Behaviour Model Elaboration using Partial Labelled Transition Systems

Sebastian Uchitel, Jeff Kramer, Jeff Magee
Department of Computing, Imperial College London
Huxley Building, South Kensington Campus
London, SW7 2AZ, U.K.

[s.uchitel, jk, jnm]@doc.ic.ac.uk

ABSTRACT
State machine based formalisms such as labelled transition systems (LTS) are generally assumed to be complete descriptions of system behaviour at some level of abstraction: if a labelled transition system cannot exhibit a certain sequence of actions, it is assumed that the system or component it models cannot or should not exhibit that sequence. This assumption is a valid one at the end of the modelling effort when reasoning about properties of the completed model. However, it is not a valid assumption when behaviour models are in the process of being developed. In this setting, the distinction between proscribed behaviour and behaviour that has not yet been defined is an important one. Knowing where the gaps are in a behaviour model permits the presentation of meaningful questions to stakeholders, which in turn can lead to model exploration and thus more comprehensive descriptions of the system behaviour. In this paper we propose using partial labelled transition systems (PLTS) to capture what remains to be defined of the system behaviour. In the context of scenario synthesis, we show that PLTSs can be used to support the iterative incremental elaboration of behaviour models.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Requirements, Specifications – *Languages, tools.*

General Terms
Design, Languages, Verification.

Keywords
Partial LTS, Model elaboration, Synthesis, Scenarios

1. INTRODUCTION
1.1 Motivation
Behaviour models are precise, abstract descriptions of the intended behaviour of a system. Behaviour models have solid mathematical foundations that can be used to support rigorous analysis and mechanical verification of properties. Effective techniques and tools have been developed for this purpose and have shown that behaviour modelling and analysis are successful in uncovering the subtle errors that can appear when designing concurrent and distributed systems [5, 6].

Although there is substantial benefit to be gained in using behaviour models for developing complex systems, adoption of behavioural modelling and verification technologies has been slow. One of the main reasons for this is that the construction of behaviour models remains a difficult task that requires considerable expertise and effort. *Our aim is to develop methods and tools that support the construction and elaboration of behaviour models*. Such methods and tools are crucial for the adoption of sound model-based engineering methods for distributed software.

State machine based formalisms such as labelled transition systems (LTS) [12] are commonly used to describe system behaviour. These formalisms are generally assumed to be complete descriptions of system behaviour at some level of abstraction (a fixed alphabet of actions): if a labelled transition system cannot exhibit a certain sequence of actions in its alphabet, it is assumed that the system or component it models cannot or should not exhibit that sequence.

Although this assumption can be a valid one when reasoning about the properties of a finished design proposal, it is not so when behaviour models are in the process of being developed, such as during the requirements process. In this setting the distinction between proscribed behaviour and behaviour that has not yet been defined is an important one. Knowing where the gaps are in a behaviour model permits the presentation of meaningful questions to stakeholders, which in turn can lead to model exploration and potentially more comprehensive descriptions of the system behaviour [19]

In this paper we propose using partial labelled transition systems (PLTS) to capture what remains undefined of the system behaviour. PLTSs extend LTSs by explicitly modelling in each state the set of actions that may not occur, i.e. the set of proscribed actions at each state. Given a state, actions that are neither in the state's set of proscribed actions nor have an outgoing transition from the state are actions for which the system behaviour at that state is unknown. In this paper, we demonstrate the usefulness of such models in the context of scenario synthesis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE '03, September 1 - 9, 2003, Helsinki, Finland
Copyright 2000 ACM 1-58113-743-5/03/0009…$5.00.

1.2 Context

Scenario-based specifications (e.g. [1, 10]) are partial descriptions of system behaviour. A scenario conveys instance level information; it depicts an example of how system components should interact. Hence, a scenario-based specification will typically have many scenarios that cover most common system behaviours and possibly some exceptional ones too. In contrast to behaviour models, scenario-based specifications are not particularly well suited for exhaustive description of all possible system traces and it is natural to assume that the absence of a scenario in a specification does not imply that it is an undesired system trace.

Some scenario-based notations provide mechanisms for explicit specification of undesired system behaviour (e.g. conditions [8, 11], negative scenarios [19]). Nevertheless scenario-based specifications will generally leave gaps in the specification; that is, examples of system behaviour that have not been described explicitly as positive (intended) or negative (unintended) system behaviour.

The difference in interpretation of scenarios and state machines is one of the causes for the former to be used in early requirements phases of the development life-cycle, where system descriptions are relatively partial and require elaboration; while the latter tend to be used at more advanced stages such as design, where a more comprehensive knowledge of the system is available. This separation has led to significant efforts in developing synthesis techniques that support the construction of state-machine models from scenario-based descriptions [1, 16, 20].

The issue of moving from a partial to complete specifications is addressed by scenario synthesis techniques in many different ways. However, they all make assumptions on what to do with unspecified system behaviour because their target output requires behaviours be either positive or negative behaviour.

Consequently, from the perspective described above, approaches to scenario synthesis lose the distinction between positive and negative behaviour and behaviour that has not yet been defined. In the context of supporting the elaboration of behaviour models this is at best a missed opportunity and at worst misleading. The gaps in a behaviour model, if detected, can help raise questions to stakeholders, which in turn can lead more comprehensive descriptions of the system behaviour [19].

Hence, there is a case for using, in the context of scenario synthesis, an extended notion of state machine that can explicitly capture undefined behaviour and support reasoning about aspects of system behaviour that need further elaboration.

1.3 Summary

In this paper we use PLTSs as the target model for scenario synthesis. We follow Whittle and Shumman's approach [20] to synthesis, but use PLTS models instead of standard LTS ones. We show how additional and relevant feedback can be obtained when using PLTSs.

More specifically, we start from a set of sequence diagrams [18] of an ATM machine and an Object Constraint Language (OCL) [18] based specification of message pre- and post-conditions. We use the synthesis approach of [20] to build a LTS, and then extend it to a PLTS that models which message preconditions do not hold on each state. We show that the resulting model can be used to identify behavioural aspects of the ATM system that were under-specified in the original specification. These undefined scenarios are not differentiated from proscribed behaviours in the synthesised model produced in [20]; hence missing an opportunity for model elicitation and elaboration. We show how composition of PLTS can be used to combine different partial behaviour models and (potentially) reduce the number of undefined system scenarios. This supports detection and validation of gaps in the system behaviour, rather than focusing on details of specific components which may be irrelevant to the overall system behaviour. Finally we discuss tools support for PLTSs and conclude this paper with some comments on future work.

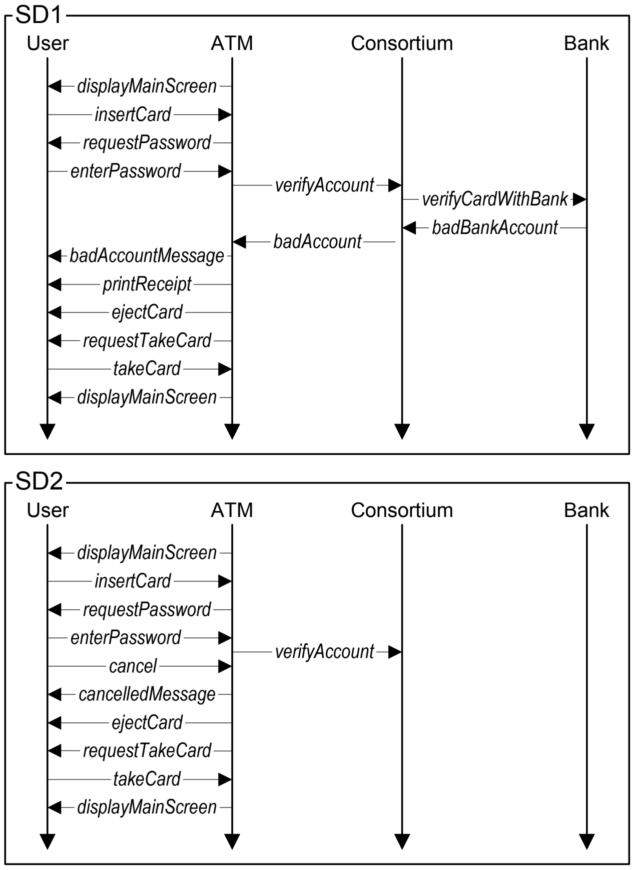

Figure 1 - Scenarios SD1 and SD2

2. LTS Synthesis

The example we use to illustrate our approach is based on a version of the ATM case study presented by Whittle and Schumann in [20]. A number of sequence diagrams (depicted in Figure 1 and Figure 2) describe how a user operates a bank account by interacting with an ATM. The ATM is connected to a network run by a consortium, which in turn interacts with the bank. In addition to the scenarios, pre- and post-conditions for some scenario messages are given in OCL (Figure 3). The post-conditions specify how messages modify the values of a set of ATM state variables (*cardIn*, *cardHalfway*, *passwdGiven*, *card*, and *passwd*). The pre-conditions specify the values these variables are expected to have before a message occurs.

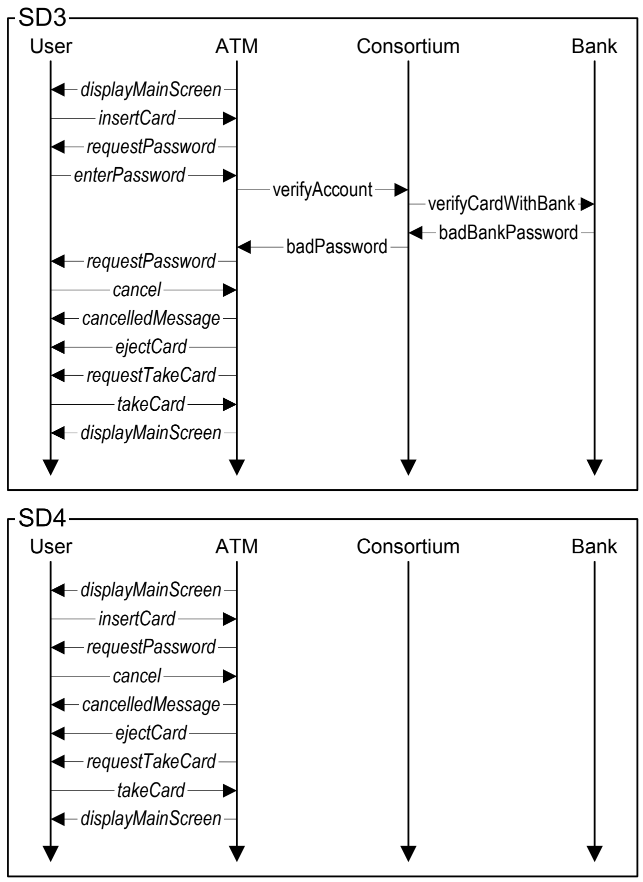

Figure 2 - Scenarios SD3 and SD4

```
cardIn, cardHalfway, passwdGiven : Boolean
card : Card
passwd : Sequence
insertCard(c : Card)
pre : cardIn = false
post: cardIn = true and card = c
enterPassword(p : Sequence)
pre : passwdGiven = false
post: passwdGiven = true and passwd = p
takeCard()
pre : cardHalfway = true
post: cardHalfway = false and cardIn = false
displayMainScreen()
pre: cardIn = false and cardHalfWay = false
post:
requestPassword()
pre : passwdGiven = false
post:
ejectCard()
pre : cardIn = true
post: cardIn = false and cardHalfway = false and card
= null and passwd = null and passwdGiven = false
requestTakeCard()
pre : cardHalfway = true
post:
canceledMessage()
pre : cardIn = true
post:
```

Figure 3 - OCL pre- and post-conditions

In [20] a synthesis procedure is presented for automatically generating LTSs from the combination of the scenarios and the pre- and post-conditions. Using the pre- and post conditions, the procedure first infers the value of state variables at specific points of the scenarios. For example, for *displayMainScreen*, the first message in SD1, the OCL specification states a pre-condition that allows inferring that the value of *cardIn* and *cardHalfway* should be false at the beginning of SD1.

By considering all message pre- and post-conditions and using the unification and frame action techniques defined in [20] it is possible to infer further information on the value of state variables throughout the available scenarios. Consequently, it is possible to assign a (possibly partial) valuation of state variables to every scenario state (the gap in a scenario instance between two consecutive events). The valuations are then used to infer which scenario states should be modelled with one state in the LTS to be synthesised. For more details concerning the synthesis procedure, the interested reader can refer to [20].

3. Undefined Behaviour

3.1 Motivation

Figure 4 depicts the LTS for the ATM component synthesised from the scenario and OCL specification. As expected, the model captures the sequences of interactions the ATM component performs in the scenarios. For instance the LTS models an ATM that is capable of performing the sequence of actions <*displayMainScreen*, *insertCard*, *requestPassword*, *enterPassword*, *verifyAccount*…> of scenario SD1. Additionally, by omission, the LTS also models the sequences of actions that the ATM cannot perform. Thus, the ATM cannot perform sequences with prefix <*displayMainScreen*, *insertCard*, *insertCard*> because after performing *displayMainScreen* and *insertCard* the LTS is in state 2, which does not have any outgoing transitions labelled *insertCard*. For exactly the same reasons, the ATM LTS cannot perform the following sequence: <*displayMainScreen*, *insertCard*, *ejectCard*>.

However, a closer inspection of the LTS and the OCL specification reveals that the LTS is over-specifying the behaviours that the ATM should prohibit. Table 1 shows the value of the OCL variables in each state of the ATM LTS (where t, f, p, c, and – are respectively true, false, a password, a card, and null). Considering that the pre-condition of message *insertCard* requires variable *cardIn* to be false, we can infer that in state 2 *insertCard* should not occur. This is consistent with the fact that the LTS for the ATM component does not allow <*displayMainScreen*, *insertCard*, *insertCard*>. On other hand, message *ejectCard* requires *cardIn* to be true, hence its precondition is satisfied in state 2. Consequently, there is no reason to dismiss the possibility of the ATM performing the sequence <*displayMainScreen*, *insertCard*, *ejectCard*>. However, the LTS for the ATM component does not allow this sequence. Clearly, our knowledge of sequences <*displayMainScreen*, *insertCard*, *insertCard*> and <*displayMainScreen*, *insertCard*, *ejectCard*> is different. We know the first one should not occur because it would violate the *insertCard* pre-condition. For the second sequence we know it does not violate any pre-conditions, thus it may be a valid ATM behaviour. This means that it may well be a situation that has not been explicitly specified or that has not even been considered by stakeholders. Hence it is an opportunity for providing feedback that may trigger new scenarios or strengthened pre-conditions. Either way, it is an opportunity for eliciting further information and elaborating the system's behaviour model.

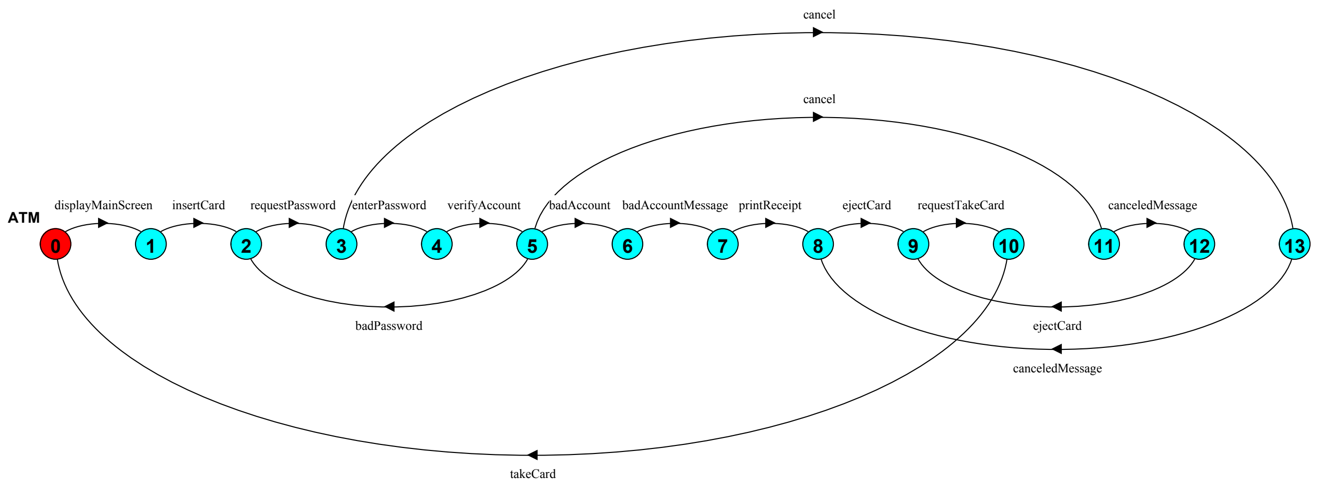

Figure 4 - Synthesized LTS for the ATM component

Table 1 - Valuation of variables on states

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
CardIn	f	f	t	t	t	t	t	t	t	f	f	t	t	t
cardHalfway	f	f	f	f	f	f	f	f	f	t	t	f	f	f
passwdGiven	t	t	f	f	t	t	t	t	t	f	f	t	t	f
Card	-	-	c	c	c	c	c	c	c	-	-	c	c	c
Passwd	-	-	-	-	p	p	p	p	p	-	-	p	p	-

3.2 Tool Support

We used the Labelled Transition System Analyser (LTSA) [15] to automate the detection of undefined behaviour in the LTS generated by the scenario synthesis approach of [20]. The LTSA tool provides model checking and animation functionality over behaviour models written in the Finite State Processes (FSP) process algebra [15]. For this we first we extended the LTS depicted in Figure 4 with an extra sink state to capture when an action is not enabled in a state. In other words, we extended a the LTS (S, L, Δ, q) into a LTS $(S \cup \{\bot\}, L \cup \{undef\}, \Delta', q)$ where $\Delta' = \Delta \cup \{(\bot, undef, \bot)\} \cup \{(s, l, \bot) \mid \forall s'. (s, l, s') \notin \Delta)$. In this extended model, all traces that lead to the sink state are potential examples of traces that lead to states in which an action is undefined. However, they will only be true examples if the precondition for an action *l* that leads to the sink state holds (because if it holds, the action could be fired, yet the synthesized LTS did not have a transition for it). Hence, detection of examples of traces that lead to states in which actions are undefined depends on analysis of pre-conditions.

The approach to detection of undefined behaviour relies on the fluent linear temporal logic (FLTL) model checking capabilities of LTSA. FLTL is a linear temporal logic that allows reasoning on the effects of actions on the state of the system. A fluent is an abstract state that is defined on the occurrence of visible system actions. FLTL allows expressing temporal properties over fluents.

We defined fluents to capture the value of the state variables that appeared in the OCL specification of Figure 3. For instance variable *cardIn* was modelled with the following fluent definition:

```
fluent CARDIN=<insertCard, {ejectCard , takeCard}>
```

The definition states that the fluent CARDIN becomes true when *insertCard* occurs and remains true until either *ejectCard* or *takeCard* occur. The fluent becomes false once *ejectCard* or *takeCard* occur and remains false until *insertCard* occurs. Note that the fluent CARDIN is defined from the OCL post-conditions of Figure 3 in which variable *cardIn* appears: *insertCard*, *ejectCard*, and *takeCard*.

Fluent generation was performed manually, however automating this step, for a subset of OCL predicates, is possible.

Having modelled OCL state variables with fluents, FLTL properties can be used to detect traces leading to states in which the precondition of a certain message holds yet the message is undefined in the state. For example, the following FLTL formula asserts that it is always true that if the variable cardIn is not true, then the sequence *insertCard*, *undef* does not occur[1].

```
assert INSERTCARD_UNDEF =
         [](!CARDIN -> !X(insertCard && X undef ))
```

A violation to this assertion would be a trace modelling a situation in which variable *cardIn* becomes false and action *insertCard* is undefined. Using LTSA, the assertion can be checked producing the following output.

```
Trace to property violation in INSERTCARD_UNDEF:
      displayMainScreen
      insertCard             CARDIN
      requestPassword        CARDIN
      cancel                 CARDIN
      canceledMessage        CARDIN
      ejectCard
      insertCard             CARDIN
      undef                  CARDIN
   Analysed in: 40ms
```

The left column of the output is a trace that violates the FLTL assertion. The violation is a scenario in which after a session, the ATM ejects the card, which is left halfway in the machine, and the user instead of taking the card pushes it back in. This scenario is the one discussed previously and depicted in Figure 5. However, in this case we have detected and generated the scenario automatically using FLTL model checking. The right column of the LTSA output shows CARDIN on lines in which the

[1] `[]` and `X` correspond to the temporal operators always and next, `!` and `->` correspond to logical negation and logical consequence.

variable is true at that stage of the violation trace. Hence, the variable is false until *insertCard* occurs, and remains so until eject card occurs. Then *insertCard* makes the variable true again and it remains true until the end of the violation trace.

4. Partial Labelled Transition Systems

As described above, there is benefit to be gained from differentiating in LTS models behaviour that is known to be undesired from behaviour that is not yet known to be positive or negative. In the previous section, we used LTSs with FLTL to detect undefined behaviour. However, the undefined behaviour is not explicitly represented in the LTS model, it is inferred using information that is in the OCL specification of Figure 3. As we show in subsequent sections, distinguishing undefined behaviour from proscribed behaviour explicitly in a behaviour model can provide additional advantages.

To distinguish proscribed and undefined behaviour we extend the notion of LTS to support explicit modelling of proscribed behaviour. Each LTS state is associated with a set of labels. These labels model actions that are explicitly proscribed at that state.

More formally, we define *States* as the universal set of states. We define *Labels* as the universal set of action labels.

Definition 1 (Labelled Transition Systems) A labelled transition system (LTS) P is a structure (S, L, Δ, q) where:
- $S \subseteq \mathit{States}$ is a finite set of states
- $L = \alpha(P)$ where $\alpha(P) \subseteq \mathit{Labels}$ is a set of labels that denotes the communicating alphabet of P
- $\Delta \subseteq (S \times L \times S)$ defines the labelled transitions between states
- $q \in S$ is the initial state

Definition 2 (Partial Labelled Transition Systems) A partial labelled transition system (PLTS) P is a structure (S, L, Ψ, Δ, q) where $P' = (S, L, \Delta, q)$ is a LTS, $\Psi \subseteq (S \times \alpha(P'))$ defines the proscribed labels on states and $(s, a) \in \Psi$ implies that $(s, a, s') \notin \Delta$ for all $s' \in S$.

Given a PLTS $P = (S, L, \Psi, \Delta, q)$, we use $P \xrightarrow{a} P'$ if $P' = (S, L, \Psi, \Delta, q')$ and $(q, a, q') \in \Delta$. We say that a is enabled in P if there is P' such that $P \xrightarrow{a} P'$. We also say that a is proscribed in P, denoted $P \not\xrightarrow{a}$ if $(q, a) \in \Psi$.

Note that if an action is proscribed at a state then we require there be no outgoing transitions from that state with the same label. Consequently, we have that in each state every label of the PLTS alphabet is enabled, proscribed or undefined. That is, the following equation holds for each state s:

$$\alpha(P) = \mathit{enabled}(s) \cup \mathit{proscribed}(s) \cup \mathit{undefined}(s)$$

where:

- $\mathit{enabled}(s) = \{a \in \alpha(P) \mid \exists s'.\ (s,a,s') \in \Delta\}$,

- $\mathit{proscribed}(s) = \{a \in \alpha(P) \mid (s,a) \in \Psi\}$,

- $\mathit{undefined}(s) = \{a \in \alpha(P) \mid (s,a) \notin \Psi \wedge \forall s'.\ (s,a,s') \notin \Delta\}$.

In a sense, the set of proscribed actions on a state can be seen as the refusal set in CSP semantics [9]. Apart from the fact that refusals are defined considering unobservable actions, here we do not require an action for which there is no enabled transition to be refused. In addition, note that if for all s in S we have $\mathit{undefined}(s) = \emptyset$, the PLTS can be considered a LTS.

If we contrast the preconditions of Figure 3 with the valuation of state variables for each state of the ATM LTS (Table 1) we can determine which messages should not occur in each state. For instance, the precondition of *insertCard* determines that it cannot occur in states 2 to 8, 11 to 13. Consequently, we can extend the LTS of Figure 3 with a set Ψ modelling the messages proscribed at each state. Cells marked "*p*" in Table 2 represent the pairs $(s, a) \in \Psi$ for the ATM PLTS.

We can then add to Table 2, the information on enabled messages for each state (cells marked "*e*"). For example, as a transition labelled *insertCard* has been defined from state 1 to state 2, *insertCard* is marked as enabled in state 1 in Table 2.

Pairs of messages and labels that are not marked with either "*e*" or "*p*" are highlighted with "?" and model the states where messages could occur (according to their preconditions), but the consequences of such occurrences are not yet known. Thus, we have states 0, 9 and 10 modelling that *insertCard* could occur, but the state to which its occurrence would lead to is not known.

Table 2 can now be used to prompt stakeholders on hypothetical situations. For instance, the fact that in state 0, insert card is undefined may prompt the following question: Can a card be inserted into the ATM before a message is displayed?

Table 2 - Classification of ATM states

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
insertCard	?	e	p	p	p	p	p	p	p	?	?	p	p	p
enterPassword	?	?	?	e	p	p	p	p	p	p	p	p	p	?
takeCard	p	p	p	p	p	p	p	p	p	?	e	p	p	p
displayMainScreen	e	?	p	p	p	p	p	p	p	p	p	p	p	p
requestPassword	?	?	e	?	p	p	p	p	p	p	p	p	p	?
ejectCard	p	p	?	?	?	?	?	?	e	p	p	?	e	?
cancelledMessage	p	p	?	?	?	?	?	?	?	p	p	e	?	e
requestTakeCard	p	p	p	p	p	p	p	p	p	e	?	p	p	p

Note that PLTS differ from multi-valued state-machines (e.g. [4]), in that in the latter transitions are assigned truth values (e.g. true, false, unknown) rather than transition labels being undefined at states. Thus, multi-valued state-machines require much finer grained knowledge about what is unknown. This is discussed further in the section on related work.

5. Relations on PLTSs

In this section we discuss equivalence and simulation relations between PLTSs. Equivalence relations provide a semantic framework for constructing and comparing the behaviour represented by PLTSs. Various notions of equivalence have been used to compare the behaviour represented by two LTSs, these include strong and weak (or observational) equivalence [17] and trace and failures-divergence equivalence [9]. Equivalence relations can also be used when reducing the state space LTSs to simplify model analysis. We use the notion of PLTS equivalence in the next section to simplify the discussion on how parallel composition of PLTSs works. In this section we also define a simulation relation between PLTSs. Simulation can be used to capture the elaboration process of PLTSs in which the undefined behaviour is gradually defined as either positive or negative information, to yield a PLTS with no undefined behaviour, in other words, to yield a LTS.

We first define strong equivalence, which equates PLTSs that have identical structure. For this we define $\wp$ to be the set of all PLTSs.

Definition 3 (Strong Equivalence) The strong equivalence "$\sim$" is the union of all relations $R \subseteq \wp \times \wp$ satisfying that $(P, Q) \in R$ implies

- $\alpha(P)=\alpha(Q)$
- $\forall a \in \alpha(P)$.
    - $P \xrightarrow{a} P'$ implies $\exists Q'.\ Q \xrightarrow{a} Q'$ and $(P', Q') \in R$
    - $Q \xrightarrow{a} Q'$ implies $\exists P'.\ P \xrightarrow{a} P'$ and $(P', Q') \in R$
- $\forall a \in \alpha(P).\ P \not\xrightarrow{a}$ if and only if $Q \not\xrightarrow{a}$

Strong equivalence of PLTSs extends that of LTSs in that equivalent PLTSs are not only required to have transitions that lead to equivalent PLTSs but also are required to proscribe the same set of actions. A consequence of the requirement of equivalent PLTSs to have the same alphabet is that they will also be undefined on the same actions.

Weaker notions of equivalence can be defined on PLTS. If some notion of abstraction is introduced, possibly distinguishing between actions that are observable and unobservable for the environment of a PLTS, then an equivalence relation similar to that of observational equivalence between LTSs can be defined. A weaker equivalence relation that may also prove to be useful is that of trace equivalence.

A useful notion in the context of PLTSs and behaviour elaboration is that of a PLTS being "more defined" than another PLTS. The elaboration process can thought of as producing a sequence of process in which each one is "more defined" than the previous one. In principle, the process would start with a completely undefined PLTS (i.e. $(\{q\}, Labels, \emptyset, \emptyset, q)$) and finish with a PLTS where all actions are defined on all states, in other words the process would finish with a LTS: (S, L, Ψ, Δ, q) where for all state s in S, $undefined(s) = \emptyset$.

In essence, an elaboration process consists in iteratively replacing unknown behaviour with either positive or negative behaviour until there is a complete model of the system. This progression from the PLTS in which everything is unknown to the one in which there are no undefined actions can be captured with the notion of simulation.

Definition 4 (Simulation) The simulation relation "$\leq$" is the union of all relations $R \subseteq \wp \times \wp$ satisfying that $(P, Q) \in R$ implies that

- $\forall a \in \alpha(P)$
    - $P \xrightarrow{a} P'$, implies $\exists Q'.\ Q \xrightarrow{a} Q'$ and $(P', Q') \in R$,
    - $P \not\xrightarrow{a}$ implies $Q \not\xrightarrow{a}$

Intuitively, simulation captures the notion of "more defined than". If Q simulates P ($P \leq Q$) then Q has all of P's positive and negative behaviour, but may be proscribe additional actions or exhibit additional transitions. Additional positive or negative behaviour corresponds to a step in the elaboration process were additional information on system behaviour is acquired, possibly resulting from user feedback. Note that we do not require the alphabets of P and Q to be equal. The alphabet of Q may drop an action of P as long as it was undefined in all of P's states. Conversely, Q may introduce actions that were not in the alphabet of P.

In the context of scenario-based synthesis approaches, preservation of the simulation relation between synthesised models is desirable. Suppose we have scenario specification from which a PLTS model is synthesised. In addition, suppose the PLTS is used to generate feedback on behavioural aspects that need further elaboration and as a consequence new scenarios (either positive or negative) are added to the specification. In principle, it would be reasonable to expect the new scenario specification to yield a PLTS that can simulate (that is "more defined than") the PLTS that was synthesized from the original specification. We have not yet attempted to prove that the simulation relation is preserved by the specific synthesis approach we have used based on [20]. In this paper, our aim is to show how PLTSs can support the elaboration of behaviour models. Research into specific scenario synthesis approaches using PLTSs is something we intend to do as future work

6. Composition of Partial Behaviour Models

Although benefits may be obtained from inspecting a PLTS, a more appealing approach is to generate feedback in the form of scenarios. We wish to compose partially specified models of the system components appearing in scenarios, to reason about how they interact, and to detect whether or not they reach states for which certain message labels are undefined. For instance, if a PLTS for the user, consortium and bank where built, the scenario of Figure 5 could automatically be generated through some form of reachability analysis. The scenario depicts the case where after a session, the ATM ejects the card, which is left halfway in the machine, and instead of taking the card the user pushes it back in.

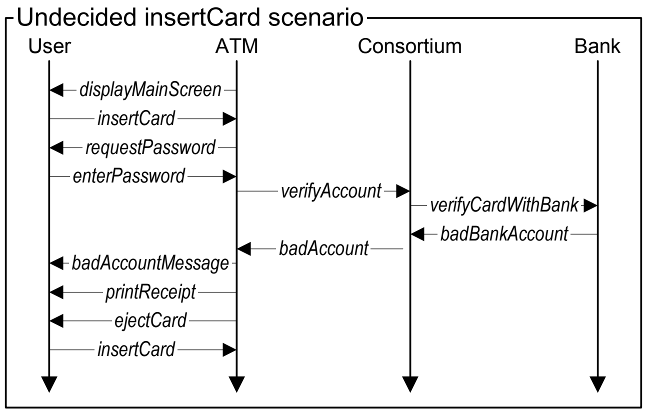

Figure 5 - Undefined *insertCard* scenario

We therefore need to extend the notion of parallel composition of LTSs to PLTSs. Intuitively, parallel composition of LTSs models a system in which components execute asynchronously and synchronize on shared message labels. Given a shared label l, one LTS can take an l-labelled transition if and only if the LTS it is being composed with can do so too. Consequently, a LTS in a state where l is not enabled will prevent the other LTS from taking a transition labelled l.

In the parallel composition of PLTSs this changes because l not being enabled does not imply that l is proscribed. For instance, suppose we are composing PLTSs P and Q, which are in states p

and q respectively. In addition suppose there is a shared label l that is enabled in p. If l is undefined in state q then l should also be undefined in the composite process because we do not know if Q can synchronize on l when in q. Clearly, if l is proscribed in q, then l should be also proscribed in the composite process (as with standard LTSs).

On the other hand, consider that l is undefined in state p. If l has been explicitly proscribed on state q then l should also be proscribed in the composite process. This means that the fact that l is undefined in p is irrelevant with respect to the composite behaviour. In other words, although we have a gap in the specification of component P, providing feedback concerning it is not necessary in the context of Q. In essence, proscribed takes precedence over undefined that takes precedence over enabled.

Table 3 provides a summary intuition as to how enabled, proscribed and undefined message labels work together in parallel composition of PLTSs.

Table 3 - Proscribed, enabled and undefined messages in PLTS parallel composition

	Enabled	*Proscribed*	*Undefined*
Enabled	Enabled	Proscribed	Undefined
Proscribed	Proscribed	Proscribed	Proscribed
Undefined	Undefined	Proscribed	Undefined

Definition 5 (Parallel Composition) Let P_1 and P_2 be PLTSs where $P_i = (S_i, L_i, \Psi_i, \Delta_i, q_i)$. Their parallel composition denoted $P_1||P_2$ is a PLTS $(S_1 \times S_2, L_1 \cup L_2, \Psi, \Delta, (q_1, q_2))$ where Δ and Ψ are the smallest relation that satisfies rules in Figure 6 and Figure 7 respectively.

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \parallel P_2 \xrightarrow{a} P_1' \parallel P_2'} \quad (a \notin \alpha(P_2))$$

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \parallel P_2 \xrightarrow{a} P_1' \parallel P_2'} \quad (a \notin \alpha(P_1))$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \parallel P_2 \xrightarrow{a} P_1' \parallel P_2'}$$

Figure 6 - Rules for PLTS transition relation

$$\frac{P_1 \not\xrightarrow{a}}{P_1 \parallel P_2 \not\xrightarrow{a}} \qquad \frac{P_2 \not\xrightarrow{a}}{P_1 \parallel P_2 \not\xrightarrow{a}}$$

Figure 7 - Rules for PLTS proscribed relation

More generally, parallel composition of PLTSs allows us to combine different partial behaviour models and (potentially) reduce the number of unclassified system scenarios. Allowing detection and validation of gaps in the behaviour specification facilitates focusing on the emerging behaviour of system components working together, rather than on details of components that may be irrelevant to the overall system behaviour.

Consider the ATM example, and suppose we have additional information as to how the card is managed between the ATM and the user, which for short we call "the card protocol". We describe the behaviour with a PLTS depicted in Figure 8 and Table 4.

If we compose the behaviour models for the card protocol and the ATM, the resulting PLTS will have, for instance prohibit the following trace <*displayMainScreen*, *insertCard*, *insertCard*> which, as explained previously, was undefined behaviour in the ATM. Let us see why. The state space of the composite PLTs is the Cartesian product of the state spaces of the card protocol PLTS and the ATM PLTS. We shall refer to states of the composite model as pairs (x, y) where x and y are digits referring to states of the card protocol PLTS (Figure 8) and the ATM PLTS (Figure 4) respectively.

The composite model starts in state $(0, 0)$ according to Definition 5. As the ATM has *displayMainScreen* enabled in state 0, and *displayMainScreen* is not in the alphabet of the card protocol PLTS, then according to the first second rule of in Figure 6, the composite model can perform *displayMainScreen* and transition into state $(0, 1)$. Because *insertCard* is enabled in both state 0 and 1 of PLTSs Card Protocol and ATM, the third rule in Figure 6 indicates that the composite PLTS can transition to $(1, 2)$. Now the Card Protocol PLTS proscribes message *insertCard* in state 1 (see Table 4) while the ATM considers the same label as undefined in state 2 (see Table 2). Thus, from the first rule of Figure 7, the composite process will not be able to transition on label *insertCard*. Considering that both the ATM and the Card Protocol are deterministic it is easy to see that the composite PLTS can never exhibit the trace <*displayMainScreen*, *insertCard*, *insertCard*>

If we compute the composite model and minimise with respect to the strong semantic equivalence (defined in the previous section), the resulting PLTS will no longer have the following pairs of undefined behaviour (compare with Table 2): {(*insertCard*, 2), (*insertCard*, 9), (*insertCard*, 10), (*ejectCard*, 3), (*ejectCard*, 4), (*ejectCard*, 5), (*ejectCard*, 6), (*ejectCard*, 7), (*requestTakeCard*, 10), (*takeCard*, 9)}.

As a consequence, we now have a composite behaviour model that has fewer gaps requiring stakeholder intervention.

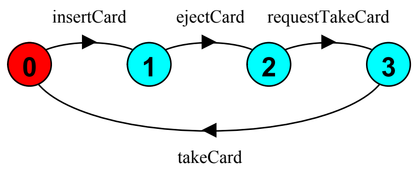

Figure 8 - Card Protocol

Table 4 - Classification of Card Protocol states

	0	1	2	3
`insertCard`	e	p	p	p
`takeCard`	p	p	?	e
`ejectCard`	p	e	p	p
`requestTakeCard`	p	p	e	p

Although the preceding example reduces the number of undefined pairs of states and labels (compared to the ATM component on its own), this is not always the case. Clearly, composition of PLTSs can introduce new cases of undefinedness in the composite behaviour. Thus, parallel composition does not always reduce the number of gaps in the overall specification. This raises the following issue: If gaps are used to generate feedback for users, we risk generating an unmanageable number of scenarios that a user must validate. Although this is true, one might argue that if there is a significant portion of the system behaviour that is unknown surely assuming an answer from the user in order to not overburden them can be dangerous. In addition, the number of queries made to users can be reduced if system level properties and constraints are modelled as LTSs and composed in parallel with the partial behaviour model.

7. Related Work

An area that is closely related to the work presented in this paper is that of multi-valued logics. Traditionally, logics allow only two possible truth-values for any proposition, true and false. Multi-valued logics allow for a range of truth-values. For instance three values can be used to model uncertainty, disagreement [4], and 'unknown' [14]. There are, however, several important differences with respect to our work. Firstly, PLTS are compositional specifications that allow components to be specified individually, composed into sub-systems and minimised with respect to behaviour equivalence. In addition, PLTS do not specify the internal component state; behaviour is described in terms of the actions a component can and cannot perform, and those for which it is not yet known if and how the component would react. In approaches to multi-valued logics atomic propositions are valued in each state, thus properties on actions that produce state changes can only be modelled indirectly with respect to state propositions.

Another related area is that of multi-valued state-machines (e.g. [4]). In these machines multiple truth-values are assigned to transitions. In a three-valued state machine values could be used to model the positive, negative and unknown behaviour of the state machine. With this interpretation, three-valued state-machines require a much finer grain knowledge concerning what is unknown. At a given state, one does not model that the component cannot react to a certain action; rather, it is necessary to model all the target states to which the component could reach through the action, but is not yet known to do so. For instance, in the ATM example we would have to speculate on all the possible destinations of *insertCard* from state 0: transitions labelled *insertCard* with value unknown would be needed from state 0 to all other states. In the setting we propose, this is not useful as the true transition from state 0 for insert card –supposing that it should exist, but has not appeared in the given scenarios– could lead to a new PLTS state altogether.

We envisage using PLTSs to support the elaboration of behaviour models. Unknown behaviour can be modelled explicitly, and then models can be used to query users on whether a particular scenario is possible or not. Our previous work on implied scenarios [19] shares this approach to model elaboration based on scenario generation and validation. However, implied scenarios address a very specific aspect of partial scenario-based descriptions while PLTSs provide a more general framework for model elaboration.

The use of PLTS in the context of scenario synthesis is related to that of Mäkinen and Systä [16] who have worked on the iterative construction of scenario-based specifications. However, scenarios that are fed back to users are the result of over-generalisations of the synthesis procedures used. The scenarios we generate are a result of the behavioural aspects that have been under-specified. In addition, in [16] feedback is given at a component level (i.e. component traces) rather than at a system level.

8. Conclusions and Future Work

In this paper we have demonstrated the utility of using partial labelled transition systems to support the elaboration of behaviour models. By explicitly modelling the aspects of system behaviour that are unknown, it is possible to generate meaningful feedback to users leading to more comprehensive descriptions of the system behaviour.

We have exemplified our approach using scenario-based notations and scenario synthesis because they are being used increasingly to support behaviour model construction. In particular, in this paper we have used an example based on scenarios and OCL pre- and post-conditions. However, PLTS can be used in a similar way to support behaviour model elaboration in the context of other scenario synthesis approaches that use MSC conditions [8, 11] or negative scenarios [19] instead of OCL. Furthermore, we believe that PLTSs can be used successfully to support behaviour model elaboration in settings other than scenario synthesis.

We are currently experimenting with PLTSs as the target for scenario synthesis. We aim to implement synthesis procedures and to develop methods for constructing and reasoning on the partial behaviour models. This work will extend our existing MSC-LTSA tool [2]. We have ongoing work on modelling PLTSs onto LTSs to allow some forms of PLTS analysis reusing existing model checking technology. The difficulty of here is to ensure the encoding is preserved by the parallel composition and equivalence relations on LTSs. We are also looking into using PLTSs in the context of our simulation tool [3] to guide model exploration and elaboration. PLTSs could also provide a framework for guided play-in scenarios [7] where a user using a PLTS model-based simulation moves seamlessly from replaying the already defined model to extending it with previously undefined behaviour. Finally, we intend to use PLTSs to combine goal directed requirement elaboration techniques such as in [13] with scenario synthesis approaches.

9. Acknowledgements

We gratefully acknowledge the support of the European Union under grant STATUS (IST-2001-32298), and EPSRC grant READS GR/S03270/01.

10. References

[1] *First ICSE Workshop on Scenarios and State Machines: Model, Algorithms and Tools (SCESM'02)*. 2002.

[2] Chatley, R., *et al. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios* in *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. 2003. Warswaw.

[3] Chatley, R., *et al. Model-based Simulation of Web Applications for Usability Assessment* in *ICSE Workshop on "Bridging the Gaps Between Software Engineering and Human-Computer Interaction"*. 2003. Portland, May 2003.

[4] Chechik, M., S. Easterbrook, and B. Devereux. *Model Checking with Multi-Valued Temporal Logics* in *31st IEEE International Symposium on Multiple Valued Logics (ISMVL'01)*. 2001. Warsaw.

[5] Clarke, E.M. and J.M. Wing, *Formal Methods: State of the Art and Future Directions.* ACM Computing Surveys, 1996. **28**(4): p. 626-643.

[6] Cleaveland, R. and S.A. Smolka, *Strategic Directions in Concurrency Research.* ACM Computing Surveys, 1996. **28**(4): p. 607-625.

[7] Harel, D., *From Play-In Scenarios to Code: An achievable Dream.* IEEE Software, 2001. **34**(1): p. 53-60.

[8] Harel, D. and W. Damm. *LSCs: Breathing Life into Message Sequence Charts* in *3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*. 1999. New York: Kluwer Academic.

[9] Hoare, C.A.R., *Communicating Sequential Processes*. 1985, Englewood Cliffs, New Jersey: Prentice Hall.

[10] ITU, *Message Sequence Charts*, 1996, International Telecommunications Union. Telecommunication Standardisation Sector.

[11] ITU, *Message Sequence Charts*, 2000, International Telecommunications Union. Telecommunication Standardisation Sector.

[12] Keller, R., *Formal verification of parallel programs.* Communications of the ACM, 1976. **19**(7): p. 371-384.

[13] Lamsweerde, A.v., R. Darimont, and P. Massonet. *Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt* in *Second IEEE International Symposium on Requirements Engineering*. 1995. York: IEEE CS Press.

[14] Lukasiewicz, J., *Selected Works*. 1970, Amsterdam: North-Holland.

[15] Magee, J. and J. Kramer, *Concurrency: State Models and Java Programs*. 1999, New York: John Wiley & Sons Ltd.

[16] Mäkinen, E. and T. Systä. *MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML,* in *23rd IEEE International Conference on Software Engineering (ICSE '01)*. 2001. Toronto.

[17] Milner, R., *Communication and Concurrency*. 1989, London: Prentice-Hall.

[18] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. 1999, Harlow: Addison-Wesley.

[19] Uchitel, S., J. Kramer, and J. Magee. *Negative Scenarios for Implied Scenario Elicitation* in *10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'02)*. 2002. Charleston.

[20] Whittle, J. and J. Schumann. *Generating Statechart Designs from Scenarios* in *22nd International Conference on Software Engineering (ICSE'00)*. 2000. Limerick, Ireland.