Modes in component behavior specification via EBP and their application in product lines

Jan Kofroň [a,b] František Plášil [a,b] Ondřej Šerý [a]

[a] *Charles University in Prague*
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
Tel: +420 221 914 266, Fax: +420 221 914 323
{jan.kofron, frantisek.plasil, ondrej.sery}@dsrg.mff.cuni.cz

[b] *Academy of Sciences of the Czech Republic, Institute of Computer Science*
Pod Vodárenskou věží 2, 182 07 Praha 8, Czech Republic
Tel: +420 266 053 830, +420 286 585 789
{jan.kofron, frantisek.plasil}@cs.cas.cz

Abstract

The concept of software product lines (SPL) is a modern approach to software development simplifying construction of related variants of a product thus lowering development costs and shortening time-to-market. In SPL, software components play an important role. In this paper, we show how the original idea of component mode can be captured and further developed in behavior specification via the formalism of Extended Behavior Protocols (EBP). Moreover, we demonstrate how the modes in behavior specification can be used for modeling behavior of an entire product line. The main benefits include (i) the existence of a single behavior specification capturing the behavior of all product variants, and (ii) automatic verification of absence of communication errors among the cooperating components taking the variability into account. These benefits are demonstrated on a part of a non-trivial case study.

Key words: Behavior specification, component modes, software product lines
PACS:

1 Introduction

The concept of software components has been around for more than a decade. Component models range from relatively simple, flat component models (e.g., EJB [32]) to more sophisticated hierarchical component models such as Fractal [5] and Koala [30,31]. The latter one is based on Darwin [19] which coined

the concept of provided and required interfaces and primitive and composed components allowing component nesting (forming a hierarchy).

Typically, a part of a component-based application involves several operational variants, especially when the part was subject to reuse. This may be reflected both by architecture and behavior variants. However, there is no general consensus on handling this variability. The well-known approaches include modes and product lines.

A mode, as introduced in [13], defines (at design time) a particular alternative of a component's architecture. At runtime, transitions among the modes of the component may take place, however. In principle, a mode is part of a static view on component architecture; it determines the component's internal architecture, the mode of internal components, and is associated with a specific behavior of the component (not necessarily specified in detail). At the same time, the modes of internal components determine a specific mode of the parent.

Software components also play an important role in software product lines (SPL) [30,12]. Work in this field aims at supporting development of software for a set of closely related and likely further evolving products such as consumer electronics. Therefore, capturing variability at different levels of abstraction and stages of software development is a key goal here [7,29,26,9,6,28]. These stages range from modeling software requirements, over design and architecture specification, to code. Consequently, there are many modeling methods targeting variability within such different software artifacts; the key related abstractions include features [23], and, in particular, variation points and their resolution when a specific product is to be instantiated [28,27]. In addition to aspect-oriented programming [24], software components play an important role when considering variability at the level of software architecture. It should be emphasized that a SPL needs to address both variability (configurability) and evolution (modifiability). In a component-based architecture, the variability is reflected in general by some kind of variation points (resolved by configuration parameters) and evolution by the option to replace a component at various levels of component hierarchy.

1.1 Problem statement

Unfortunately, there has been no general consensus on handling architecture variants with respect to component behavior. Specifically, the mode concept is an approach to switching statically determined architecture variants at runtime; however, there is no abstraction to capture how switching among the variants is related to component behavior. If this were determined, automated

checks could be employed to verify whether the intended transitions among architecture variants are safely possible in a particular state of the involved components.

Similarly, in SPL, most of the focus is on variability in software architecture, but little attention is paid to variability of a component's behavior in support of its reuse in different architectural variants.

1.2 Goals and structure of the paper

In our group, we have developed a technique of specifying component behavior in a simple process algebra style. In its first variant, Behavior Protocols (BP) [1], the events specified are purely related to issuing request and responses of method calls on the component interfaces, while its more elaborated version, Extended Behavior Protocols (EBP) [16], allows us to capture a component state encoded as a n-tuple of values of enumeration types. The actual expressive power of EBP was tested on a non-trivial component based application developed in the CoCoME component models' contest [22]. There are model checking tools of BP and EBP which can verify correctness of communication among components. The goal of this paper is threefold—to show how

- (i) EBP can be used to specify the behavior associated with a specific mode, and also capture the transitions among modes,
- (ii) EBP can be employed in product line architecture specification and help derive the behavior and architecture of a particular product variant, and
- (iii) the EBP model checking tools can be used for verification of communication correctness of architecture variants.

2 Background

There are a number of formalisms designed for formal behavior specification of software, including set theoretic (e.g., Z), algebraic (e.g., VDM), and process algebraic (e.g., CCS, CSP, FSP [20]) formalisms. As to component-based software, its structuring of code into components with clearly defined interfaces and bindings encourages modular reasoning, on the one hand. On the other hand, the formalism for behavior specification of software components should reflect the key abstractions commonly used in component models. These abstractions involve methods grouped into interfaces, both provided and required, communication among assembled components via bindings of interfaces, and support of component hierarchies. In addition to BP and EBP,

such specialized formalisms include Interface automata [8], Component interaction automata [4], and SEFF [11].

2.1 EBP Example

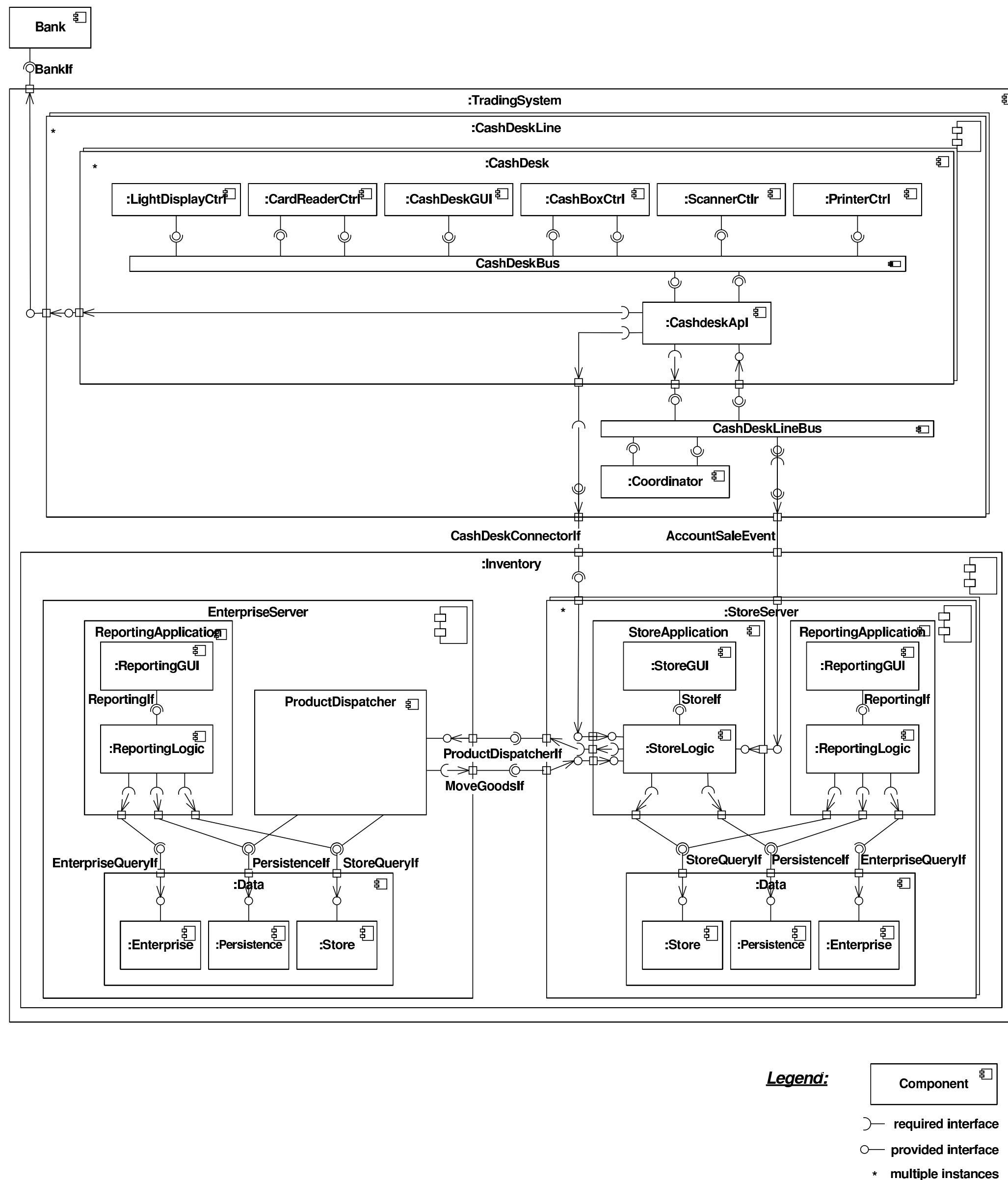

Fig. 1. Architecture of the CoCoME application

First, we shortly describe the CoCoME (Common Component Modeling Example) contest assignment, as the examples in this paper stem from our solution to this assignment [22]. CoCoME was motivated by the need for having a nontrivial canonical example of a component based application that would enable an assessment of strengths and weaknesses of different features and

their comparison. Previously, only simple examples (even toy ones) had been used as a proof-of-concept for this purpose.

The assignment of CoCoME is an application for managing a set of stores, each equipped with a cashdesk line. UML component diagram [33] providing an overview of the whole application is in Fig. 1. The assignment consists of a UML specification (component, deployment, sequence, and use-case diagrams), a prototype implementation in Java, and specification of extra-functio-nal properties. A number of teams applied their modeling techniques to the assignment; the results were subsequently evaluated by a jury, pointing out pros and cons of each modeling technique [22].

The example in Fig. 2 provides an intuitive insight and a brief overview of EBP. The example is a slightly simplified version of the EBP specification of the CoCoME CashDeskApplication component. The component contains the actual business logic of a single cash desk in a store; e.g., it maintains information about the progress of a current sale.

An EBP specification of a component consists of three sections: *types*, *vars*, and *behavior*. In the *types* section, the enumeration types of the components' state variables and method parameters are defined. In our case, there is a single enumeration type *states*, which captures the possible states of sale (line 3). The state variables are listed in the *vars* section. In the example, *state* captures the state of a current sale (line 6); it is initialized to SALE_STARTED.

The actual behavior is specified in the *behavior* section. Basic building blocks are: accepting a method call *?interface.method(parameters)* {*reaction*}, issuing a method call *!interface.method(parameters)*, assignment to a state variable *variable* $<-$ *value*, and the *switch* statement, which is used to direct the control flow depending on the value of a state variable. More complex expressions are constructed using the '+' alternative, ';' sequence, '$*$' repetition, and '|' parallel (interleaving, no synchronization) operators. They are described in more detail in [21] and [16].

The behavior section of the specification describes which method calls the component can accept and how it reacts on them. In Fig. 2 the reaction depends on the actual value of the *state* variable (note the *switch* statements). In its initial state SALE_STARTED, the CashDesk component can accept bar codes of sale items (ProductBarcodeScannedEvent—line 10) on which it reacts by querying StoreServer about the items and by updating the current total. This can be repeated (line 62). The state is switched to SALE_FINISHED, when the end of BarCode entering is signaled by the GUI component (the corresponding event SALE_FINISHED_EVENT mediated by CashDeskBus is accepted at line 20). After that, the purchase is paid either by cash or a credit card. The former case is reflected by accepting a call from GUI (mediated by CashDeskBus as

```
 1: component CashDeskApplication {
 2:   types {
 3:     states = { SALE_STARTED, SALE_FINISHED, CREDIT_CARD_SCANNED, PAID }
 4:   }
 5:   vars {
 6:     states state = SALE_STARTED
 7:   }
 8:   behavior {
 9:     (
10:       ?CashDeskAppHandleIf.onEvent(ProductBarcodeScannedEvent) {
11:         switch (state) {
12:           SALE_STARTED: {
13:             !CashDeskConnectorIf.getProductWithStockItem;
14:             (
15:               !CashDeskAppDispatchIf.send(ProductBarcodeNotValidEvent) +
16:               !CashDeskAppDispatchIf.send(RunningTotalChangedEvent)
17:             )
18:       } } }
19:       +
20:       ?CashDeskAppHandleIf.onEvent(SaleFinishedEvent) {
21:         switch (state) {
22:           SALE_STARTED: { state <- SALE_FINISHED }
23:       } }
24:       +
25:       ?CashDeskAppHandleIf.onEvent(CashAmountCompletedEvent) {
26:         switch (state) {
27:           SALE_FINISHED: {
28:             !CashDeskAppDispatchIf.send(ChangeAmountCalculatedEvent);
29:             state <- PAID
30:       } } }
31:       +
32:       ?CashDeskAppHandleIf.onEvent(CashBoxClosedEvent) {
33:         switch (state) {
34:           PAID: {
35:             !CashDeskAppDispatchIf.send(SaleSuccessEvent);
36:             !CashDeskDispatchIf.send(AccountSaleEvent);
37:             state <- SALE_STARTED
38:       } } }
39:       +
40:       ?CashDeskAppHandleIf.onEvent(CreditCardScannedEvent) {
41:         switch (state) {
42:           SALE_FINISHED: { state <- CREDIT_CARD_SCANNED }
43:       } }
44:       +
45:       ?CashDeskAppHandleIf.onEvent(PINEnteredEvent) {
46:         switch (state) {
47:           CREDIT_CARD_SCANNED: {
48:             !BankIf.validateCard;
49:             (
50:               !CashDeskAppDispatchIf.send(InvalidCreditCardEvent)
51:               +
52:               !BankIf.debitCard;
53:               (
54:                 !CashDeskAppDispatchIf.send(InvalidCreditCardEvent);
55:                 (NULL + state <- SALE_FINISHED)
56:                 +
57:                 !CashDeskAppDispatchIf.send(SaleSuccessEvent);
58:                 !CashDeskDispatchIf.send(AccountSaleEvent);
59:                 state <- SALE_STARTED
60:             ) )
61:       } } }
62:     )*
63:   }
64: }
```

Fig. 2. EBP specification of the CashDeskApplication component

CashAmountCompletedEvent—line 25) reporting the cash amount paid by a customer. As a result, GUI is called (again mediated by CashDeskBus—line 28) to report the change to be returned to the customer and the state is switched to SALE_PAID. As soon as the change is returned and the cashbox is closed (CashBoxClosedEvent—line 32), the StoreServer records are updated and the state is set back to SALE_STARTED (lines 35-37). Payment by credit card is initiated by swiping a credit card (CreditCardScannedEvent—line 40), switching the state to CREDIT_CARD_SCANNED, and then either finalized by entering a valid PIN (PINEnteredEvent—line 45), or canceled by switching back to the state SALE_FINISHED.

2.2 Detecting component communication errors via EBP

After an EBP specification of each component has been finished, the tools designed for EBP are applied. They serve to verify correctness of communication among components via checking absence of the following communication errors: *bad activity*, *no activity*, and *unbound requires error*. In general (with a slight simplification), whenever a component calls ('!') a method and the target component is not ready to accept ('?') the call, the bad activity error occurs. No activity represents the situation, when there are components waiting for a method call, while no component is able to emit any. The last error, unbound requires, may be viewed as a special case of the bad activity error, as it represents the situation, when a component issues a call on an unbound required interface. We have developed a tool (Sect. 2.3) which automatically identifies the communication errors and produces a corresponding error trace.

For a composed component, a typical question is whether a *component frame* (the set of externally visible interfaces) is correctly implemented by *component architecture* (composition of subcomponents). The tool can answer the question on the behavioral level, i.e., it reports whether an EBP specification of a composed component is correctly implemented by the behavior of its subcomponents (as described in [1,16,18], compliance of the frame and architecture EBP protocol is verified by a tool).

Considering the CoCoME architecture, the tool can be, e.g., used to show that the EBP specification of CashDesk is correctly implemented by composing CashDeskApplication, CashDeskGUI, LightDisplayCtrl, CardReaderCtrl, CashBoxCtrl, ScannerCtrl, PrinterCtrl, and CashDeskBus (the architecture of CashDesk). Correctness of an entire application can by verified by applying this idea at each level of component nesting.

In principle, a protocol in EBP is a textual definition of a finite automaton specifying behavior of a component. The essence of the behavior part (as seen in Fig. 2) are expressions forming a simple process algebra close to CSP and partially to CCS. Such an expression generates a set of traces—a trace in EBP is a finite sequence of labels representing the atomic events related to method invocations (the label of a form ?a↑ stands for accepting an invocation of a method with (composed) name a, !a↑ for issuing an invocation, ?a↓ means accepting the response (end) of a method execution, !a↓ means issuing the response). Syntactically, an expression in EBP is composed of labels, operators, and parenthesis ‘()’ and ‘{}’. The basic operators are: ‘;’ sequencing, ‘+’ alternative, and ‘|’ parallel interleaving with similar semantics as in CSP. However, recursive definitions are not allowed; instead, the repetition operator ‘*’ similar to regular expressions is employed. Therefore only finite traces are considered. Moreover, the parenthesis ‘{}’ serve to easily encode method calls and functionality of methods in the following way: ‘?a’ stands for ‘?a↑; !a↓’, while ‘?a{P}’ stands for ‘?a↑; P; !a↓’, and similarly for ‘!a’ and ‘!a{P}’, where P is again an expression in EBP. For illustration, consider the example in Fig. 3.

```
45: ?CashDeskAppHandleIf.onEvent {
46:
47:
48:   !BankIf.validateCard;

49:   (
50:     !CashDeskAppDispatchIf.send

51:     +
52:     !BankIf.debitCard;

53:     (
54:       !CashDeskAppDispatchIf.send

55:
56:       +
57:       !CashDeskAppDispatchIf.send;

58:       !CashDeskDispatchIf.send

59:
60:   ) )
61: }
```

```
45: ?CashDeskAppHandleIf.onEvent↑;
46:
47:
48:   !BankIf.validateCard↑;
49:   ?BankIf.validateCard↓;
50:   (
51:     !CashDeskAppDispatchIf.send↑;
52:     ?CashDeskAppDispatchIf.send↓
53:     +
54:     !BankIf.debitCard↑;
55:     ?BankIf.debitCard↓;
56:     (
57:       !CashDeskAppDispatchIf.send↑;
58:       ?CashDeskAppDispatchIf.send↓
59:
60:       +
61:       !CashDeskAppDispatchIf.send↑;
62:       ?CashDeskAppDispatchIf.send↓;
63:       !CashDeskDispatchIf.send↑;
64:       ?CashDeskDispatchIf.send↓
65:
66:   ) );
67: !CashDeskAppHandleIf.onEvent↓
```

Fig. 3. EBP specification of the CashDeskApplication component

In the left column, there is a stripped-off version of the EBP specification in Fig. 2, which was obtained by ignoring the switch constructs and method parameters for simplicity. An equivalent expression where only atomic events are used is in the right column. In principle, this is also a valid CSP specification,

in which the events feature composed names (such as !BankIf.validateCard↑)—notice that here a dot means name composition and not CSP prefixing. This simple example illustrates only the use of the '+' and ';' . The operator '|' will be applied later (Fig. 5-10). Obviously, since an expression in CSP determines an LTS [25], an expression in EBP determines also an LTS (more specifically a finite automaton, since recursion in the EBP specification is not employed). Notice that the constructs not employed in Fig. 3 (switches and method parameters) are based on enumeration types, only constants can be assigned to variables of these types, the variables are used only to control switch alternatives, etc., so that the rules for converting them into a finite automaton can be easily articulated.

Composed behavior of two components is determined by parallel composition of their EBP behavior specifications. For this purpose, the binary operator consent (∇_M) is introduced in EBP. In principle, its semantics is similar to the "interface parallel" composition with restriction ($P \mid\mid_M Q$) as known from CSP [25], i.e. in the interleaving of events from P and Q those with names in M synchronize and restriction makes them τ events in the corresponding LTS. However, there are two key semantic differences in case of $P \nabla_M Q$:

(i) The synchronization is based on pair-wise complementarity of the event names (similar to CCS)—the names which differ only in their prefixes '!' and '?' are complementary. For example, events !BankIf.validateCard↑ and ?BankIf.validateCard↑ would synchronize and produce τ.

(ii) Contrary to CSP, if there is no counterpart for an event in M in the other operand of ∇_M, such situation triggers a communication error (an erroneous trace is produced by definition). As mentioned in Sect. 2.2, the following communication errors are defined: *bad activity*, *no activity*, and *unbound required error*.

In our group, for BP we have developed several variants of a model checking tool which identifies communication errors in a ∇_M composition and produces a corresponding error trace [18]. Moreover, for EBP verification, we use a tool transforming EBP specifications into Promela—the input language of the Spin model checker [14]. The reason for choosing Spin is an easier support for future extensions of the EBP language and the efficiency and maturity of Spin—it is a state-of-the-art explicit model checker featuring LTL checking abilities, bit-state hashing, and quite friendly user interface. In addition, there are a number of extensions of Spin, e.g., dSpin extending the Promela language by functions, exceptions, etc. The reason for choosing Spin is an easier support for future extensions of the EBP language.

3 Expressing modes and product lines in EBP

3.1 Behavior modes

To illustrate the way modes are reflected in EBP behavior specification, we present a fragment of the CoCoME component architecture (Fig. 1)—the CashDeskApplication component (Fig. 2). CashDeskApplication works in two operational variants, EXPRESS mode and NORMAL mode. Each of these variants is characterized by a modification of both behavior and architecture of the involved components. In Fig. 4, the architectural variants corresponding to these modes are depicted (we omit other components connected to the CashDeskBus for simplicity).

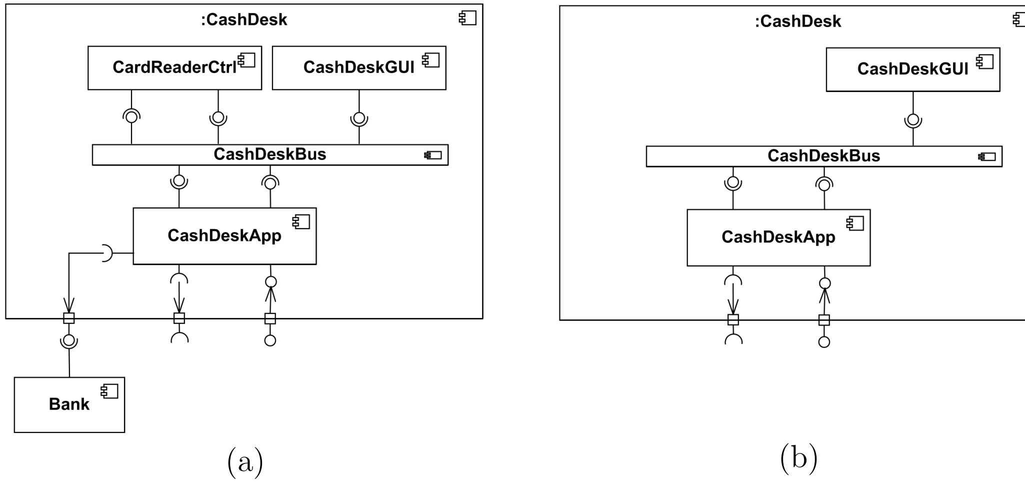

Fig. 4. CashDeskApplication in NORMAL mode (a) and in EXPRESS mode (b)

The EBP snippet in Fig. 5 shows how switching between the modes is reflected in behavior specification of CashDeskApplication. It contains the changes to the CashDeskApplication frame protocol necessary to describe switching to and from the EXPRESS mode. Basically, the original protocol from Section 2.1 is extended by an additional state variable CDAmode (containing the value NORMAL or EXPRESS). The component starts computation in the NORMAL mode (line 8); after receiving an onEvent method call with the parameter value ExpressModeEnabledEvent (line 27), it switches to the EXPRESS mode (line 28).

This example shows that a mode is captured as a specific value of a single state variable (CDAmode in this case). However, in general, a behavior mode of a component can be encoded as an n-tuple of local state variables, each of a specific enumeration type.

The EBP snippet in Fig. 6 describes behavior of the CashDesk component—the parent component of CashDeskApplication. Generally, the behavior modes

```
 1: component CashDeskApplication {
 2:  types {
 3:   states = { SALE_STARTED, SALE_FINISHED, CREDIT_CARD_SCANNED, PAID },
 4:    modes = { NORMAL, EXPRESS}
 5:  }
 6:  vars {
 7:    states state = SALE_STARTED,
 8:    modes CDAmode = NORMAL
 9:  }
10:  behavior {
11:    (
12:
13:      . . .
14:
15:      +
16:      ?CashDeskAppHandleIf.onEvent(CreditCardScannedEvent) {
17:        switch (CDAmode) {
18:          NORMAL: {
19:            switch (state) {
20:              SALE_FINISHED: { state <- CREDIT_CARD_SCANNED }
21:      } } } }
22:      +
23:
24:      . . .
25:
26:    )* | (
27:      ?CashDeskAppHandleIf.onEvent(ExpressModeEnabledEvent) {
28:        CDAmode <- EXPRESS
29:      }
30:    )* |  (
31:      ?CashDeskAppHandleIf.onEvent(ExpressModeDisabledEvent) {
32:        CDAmode <- NORMAL
33:      }
34:    )*
35:  }
36: }
```

Fig. 5. EBP specification of the CashDeskApplication component with behavior modes

of a parent and its child components are independent—in each component, there may be state variables capturing the actual behavior mode. In the case of the CashDesk and CashDeskApplication components, however, the modes of both parent and child components are switched simultaneously as a reaction to the ExpressModeEnabled/ExpressModeDisabled events mediated by CashDeskBus.

To provide an example of really independent behavior modes in nested components, let us consider a modified CashDesk component featuring again two behavior modes (CASH and CARD mode) to capture how a customer has decided to pay. Since customers are allowed to pay by cash in both the EXPRESS and NORMAL modes of CashDeskApplication, the behavior mode of CashDesk is not determined by the behavior mode of CashDeskApplication (and vice versa) in this case. Of course, since the CARD mode makes sense only in the NORMAL mode, the modes of CashDeskApplication and CashDesk are not logically independent—the former influences the latter.

```
 1: component CashDesk {
 2:  types {
 3:    modes = { NORMAL, EXPRESS}
 4:  }
 5:  vars {
 6:    modes CashDeskMode = NORMAL
 7:  }
 8:
 9:  behavior {
10:    # Sale related stuff
11:    (
12:      !CashDeskConnectorIf.getProductWithStockItem*;
13:        ( # the bank interface is used only in the EXPRESS mode
14:          switch (CashDeskMode) {
15:            NORMAL: {
16:                      !BankIf.validateCard*;
17:                      !BankIf.validateCard;
18:                      !BankIf.debitCard;
19:            }
20:
21:            EXPRESS: { NULL }
22:          }
23:        )*;
24:      !CashDeskEventDispatcherIf.send(AccountSaleEvent);
25:      !CashDeskEventDispatcherIf.send(SaleRegisteredEvent)
26:    )*
27:    |
28:    # Express mode switch
29:    ?CashDeskAppEventHandlerIf.onEvent(ExpressModeEnabledEvent) {
30:      CashDeskMode <- EXPRESS
31:    }*
32:    |
33:    # Normal mode switch
34:    ?CashDeskAppEventHandlerIf.onEvent(ExpressModeDisabledEvent) {
35:      CashDeskMode <- NORMAL
36:    }*
37:  }
38: }
```

Fig. 6. EBP specification of the CashDesk component with modes

In composition via ∇_M (and the corresponding verification), the behavior modes (state variables) themselves are taken into account indirectly, via the sequences of method calls that are determined by the switch variants which trigger mode switching. In our example, the NORMAL mode is reflected by calling the methods on the BankIf interface (lines 16-18), which is not allowed in the EXPRESS mode (line 21).

3.2 Software product lines

In this section, we show how EBP and behavior modes can be beneficially employed in product line software engineering. To get an idea how EBP can help in SPL architectural design and verification, consider the following example (a slightly generalized part of the CoCoME example). In each store, there is a server maintaining the list of items on and out of stock (Fig. 7). Since the enterprise operates multiple stores, when some goods are running out of

stock in a store, they may be brought there from another store (if available) and/or ordered from a supplier. The decision on moving/ordering the goods is realized by a dedicated enterprise server. Depending on whether there is an outlet store at the enterprise premises, the enterprise server either provides also the functionality of the store server (Fig. 9), or it does not (Fig. 8).

These functional alternatives are reflected as architecture variants of a generic server (Fig. 9). The ability to serve as a store server is embodied in the presence of the StoreApplication component, while the enterprise functionality is implemented by the ProductDispatcher component. The other parts, i.e., the ReportingApplication and Data components, are present in all variants.

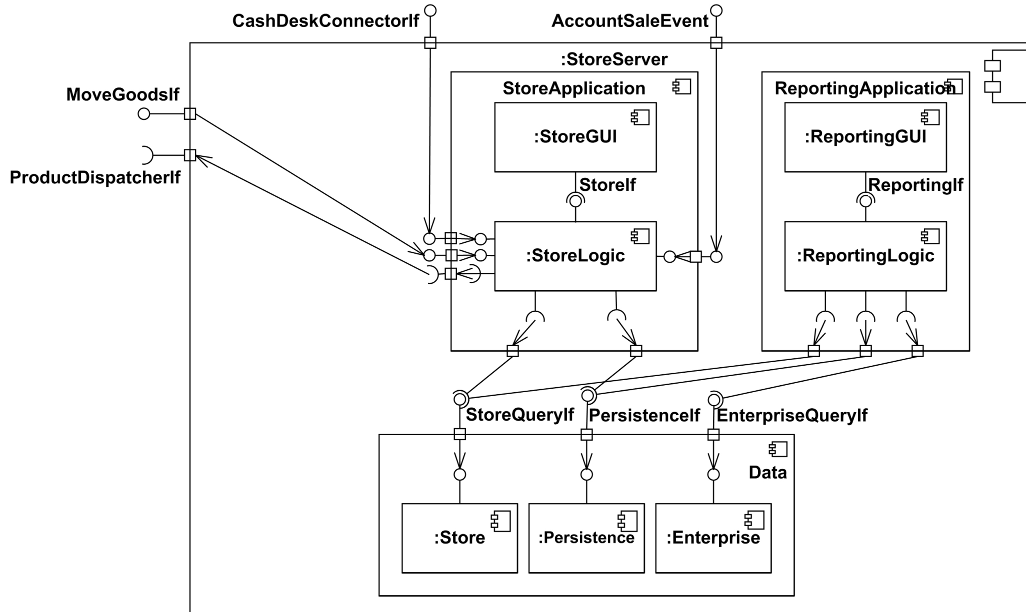

Fig. 7. Store server—present at common stores

In Fig. 10, an EBP specification of the generic server component is provided. Such a behavior specification takes advantage of the fact that some parts of the functionality (behavior) are shared by several variants. The key idea is that the variation points are represented by the switch construct employing the state variables store and enterprise which serve for resolution of these variation points. Both of them can be set to YES or NO; since they are supposed to be set at the design phase by the system architect according to the desired role of the component instance within the system, they are read-only. Particular combinations of the values of store and enterprise reflect the architecture variants described above (except for the combination NO-NO, which results in empty behavior). Such initial assignments to the variables determine the resolution statically.

Behavior modes are suitable for SPL design of unrelated variation points (where state variables are independent). The only difference between a variable determining a product line variant and a variable representing a runtime

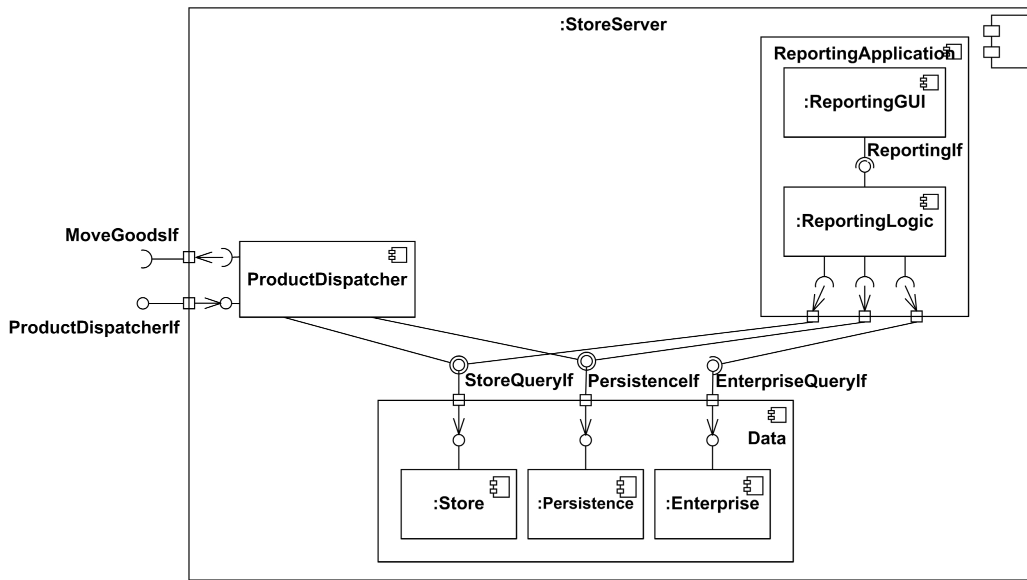

Fig. 8. Enterprise server—present at the enterprise with no store

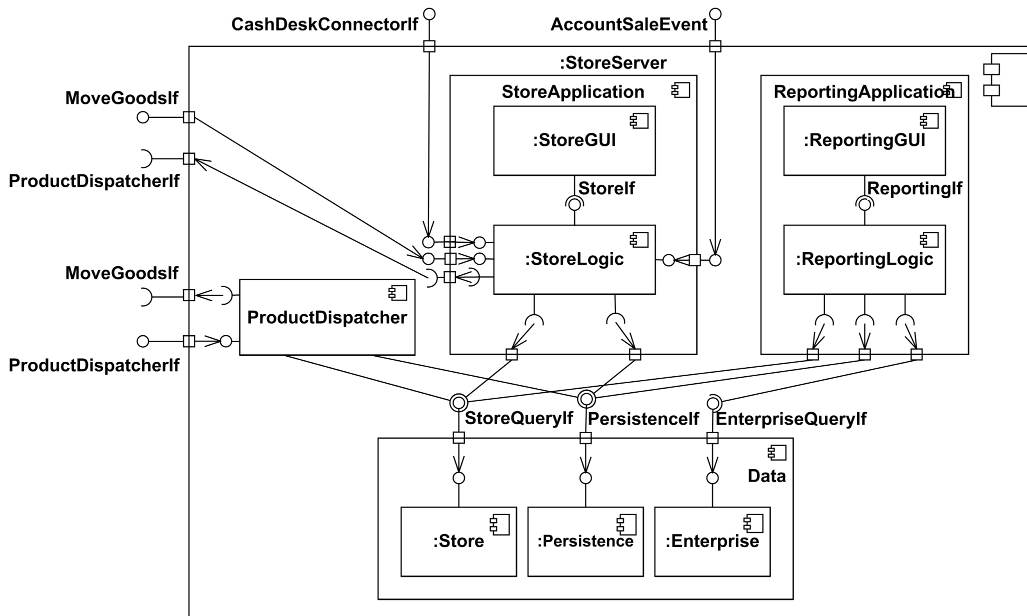

Fig. 9. Generic server

mode is that the value of the former is not modified in the behavior specification. There are no means to enforce this within the EBP language itself; it is up to the designer to obey this principle.

In general, there may be several state variables, each capturing variants of a specific aspect of component behavior and a product line variant. Before the deployment phase, the values of (read-only) variables representing a product line variant are determined to reflect the actual role of each component in the system (resolution of variation points)—it is the case of the store and

enterprise variables in Fig. 10. Next, at runtime, the other variables are used to express different behavior modes (and transitions among them) of particular components, as illustrated by the state and CDAmode variables in Fig. 5.

```
component Server {
  types {
    STORE { YES, NO }
    ENTERPRISE { YES, NO }
  }

  vars {
          STORE store = YES
          ENTERPRISE enterprise = NO

  }

  behavior {
    switch (enterprise) {
      YES: {
          (
              ?ProductDispatcherIf.orderProductsAvailableAtOtherStores{
                  !MoveGoodsIf.queryGoodAmount;
                  ((!MoveGoodsIf.acceptFromOtherStore;
                    !MoveGoodsIf.sendToOtherStore)
                    + NULL
                  )
              }
          )*
      }

      NO: { NULL }
    }
    |
    switch (store) {
      YES: {
        (
          ?CashDeskConnectorIf.getProductWithStockItem*;
          (?CashDeskLineEventHandlerIf.onEvent(AccountSaleEvent){
                !ProductDispatcherIf.orderProductsAvailableAtOtherStores +
                NULL
           }
          +
            NULL
          )
        )*
        |
        (
            ?MoveGoodsIf.queryGoodAmount
            +
            ?MoveGoodsIf.sendToOtherStore
        )*
        |
        ?MoveGoodsIf.acceptFromOtherStore*
      }

      NO : { NULL }
    }
  }
}
```

Fig. 10. EBP specification of the Server component with variation points

4 Evaluation and related work

There are several differences between the behavior modes proposed by this paper and the modes introduced in [13]. First, a mode in [13] is a part of the static view on a component's architecture. Moreover, in [13] a mode of a composite component determines both the architecture implementing its frame (i.e., the subcomponents and their bindings) and the modes of subcomponents. Since there is a one-to-one mapping of the mode of a frame and the modes of the subcomponents and their bindings, the architecture and the modes of subcomponents also determine the mode of the encapsulating frame. Regarding the behavior corresponding to different modes, the authors of [13] just assume that every mode of a component is associated with a specific behavior of the component (however, no further details on behavior, or on how modes are switched among, are provided). In other words, modes are a part of the architecture description (technically, they are expressed as (i) labels associated with components and (ii) specification of component instances and their bindings [13]). This way, behavior is determined only at an abstract level. Moreover, employment of the mode idea is distributed over several non integrated tools: Architecture labeling (Darwin), Alloy specification of mode switching constraints [15], and Ponder for specifying runtime policies (such as management and security).

In our approach, on the other hand, mode is a part of behavior specification and it relates to the static view (architecture) only indirectly. Moreover, there is no direct dependency between the behavior mode of a frame and the behavior modes of the subcomponents in its relevant architecture. The only requirement is that the architecture protocol has to be compliant to the frame protocol. It should be emphasized that the compliance evaluation takes into account involvement of mode switching both in the architecture and the frame protocol, i.e., it is verified that these mode switches correspond to each other well. In other words, compliance evaluation helps verify that parent mode switching is correctly reflected in children's behavior. As an aside, an additional benefit of EBP is that the specification scales well due to its textual form (no diagrams) and separation of abstraction layers (frame vs. architecture protocol) which helps keep the state space subject to model-checking in manageable limits.

As to SPL, to our knowledge, there has been no attempt to directly support variation points in behavior specification of a product line. The EBP language allows specifying the behavior of all potential resolutions of variation points in a component frame within a single specification. This is beneficial since significant parts of the specification are typically shared by several variants. An additional benefit of our approach is that from the generic behavior specification of a product line (like the specification of the generic server in Fig. 9)

both the architecture and its behavior specification of a variant can be derived; e.g., the architectures in Fig. 7 and Fig. 8 can be inferred in an automatized way.

In addition to [13] discussed above, probably the most related work is [3] emphasizing the need for dynamic software product lines in mobile applications, particularly in the context of self adaptation. We can imagine modifying the variant selection algorithm proposed in [3] in such a way that it would convert its output to trigger an event which could determine a corresponding "dynamic" behavior mode. In our approach, due to the ability to combine both static and non-static variables in the n-tuple determining a behavior mode, we can express behavior of both static (classical) and dynamic product lines.

In [10], dynamic architecture reconfiguration in the context of SPL is modeled as applying the reconfiguration patterns defined at design phase. However, no verification of correctness properties is considered. In [2], behavior of product line variants is modeled in a CSP-based algebra. Nevertheless, all the architecture variants are considered statically and their correctness in terms of deadlock related architectural mismatches is verified separately for each variant. To our knowledge, this is the only process algebra used in SPL.

When compared to general CCS and CSP-like process algebras, EBP is specialized for specification of components. It offers key component abstractions such as methods grouped into interfaces, both provided and required, communication among assembled components via bindings of interfaces, and support of component hierarchies. Another important aspect is representation of a method call by four separate atomic events $!iface.method\uparrow$, $?iface.method\downarrow$, $?iface.method\uparrow$, and $!iface.method\downarrow$, which allows for precise specification of interleaving and nesting of method calls.

As an aside, there is no means for expressing behavior modes in the other component behavior modeling formalisms known from literature, such as Interface automata [8], I/O automata [17], and Component interaction automata [4].

5 Conclusion

We have presented a contribution to handling architecture variants with respect to component behavior. The key idea is to use the behavior specification in EBP as a basis for specification of behavior variants. We have shown how EBP can be used to specify the behavior associated with a specific mode of a component and to capture the transitions among the modes at runtime. Moreover, the employment of EBP in the architecture of a product line behavior specification, and the derivation of the architecture and its behavior

for a particular product variant were presented. Finally, we have described the use of model checking tools designed for EBP in verifications of component communication correctness in architectural variants.

6 Acknowledgements

This work was partially supported by the Czech Republic project 201/08/0266 and the Czech Academy of Sciences project 1ET400300504.

References

[1] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 2004.

[2] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.

[3] G. Brataas, S. Hallsteinsen, R. Rouvoy, and F. Eliassen. Scalability of decision models for dynamic product lines. In *Proceedings In International SPLC Workshop on Dynamic Software Product Line (DSPL'07)*, page 10, 2007.

[4] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Softw. Eng. Notes*, 31(2):4, 2006.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In *CBSE*, pages 7–22, 2004.

[6] S. Buhne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 41–52, Washington, DC, USA, 2005. IEEE Computer Society.

[7] P. Clements and L. Northrop. *Software product lines: practices and patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the.* ACM Press, January 2001.

[9] A. Garg, M. Critchlow, P. Chen, C. V. der Westhuizen, and A. van der Hoek. An environment for managing evolving product line architectures. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 358, Washington, DC, USA, 2003. IEEE Computer Society.

[10] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[11] J. Happe, H. Koziolek, and R. H. Reussner. Parametric Performance Contracts for Software Components with Concurrent Behaviour. In F. S. de Boer and V. Mencl, editors, *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS06), Prague, Czech Republic*, Electronical Notes in Computer Science, pages 41–55, September 2006.

[12] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[13] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.

[14] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[15] J. S. Kim and D. Garlan. Analyzing architectural styles with alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM.

[16] J. Kofron. *Behavior Protocols Extensions.* PhD thesis, Charles University in Prague, 2007.

[17] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, November 1988.

[18] M. Mach, F. Plášil, and J. Kofroň. Behavior protocol verification: Fighting state explosion. *International Journal of Computer and Information Science*, 2005(1):22–30, 2005.

[19] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[20] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* Wiley, 1999.

[21] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

[22] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS.* Springer, Heidelberg, 2008.

[23] M. Riebisch, D. Streitferdt, and I. Pashov. Modeling variability for object-oriented product lines. In *LNCS 3013*, pages 165–178. Springer, 2004.

[24] D. B. Roberto E. Lopez-Herrejon. Modeling features in aspect-based product lines with use case slices:an exploratory case study., October 2006.

[25] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[26] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, pages 197–213. Springer-Verlag, 2004.

[27] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Managing variability in software product families. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVMG 2004)*, 2004.

[28] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[29] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 130–153, London, UK, 2002. Springer-Verlag.

[30] R. van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM.

[31] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[32] Sun Enterprise Java Beans, `http://java.sun.com/products/ejb`.

[33] Unified Modeling Language, `http://www.uml.org/`.