Merging Features in Featured Transition Systems

Joanne M. Atlee, Sandy Beidu, Uli Fahrenberg, Axel Legay

Merging Features in Featured Transition Systems

Joanne M. Atlee
University of Waterloo
jmatlee@uwaterloo.ca

Sandy Beidu
University of Waterloo
sandybeidu@yahoo.com

Uli Fahrenberg
Inria Rennes
ulrich.fahrenberg@inria.fr

Axel Legay
Inria Rennes
axel.legay@inria.fr

***Abstract*—Featured Transition Systems (FTSs) is a popular representation for software product lines: an entire product line is compactly represented as a single transition-machine model, in which feature-specific behaviour is guarded by feature expressions that are satisfied (or not) by the presence or absence of individual features. In previous work, FTS models were monolithic in the sense that the modeller had to construct the full FTS model of the product line in its entirety. To allow for modularity of FTS models, we propose here a language for extending an existing FTS model with new features. We demonstrate the language using a running example and present results about the language's expressivity, commutativity of feature extensions, feature interactions, and resolution of such interactions.**

I. Introduction

Software product lining is a software development paradigm in which a family of related software products (*e.g.*, similar smart phones, or payroll systems) share a common set of mandatory features and individual products are differentiated by the presence or absence of optional features. Ideally, features are increments of functionality that can be developed independently and combined into products or a product line.

Featured Transition Systems (FTSs) [6] is the prevalent mathematical model for expressing the behaviour of a software product line. An FTS resembles a traditional state-transition machine, except that transitions are guarded by feature expressions (as well as annotated with actions): the feature expression specifies that the transition exists in a product only if the product's features satisfy the feature expression.

As an example, Fig. 1 displays an FTS model of an automated teller machine (ATM). It has a mandatory feature that consists of a cycle of card insertion, PIN entrance, amount specification, cash retrieval, and card retrieval. Hence these transitions will be present in all products. Also shown is an optional "Cancel" feature which allows to cancel transactions, and another optional "Balance" feature which permits the customer to get an account balance. Thus, the transitions belonging to feature Ca will be present only in products which include the Ca feature; the balance-labeled transition from state 2 to state 4 will be present only in products that include the B feature.

In previous work, FTS models were monolithic models of full product lines. There was no means of modelling individual features and composing them into products or product-line models, or of specifying feature increments to an existing product-line model. As such, FTSs could not be the mathematical basis for modelling technologies that support feature decomposition, composition, or incremental evolution of a product line.

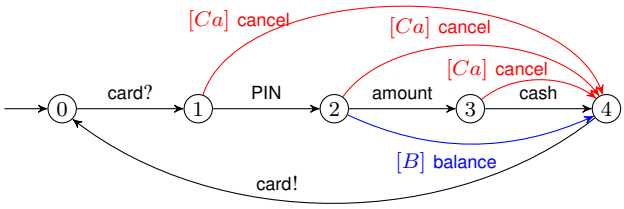

Fig. 1. ATM example

To allow for modularity of FTS models, we propose here a language of *feature merge expressions (FMEs)* for specifying new feature extensions to an existing FTS model. Specifically, given a base FTS model of mandatory behaviour, an FME is a sequence of FTS transformations that add transitions and states and manipulate feature expressions, thereby embedding new feature-specific behavior into the existing FTS. We show that a relatively small set of FME constructs is sufficiently powerful to express any legal extension of an FTS.

From a methodological point of view, we envision application of FMEs to FTSs to be automatized, and we will show below how to achieve this. Once such automatization is available, FMEs should become a powerful tool for product line specification.

In the following we give a brief overview of FTSs. In Section III, we introduce a preliminary language of simple FMEs and demonstrate its utility on a running example; we conclude that section with a proof of expressivity of the simple language. In Section IV, we extend our preliminary language to include variables, which enable the specification of more general model-transformation rules that (may) apply to multiple locations in an existing FTS. We discuss in Section V the issue of commutativity of feature extensions, interactions among features, and how to use FMEs to specify resolutions to interactions. In Section VI we conclude and describe future directions of this work.

II. Featured Transition Systems

A *transition system* (TS) $\mathcal{S} = (S, \Sigma, I, T, \tau)$ consists of a finite set of states S, a nonempty set of initial states $I \subseteq S$, a finite set of actions Σ, and a finite set of transitions T together with a mapping $\tau : T \to S \times \Sigma \times S$. We write $t : s \xrightarrow{a} s'$ to indicate a transition $t \in T$ with $\tau(t) = (s, a, s')$. When necessary, we will distinguish input actions from output labels

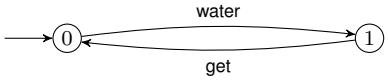

Fig. 2. Vending machine example: base feature

in Σ; in Fig. 1 for example, "card?" denotes the card being input into the ATM, whereas "card!" denotes the card being returned.

Let N be a set of *features*. A *feature expression* is a Boolean expression over N; $\mathbb{B}(N)$ denotes the set of such feature expressions. A *product* is a subset $p \subseteq N$. We denote by **tt** and **ff** the universally true, respectively false, feature expressions. For a product p and a feature expression ϕ, we write $p \models \phi$ if p satisfies ϕ.

A *featured transition system* (FTS) $\mathcal{F} = (S, \Sigma, I, T, \tau, \gamma)$ consists of a TS (S, Σ, I, T, τ) and a mapping $\gamma : T \to \mathbb{B}(N)$ which assigns *feature guards* to transitions.

FTSs are used as compact specifications of *product lines*, or sets of transition systems depending on products. To be precise, the *semantics* of an FTS $\mathcal{F} = (S, \Sigma, I, T, \tau, \gamma)$ is the mapping $[\![\mathcal{F}]\!]$ from products $p \subseteq N$ to TSs given by $[\![\mathcal{F}]\!](p) = (S, \Sigma, I, T', \tau)$ with $T' = \{t \in T \mid p \models \gamma(t)\}$. That is, the TS $[\![\mathcal{F}]\!](p)$ contains precisely those transitions from $\mathcal{F}$ which are enabled for the product p. Two FTSs $\mathcal{F}$, $\mathcal{F}'$ are called *semantically equivalent*, denoted $\mathcal{F} \equiv \mathcal{F}'$, if $[\![\mathcal{F}]\!](p) = [\![\mathcal{F}']\!](p)$ for every product $p \subseteq N$.

III. Simple Feature Merge Expressions

We want to merge a new feature into an already given FTS. The idea is that we specify *rules* which may add states to the FTS, add transitions, and modify existing transitions.

A. Example

We introduce feature merge expressions gradually through a running example. Figure 2 displays a (very) simple vending machine which hands out water to the thirsty traveler, for free. Now we would like to add "Coffee" and "Tea" features to it, noting that these hot beverages do not come for free. To do so, we use a *feature merge expression* (FME) as follows.

$$\begin{aligned} C: &\quad \text{add } 0 \text{ coin } 2 \\ &\quad \text{add } 2 \text{ coffee } 1 \\ T: &\quad \text{add } 0 \text{ coin } 2 \\ &\quad \text{add } 2 \text{ tea } 1 \end{aligned}$$

We shall give a formal syntax of FMEs below, but we can give an intuitive explanation here. The first rule of the C expression specifies that a new, coin-labeled, transition is to be added to the FTS, leading to a state 2 which does not exist. Hence this state is added to the FTS. The second rule then adds a coffee-labeled transition from state 2 to state 1. The result is displayed in Fig. 3.

The FME for T specifies to add another coin-labeled transition from state 0 to state 2, but as there is one such transition already, the effect is a *weakening* of the feature guard on this transition. Additionally, a new tea-labeled transition is added (see Fig. 4).

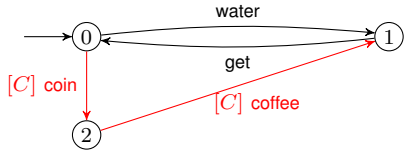

Fig. 3. Vending machine example: base feature plus Coffee

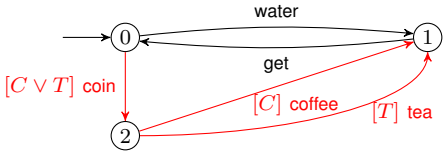

Fig. 4. Vending machine example: base feature plus Coffee and Tea

Next we would like to add a "Credit$C$ard" feature to the vending machine FTS. The FME is as follows:

$$\begin{aligned} CC: &\quad \text{str } 0 \text{ coin } 2 \neg CC \\ &\quad \text{add } 0\ C \vee T \text{ card } 3 \\ &\quad \text{add } 3\ C \vee T \text{ PIN } 2 \end{aligned}$$

The first rule specifies that the coin-labeled transition from state 0 to state 2 is to be strengthened with a feature guard $\neg CC$. Hence the Credit$C$ard feature will disable the coin transition, making credit card the only payment option. If we wanted an inclusive CC feature instead, which does not disable coin payment, then we could have omitted this strengthening rule. Also note that strengthening rules are a necessary part of merge expressions; whereas addition rules *introduce* behaviour, strengthening rules *restrict* behaviour.

The last two rules of the FME specify that a new state is to be added together with two transitions pertaining to the CC feature. Note the new piece of syntax in the addition rules: the transitions added should only be enabled if either the Coffee or Tea feature are present, so they have to be guarded by the feature expression $C \vee T$. The result of the merge is depicted in Fig. 5.

B. Formal Syntax and Semantics

In general, a feature merge expression for a feature F consists of rules of two types:

- addition rules of the form

$$\text{add } s\ \langle \phi \rangle\ a\ s'$$

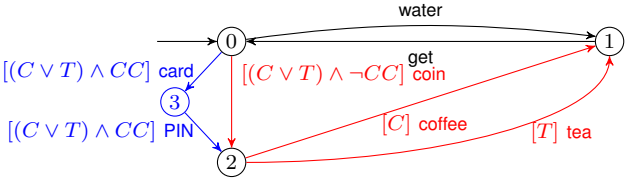

Fig. 5. Vending machine with Coffee, Tea and Credit$C$ard

(the "$\langle\phi\rangle$" denotes an optional part) which specify to *add* a transition (and possibly the states) from s to s' with label a and feature guard $\phi \wedge F$;
- strengthening rules of the form

$$\text{str } s\ a\ s'\ \psi$$

which specify that any existing transition from s to s' with label a has to have its guard *strengthened* by ψ.

As we saw in the example above, addition rules can also (implicitly) specify *weakening* clauses in feature guards: Adding a transition (s, a, s') with feature guard ψ to an FTS which already has a transition (s, a, s'), with feature guard ϕ, has the same semantic effect as weakening the feature guard of the latter transition to $\phi \vee \psi$.

The formal syntax for an FME E is as follows.

$$\begin{aligned} E &::= \phi : IE \mid E; E \\ IE &::= \text{add } s\ a\ s' \mid \text{add } s\ \psi\ a\ s' \\ &\quad \mid \text{str } s\ a\ s'\ \psi \mid IE; IE \end{aligned}$$

Here ϕ can be any feature expression, but will usually be only a single feature. In an FME $\phi : IE$, we call IE an *inner expression*. Hence the first line above specifies the syntax of merge expressions, whereas the second line specifies the syntax of inner expressions.

We give denotational semantics to FMEs. Let $\mathcal{F} = (S, \Sigma, I, T, \tau, \gamma)$ be an FTS and E an FME, then the effect $[\![E]\!](\mathcal{F})$ is a new FTS $\mathcal{F}'$ which is given by structural induction on E:

$$\begin{aligned} [\![E_1; E_2]\!](\mathcal{F}) &= [\![E_2]\!]([\![E_1]\!](\mathcal{F})) \\ [\![\phi : IE]\!](\mathcal{F}) &= [\![IE]\!]_\phi(\mathcal{F}) \\ [\![IE_1; IE_2]\!]_\phi(\mathcal{F}) &= [\![IE_2]\!]_\phi([\![IE_1]\!]_\phi(\mathcal{F})) \\ [\![\text{add } s\ a\ s']\!]_\phi(\mathcal{F}) &= [\![\text{add } s\ \mathbf{tt}\ a\ s']\!]_\phi(\mathcal{F}) \\ [\![\text{add } s\ \psi\ a\ s']\!]_\phi(\mathcal{F}) &= (S \cup \{s, s'\}, \Sigma, I, T \cup \{t_{\text{new}}\}, \tau', \gamma') \\ &\quad \text{where } t_{\text{new}} \notin T \text{ is a new transition and} \\ &\quad \tau'(t) = \begin{cases} \tau(t) & \text{for } t \neq t_{\text{new}} \\ (s, a, s') & \text{for } t = t_{\text{new}} \end{cases} \text{ and} \\ &\quad \gamma'(t) = \begin{cases} \gamma(t) & \text{for } t \neq t_{\text{new}} \\ \phi \wedge \psi & \text{for } t = t_{\text{new}} \end{cases} \\ [\![\text{str } s\ a\ s'\ \psi]\!]_\phi(\mathcal{F}) &= (S, \Sigma, I, T, \tau, \gamma') \quad \text{with} \\ &\quad \gamma'(t) = \begin{cases} \gamma(t) & \text{if } \tau(t) \neq (s, a, s') \\ \gamma(t) \wedge \psi & \text{if } \tau(t) = (s, a, s') \end{cases} \end{aligned}$$

Here the first line shows how to concatenate merge expressions, and the second line gives the rule for how to access an inner expression IE inside an FME $E = \phi : IE$. The semantics of this inner expression is then dependent on its feature guard ϕ, *i.e.*, the meaning of $[\![IE]\!]_\phi(\mathcal{F})$ depends on ϕ.

The third line then shows how to concatenate inner expressions, and the fourth and fifth lines give semantics to addition rules. Note that the second addition rule specifies that a new transition t_{new} be added to the FTS, whereas the states s and s' only are added if they do not already exist in S.

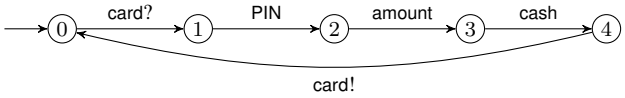

Fig. 6. ATM example: base feature

C. Expressivity

We show here that FMEs are powerful enough to make any legal extension to an FTS.

Theorem 1 *For all FTS $\mathcal{F}$, $\mathcal{F}'$, there exists an FME E such that $[\![E]\!](\mathcal{F}) \equiv \mathcal{F}'$.*

Proof: Let $\mathcal{F} = (S, \Sigma, I, T, \tau, \gamma)$, $\mathcal{F}' = (S', \Sigma', I', T', \tau', \gamma')$. By renaming states if necessary, we can assume that $S \cap S' = \emptyset$. We construct $E = \mathbf{tt} : IE; IE'$, where IE and IE' are inner expressions.

We first strengthen all transitions in $\mathcal{F}$ to make their feature guards unsatisfiable. Write $T = \{t_1, \ldots, t_n\}$ and define $IE_j = \text{str } \tau(t_j)\ \mathbf{ff}$ for $j = 1, \ldots, n$. Let $IE = IE_1; \ldots; IE_n$. It is clear that every transition in $[\![I]\!]_{\mathbf{tt}}(\mathcal{F})$ has unsatisfiable feature guard.

Now we add all transitions in T' to $[\![I]\!]E_{\mathbf{tt}}(\mathcal{F})$. Write $T' = \{t'_1, \ldots, t'_m\}$ and define $IE'_j = \text{add } \tau'(t'_j)\ \gamma'(t'_j)$ for $j = 1, \ldots, m$. Let $IE' = IE'_1; \ldots; IE'_m$. Then $[\![IE; IE']\!]_{\mathbf{tt}}(\mathcal{F}) = [\![E]\!](\mathcal{F})$ is semantically equivalent to $\mathcal{F}'$. ■

IV. Extended Feature Merge Expressions

In order to allow for more generic feature merge expressions, we propose to extend the syntax of FMEs to include variables. We again introduce these through a running example.

Recall the ATM example from Fig. 1, where we for now ignore the $Cancel$ and $Balance$ features. Figure 6 shows the ATM base feature.

Now we would like to add a $Receipt$ feature, which models an ATM where the customer can obtain a receipt after her transaction. Hence R should add a transition labeled rec? from state 4 to a new state 5, asking whether the customer wants a receipt. From state 5, a rec! transition will issue a receipt, leading to another state 6 from which there is a card! transition back to state 0. For the case that the customer does not want a receipt, there should be a transition labeled norec? from state 4 to state 6; also, the R merge expression should strengthen the existing card!-labeled transition.

In order to make our merge expression more generic, we will avoid making explicit reference to existing states like above. Instead, we will specify that the new rec?-labeled transition is to start at the target state of the existing card!-labeled transition; similarly, the new card!-labeled transition is to have the same target state as the existing one. This is

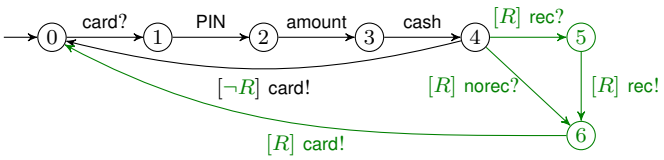

Fig. 7. ATM example: base feature plus *R*eceipt

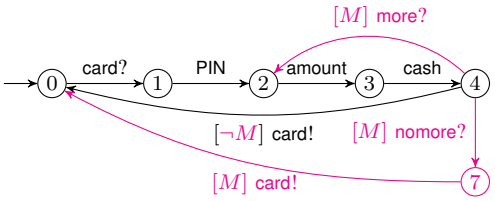

Fig. 8. ATM example: base feature plus *M*ore

achieved by the following *extended feature merge expression* (EFME); the result of the merge is displayed in Fig. 7.

$$
\begin{aligned}
R: \quad & \text{foreach } t \text{ labeled card!} \\
& \quad \text{str } t\ \neg R \\
& \quad \text{add } \mathit{src}(t)\ \mathit{guard}(t)\ \text{rec? } 5 \\
& \quad \text{add } \mathit{src}(t)\ \mathit{guard}(t)\ \text{norec? } 6 \\
& \quad \text{add } 5\ \mathit{guard}(t)\ \text{rec! } 6 \\
& \quad \text{add } 6\ \mathit{guard}(t)\ \text{card! } \mathit{tgt}(t)
\end{aligned}
$$

In the above EFME, the line "foreach t labeled card!" loops over all card!-labeled transitions, binding them to the variable t. The lines within the loop's scope can then access the source state $\mathit{src}(t)$ and the target state $\mathit{tgt}(t)$ of t and its feature guard $\mathit{guard}(t)$.

We would also like to add a More feature to the original ATM model, which allows the customer to make more than one transaction in a session. Hence M should add a transition labeled more? back from the start of any card!-labeled transition (this signifying the end of a session) to state 2 and modify other parts of the FTS accordingly. The EFME for M looks as follows, with the ATM+More displayed in Fig. 8.

$$
\begin{aligned}
M: \quad & \text{foreach } t \text{ labeled card!} \\
& \quad \text{str } t\ \neg M \\
& \quad \text{add } \mathit{src}(t)\ \mathit{guard}(t)\ \text{more? } 2 \\
& \quad \text{add } \mathit{src}(t)\ \mathit{guard}(t)\ \text{nomore? } 7 \\
& \quad \text{add } 7\ \mathit{guard}(t)\ \text{card! } \mathit{tgt}(t)
\end{aligned}
$$

Note how this permits the customer to indicate that she wants to make another transation (more?) versus that she is done (nomore?).

The semantics of EFMEs is given by translating them to simple FMEs. Specifically, let

$$\phi : \text{foreach } t \text{ labeled } a : IE$$

be an EFME, denote by $\{t \in T \mid \tau(t) = (s, a, s')\} = \{t_1, \ldots, t_n\}$ all the a-labeled transitions of the FTS under consideration and write $\tau(t_i) = (s_i, a, s_i')$ for all $i = 1, \ldots, n$. Then the FME translation is $\phi : IE_1'; \ldots; IE_n'$, where each IE_j' is obtained from IE by the following syntactic replacements:

$$\mathit{src}(t) \mapsto s_i \qquad \mathit{tgt}(t) \mapsto s_i' \qquad \mathit{guard}(t) \mapsto \gamma(t_i)$$

V. Feature Interactions

It is now a natural question to ask how different feature merges interact with each other. This should be phrased as a *commutativity* question: does it matter whether one feature is merged after another or vice versa, or do the merges commute? Hence we are interested in the question whether application of two FMEs to FTS is commutative, that is, given an FTS $\mathcal{F}$ and two FMEs E_1, E_2, whether $[\![E_1; E_2]\!](\mathcal{F}) \equiv [\![E_2; E_1]\!](\mathcal{F})$.

A. Commutativity of Feature Merges

We will consider the commutativity question on our running ATM example. First, note that the Balance feature (see Fig. 1) does *not* interact with the Receipt feature: if B is specified using the FME

$$B: \quad \text{add } 2 \text{ balance } 4$$

then merging B after R will give the same result as merging B before R. Similarly, B does not interact with M. Hence in these cases, the feature merges are commutative and the merges do not interact.

Much more interesting is the question whether the More and Receipt feature merges commute. If we first apply the FME for M and then the one for R to the ATM base, then the result is the FTS ATM+M+R displayed in Fig. 9. As the merge expression for M introduces a second card!-labeled transition into the FTS, the R expression is now applied to *two* card!-labeled transitions, one from state 4 to state 0 and one from state 7 to state 0. Hence this introduces two copies of the triangle of rec?-, norec?-, and rec!-labeled transitions, one anchored in state 4 and the other in state 7.

Also two new card!-labeled transitions are introduced, one with feature guard $\neg M \wedge R$ and the other with feature guard $M \wedge R$; but as they both connect state 6 to state 0, they can be merged into one card!-labeled transition with feature guard R. Note how this ATM model lets the customer do several transactions using the more?-labeled transition, and only after indicating that she is done (nomore?), the ATM asks whether she wants a receipt.

If we instead apply the FMEs for M and R in the opposite order, *i.e.*, first R and then M, then the result is the FTS ATM+R+M displayed in Fig. 10. Its behaviour is different from ATM+M+R: now the customer is asked whether she wants a receipt after every transaction in a session, instead of collecting one receipt at the end.

Hence the merges of R and M do not commute, but both orders of merging result in reasonable behaviour: ATM+M+R lets the customer do multiple transactions and asks whether it should print a receipt once the customer has done all her transactions; AMT+R+M instead asks whether a receipt is desired after every single transaction.

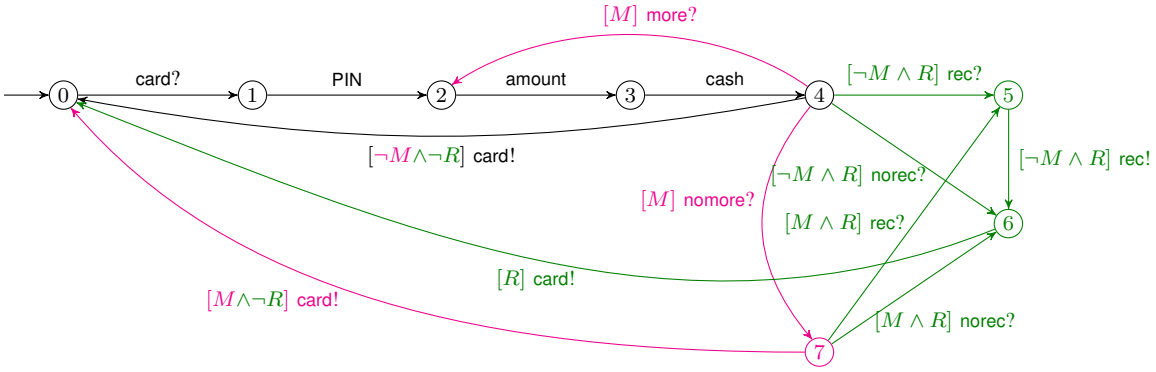

Fig. 9. ATM+M+R: ATM base feature plus More plus Receipt, in this order

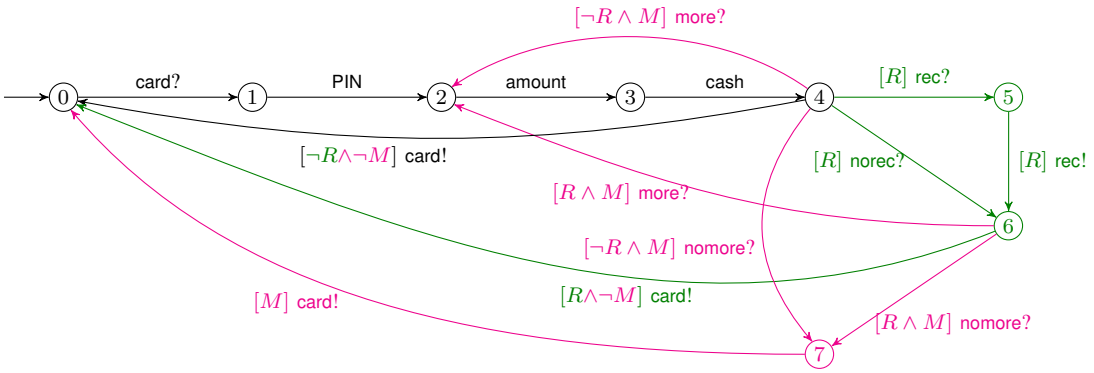

Fig. 10. ATM+R+M: ATM base feature plus Receipt plus More, in this order

B. Fixing Feature Interactions

From a designer's point of view, non-commutativity of feature merges is a sign that feature interactions are taking place and the design needs to be fixed. We propose such a fixed ATM design, called ATM+(MR), in Fig. 11. This FTS lets the customer do several transactions, each time giving her a choice whether to collect a receipt or not. Once she is done, she can collect one more receipt and then get her card back.

In order to fix the interaction of the M and R features, we present an *interaction FME* which turns both ATM+M+R and ATM+R+M into the fixed ATM+(MR). The following FME carefully adjusts and extends both ATM+M+R and ATM+R+M into ATM+(MR):

$R \wedge M$:	add 4 more? 2	add 4 rec? 5
	str 4 norec? 6 $\neg M$	add 4 nomore? 7
	add 5 rec! 6	add 6 more? 2
	add 6 nomore? 9	str 6 card! 0 $\neg M$
	add 7 rec? 8	add 7 norec? 9
	str 7 card! 0 $\neg R$	add 8 rec! 9
		add 9 card! 0

We show the complete feature merge expression for the ATM example in Fig. 12. Note that this also includes an FME for the base feature, showing that FMEs can be used to construct FTSs from the bottom up.

VI. Conclusion and Further Work

We have introduced a simple language of feature merge expressions (FMEs) and shown that these can be used for merging new features into featured transition systems (FTSs). We have shown that FMEs can be used to introduce any legal extension to an FTS. For practical usability, we have introduced extended FMEs (EFMEs), which allow for more generic merge expressions, and shown how to automatically translate EFMEs to simple FMEs.

We have shown a comprehensive example of the use of FMEs to construct and extend FTSs. For further evaluation, we plan to implement our feature merge mechanism within ProVeLines [7]. Note that in engineering practice, application of feature merges can be entirely automatic, except when feature interactions are detected: in that case, our proposed tool will notify the engineer who must then solve any problems manually, using interaction FMEs.

We have also seen that feature merges are not always commutative, and that such (typically undesirable) failures of commutativity can be resolved using an extra FME. Non-commutativity of feature merges also appears in other contexts, in particular in [3] which uses a much richer modelling language than ours but otherwise is concerned with similar modularity problems.

We have shown that non-commutativity of feature merges is related to feature interactions, which opens up connections to recent work on detecting and fixing feature interactions [1],

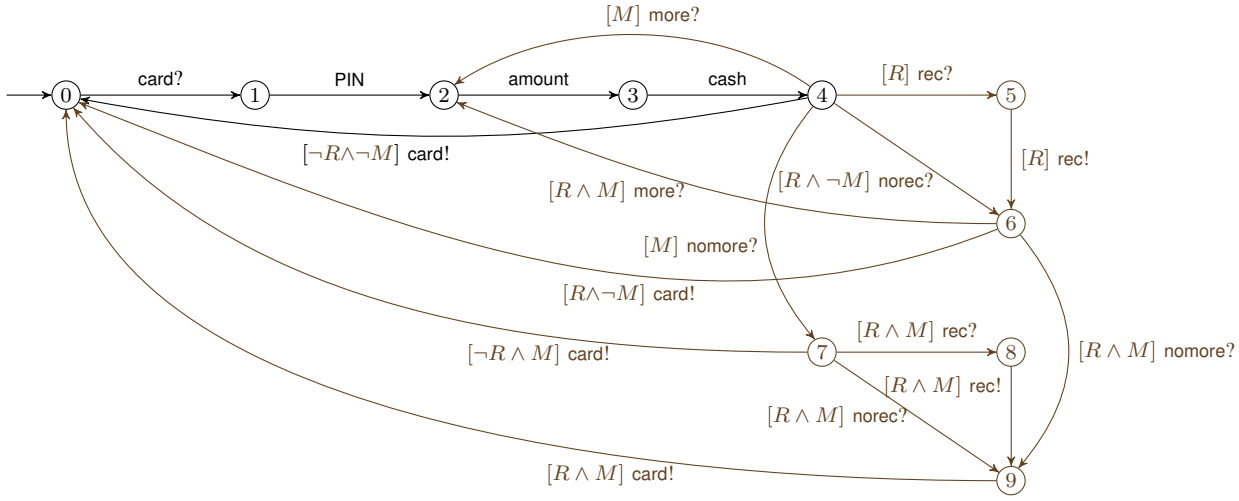

Fig. 11. ATM+(MR): ATM base feature plus Receipt plus More, with feature interactions fixed

tt :
add 0 card? 1
add 1 PIN 2
add 2 amount 3
add 3 cash 4
add 4 card! 0

Ca :
add 1 cancel 4
add 2 cancel 4
add 3 cancel 4

B :
add 2 balance 4

M :
foreach t labeled card!
str t $\neg M$
add *src*(t) *guard*(t) more? 2
add *src*(t) *guard*(t) nomore? 7
add 7 *guard*(t) card! *tgt*(t)

R :
foreach t labeled card!
str t $\neg R$
add *src*(t) *guard*(t) rec? 5
add *src*(t) *guard*(t) norec? 6
add 5 *guard*(t) rec! 6
add 6 *guard*(t) card! *tgt*(t)

$R \wedge M$:
add 4 more? 2
str 4 norec? 6 $\neg M$
add 5 rec! 6
add 6 nomore? 9
add 7 rec? 8
str 7 card! 0 $\neg R$
add 4 rec? 5
add 4 nomore? 7
add 6 more? 2
str 6 card! 0 $\neg M$
add 7 norec? 9
add 8 rec! 9
add 9 card! 0

Fig. 12. Complete feature merge expression for ATM example

[2], [4], [5], [8]–[11]. In future work, we intend to further explore this link between feature interactions and non-commutativity both from a theoretical and from a practical point of view. As a simple example, one could use the method we have introduced in [2], which can measure the number of feature interactions, to give a quantitative score to non-commutativity; this may be useful from a developer's point of view.

References

[1] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting dependences and interactions in feature-oriented design," in *ISSRE*, 2010, pp. 161–170.

[2] J. M. Atlee, U. Fahrenberg, and A. Legay, "Measuring behaviour interactions between product-line features," in *FormaliSE*. IEEE, 2015, pp. 20–25.

[3] S. Beidu, J. M. Atlee, and P. Shaker, "Incremental and commutative composition of state-machine models of features," in *MiSE*. IEEE, 2015, pp. 13–18.

[4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[5] M. Calder and A. Miller, "Feature interaction detection by pairwise analysis of LTL properties - A case study," *Formal Methods in System Design*, vol. 28, no. 3, pp. 213–261, 2006.

[6] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin, "Featured transition systems," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1069–1089, 2013.

[7] M. Cordy, A. Classen, P. Heymans, P. Schobbens, and A. Legay, "ProVeLines: a product line of verifiers for software product lines," in *SPLC workshops*. ACM, 2013, pp. 141–146.

[8] A. L. J. Dominguez, "Feature interaction detection in the automotive domain." in *ASE*, 2008, pp. 521–524.

[9] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model composition in product lines and feature interaction detection using critical pair analysis," in *MoDELS*, 2007, pp. 151–165.

[10] W. Scholz, T. Thüm, S. Apel, and C. Lengauer, "Automatic detection of feature interactions using the Java modeling language: an experience report." in *SPLC Workshops*, 2011, p. 7.

[11] P. Shaker and J. M. Atlee, "Behaviour interactions among product-line features," in *SPLC*, S. Gnesi, A. Fantechi, P. Heymans, J. Rubin, and K. Czarnecki, Eds., 2014, pp. 242–246.