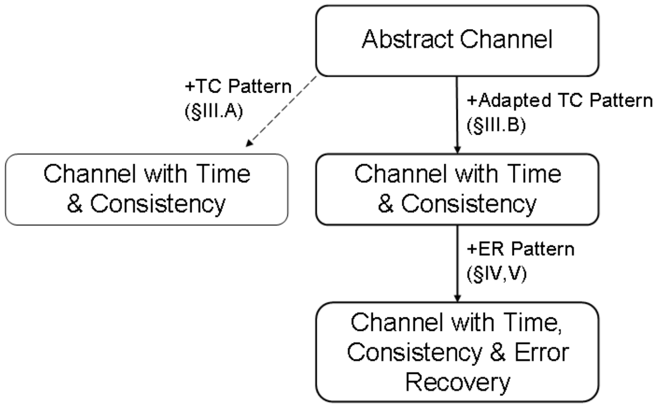
Abstract Channel
+TC Pattern (§III.A)
+Adapted TC Pattern (§III.B)
Channel with Time & Consistency
Channel with Time & Consistency
+ER Pattern (§IV,V)
Channel with Time, Consistency & Error Recovery

Fig. 1. Structure of the developed models.

We identify two processes, a sender and a receiver, and consider a channel carrying messages between them. The channel could be a lower-level off-the-shelf component, or a portion of the business network containing a number of other processes. We consider only messages that identify inconsistencies to be resolved. At this level of abstraction, receipt of a message models correction of the inconsistency. We identify a set $cons$ of messages. A message is in this set if the inconsistency to which it refers has been resolved. We refer to these messages as consistent.

We model the transmission and reception of messages over the channel (events snd and rcv in Fig. 2.) The variables $sent$, $chan$ and $cons$ are all subsets of $MSG$ (a carrier set representing the set of all messages identifying inconsistencies). Variables $sent$ and $chan$ represent messages that have been sent and the contents of the channel respectively. In this abstract machine, and in subsequent developments, we will state *consistency invariants*. These are normal Event-B invariants, and their purpose is to capture what we can prove about consistency in the different machines. The (untimed) consistency invariant is given in Fig. 2. Consistent messages are those that have been removed from the channel.

```
snd
  any  $m$  where
    $m \in MSG \land m \notin sent$
  then
    $sent := sent \cup \{m\}$
    $chan := chan \cup \{m\}$
  end

rcv
  any  $m$  where
    $m \in chan$
  then
    $cons := cons \cup \{m\}$
    $chan := chan \setminus \{m\}$
  end

invariants
    $cons = sent \setminus chan$
```

Fig. 2. The abstract channel.

## III. An Adaptation of the Time Constraint Pattern

In this section, we describe the timed behaviour of the channel. Time is not an explicit part of the Event-B language, so in order to describe time-bounded inconsistency, we begin by examining an important pattern for representing time.

We will go on to model recovery from time-bounded inconsistency. At this stage we do not wish to set a hard upper limit on the period on inconsistency, so we do not set an upper bound on the duration of message transmission. Instead, we associate a real-time period $limit$ with each message in the channel. This limit is the time limit of "acceptable" inconsistency. If a message arrives after $limit$ then recovery will be necessary.

## *A. The Time Constraint Pattern*

Several extensions have been proposed for modelling time in B and Event-B (e.g. [9], [10]). However, we prefer to work entirely within the Event-B language in order to take advantage of the tools provided.

The Time Constraint Pattern presented in Cansell et al [8] is a promising Event-B pattern for introducing time constraints into a development. An elaboration is given by Rehm [11]. In the TCP, time progression is measured by the increase of a dedicated variable $time$ of type $NAT$. The passage of time is modelled by a separate event tick_tock, see Fig 3. A collection of "active times" is maintained to represent the times in the future when certain events must be performed. (Rehm [11] associates events with these times, whereas the earlier paper of Cansell et al. [8] just records the set of active times in the variable $at$.) The value $tm$ is the new value for $time$ after the event tick_tock. Time must not progress beyond the point at which the next event is scheduled to be performed, as guaranteed by the third guard in tick_tock. Time constraints are introduced to and removed from the set of active times by the events post_time and process_time respectively (Fig. 3).

The TCP is not designed to be applied directly, but must be adapted to a match a specific development. Here, we wish to use the pattern to allow us to record constraints that messages are handled within deadlines: each message should be received

```
post_time
  any  $tm$  where
    $tm \in NAT \wedge tm > time$
  then
    $at := at \cup \{tm\}$
  end

process_time
  when
    $time \in at$
  then
    $at := at \setminus \{time\}$
  end

tick_tock
  any  $tm$  where
    $tm \in NAT \wedge tm > time$
    $(at \neq \varnothing \Rightarrow tm \leq \min(at))$
  then
    $time := tm$
  end
```

Fig. 3. The TCP events.

within $limit$ time units. The constant $limit$ (defined as a natural number in the associated context) is the upper limit on the delivery of messages before recovery is necessary.

In Fig. 4 we show the machine *TCP-channel* resulting from an application of the TCP to our abstract channel of Fig. 2. The variable $now$ (representing elapsed time) matches the TCP variable $time$. The correspondence between variable $deadline$ and TCP variable $at$ is less immediate. We use $deadline$ to record the time at which messages must leave the channel. The variable $at$ therefore matches $rng(deadline)$. The map structure associating messages and times is similar to the presentation given in [11].

The variable $timesent$ is a total function from sent messages to time, and records the time at which messages are placed on the channel. The variable $timercvd$ records the time at which messages are received from the channel. All messages in $chan$ have an associated deadline, given by $deadline(m)$.

We refine the snd event to record the time at which a message is sent, as suggested by the event post_time from the TCP. Event snd also adds the message to the channel, and adds timing information in the form of the maplet $\{m \mapsto now + limit\}$ to the variable $deadline$. The time posted in $deadline$ is $now + limit$, giving a real-time deadline for receipt of the message.

The concrete rcv event corresponds to the process_time event. We generalise the timed guard from process_time to $now \leq deadline(m)$, to allow messages to be received at any point before their deadline is reached. Event rcv removes a message from the channel and the associated maplet from $deadline$, adds the message to $cons$, and records the time at which the message was received. The tick event is a refinement of the tick_tock event of TCP. When tick fires, time progresses by one unit.

For this machine timely receipt of a message represents time-bounded recovery from inconsistency. A timed consistency invariant is given in Fig. 5.

Any sent messages for which the deadline has passed ($timesent(m) + limit < now$) must be in the consistent set $cons$.

This direct application of TCP produces a machine that excludes the possibility of messages being delayed, since $deadline(m)$ gives a hard upper bound on the duration of transmission of message $m$. However, as noted previously, the application developer is not designing the message-carrying layer, and may not know this information at the time of development. In this case, he cannot assume that all messages *will* be delivered within a certain deadline. He may only be able to say that messages *should* be delivered within some time, and that messages which take longer should be subject to error-recovery. We therefore adapt the TCP, to allow him to specify from this point of view.

```
snd
  any  $m$  where
    $m \in MSG$
    $m \notin dom(timesent)$
  then
    $timesent := timesent \cup \{m \mapsto now\}$
    $chan := chan \cup \{m\}$
    $deadline := deadline \cup \{m \mapsto now + limit\}$
  end

rcv
  any  $m$  where
    $deadline \neq \varnothing$
    $m \in chan$
    $now \leq deadline(m)$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $chan := chan \setminus \{m\}$
    $cons := cons \cup \{m\}$
    $timercvd := timercvd \cup \{m \mapsto now\}$
  end

tick
  when
    $dom(deadline) \neq \varnothing \ \Rightarrow \ now < min(ran(deadline))$
  then
    $now := now + 1$
  end

invariants
  $now \in \mathbb{N}$
  $sent = dom(timesent)$
  $timesent \in dom(timesent) \rightarrow \mathbb{N}$
  $deadline \in dom(timesent) \rightarrowtail \mathbb{N}$
  $timercvd \in dom(timesent) \rightarrowtail \mathbb{N}$
  $dom(deadline) = chan$
```

Fig. 4. The invariants and events of the TCP-channel.

$$\forall m \ \cdot \ m \in dom(timesent) \Rightarrow \\ (timesent(m) + limit < now \Rightarrow m \in cons)$$

Fig. 5. A timed consistency invariant in TCP-channel.

## *B. Adapting the Time Constraint Pattern for Modelling Bounded Inconsistency*

We adapt the TCP by removing the guard on tick and the third guard on rcv, giving events ungrd_tick and untmd_rcv (Fig. 6). By allowing time to progress at any stage in the execution of a machine, we allow any message to be delayed beyond $deadline(m)$. In the machine that results, with concrete events snd (unchanged from *TCP-channel*, Fig. 4), untmd_rcv (Fig. 6) and ungrd_tick (Fig. 6), all messages received at any time are consistent. A possible consistency invariant is also given in Fig. 6.

```
untmd_rcv
  any  $m$  where
    $deadline \neq \varnothing$
    $m \in chan$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $chan := chan \setminus \{m\}$
    $cons := cons \cup \{m\}$
    $timercvd := timercvd \cup \{m \mapsto now\}$
  end

ungrd_tick
  begin
    $now := now + 1$
  end

invariants
   $dom(timercvd) = cons$
```

Fig. 6. Events untmd_rcv and ungrd_tick and invariant.

We now need to separate normal and recovery behaviour, distinguishing the receipt of message $m$ before and after $deadline(m)$. We propose an error recovery pattern to deal separately with this error recovery behaviour.

## IV. An Error Recovery Pattern

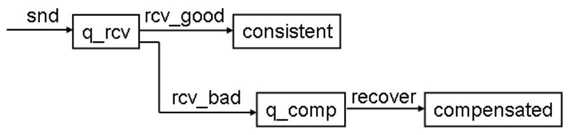

Fig. 7. Structure of Concrete Error Recovery Pattern.

Effective modelling and analysis of fault tolerance relies on a clear separation between normal and abnormal states and behaviours. This structuring reduces the complexity of system design. It allows developers to reason explicitly about erroneous states and to associate appropriate recovery actions with them. Moreover, it provides a consistent means of expressing the switchover between normal and abnormal modes of execution, the initiation of recovery actions and the management of success and failure of this recovery.

In this section we present a pattern modelling error recovery. We will refer to this as the Error Recovery Pattern (ERP). The ERP has one abstract and one concrete machine. The abstract machine is the original description of the channel given in Fig. 2. To model error recovery, we begin by distinguishing the possibilities that the message transmission is either correct or faulty. If the transmission is faulty (the message is corrupted, late, etc.), then some recovery action is required. The concrete part of the pattern is represented diagrammatically in Fig. 7. Event rcv from Fig. 2 is now refined into three events (rcv_good, rcv_bad and recover, Fig.8), and the channel of the abstract machine is split into two queues. $q\_rcv$ is the queue of messages waiting to be received, and $q\_comp$ is the queue of messages which have been received and for which compensation is required. The variable $consistent$ represents those messages which were immediately consistent on arrival, and $compensated$ represents those messages for which the system had to perform recovery.

```
rcv_good
  refines  $rcv$
  any  $m$  where
    $m \in q\_rcv$
  then
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

rcv_bad
  any  $m$  where
    $m \in q\_rcv$
  then
    $q\_comp := q\_comp \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

recover
  refines  $rcv$
  any  $m$  where
    $m \in q\_comp$
  then
    $compensated := compensated \cup \{m\}$
    $q\_comp := q\_comp \setminus \{m\}$
  end

invariants
   $cons = consistent \cup compensated$
   $chan = q\_comp \cup q\_rcv$
```

Fig. 8. A concrete Error Recovery Pattern.

In the concrete machine of the ERP given in Fig.8, the event rcv_good models the receipt of a good message from $q\_rcv$. No further action is required and the part of the system to which it refers is immediately consistent. Event rcv_bad models the receipt of a faulty message and its placement in $q\_comp$ – the queue of messages which need to be compensated. Event recover performs the recovery action, removing the faulty message from $q\_comp$ and placing it in the set of compensated messages $compensated$. The event snd (not given) is almost unchanged, except that sent messages are placed in the queue for the receiving component $q\_rcv$ instead of $chan$.

As well as the general role that careful structuring plays in the provision of effective fault tolerance, recursive structuring supports the description of the propagation of responsibility for error handling through system levels [12]. An example of this hierarchical error recovery is Fault Detection, Isolation and Recovery (FDIR) [13]. If, rather than just consider the case where compensation is successful, we consider as well the case where it fails, we can reapply the ERP to itself to model hierarchical error recovery. A single application of ERP

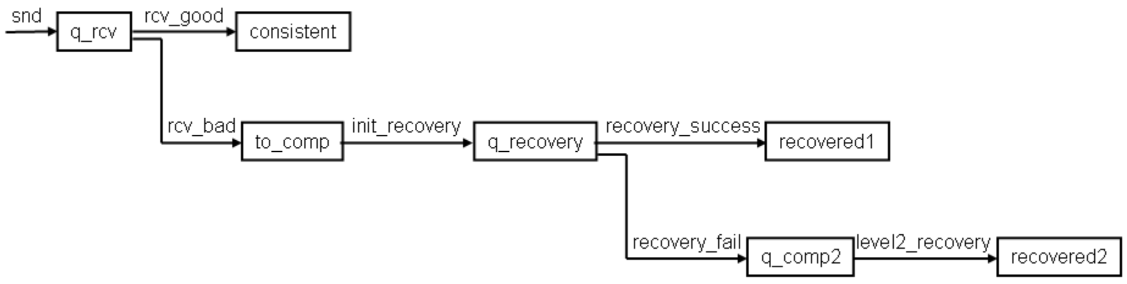

Fig. 9. Hierarchical Error Recovery.

to itself results in the machine summarised diagrammatically in Fig. 9, and presented in full in Fig. 11.

## V. The Timed Error Recovery Pattern

We combine the ERP and the adapted TCP to generate the concrete part of the Timed Error Recovery Pattern, guaranteeing recovery from time-bounded inconsistency arising from the late arrival of messages. We distinguish the timely and late arrival of messages. Event rcv_good becomes rcv_ontime (Fig. 12.) The addition of the timing guard $deadline(m) \geq now$ ensures that this event applies only to messages whose deadline has not yet passed. Event rcv_late is derived from the rcv_bad event of the ERP, with the addition of the guard $deadline(m) < now$. Event compensate deals with a late message which requires compensation. It places the message in the set of messages which have been compensated, and removes the deadline information from $deadline$. We also record the time at which a message is received ($timercvd$). With these, we can then prove the timed consistency invariants in Fig 10.

The Timed Error Recovery pattern has two refinement stages which need to be applied to guarantee recovery from time-bounded inconsistency. The first changes events snd and rcv to the three events in Fig. 8 and adds the event ungrd_tick (Fig. 6) The second refines these events with the events in Fig. 12. The total pattern contains a total of 120 proof obligations, 102 of which have been proved automatically. These proof obligations do not need to be reproved when the pattern is applied.

For example, the proof obligation generated to demonstrate that the event rcv_ontime preserves the first consistency invariant of Fig 10 is given in Fig 13. To prove this, we may appeal to axioms from the context and other invariants as well as the guards and behaviour of the event rcv_ontime.

## VI. Application

Prandi et. al. [14] give a description of a Credit Request Process (CRP). A bank customer uses a credit portal to request credit from a bank. If authentication is successful, he creates a loan request, which is sent to the bank. The bank sends the information provided to a validation service which performs some checks and returns the result to the bank. This result is then returned to the customer. If the customer is unsuccessful, they may modify the amount requested and reapply. Fig. 14 shows the part of the CRP after authentication is complete. The notation is Business Process Modelling Notation (BPMN). The labelling of tasks (Tx) and events (Ex) matches the labelling from [14].

We begin with a systematically produced Event-B model of the CRP. We focus on the consistency of loan requests between the customer and the credit portal. We wish to ensure that credit requests from the customer either arrive within a specified time or are compensated for in some (as yet unspecified) way. We therefore apply the Timed Error Recovery Pattern to the channel indicated between T5 and T9. This application of the pattern was done automatically, using the approach and tool described in [6]. This produces a refined version of the CRP Event-B machine, in which the invariants of Fig. 10 hold.

Fig. 15 shows the relevant events and further invariants in the Event-B model of the Credit Request Portal. The event cSend models the customer sending a credit request to the portal. The guard $tk\_cSend > 0$ ensures that a loan request has been created or modified (process T4 or T6). The second two guards match the guards of the sending event in the abstract part of the Timed Error Recovery Pattern (event snd in Fig. 2). cSend adds a loan request identifier ($ln$) on to channel ($ch\_req\_ids$). It also deals with the control flow tokens and adds a $CreditReq$ object on to the channel between the customer and the credit portal, and records $ln$ in the set $sent$.

The event pReceive models the reception of a loan request by the portal. A loan request may arrive when the token for process T9 has been set ($tk\_pReceive > 0$). This is set at the completion of the previous process in the Credit Portal. The presence of the $CreditReq$ object in $ch\_C2P$ is an indication that a credit request has been made. The variable $ch\_C2P$ is the communication channel between the customer and the portal. The incoming loan request identifier $ln$ must be on the request identifier channel $ch\_req\_ids$.

When pReceive fires the token for that stage ($tk\_pReceive$) is reduced and the token for the next stage (($tk\_pVeriSend$) is set. The credit request object is removed from the channel $ch\_C2P$ and the loan request is added to the consistent set $cons$ and removed from the channel $ch\_req\_ids$.

For the Timed Error Recovery Pattern to be applied, cSend and pReceive are substituted for snd and rcv from Fig. 2. The guards and actions in each event which are substituted in this application are labelled in Fig. 15 with the word pattern. The variable $limit$ is set to the maximum time that loan

$$
\begin{array}{lcl}
\forall m \cdot (m \in dom(timercvd) \;\wedge timercvd(m) - timesent(m) \leq limit) & \Leftrightarrow & m \in consistent \\
\forall m \cdot (m \in dom(timercvd) \;\wedge timercvd(m) - timesent(m) > limit) & \Leftrightarrow & m \in q\_comp \cup compensated
\end{array}
$$

Fig. 10. Timed Error Recovery Pattern Consistency Invariants.

```
snd
  refines  $snd$
  any  $m$  where
    $m \in MSG$
    $m \notin sent$
  then
    $sent := sent \cup \{m\}$
    $q\_rcv := q\_rcv \cup \{m\}$
  end

rcv_good
  refines  $rcv\_good$
  any  $m$  where
    $m \in q\_rcv$
  then
    $consistent :=$
        $consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

rcv_bad
  refines  $rcv\_bad$
  any  $m$  where
    $m \in q\_rcv$
  then
    $to\_comp :=$
        $to\_comp \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

init_recovery
  refines  $init\_recovery$
  any  $m$  where
    $m \in to\_comp$
  then
    $to\_comp :=$
        $to\_comp \setminus \{m\}$
    $q\_recovery :=$
        $q\_recovery \cup \{m\}$
  end

recovery_success
  refines  $term\_recovery$
  any  $m$  where
    $m \in q\_recovery$
  then
    $q\_recovery :=$
        $q\_recovery \setminus \{m\}$
    $recovered1 :=$
        $recovered1 \cup \{m\}$
  end

recovery_fail
  any  $m$  where
    $m \in q\_recovery$
  then
    $q\_comp2 :=$
        $q\_comp2 \cup \{m\}$
    $q\_recovery :=$
        $q\_recovery \setminus \{m\}$
  end

level2_recovery
  any  $m$  where
    $m \in q\_comp2$
  then
    $recovered2 := recovered2 \cup \{m\}$
  end
```

Fig. 11. Hierarchical Error Recovery events.

```
tick
  begin
    $now := now + 1$
  end

rcv_ontime
  refines  $rcv$
  any  $m$  where
    $m \in q\_rcv$
    $deadline(m) \geq now$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

rcv_late
  any  $m$  where
    $m \in q\_rcv$
    $deadline(m) < now$
  then
    $q\_rcv := q\_rcv \setminus \{m\}$
    $q\_comp := q\_comp \cup \{m\}$
  end

compensate
  refines  $rcv$
  any  $m$  where
    $m \in q\_comp$
  then
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $compensated := compensated \cup \{m\}$
    $q\_comp := q\_comp \setminus \{m\}$
  end

snd
  refines  $snd$
  any  $m$  where
    $m \in MSG$
    $m \notin dom(timesent)$
  then
    $timesent := timesent \cup \{m \mapsto now\}$
    $deadline := deadline \cup \{m \mapsto now + limit\}$
    $q\_rcv := q\_rcv \cup \{m\}$
  end
```

Fig. 12. The events of the concrete part of Timed Error Recovery Pattern.

requests may be in the channel $ch\_req\_ids$ without requiring compensation.

The application of the pattern results in the creation of the pReceive_ontime, pReceive_late and compensate events within the refined Credit Request Process, given in Fig. 16. The bounded consistency invariants given in Fig. 10 hold, and do not have to be reproved.

We can also apply the pattern in a way that maintains consistency of information within a component. For example, to consider time bounded consistency of credit requests within the credit portal, we can match the snd event with T9 (receive_credit_request) and the rcv event with T11 (send_preliminary_decision). There is no channel directly between these two events, so we introduce a virtual channel connecting them. Loan requests join this channel when they arrive at T9, and leave when the decision is sent back to the customer at T11. Loan requests which take greater than a certain time limit to travel between these two points may be compensated by refining the event compensate.

## VII. Related Work

Approaches to the formal modelling and analysis of workflows over service-oriented architectures, particularly for

$$m0 \in dom(timercvd) \cup \{m \mapsto now\} \wedge$$
$$(timercvd \cup \{m \mapsto now\})(m0) - timesent(m0) \leq limit$$
$$\Leftrightarrow$$
$$m \in consistent \cup \{m \mapsto now\}$$

Fig. 13. A proof obligation in Timed Error Recovery Pattern.

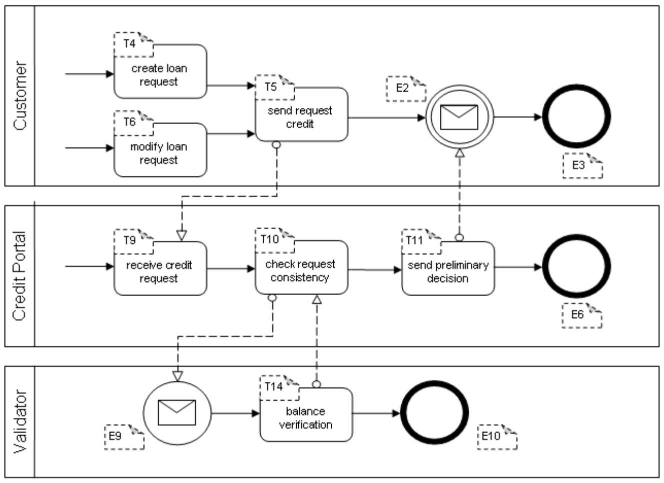

Fig. 14. The Final Stage of the Credit Request Process.

```
invariants
    $sent \subseteq reqId$
    $ch\_req\_ids \subseteq reqId$
    $cons \subseteq reqId$
    $cons \cap ch\_req\_ids = \varnothing$
    $sent = ch\_req\_ids \cup cons$
    $cons = sent \setminus ch\_req\_ids$

cSend
  any $ln$ where
    $tk\_cSend > 0$
    $ln \in reqId$                                  pattern
    $ln \notin sent$                                pattern
  then
    $tk\_cSend := tk\_cSend - 1$
    $ch\_C2P := ch\_C2P \cup \{CreditReq\}$
    $tk\_cGate := tk\_cGate + 1$
    $sent := sent \cup \{ln\}$                      pattern
    $ch\_req\_ids := ch\_req\_ids \cup \{ln\}$      pattern
  end

pReceive
  any $ln$ where
    $tk\_pReceive > 0$
    $CreditReq \in ch\_C2P$
    $ln \in ch\_req\_ids$                           pattern
  then
    $tk\_pReceive := tk\_pReceive - 1$
    $ch\_C2P := ch\_C2P \setminus \{CreditReq\}$
    $tk\_pVeriSend := tk\_pVeriSend + 1$
    $cons := cons \cup \{ln\}$                      pattern
    $ch\_req\_ids := ch\_req\_ids \setminus \{ln\}$ pattern
  end
```

Fig. 15. Part of the Credit Request Process.

BPEL, use a variety of notations and tools. Many base formal models on transition systems and Petri Nets, allowing the analysis of path and termination properties [15], [16]. Finite automata encoded in Promela have also been used for deadlock detection [17]. Process Calculi such as FSP [18], LOTOS [19] and Abstract State Machines [20] have also been used to model workflow supporting a range of properties including aspects of fault handling and compensation.

Our approach focuses on the use of proof to give semantic analysis of functional and timing aspects of workflows over distributed business systems. The closest work to our own is perhaps that of Aït-Sadoune and Aït-Ameur [21] which presents an Event-B translation of the BPEL language including fault handling. This allows for proof-based validation of properties such as deadlock-freeness. Ball and Butler [5] identify design patterns relating fault tolerance behaviour in the family of agent interaction protocols dealing with agent contracts. The patterns codify possible responses to problems which agents can encounter when exchanging requests to carry out actions and are presented as blueprints, capturing typical elements of the agent models. Our work differs from the approaches above by addressing application-level semantic consistency, focusing on patterns that can be included in the Event-B model derived from workflow and having an explicit model of real time.

Clearly, the Time Constraint Pattern of Cansell et al. [8] is closely linked to this work, and applies to cases where the developer can be confident that the message transportation layer can guarantee certain timing properties. The pattern we develop here covers the case where these timing characteristics are not known. The timing part of our pattern is simpler than the TCP. If these characteristics were settled later in the development, it would therefore be possible to add this detail to the Event-B development, and return to the full strength of the Time Constraint Pattern.

Besides time and consistency, our approach makes use of a simple error recovery pattern. There are several approaches to patterns for fault tolerance in B and Event-B. Laibinis and Troubitsyna [22] propose a formal specification pattern (in B) that can be recursively applied to specify exception raising and handling at various architectural layers of component-based systems. The approach is developed for systems in which components interact by issuing synchronous calls and which mainly face hardware failures and human errors. Iliasov [23], [7] offers a general approach to defining fault tolerance refinement patterns assisting system developers in disciplined application of software fault tolerance mechanisms during rigorous system design. Our work additionally requires modelling of temporal constraints. These patterns are formally defined as model transformations producing correct model refinements. The approach is backed by a tool and a theory of pattern composition.

## VIII. Conclusions and Further Work

We have presented a pattern that may be used to describe time-bounded semantic consistency properties for distributed business information systems. Our approach adapts the Timed

```
pReceive_ontime
  refines  $pReceive$
  any  $m$  where
    $tk\_pReceive > 0$
    $m \in q\_rcv \cup q\_comp$
    $m \in q\_rcv$
    $deadline(m) \geq now$
  then
    $tk\_pReceive := tk\_pReceive - 1$
    $ch\_C2P := ch\_C2P \setminus \{CreditReq\}$
    $tk\_pVeriSend := tk\_pVeriSend + 1$
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $consistent := consistent \cup \{m\}$
    $q\_rcv := q\_rcv \setminus \{m\}$
  end

pReceive_late
  any  $m$  where
    $m \in q\_rcv$
    $deadline(m) < now$
  then
    $q\_rcv := q\_rcv \setminus \{m\}$
    $q\_comp := q\_comp \cup \{m\}$
    $timercvd := timercvd \cup \{m \mapsto now\}$
  end

compensate
  refines  $pReceive$
  any  $m$  where
    $tk\_pReceive > 0$
    $m \in q\_rcv \cup q\_comp$
    $m \in q\_comp$
  then
    $tk\_pReceive := tk\_pReceive - 1$
    $ch\_C2P := ch\_C2P \setminus \{CreditReq\}$
    $tk\_pVeriSend := tk\_pVeriSend + 1$
    $deadline := deadline \setminus \{m \mapsto deadline(m)\}$
    $compensated := compensated \cup \{m\}$
    $q\_comp := q\_comp \setminus \{m\}$
  end
```

Fig. 16. Events in the Concrete CRP after application of the TERP.

Consistency pattern of Cansell et al., adding (optionally hierarchical) error recovery.

Although our approach was inspired by business information systems applications, we conjecture that the same approach could be used for a wider class of distributed applications. The work reported in this paper concentrates on business process models and not the choreography models that describe communications protocols between business processes. Our previous work [1] modelled the effect of faults in communications middleware on the completion of communications protocols at the choreography level that underpins applications. An important direction for future work is to link these two levels, in order to provide more complete analysis, including error propagation and recovery.

We intend to apply the patterns for modelling temporary semantic inconsistency to realistic business processes typically implemented in SAP software and to custom business processes modelled with the help of BPMN. To this end we are investigating ways of representing these processes in a formal language like Event-B, similar to the general approach of Aït-Sadoune and Aït-Ameur [21]. Automating and providing tool support for the translation into a formal language and for the application of the error recovery pattern may give business process designers the most convenient support for their work. There is preliminary evidence that the re-use of patterns in Event-B can save significantly in proof effort [6]; we would wish to gather more data from other applications of our patterns in order to confirm this. It is in our plans for the future work to evaluate the applicability of general pattern approaches such as those of Iliasov [7] to the development of the time constraint and error recovery patterns proposed in this paper. The use of good labour-saving patterns is likely to play a significant part in the achievement of that goal.

*Acknowledgements:* This work has been supported by the EC FP7 Integrated Project *Deploy*. We are grateful to our colleagues in the project for their continuing fruitful collaboration and especially to Wei Wei at SAP Research for providing the systematically generated CRP model.

## References

[1] J. Bryans, J. Fitzgerald, A. Romanovsky, and A. Roth, “Formal Modelling and Analysis of Business Information Applications with Fault Tolerant Middleware,” in *Proc. 14th IEEE Intl. Conf. Conference on Engineering of Complex Computer Systems*, June 2009, pp. 68–77.

[2] J. H. Snabe, A. Rosenberg, C. Moller, and M. Scavillo, *Business Process Management: The SAP Roadmap*. SAP PRESS, 2008.

[3] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009, to appear. See also http://www.event-b.org.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, “An open extensible tool environment for event-b,” in *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods, ICFEM 2006*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, November 2006, pp. 588–605.

[5] E. Ball and M. Butler, “Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction,” in *Methods, Models and Tools for Fault Tolerance*, ser. Lecture Notes in Computer Science, M. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, Eds., 2009, vol. 5454, pp. 104–129.

[6] T. S. Hoang, A. Fürst, and J.-R. Abrial, “Event-b patterns and their tool support,” in *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, D. V. Hung and P. Krishnan, Eds. IEEE Computer Society, 2009, pp. 210–219.

[7] A. Iliasov, “Design Components,” Ph.D. dissertation, School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom, 2008.

[8] D. Cansell, D. Méry, and J. Rehm, “Time Constraint Patterns for Event B Development,” in *B 2007: Formal Specification and Development in B, 7th Intl. Conf. of B Users, Besançon, France, January 17-19, 2007*, ser. Lecture Notes in Computer Science, J. Julliand and O. Kouchnarenko, Eds., vol. 4355. Springer, 2007, pp. 140–154.

[9] M. Rached, J.-P. Bodeveix, M. Filili, and O. Nasr, “A Timed B Method for Modelling Real Time Reactive Systems,” in *Proc. 2nd South-East European Workshop on Formal Methods*, Nov 2005, pp. 181–195.

[10] J. Rehm, “A Duration Pattern for Event-B Method,” in *Proc. 2nd Junior Workshop on Real-Time computing*, 2008.

[11] ——, “From absolute-time to relative-countdown: Patterns for model-checking,” Hyperarticles en Ligne, Article hal-00319104 at `hal.archives-ouvertes.fr`, May 2008.

[12] B. Randell, “Recursively Structured Distributed Computing Systems,” in *Symposium on Reliability in Distributed Software and Database Systems*, 1983, pp. 3–11.

[13] A. M. Salkham, “Fault Detection, Isolation and Recovery (FDIR) in On-Board Software,” Master’s thesis, Chalmers University of Technology, 2005.

[14] D. Prandi, P. Quaglia, and N. Zannone, *Coordination Models and Languages*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 5052, ch. Formal analysis of BPMN via a translation into COWS, pp. 249–263.

[15] S. Hinz, K. Schmidt, and C. Stahl, "Transforming BPEL to Petri Nets," in *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds., vol. 3649. Springer-Verlag, 2005, pp. 220–235.

[16] H. Verbeek and W. van der Aalst, "Analyzing BPEL Processes using Petri Nets," in *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, Florida International University, Miami, Florida, USA, 2005*, D. Marinescu, Ed., 2005, pp. 59–78.

[17] S. Nakajima, "Model-Checking Behavioral Specification of BPEL Applications," *Electronic Notes in Theoretical Computer Science*, vol. 151, no. 2, pp. 89–105, 2006, proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005).

[18] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel, "Model checking service compositions under resource constraints," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 225–234.

[19] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services using Process Algebra," in *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 2004, pp. 43–50.

[20] R. Farahbod, U. Glässer, and M. Vajihollahi, "Specification and validation of the business process execution language for web services," in *Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop, ASM 2004, Lutherstadt Wittenberg, Germany, May 24-28, 2004. Proceedings*, ser. Lecture Notes in Computer Science, W. Zimmermann and B. Thalheim, Eds., vol. 3052. Springer-Verlag, 2004, pp. 78–94.

[21] I. Aït-Sadoune and Y. Aït-Ameur, "A proof based approach for modelling and verifying web services compositions," in *Proc. 14th IEEE Intl. Conf. on Engineering of Complex Computer Systems*, June 2009.

[22] L. Laibinis and E. Troubitsyna, "Fault Tolerance in a Layered Architecture: A General Specification Pattern in B," in *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*. IEEE Computer Society, 2004, pp. 346–355.

[23] A. Iliasov, "Refinement patterns for rapid development of dependable systems," in *Proc. 2007 Workshop on Engineering Fault Tolerant Systems*, N. Guelfi, H. Muccini, P. Pelliccione, and A. Romanovsky, Eds. ACM, 2007.