Journal of Theoretical and Applied Information Technology
15th February 2018. Vol.96. No 3
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A METHOD FOR MATCHING MODELS IN UML-B

[1] MUHAMMED BASHEER JASSER, [2] MAR YAH SAID,
[3] ABDUL AZIM ABDUL GHANI, [4]PATHIAH ABDUL SAMAT
[1]Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 UPM Serdang, Selangor, Malaysia
E-mail: [1]mbjasser@m.ieice.org/mbjasser@gmail.com, [2]maryah@upm.edu.my

ABSTRACT

UML-B is a graphical front-end formal modelling language of the formal method Event-B. UML-B models semantics are given by the corresponding generated Event-B. Identifying similarities between models has several benefits such as model comparison, integration and evolution. Several matching and comparison methods have been done in the context of model driven software engineering. However, matching models via a systematic method is not supported yet in UML-B. In this work, we propose a matching method for UML-B elements based on their semantics. This method includes variable-based matching, event-based matching and state-machine matching. The variable-based matching provides rules for matching UML-B classes, attributes, states and variables. The event-based matching provides rules and cases for matching UML-B transitions and class-events. The state-machine matching provides rules for matching UML-B state-machines based on the state and transition matching rules. The matching rules are formalized by means of the generated corresponding Event-B specifications. The correctness of the rules is justified via preserving the compatibility of the matched state-variables and corresponding modifying events including their matched guards and actions. These rules are illustrated via a communication-based case study to show their applicability.

Keywords: *Visual modeling languages, Formal specification, Event-B, UML-B, Model Matching*

1. INTRODUCTION

Model-Driven Software Engineering MDSE [1] is part of the software engineering discipline where models are considered as the primary elements representing the abstract view of the systems to be handled with. Model matching is an important process for model management, evolution and integration. A correct and accurate model matching leads to a better model integration.

Several matching approaches and methods have been proposed in the context of MDSE. They stand on specifying the model differences through three phases: calculation, representation and visualization [2]. The calculation stands for comparing models, the representation is to provide the outcome of calculation for further manipulation, and the visualization is to represent the model differences in a visualized way.

Formal modelling is part of the software engineering [3], [4] which provides an accurate way of modelling and verifying systems. This is by the precise specification and the mathematical basis which the formal languages are based on. Event-B [3] is a formal method which is based on the set-theory, first order logic and action systems [5]. It allows modelling correct-by-construction systems that are verified by the theorem provers offered by Rodin platform [6].

UML [7] is a semi-formal language for modeling object-oriented systems. UML-B [8], [9], [10], [11] is a graphical front-end of the formal method Event-B. It combines the semi-formal properties of UML and the formal ones of Event-B. UML-B models semantics are given by their corresponding Event-B models generated from the translation process and used for verification purposes.

Matching UML-B models is interesting as it has several benefits such as: first, matching helps observing the compatibility and difference between the models in UML-B, especially what is related to mathematical types, second, matching provides a good potential for reusing the compatibly matched UML-B elements that could be extracted and reused avoiding the remodelling and reproving effort of these elements. Currently, matching models via a systematic method is not supported yet in UML-B. The research question that we tackle in this work is:

How models in UML-B could be matched compatibly and consistently?. To answer this question, in this paper, we propose a method for matching UML-B models based on their semantics that are given by their corresponding Event-B models represented by the generated Event-B state-variables and events. The method introduces rules for matching the different structures offered by UML-B. This includes UML-B class, UML-B attribute, UML-B state, and UML-B variable that become Event-B variable, and UML-B transition and UML-B class-event that become Event-B event. The proposed rules guide the user to a correct model matching which should lead to correct model integration and management. The proposed method is formalized and the justification for correctness is provided through preserving the compatibility of the matched UML-B elements. The method applicability is illustrated through a case study. The contributions of this paper are as follows:

- Variable-based matching rules.
- Event-based matching rules.
- State-machine matching rules.
- A formalization of the method rules.
- A communication-based case study to illustrate the method applicability.

This paper is structured as follows. Section 2 reviews a background related with Event-B, matching in Event-B, and UML-B. Section 3 presents our proposed method. Section 3.1 presents the variable-based matching rules for UML-B class, attribute, state and variable. Section 3.2 presents the matching rules for UML-B transition and class-event. Section 3.3 introduces the UML-B state-machine matching rules. Section 4 presents a formalization and a justification for the method correctness. Section 5 overviews a communication-based case-study to show the method applicability. Section 6 exhibits some of the related work regarding model matching in the context of Model-Driven Software Engineering. Section 7 exhibits some benefits and limitations of the method and some proposed future works. Section 8 concludes the work.

2. BACKGROUND

This section discusses Event-B, matching in Event-B, UML-B diagrams and its semantics.

2.1 Formal Modeling and Matching in Event-B

Event-B [3] is a variant of the B-method [4], [12] and is based on Action Systems [5]. An action system is a collection of actions on some set of state variables. Every action has enabling condition guard which depends on any system variable. The mathematical notation used in both Event-B and B-method is based on set-theory [13]. Event-B models are described by contexts and machines where they represent the static and dynamic parts respectively. The context contains the types, axioms and constants, while the machine contains the state variables, invariants and events that can be called to change the machine state. Invariants represent the system conditions on variables and should be preserved for each event that has an effect on the invariant-related variables. Event-B is supported by Rodin tool [6] for modeling and proving.

In [14], pattern matching guidelines are introduced in Event-B. The user has to define which problem variables are to be matched with all the pattern variables. The compatibility of the matching should be guaranteed following specific checks. First, there is no non-matched events that alter matched variables. Second, in each pattern event, all the guards and actions, that depend naturally on only the matched pattern variables, are syntactically matched (i.e. $a:=a+1$ is syntactically matched with $b:=b+1$).

Our proposed method for matching UML-B models is based on some of the Event-B matching aspects that are related with matching the events and the syntactical matching of guards and actions. However, UML-B is a higher level graphical language, and we provide rules for matching UML-B class, attribute, state and variable based on their implicit semantic types, rules for matching UML-B transitions and class-events based on their semantics, and rules for matching UML-B state-machines. In addition, our proposed method considers the partial matching between the matched UML-B machines.

2.2 UML-B Diagrams and Semantics

UML-B [8], [9], [10], [11] is a graphical front-end for the formal method Event-B. It shares similar properties with UML [7], [15], but UML-B has its own meta-model. UML-B offers four diagrams: package, context, class, and state-machine. These diagrams are translated to Event-B for verification by Rodin theorem provers.

In package diagram, contexts, machines and their interconnecting relationships are represented. a machine refines another machine and a context extends another context while machines sees contexts. Figure 1 shows an example of a package diagram which has two machines *DeviceAbs* and *DeviceRef*, and two contexts *CTX1* and *CTX2*. Machines relate with each other by *refines* relationship such as *DeviceRef* refines *DeviceAbs*. Contexts relate by *extends* relationship such as *CTX2* extends *CTX1*. Machine and context relate by *sees* relationship such as *DeviceRef* sees *CTX2*. Figure 2 shows an example of the context diagram. Three class types are defined which are *DEVICE*, *CONTROLLER* and *ControllerType*.

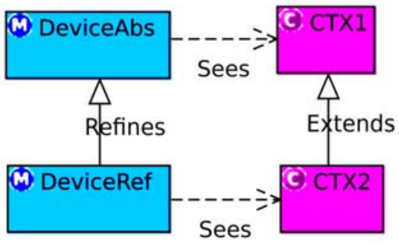


Figure 1: UML-B package diagram

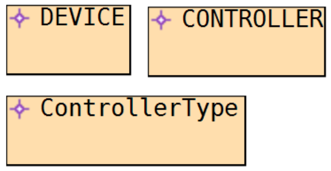


Figure 2: UML-B context diagram

In the class diagram, a system dynamic behavior is represented by classes, variables, events and invariants. Classes contain attributes, events, state machines, invariants and theorems. The class attribute may be of a *classtype* defined in the context or predefined type. The class event is executed whenever guards hold true and executes actions changing classes, attributes or machine variables values. Figure 3 shows the UML-B class diagram of a device system. Two classes are defined: *Controller* and *Device*. The association relationship *DController* relates the classes *Device* and *Controller* and represented as an attribute in the class *Device*. The class *Controller* has one attribute *Type* that is *ControllerType*. The association relationship attribute *DController* has the type of the target class *Controller*. *DController* has the multiplicity of 1..1 in the side of *Controller* class and 1..1 in the side of *Device* class allowing a device to have only one controller, and a controller to be related with one device only. The class *Device* has one class event *CreateDevice*.

In the state machine diagram, system changes its state by executing transitions. State-machine may be attached to a class or defined at the machine level.

A class partitions its behaviour into different states in class state machine. System changes its current state when firing transitions. These transitions are similar to the class events except an additional guard and additional action represented by the source and target states respectively. Figure 4 shows *ControllerSM* class state machine attached to the class *Controller*. Two different disjoint states are defined *Idle* and *Active*. Five transitions change the system states *TurnOn*, *ShutDown*, *Activate*, *DeActivate* and *ExecuteCommand*.

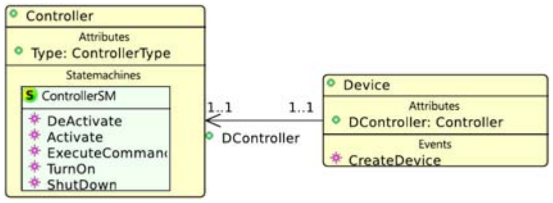


Figure 3: UML-B class diagram

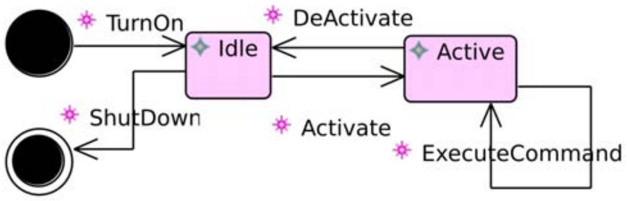


Figure 4: UML-B state machine diagram

2.2.1 UML-B semantics

The semantics of UML-B models are given by the generated corresponding Event-B models obtained from U2B translator [16]. U2B has a specific translation rule for each UML-B element. In this section, we provide a background regarding some rules to comprehend the proposed matching method.

In this article, we differentiate the UML-B implicit guards and actions from the explicit ones. The implicit (semantic) guards and actions are related with the graphical UML-B structures and generated according to the transition/class-event kind. The explicit guards and actions are the ones added explicitly to the transition/class-event/machine-event (via the *Properties* section), and they are not generated from the graphical structures. The implicit and explicit invariants are discussed similarly.

UML-B class, attribute, state and variable are translated to Event-B variable, while UML-B machine event, class event and state machine transition are translated to Event-B event. The implicit invariants and specifications for the UML-B elements are discussed in the next paragraphs.

UML-B classes, attributes, states: The classes *Controller* and *Decive*, the class attributes *Type* and *DController*, and the states *Idle* and *Active* are translated to Event-B variables.

Typing and Class Partition Invariants: Typing invariants are defined in *INVARIANTS* section. *Controller* and *Device* are typed as subsets of *CONTROLLER* and *DEVICE* respectively. *Type* is typed as total function with *Controller* and *ControllerType* as the domain and range respectively. *DController* is typed as bijective function with *Device* as the domain and *Controller* as the range. *partitions(Controller, Active, Idle)* is the class partition invariant which partitions the states of class *Controller*. This is equivalent to $((Controller = Idle \cup Active) \wedge (Idle \cap Active = \emptyset))$.

UML-B transitions and class-events: The *ControllerSM* state machine transitions *Activate*, *TurnOn* and *ShutDown*, and the class event *createDevice* are translated to Event-B events. Three kinds of transitions exist which are the constructor (transition with initial source state), the destructor (the transition with final target state) and the normal (the transition with normal source and target states). The generated implicit Event-B specifications of these transitions are as follows:

TurnOn $\triangleq$
Any *self*. **WHERE** $self \in CONTROLLER \setminus Controller$ **THEN** $Controller := Controller \cup \{self\}$ $Idle := Idle \cup \{self\}$ **END**

ShutDown $\triangleq$
Any *self*. **WHERE** $self \in Controller$ $self \in Idle$ **THEN** $Controller := Controller \setminus \{self\}$ $Idle := Idle \setminus \{self\}$ $Type := \{self\} \lhd Type$ **END**

Activate $\triangleq$
Any *self*. **WHERE** $self \in Controller$ $self \in Idle$ **THEN** $Idle := Idle \setminus \{self\}$ $Active := Active \cup \{self\}$ **END**

self represents the instance parameter of the class *Controller*. The same name *self* is used to represent in general any instance parameter of UML-B classes in this article.

TurnOn has the initial source state and the target state *Idle*. A guard related with the containing class, and two actions related with the containing class and the target state are generated. $self \in CONTROLLER \setminus Controller$ is the guard generated for the class Controller. $Controller := Controller \cup \{self\}$ and $Idle := Idle \cup \{self\}$ are the actions generated for the class *Controller* and the state *Idle* respectively. *self* represents the class instance parameter.

ShutDown has the source state *Idle* and the final target state. Two guards related with the containing class and the source state, and three actions related with the containing class, the source state and the class attribute *Type* are generated. $self \in Controller$ is the guard generated for the class Controller. $self \in Idle$ is the guard generated for the source state. $Controller := Controller \setminus \{self\}$ is the destruction action generated for the class *Controller*. $Idle := Idle \setminus \{self\}$ is the action generated for the source state. $Type := \{ self \} \lhd Type$ is the generated destruction action for the class attribute *Type*, which is necessary to preserve its functional type.

Activate has the normal source and target states *Idle* and *Active* respectively. Two guards related with the containing class and the source state, and two actions related with the source and target states are generated. $self \in Controller$ is the guard generated for the containing class *Controller*. $self \in Idle$ is the guard generated for the source state. $Idle := Idle \setminus \{self\}$ and $Active := Active \cup \{self\}$ are the actions generated for *Idle* and *Active* states respectively.

CreateDevice is a constructor class event contained in the class *Device*. This is discussed similarly to the transition *TurnOn* except that *CreateDevice* does not use states.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3. THE PROPOSED METHOD FOR MATCHING MODELS IN UML-B

The proposed method is based on UML-B semantics given by the corresponding generated Event-B state-variables and events.

The matching rules treat the UML-B models as a collection of generated state-variables and corresponding events changing their values. UML-B class, state, attribute and variable represent the state-variable, and UML-B transition, class-event and machine-event represent the corresponding event. The outcome of the matching method is a boolean identity (matched/non-matched) for the state-variables, and their related events including their guards and actions.

The matching of these UML-B state-variables and their modifying events is based on the set-theory and some aspects of the Event-B matching in [14] as follows:

- The set-theory is the basis of variables typing in Event-B which is to guide the matching of UML-B class, attribute, state and variable based on their semantic Event-B types.
- The Event-B matching steps in [14] preserve the compatibility of the matched Event-B variables by checking the related modifying events. It is assumed in [14] that one of the matched Event-B models should be completely matched with the other one. In this method, the compatibility of the matched UML-B class, attribute, state, and variable is also preserved following a similar concept via matching their modifying UML-B transitions, class-events and machine-events. However, in this method the partial matching of models is also treated, and matching rules are proposed to handle the cases that are related with both the complete and partial matching.

The proposed matching rules are classified to variable-based, event-based and state-machine matching. These are introduced in Sections 3.1, 3.2 and 3.3 respectively. Briefly, they are as follows:

- The variable-based matching rules concern matching the state-variables (UML-B class, attribute, state and variable).
- The event-based matching rules are related with the variable-based matching rules and concern matching the corresponding events (UML-B transition and class-event). The transition and class-event matching cases decide the compatibility of the related state-variables based on the variable-based matching rules.
- The state-machine matching is proposed based on the UML-B state and transition matching rules.

The matching is performed between two models *M1* and *M2*. As shown in Table 1, *M1C, M1A, M1S* and *M1V* represent the model *M1* class, attribute, state and variable respectively, and *M1T* and *M1CE* represent the model *M1* transition and class event. *M1CASVm* represents the matched state-variables that are the matched class, attribute, state and variable. *M1CASVnm* represents the non-matched state-variables. *M1Em* represents the generated matched event from transition, class-event and machine-event. *M1Enm* represents the non-matched generated event. The UML-B elements (generated state-variables and corresponding events) of the model *M2* are discussed similarly.

Table 1: UML-B Matching Frequently Used Symbols

Model/ UML-B element, Event-B Generated State-variables and events	***M1***	***M2***
UML-B Class	***M1C***	***M2C***
UML-B Attribute	***M1A***	***M2A***
UML-B State	***M1S***	***M2S***
UML-B Variable	***M1V***	***M2V***
UML-B Transition	***M1T***	***M2T***
UML-B Class-event	***M1CE***	***M2CE***
Event-B State-variables	***M1CASVm*** (matched), ***M1CASVnm*** (non-matched)	***M2CASVm*** (matched), ***M2CASVnm*** (non-matched)
Event-B Generated Events	***M1Em*** (matched), ***M1Enm*** (non-matched)	***M2Em*** (matched), ***M2Enm*** (non-matched)

The model matching is based on the following assumptions:

- Matching occurs between elements of the same type (i.e. a transition with a transition, a class with a class).
- Classes and attributes may be implicitly or explicitly modified by transitions, class-events or machine-events. States may only be implicitly modified by transitions.
- The implicit modifiers of the UML-B class, attribute and state are as follows:
    - The implicitly modifying transitions and class-events of classes are the constructors and destructors of the classes.
    - The implicitly modifying transitions and class-events of the attributes are the destructors of the containing classes of these attributes.
    - The implicitly modifying transitions of the states are the incoming and outgoing transitions of the states.
- The actions and guards, mentioned in the conditions of variable-based matching rules include both generated implicit and explicit ones, except the modifying actions in the state matching rule *SMatch*, because states are only implicitly modified. The implicit guards and actions are those obtained from UML-B graphical semantics and generated from U2B translator in the corresponding Event-B model. The explicit guards and actions are not part of the graphical semantics and added explicitly to the UML-B model in transitions, class-events or machine-events.

3.1. Variable-Based Matching

This section presents the matching rules for the UML-B classes (*M1C, M2C*), attributes (*M1A, M2A*), states (*M1S, M2S*) and variables (*M1V, M2V*). The conditions in each rule are to guide the matching of UML-B classes, attributes, states and variables based on their semantic Event-B types, and to maintain their compatible changes by preserving that their behavioural modifications are the same.

3.1.1 Classes matching

The class matching rule *CMatch:* The classes *M1C* and *M2C* are compatibly matched (*M1C=M2C*), if the following three conditions are met:

- ***CMatchCondition1***: *M1C* and *M2C* have the same type and the same instances as formalized as follows:

$$\boxed{\begin{array}{l}(M1C \in \mathbb{P}(TYPE1) \wedge M2C \in \mathbb{P}(TYPE2)) \wedge \\ (TYPE1=TYPE2) \wedge \\ (\forall\, self\,.\, ((self \in M1C) \Leftrightarrow (self \in M2C)))\end{array}}$$

- ***CMatchCondition2***: The implicitly and explicitly modifying transitions *M1T*, class-events *M1CE*, and machine-events of *M1C* are matched with the implicitly and explicitly modifying transitions *M2T*, class-events *M1CE*, and machine-events of *M2C*.
- ***CMatchCondition3***: In every matched transitions, class-events, and machine-events, the following three sub-conditions are met:
    - ***CMatchCondition3_{MatchedActions}***: The actions, which modify the classes (*M1C*, *M2C*), should be syntactically matched and dependent only on the classes and possibly other matched state-variables.
    - ***CMatchCondition3_{MatchedGuards}***: The guards, which depend only on: the classes (*M1C, M2C*); or other matched state-variables (*M1CASVm*, *M2CASVm*), should be enabled together and syntactically matched.
    - ***CMatchCondition3_{NonMatchedGuards}***: The guards, which depend on non-matched state-variables, and cannot be matched, should be enabled together

3.1.2 Class attributes matching

Attributes matching is considered relational, since the semantic of an attribute is recognized as a relation with the containing class of the attribute as the relation domain and the selected type as the relation range. The relation has several possible types (total, functional, surjective, injective, bijective).

The attribute matching rule *AMatch:* The attributes *M1A* and *M2A*, as in Figure 5, are compatibly matched (*M1A=M2A*) for all relational types, if the following three conditions are met:

- ***AMatchCondition1***: The containing classes of *M1A* and *M2A* are compatibly matched, selected types are the same, and the relation between domain and range is the same as formalized as follows:

$$(M1A \in M1C \leftrightarrow TYPE1 \land M2A \in M2C \leftrightarrow TYPE2) \land$$

$$(M1C=M2C) \land$$

$$(TYPE1=TYPE2) \land$$

$$(\forall\, self,t\,.\,((self \mapsto t \in M1A) \Leftrightarrow (self \mapsto t \in M2A)))$$

- ***AMatchCondition2***: The implicitly and explicitly modifying transitions *M1T*, class-events *M1CE*, and machine-events of *M1A* are matched with the implicitly and explicitly modifying transitions *M2T*, class-events *M1CE*, and machine-events of *M2A*.
- ***AMatchCondition3***: In every matched transitions, class-events, and machine-events, the following three sub-conditions are met:
  - ***AMatchCondition3MatchedActions***: The actions, which modify the attributes (*M1A, M2A*), should be syntactically matched and dependent only on the attributes and possibly other matched state-variables.
  - ***AMatchCondition3MatchedGuards***: The guards, which depend only on: the attributes (*M1A, M2A*); or other matched state-variables (*M1CASVm*, *M2CASVm*), should be enabled together and syntactically matched.
  - ***AMatchCondition3NonMatchedGuards***: The guards, which depend on non-matched state-variables, and cannot be matched, should be enabled together.

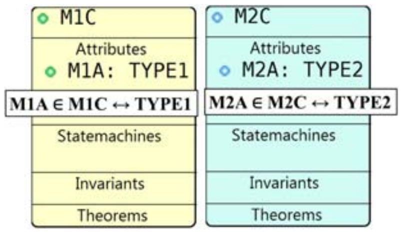


Figure 5: The Class Attributes Matching

3.1.3 States matching

This is part of state-machine matching that contains states and transitions. This section presents the rule *SMatch* for matching states, Section 3.2 presents the rule *TMatch* for matching transitions, and Section 3.3 presents the rule *SMMatch* for matching state-machines based on *SMatch* and *TMatch*.

Figure 6 shows the state machine *M1CSM* in the class *M1C*, and the state machine *M2CSM* in the class *M2C*. For simplicity, we consider matching states from flattened state-machines where no nested state-machines are allowed in the states. *M1CSM* is to be matched with *M2CSM*. *M1CSM* contains the states *M1S1, M1S2, ...M1Sn* states and their types are determined by the typing invariant *M1Si ∈ ℙ(M1C)* and class partition invariant *partition(M1C,M1S1,M1S2,..,M1Sn)*. *M2CSM* contains the states *M2S1, M2S2, ...M2Sn* states and their types are determined by the typing invariant *M2Si ∈ ℙ(M2C)* and class partition invariant *partition(M2C,M2S1,M2S2,..,M2Sn)*. The rule *SMatch* defines the conditions to match the states of the models *M1* and *M2*.

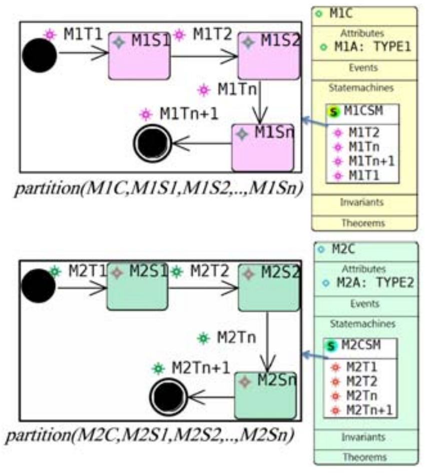


Figure 6: The States Matching

The state matching rule *SMatch:* The states *M1S* and *M2S* are compatibly matched (*M1S*=*M2S*), if the following three conditions are met:

- ***SMatchCondition1***: The containing classes *M1C* and *M2C* of *M1S* and *M2S* have the same type, and the states have the same instances as formalized as follows:

$$\boxed{\begin{array}{l}(M1S \in \mathbb{P}(M1C) \wedge M2S \in \mathbb{P}(M2C)) \wedge \\ (M1C \in \mathbb{P}(TYPE1) \wedge M2C \in \mathbb{P}(TYPE2) \wedge TYPE1=TYPE2) \wedge \\ (\forall\, self\,.\ ((self \in M1S) \Leftrightarrow (self \in M2S)))\end{array}}$$

- ***SMatchCondition2***: The implicitly modifying incoming and outgoing transitions *M1T* of *M1S* are matched with the implicitly modifying incoming and outgoing transitions *M2T* of *M2S*.
- ***SMatchCondition3***: In every matched transitions, the following three sub-conditions are met:
  - ***SMatchCondition3MatchedActions***: The actions, which modify the states (*M1S, M2S*), should be syntactically matched and dependent only on the states.
  - ***SMatchCondition3MatchedGuards***: The guards, which depend only on: the states (*M1S, M2S*); or other matched state-variables (*M1CASVm*, *M2CASVm*), should be enabled together and syntactically matched.
  - ***SMatchCondition3NonMatchedGuards***: The guards, which depend on non-matched state-variables, and cannot be matched, should be enabled together.

This is comparatively a flexible rule compared to the attribute matching rule *AMatch*, because matching the containing classes is not a necessary condition for the states to be compatibly matched. This is due to the type of a state which is a sub-set of its containing class.

3.1.4 Variables matching

UML-B variable semantic is the same of its corresponding Event-B variable. The rule *VMatch* defines the conditions to match the variables of the models *M1* and *M2*.

The variable matching rule *VMatch:* The variables *M1V* and *M2V* are compatibly matched (*M1V* = *M2V*), if the following three conditions are met:

- ***VMatchCondition1***: *M1V* and *M2V* have the same type *TYPE1=TYPE2* and possible values as formalized as follows:

$$\boxed{\begin{array}{l}(M1V \in TYPE1 \wedge M2V \in TYPE2) \wedge \\ (TYPE1=TYPE2) \wedge \\ (\forall\, x\,.\ ((M1V=x) \Leftrightarrow (M2V=x)))\end{array}}$$

- ***VMatchCondition2***: The explicitly modifying transitions, class-events, and machine-events of *M1V* are matched with the explicitly modifying transitions, class-events, and machine-events of *M2V*.
- ***VMatchCondition3***: In every matched transitions, class-events, and machine-events, the following three sub-conditions are met:
  - ***VMatchCondition3MatchedActions***: The actions, which modify the variables (*M1V, M2V*), should be syntactically matched and dependent only on the variables and possibly other matched state-variables.
  - ***VMatchCondition3MatchedGuards***: The guards, which depend only on: the variables (*M1V, M2V*); or other matched state-variables (*M1CASVm, M2CASVm*), should be enabled together and syntactically matched.
  - ***VMatchCondition3NonMatchedGuards***: The guards, which depend on non-matched state-variables, and cannot be matched, should be enabled together.

3.2. Event-Based Matching

In this section, matching UML-B transitions (*M1T, M2T*) and class-events (*M1CE, M2CE*) of the models *M1* and *M2* is discussed.

3.2.1 Transitions matching

Three transition types exist: constructor, destructor and normal. Table 2 shows the nine matching cases for the transitions of the two models *M1* and *M2*. *M1C* and *M2C* represent the containing classes of the transitions *M1T* and *M2T* respectively.

Table 2: Transition Matching Cases

Matching case/ Transition type	The Containing Classes (*M1C, M2C*)	***M2T***	***M1T***	Possibility
TCase1	Matched (*M1C = M2C*), Non-matched (*M1C ≠ M2C*)	initial source state (Constructor)	initial source state (Constructor)	Yes
-	-	initial source state (Constructor)	final target state (Destructor)	No
TCase2	Non-matched (*M1C ≠ M2C*)	initial source state (Constructor)	normal source and target states (Normal)	Yes
-	-	final target state (Destructor)	initial source state (Constructor)	No
TCase3	Matched (*M1C = M2C*), Non-matched (*M1C ≠ M2C*)	final target state (Destructor)	final target state (Destructor)	Yes
TCase4	Non-matched (*M1C ≠ M2C*)	final target state (Destructor)	Normal source and target states (Normal)	Yes
TCase2S	Non-matched (*M1C ≠ M2C*)	Normal source and target states(Normal)	initial source state (Constructor)	Yes
TCase4S	Non-matched (*M1C ≠ M2C*)	Normal source and target states(Normal)	final target state (Destructor)	Yes
TCase5	Matched (*M1C = M2C*), Non-matched (*M1C ≠ M2C*)	Normal source and target states(Normal)	Normal source and target states(Normal)	Yes

In all the possible cases, the following terms apply:

- The classes *M1C* and *M2C* should have the same type.
- It is either that the containing classes *M1C* and *M2C* of the matched transitions are matched (*M1C = M2C*) or non-matched (M1C ≠ M2C). In the case that *M1C* and *M2C* are non-matched, it is either *M1C* is a subset of *M2C* (*M1C ⊂ M2C*), *M2C* is a subset of *M1C* (*M2C ⊂ M1C*), or they share some instances/states *(M1C ∩ M2C ≠ M1C)∧ (M1C ∩ M2C ≠ M2C) ∧ (M1C ∩ M2C ≠ ∅)*. This is decided based on the matching possible cases. The compatible matching *M1C=M2C* applies when the classes *M1C* and *M2C* and their state machines *M1CSM* and *M2SCM* are matched completely, while the non-compatible matching *M1C ≠ M2C* applies when they are matched partially.
- The possible cases represent the situations where there is a chance to match some class instances and states. In other words, it should be at least possible to match some states, while this is not necessary for classes and attributes to provide flexibility for the transition matching rule.
- Generated semantic implicit guards and actions that decide the matching compatibility for class, state and attribute are the focus point of discussion, while the explicit ones are assumed that they should not violate the variable-based matching rules.

The transition matching rule *TMatch:* The rule *TMatch* defines the possible cases of matching two transitions *M1T* and *M2T* of the models *M1* and *M2*.

TCase1: a constructor *M1T1* is matched with a constructor *M2T1,* as in Figure 7.

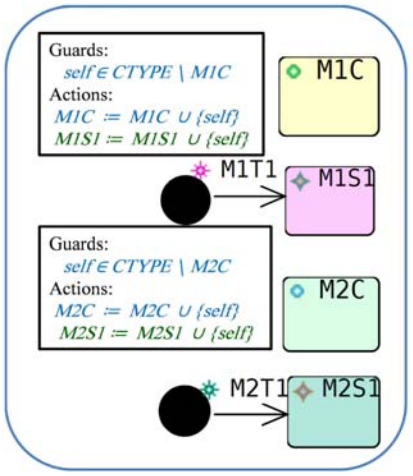


Figure 7: TCase1 Matching

TCase1 does not violate the class matching rule *CMatch*, and it is not enough to decide the preservation of *CMatch* (*M1C* and *M2C* compatible matching) looking at *TCase1* alone. *M1C* and *M2C* are compatibly matched (*M1C=M2C*) given that all the remaining classes constructors and destructors matching cases do not violate *CMatch*. *M1C* and *M2C* are non-matched (*M1C* ≠ *M2C*), if there exists one constructor/destructor matching case violates *CMatch*. Figure 7 shows the generated implicit guards and actions for the containing classes and connected states.

-Given that *M1* and *M2C* are matched, the classes related guards and actions are only related with matched classes and syntactically matched. The *M1C* class-related guard *self* ∈ *CTYPE* ∖ *M1C* and the *M1C* class-related action *M1C*≔ *M1C* ∪ *{self}* are matched with *self* ∈ *CTYPE* ∖ *M2C* and *M2C*≔ *M2C* ∪ *{self}* respectively.

-Given that *M1C* and *M2C* are non-matched, the classes related guards and actions are related with non-matched classes and cannot be matched. The *M1C* class-related guard *self* ∈ *CTYPE* ∖ *M1C* and the *M1C* class-related action *M1C*≔ *M1C* ∪ *{self}* cannot be matched with *self* ∈ *CTYPE* ∖ *M2C* and *M2C*≔ *M2C* ∪ *{self}* respectively. However, *self* ∈ *CTYPE* ∖ *M1C* and *self* ∈ *CTYPE* ∖ *M2C* could be enabled together for all possible values of *self* instance.

-*TCase1* does not violate the state matching rule *SMatch* for *M1S1* and *M2S1*. The generated implicit actions for the states *M1S1* and *M2S1* (*M1S1* ≔ *M1S1* ∪ *{self}* and *M2S1* ≔ *M2S1* ∪ *{self}*) are only related with matched states and syntactically matched given that all other incoming and outgoing transitions do not violate the state-matching rule *SMatch* making the states matched (*M1S1*=*M2S1*).

TCase2: a constructor transition *M2T1* is matched with a normal transition *M1T2,* as in Figure 8.

TCase2 is only for the case that *M1C* and *M2C* are non-matched, because there exists a constructor, which is non-matched with a corresponding constructor, violating the second condition in the class matching rule *CMatch*, and the classes-related guards and actions cannot be syntactically matched violating the third condition in *CMatch*. Figure 8 shows the generated implicit guards and actions for the containing classes and connected states.

- *M1C* and *M2C* are non-matched and the *M1C* class-related guard *self* ∈ *M1C* cannot be matched with *self* ∈ *CTYPE* ∖ *M2C*. The *M2C* class-related action *M2C* ≔ *M2C* ∪ *{self}* does not have a corresponding match in *M1T2*. However, *self* ∈ *M1C* ∧ *self* ∈ *M1S1* and *self* ∈ *CTYPE* ∖ *M2C* could be enabled together for all possible values of *self* instance.

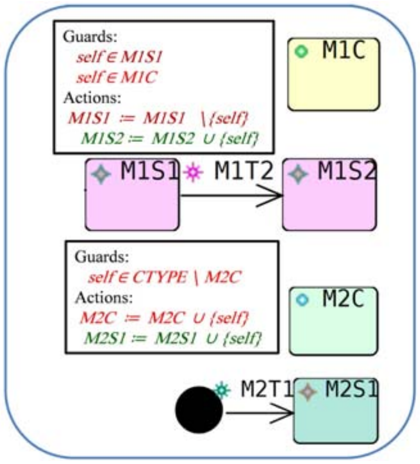


Figure 8: TCase2 Matching

-*TCase2* does not violate the state matching rule *SMatch* for states *M1S2* and *M2S1*. The generated implicit actions for the states *M1S2* and *M2S1* (*M1S2* ≔ *M1S2* ∪ *{self}* and *M2S1* ≔ *M2S1* ∪ *{self}*) are only related with matched states and syntactically matched given that all other incoming and outgoing transitions do not violate the state matching rule *SMatch* conditions making the states matched (*M1S2*=*M2S1*). The state *M1S1* is indeed non-matched, because its state matching rule *SMatch* is violated.

TCase3: a destructor *M1Tn+1* is matched with a destructor *M2Tn+1,* as in Figure 9.

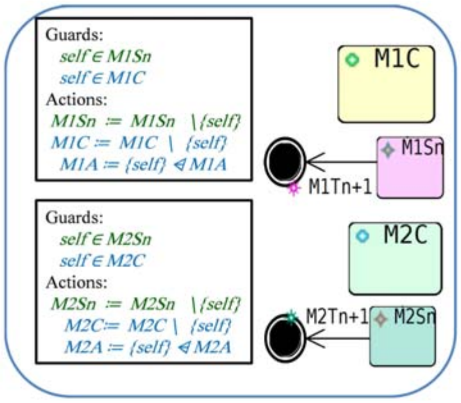


Figure 9: TCase3 Matching

Similar to *TCase1*, *TCase3* does not violate the class matching rule *CMatch*, and it is not enough to decide the compatible matching of the classes *M1C* and *M2C* looking at *TCase3* alone, as both situations are possible based on the remaining constructors and destructors matching cases. Figure 9 shows the generated implicit guards and actions for the containing classes, class attributes and connected states.

-Given that *M1C* and *M2C* are matched, the classes-related guards and actions are only related with matched classes and syntactically matched.

--Any class attribute in the matched classes *M1C* and *M2C* could be matched given that all other modifying transition matching cases do not violate the attribute matching rule *AMatch*.

-Given that *M1C* and *M2C* are non-matched, the classes-related guards and actions cannot be matched because they are related with non-matched classes. However, the guards that are related with *M1C* and *M2C* could be enabled together for all possible values of self instance.

--Any class attribute in the non-matched classes *M1C* and *M2C* are considered non-matched based on the attribute matching rule *AMatch*.

-*TCase3* does not violate the state matching rule *SMatch* for the states *M1Sn* and *M2Sn*. The generated implicit guards and actions for the states *M1Sn* and *M2Sn* are only related with matched states and syntactically matched given that all other incoming and outgoing transitions do not violate the state matching rule *SMatch* conditions making the states matched (*M1Sn=M2Sn*).

TCase4: A destructor transition *M2Tn+1* is matched with a normal transition *M1Tn,* as in Figure 10.

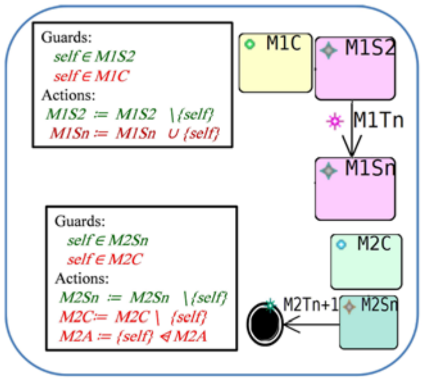


Figure 10: TCase4 Matching

TCase4 is only for the case that *M1C* and *M2C* are non-matched, because there exists a destructor, which is non-matched with a corresponding destructor, violating the second condition in the class matching rule *CMatch*, and the classes-related actions cannot be syntactically matched violating the third condition in *CMatch*. Figure 10 shows the generated implicit guards and actions for the containing classes and connected states.

- *M1C* and *M2C* are non-matched and the *M1C* class-related *self* ∈ *M1C* cannot be matched with *self* ∈ *M2C* and the *M2C* class-related destruction action *M2C* ≔ *M2C* ∖ *{self}* does not have a match in *M1Tn*. However, *self* ∈ *M1C* and *self* ∈ *M2C* could be enabled together for all possible values of *self* instance.

--Any class attribute in the non-matched classes *M1C* and *M2C* is considered non-matched based on the attribute matching rule *AMatch*.

-*TCase4* does not violate the state matching rule *SMatch* for the states *M1S2* and *M2Sn*. The generated implicit actions for the states *M1S2* and *M2Sn* are only related with matched states and syntactically matched given that all other incoming and outgoing transitions do not violate the state matching rule *SMatch* conditions making the states matched (*M1S2=M2Sn*). The state-matching rule *SMatch* for *M1Sn* is violated, because a modifying transition of *M1Sn* does not have a corresponding match violating the second condition *SMatchCondition2*.

TCase5: a normal *M1Tn* is matched with a normal *M2Tn,* as in Figure 11.

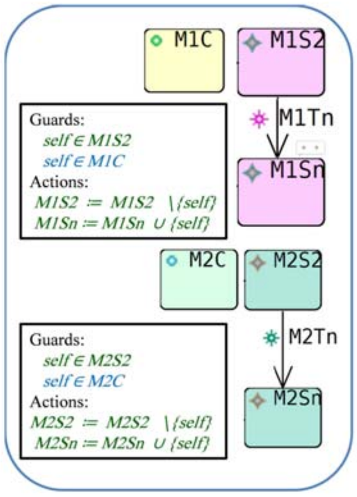


Figure 11: TCase5 Matching

Similar to *TCase1* and *TCase3*, *TCase5* does not violate the class matching rule *CMatch* where *M1C* and *M2C* could be compatibly matched (*M1C=M2C*) or non-matched (*M1C ≠ M2C*) based on the remaining matching cases. Figure11 shows the generated implicit guards and actions for the containing classes and connected states.

TCase5 does not violate the state matching rule *SMatch* when matching the states *M1S2* and *M1Sn*, with the states *M2S2* and *M2Sn*. The modified states could be compatibly matched given that all other incoming and outgoing transitions do not violate *SMatch*.

TCase2S and TCase4S: These cases are similar to *TCase2* and *TCase4* respectively and discussed similarly.

Constructor and Destructor Matching Cases: These cases are neither possible nor applicable, because it is not possible to match construction guards/actions with destruction guards/actions of matched containing classes, and matched instances and states cannot be found in these cases.

Class-events matching is similar to transition matching except that class events do not use states.

3.3. State-Machine Matching

The state matching rule *SMatch* and the transition matching cases in *TMatch* form the necessary ingredients for matching the state machines *M1CSM* and *M2CSM* in the state-machine matching rule *SMMatch*. For being systematic, the matching should be continuous which means the state-machine cannot be divided into several parts where some are matched and the others are none. This is because the state-machine represents a consecutive behaviour via a sequence of states and we do not intend the matched part to be interrupted.

M1CSM has *M1S1,...M1Sn* states and *M1T1,...M1Tn,M1Tn+1* transitions. *M2CSM* has *M2S1,...M2Sn* states and *M2T1,...M2Tn,M2Tn+1* transitions. Note that the number of states *n* and the number of transitions *n+1*, shown in *M1CSM* and *M2CSM*, represent just examples, and different numbers of states and transitions may exist in *M1CSM* and *M2CSM*.

The state-machine matching rule *SMMatch:* Two different rule cases exist *SMMatch_1* and *SMMatch_2.*

SMMatch_1: *M1CSM* and *M2CSM* are matched completely. They have the same number of states and transitions and they are all matched. In this case, *M1C* and *M2C* are compatibly matched (*M1C = M2C*).

Two possible states sequences matchings exist. We provide a general representation of these in sequences *Seq1* and *Seq2* which are represented by the transitions matching cases.

In *Seq1*, constructors are matched first as in *TCase1*, then, normal transitions are matched as in *TCase5*, and finally, destructors are matched as in *TCase3*. Figure 12 shows an example of *Seq1* matching. The dotted arrows represent the matched transitions from the state-machine *M2CSM* of the model *M2*. The dashed rectangles around the states (*M1S1, M1S2* and *M1Sn*) represent the matched states from the state-machine *M2CSM* of the model *M2*.

Seq1
TCase1(M1C = M2C), TCase5(M1C = M2C), TCase3(M1C = M2C)

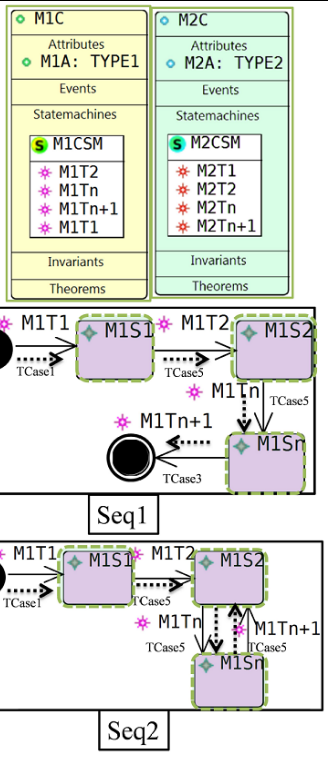


Figure 12. State-Machine Matching Seq1 and Seq2

In *Seq2*, constructors are matched first as in *TCase1*, and normal transitions are matched as in *TCase5*. This is similar to *Seq1*, except that no destructor matching case exists. Figure 13 shows an example of *Seq2* matching.

Seq2
TCase1(M1C = M2C), TCase5(M1C = M2C)

SMMatch_2: *M1CSM* and *M2CSM* are matched partially. They may have the same or different number of states and transitions where some are matched. In this case, *M1C* and *M2C* are non-matched (*M1C ≠ M2C*).

Several possible states sequence matching exist. We provide an example as in *SeqNM* in Figure 13.

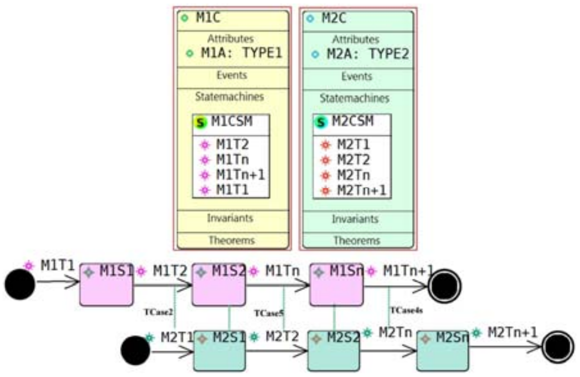


Figure 13. State-Machine Matching- SeqNM

SeqNM
TCase2, TCase5(M1C ≠ M2C), TCase4s

In *SeqNM*, a constructor *M2T1* is matched with a normal *M1T2* as in *TCase2*, then, a normal *M2T2* is matched with a normal *M1Tn* as in *TCase5*, and finally, a normal M2Tn is matched with a destructor *M1Tn+1* as in *TCase4s*. The dashed lines in Figure 13 represent the matching linkage between the matched transitions and states.

4. FORMALIZATION AND CORRECTNESS OF THE MATCHING METHOD

The correctness of the matching rules is based on the compatibility preservation of the matched state-variables, the modifying events, and their matched guards and actions predicates in the matching rules. This is based on UML-B semantics given by the generated Event-B specification for the matched UML-B class, attribute, state, variable that are discussed in the variable-based matching, and transition and class-event that are discussed in the event-based matching.

In this section, we show first the required compatibility conditions in Section 4.1, then we show the formalization of the matched machines in Section 4.2 when following the proposed matching rules, and finally we show how the compatibility required conditions are preserved in the formalization in Section 4.3.

4.1. The Required Compatibility Conditions

The following compatibility conditions should be maintained within the proposed variable-based and event-based matching rules:

- **The Compatibility Conditions of The Matched State-variables**: The following two conditions *CASVMatchCondition2* and *CASVMatchCondition3* are to preserve the compatible changes of the matched state-variables. These represent the second and third conditions in each variable-based matching rule. These conditions are *CMatchCondition2* and *CMatchCondition3* in the class matching rule *CMatch*, *AMatchCondition2* and *AMatchCondition3* in the attribute matching rule *AMatch*, *SMatchCondition2* and *SMatchCondition3* in the state matching rule *SMatch*, and *VMatchCondition2* and *VMatchCondition3* in the variable matching rule *VMatch*.
  - ***CASVMatchCondition2***: All the modifying transitions, class-events and machine-events of the matched state-variables *M1CASVm* and *M2CASVm* (class, attribute, state and variable) should be matched.
  - ***CASVMatchCondition3***: In every matched transitions, class-events and machine-events, the following conditions are preserved:
    - ▪ The actions, which modify matched state-variables *M1CASVm* and *M2CASVm* (class, attribute, state and variable), should be syntactically matched and dependent only on matched state-variables.

- The guards, which depend only on matched state-variables *M1CASVm* and *M2CASVm* (class, attribute, state and variable), should be enabled together and syntactically matched.
- The guards, which depend on non-matched state-variables, and cannot be matched, should be enabled together.

Note that these two conditions *CASVMatchCondition2* and *CASVMatchCondition3* are to help preserving the first conditions in each variable-based matching rule (*CMatchCondition1*, *AMatchCondition1*, *SMatchCondition1*, and *VMatchCondition1*).

Note also that *CASVMatchCondition2* and *CASVMatchCondition3* are related with the cases *TCase1*, *TCase2*, *TCase2s*, *TCase3*, *TCase4*, *TCase4s* and *TCase5* in the event-based matching rules. These cases decide the matching compatibility of variable-based matching.

- **The Compatibility Condition of The Matched Guards and Actions**: The following condition *EGrdActMatchCondition* is to preserve the compatibility of the matched guards and actions in the event-based matching.
  - ***EGrdActMatchCondition***: The guards and actions are only considered matched, if they are dependent only on matched state-variables and syntactically matched.

4.2. The Formalization of the Matched Machines

In the following description, an Event-B formalization of the state-variables and corresponding generated events of both matched machines *M1* and *M2* is introduced. These are based on the proposed matching rules.

In addition to the formalized first conditions of each variable-based matching rule (*CMatchCondition1*, *AMatchCondition1*, *SMatchCondition1*, and *VMatchCondition1*), the Event-B formalizations of both machines *M1* and *M2* are as follows:

```
MACHINE M1
VARIABLES M1CASVm  M1CASVnm
EVENTS
M1Em ≙
  WHEN
   M1EmGm(M1CASVm)
   M1EmGnm(M1CASVm, M1CASVnm)
 THEN
  M1CASVm :| M1EmAm(M1CASVm, M1CASVm′)
  M1CASVnm :| M1EmAnm(M1CASVm, M1CASVnm,
  M1CASVnm′)
 END
M1Enm ≙
WHEN
 M1EnmGnm(M1CASVm, M1CASVnm)
THEN
  M1CASVnm :| M1EnmAnm(M1CASVm, M1CASVnm,
  M1CASVnm′)
END
```

```
MACHINE M2
VARIABLES M2CASVm  M2CASVnm
EVENTS
M2Em ≙
  WHEN
   M2EmGm(M2CASVm)
   M2EmGnm(M2CASVm, M2CASVnm)
 THEN
  M2CASVm :| M2EmAm(M2CASVm, M2CASVm′)
  M2CASVnm :| M2EmAnm(M2CASVm, M2CASVnm,
  M2CASVnm′)
 END
M2Enm ≙
WHEN
 M2EnmGnm(M2CASVm, M2CASVnm)
THEN
  M2CASVnm :| M2EnmAnm(M2CASVm, M2CASVnm,
  M2CASVnm′)
END
```

M1CASVm and *M2CASVm* represent the matched state-variables (class, attribute, state and variable) of both machines *M1* and *M2* respectively. *M1CASVnm* and *M2CASVnm* represent the non-matched state-variables of both machines *M1* and *M2* respectively.

M1Em and *M2Em* represent the generated matched events (transitions, class-events and machine-events) of both machines *M1* and *M2* respectively.

M1EmGm in *M1Em* represent the guards that are related only with the matched state-variables *M1CASVm* and matched with their corresponding *M2EmGm* in *M2Em*.

M1EmAm in *M1Em* represent the actions that modify and depend only on the matched state-variables *M1CASVm* and matched all with their corresponding *M2EmAm* in *M2Em*.

M1EmAnm and *M2EmAnm* represent the non-matched actions that modify only the non-matched state-variables *M1CASVnm* and *M2CASVnm* respectively. *M1EmGnm* and *M2EmGnm* represent the non-matched guards.

M1Enm and *M2Enm* represent the non-matched generated events of the models *M1* and *M2* respectively. *M1EnmGnm* and *M2EnmGnm* depend on the matched and non-matched state-variables. *M1EnmAnm* and *M2EnmAnm* modify only the non-matched state-variables and may depend on both matched and non-matched state-variables.

4.3. The Preservation of the Compatibility Conditions in the Formalization

This includes the preservation of the compatible changes of the matched state-variables *M1CASVm* and *M2CASVm*, and the compatibility preservation of the matched guards and actions in the matched events *M1Em* and *M2Em*. These conditions should be preserved, as in the formalization of the machines *M1* and *M2*, when following the proposed matching rules.

4.3.1 The compatibility preservation of the matched state-variables

The compatible changes of the matched state-variables *M1CASVm* and *M2CASVm* is preserved by maintaining in the formalization the aforementioned two conditions *CASVMatchCondition2* and *CASVMatchCondition3* that are explained in Section 4.1. These conditions should be preserved as follows.

- **The condition *CASVMatchCondition2***: The matched actions *M1EmAm* and *M2EmAm* are all the modifying actions of *M1CASVm* and *M2CASVm* that only exist in the matched generated events *M1Em* and *M2Em*. This implies that all the modifying events, via actions, of *M1CASVm* and *M2CASVm* are matched.
- **The condition *CASVMatchCondition3***: In *M1Em* and *M2Em*, the following sub-conditions are preserved as follows:
  - Firstly, *M1EmAm* and *M2EmAm* are only related with *M1CASVm* and *M2CASVm*, and syntactically matched. There is no non-matched action (*M1EmAnm*, *M2EmAnm*, *M1EnmAnm*, *M2EnmAnm*) that modifies a matched state-variable (*M1CASVm* or *M2CASVm*).
  - Secondly, *M1EmGm* and *M2EmGm,* which are only related with *M1CASVm* and *M2CASVm*, are syntactically matched. Because of that, *M1EmGm* and *M2EmGm* are enabled together.
  - Thirdly, *M1EmGnm* and *M2EmGnm* do not contradict with each other and it should be always possible that they are enabled together whenever matching *M1Em* and *M2Em*.

The syntactical matching of the implicit predicates *M1EmGm* and *M1EmAm* with their corresponding *M2EmGm* and *M2EmAm*, and how they should depend on only matched state-variables have been discussed in the transition matching rule *TMatch*.

The simultaneous enabling of the implicit non-matched guards *M1EmGnm* and *M2EmGnm* are discussed in more details in this section. We focus on the implicit non-matched guards of the matching cases in *TMatch*, because these implicit guards represent the semantics of UML-B. It is up to the method user to investigate the simultaneous enabling of the explicit guards. Remember that, in all the matching cases *TCase1, TCase2, TCase2s TCase3, TCase4*, *TCase4s*, and *TCase5*, the graphical containing classes (*M1C*, *M2C*) of the transitions have the same type *CTYPE*. The simultaneous enabling of the non-matched implicit guards in the cases is discussed as follows.

In *TCase1*, a constructor transition *M2T1* of a class *M2C* is matched with a constructor transition *M1T1* of a class *M1C*. When the classes *M1C* and *M2C* are non-matched, *self* ∈ *CTYPE* ∖ *M1C* and *self* ∈ *CTYPE* ∖ *M2C* are non-matched guards. It is always possible that these guards are enabled together whenever *M1T1* and *M2T1* are matched. This is because *M1T1* and *M2T1* are both constructors, their non-matched implicit guards are simply non-belonging conditions to the classes representing the semantics of two constructors, and they do not disable each other. This is formalized as follows:

∀ *self, M2T1, M1T1, M1C, M2C* . (((*M1T1* and *M2T1* are constructors of *M1C* and *M2C*) ∧ (*M1T1* is matched with *M2T1*)) ⇒ (*self* ∈ *CTYPE* \ *M1C* ∧ *self* ∈ *CTYPE* \ *M2C*))

In *TCase2*, a constructor *M2T1* of a class *M2C* is matched with a normal transition *M1T2* of a class *M1C*. *self* ∈ *CTYPE* \ *M2C* is the non-matched guard in *M2T1*. *self* ∈ *M1C* and *self* ∈ *M1S1* are the non-matched guards in *M1T2*. It is always possible that these guards are enabled together whenever *M2T1* and *M1T2* are matched. This is because *M2T1* and *M1T2* are constructor and normal transitions, their non-matched implicit guards are simply belonging/non-belonging conditions representing the semantics of a constructor and a normal transitions, and they do not disable each other. *TCase4, TCase2S*, *TCase4S* are discussed similarly. This is formalized as follows:

∀ *self, M2T1, M1T2, M1C, M2C, M1S1*. (((*M2T1* is a constructor of *M2C* and *M1T2* is a normal transition of *M1C*) ∧ (*M2T1* is matched with *M1T2*)) ⇒ (*self* ∈ *CTYPE* \ *M2C* ∧ *self* ∈ *M1C* ∧ *self* ∈ *M1S1*))

In *TCase3*, a destructor transition *M1Tn+1* of a class *M1C* is matched with a destructor transition *M2Tn+1* of a class *M2C*. When the classes *M1C* and *M2C* are non-matched, *self* ∈ *M1C* and *self* ∈ *M2C* are the non-matched guards. It is always possible that these guards are enabled together whenever *M1Tn+1* and *M2Tn+1* are matched. This is because *M1Tn+1* and *M2Tn+1* are both destructors, their non-matched implicit guards are simply belonging conditions representing the semantics of two destructors, and they do not disable each other. This is formalized as follows:

∀ *self, M1Tn+1, M2Tn+1, M1C,M2C* . (((*M1Tn+1* and *M2Tn+1* are destructors of *M1C* and *M2C*) ∧ (*M1Tn+1* is matched with *M2Tn+1*)) ⇒ (*self* ∈ *M1C* ∧ *self* ∈ *M2C*))

In *TCase5*, a normal transition *M1Tn* of a class *M1C* is matched with a normal transition *M2Tn* of a class *M2C*. When the classes *M1C* and *M2C* are non-matched, *self* ∈ *M1C* and *self* ∈ *M2C* are non-matched guards. It is always possible that these guards are enabled together whenever *M1Tn* and *M2Tn* are matched. This is because *M1Tn* and *M2Tn* are both normal transitions, their non-matched implicit guards are simply belonging conditions representing the semantics of two normal transitions, and they do not disable each other. This is formalized as follows:

∀ *self, M1Tn, M2Tn, M1C, M2C* . (((*M1Tn* and *M2Tn* are normal transitions of *M1C* and *M2C*) ∧ (*M1Tn* is matched with *M2Tn*)) ⇒ (*self* ∈ *M1C* ∧ *self* ∈ *M2C*))

The explicitly added non-matched guards has to be checked when employing the matching rules so that they do not disable each other fulfilling the compatibility condition.

In a special case where all the model *M2* state-variables and generated events are matched with their corresponding in *M1*, *M2Em* is matched completely with *M1Em*. There are no non-matched guards in *M2Em*. In this situation, it is indeed whenever guards in *M1Em* are enabled, then the matched guards in *M2Em* are enabled as well.

4.3.2 The compatibility preservation of the matched guards and actions

The compatible matching of *M1EmGm* and *M1EmAm* with their corresponding *M2EmGm* and *M2EmAm* is preserved by maintaining in the formalization the aforementioned condition *EGrdActMatchCondition* that is explained in Section 4.1. This is preserved as follows.

The guards *M1EmGm* and actions *M1EmAm* in *M1Em* are only considered matched with their corresponding guards *M2EmGm* and actions *M2EmAm* in *M2Em*, because they are dependent only on matched state-variables (*M1CASVm*, *M2CASVm*), and syntactically matched.

The syntactical matching is decided based on the guards and actions predicates themselves referring to either candidate or confirmed matched state-variables. The dependency on only matched state-variables (*M1CASVm*, *M2CASVm*) requires studying and confirming, via the variable-based matching rules, that the appearing state-variables in these guards and actions are matched.

For example, the actions (*M1C* ≔ *M1C* ∪ *{self}* and *M2C* ≔ *M2C* ∪ *{self}*) are syntactically matched considering *M1C* and *M2C* as candidate matched classes. However, in addition to this syntactical matching, the appearing classes (*M1C*, *M2C*) should be confirmed matched via the class matching rule *CMatch*, for the actions to be matched.

5. COMMUNICATION-BASED CASE STUDY

This section, presents a communication-based case study showing the applicability of the proposed rules. Two systems *Sys1* and *Sys2* communicate through a middle ware *MW*, as shown in Figure 14. *Sys1* prepares a value *S1* to be processed and sends it to *MW*. Next, *MW* makes a backup of *S1* and forwards it to *Sys2* which in turn processes *S1* producing the resulted value *S2* in order to send it to *MW*. Then, *MW* makes a backup of *S2* and forwards it to *Sys1*, and finally *Sys1* obtains *S2*. *Sys1* communicates with *MW* by means of *Buffer1*, and *MW* communicates with *Sys2* by means of *Buffer2*.

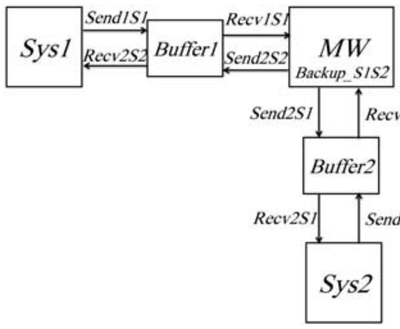


Figure 14. Communication-based Case Study Schemata

In this case-study, we show how the proposed UML-B matching rules are applied to compare two existing models *Com1* and *Com2* that have a similar behaviour related with communication. *Com1* consists of one abstract and seven refinement machines, while *Com2* consists of one abstract machine. Figure 15 shows the package diagram of the model *Com1*. We summarize each machine as follows:

ComBasedAbs: Introduces *Communication* class and the transitions to start, end and repeat the communication process.

ComBasedRef1: Introduces *Sys1* and *S1Values* classes, and the transition to prepare and send *S1* value. *Sys1* is to be a sender in the complete system scenario.

ComBasedRef2Match: Applies state machine flattening. At this level, matching is applied with an existing model *Com2* that has the functionality of a sender. At this level, state machines are completely matched.

ComBasedRef3: Introduces *Buffer1* and the communication transitions with it. *Buffer1* will serve as a communication channel with *MW* through the introduced transitions.

ComBasedRef4: Introduces *MW* class, and the transitions for sending and receiving values to and from *MW*.

ComBasedRef5Match: Applies state machine flattening. At this level, matching is applied with the model *Com2*. Matching in this case considers the partial state machines matching.

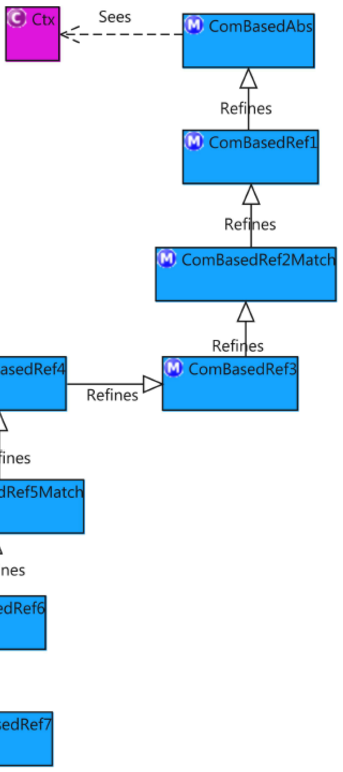


Figure 15. The Com1 Model Package Diagram

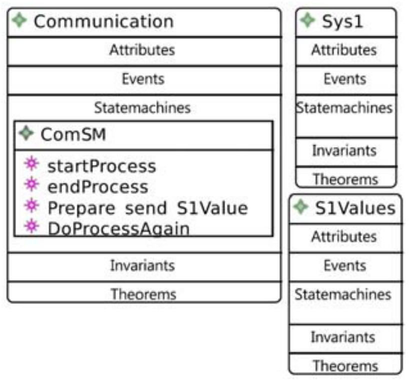
ComBasedRef6: Introduces *Buffer2* and *Sys2* classes and the communication transitions with both classes. *Buffer2* serves as a communication channel with *Sys2* through the introduced transitions.

ComBasedRef7: Introduces the class *S2Value*, and class attributes of the class *Buffer2* providing more details related with the *Sys2*.

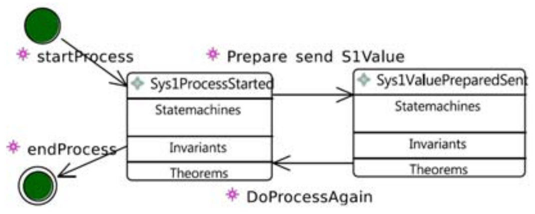
We focus in the next discussion on *ComBasedRef2Match* and *ComBasedRef5Match* where the matching rules are applied to evaluate which elements could be compatibly matched with the elements of the model *Com2*.

5.1. First Matching

Figure 16(a) and Figure 16(b) show class and state machine diagrams of *ComBasedRef2Match* machine respectively in the model *Com1*.



(a) Class Diagram



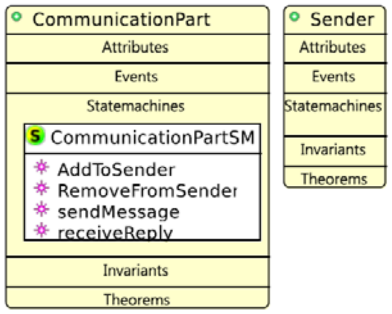
(b) State-machine Diagram

Figure 16. Com1 Model- ComBasedRef2Match

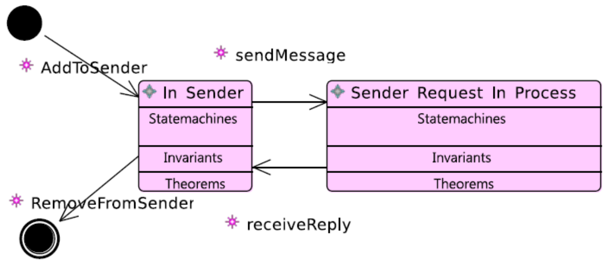
Figure 17(a) and Figure 17(b) show class and state-machine diagrams of the UML-B model *Com2* that have the behaviour of a sender initiating a request to another communication end-point.

Figure 17(a) shows the class diagram in which the class *Sender* represents the instances of the sender end-point and the class *CommunicationPart* represents the communication instances that are communicated between the *Sender* and other end-points. Figure 17(b) shows the state machine diagram in which *in_Sender* and *Sender_Request_in_Process* represent the states where *Sender* requests are prepared and processed respectively. *AddToSender* and *RemoveFromSender* are construction and destruction transitions initiating and ending the communication in *Sender* respectively. *sendMessage* and *receiveReply* transitions are to transfer the communication (from,to) the *Sender* respectively.

Matching Application: It is possible to match the machine *ComBasedRef2Match* shown in Figure 16 with the machine *Com2* shown in Figure 17. Table 3 shows the matched and non-matched UML-B elements.



(a) Class Diagram



(b) State-machine Diagram

Figure 17. The Com2 Model machine

The compatible matching of classes and states is guided by the fact that they have the same type. The class matching rule *CMatch* conditions are preserved for *Communication*= *CommunicationPart* matching. These classes have the same type containing the same states and instances, all constructors and destructors are matched, and the guards and actions that are related with *Communication* and

CommunicationPart are syntactically matched (i.e. they have the same syntactical predicates). *Sys1=Sender* compatible matching conditions are preserved and discussed similarly.

Table 3: First Possible Matching

Matching Status/ Element Type	Matched Elements	Non-Matched Elements
Class	*Communication = CommunicationPart, Sys1=Sender*	*S1Values*
Class Attribute	-	-
State	*Sys1ProcessStarted =In Sender, Sys1ValuePreparedSent= Sender _Request_ In _Process*	
Variable	-	-
Class Event	-	-
Transition	Containing classes of matched transitions are matched: *Start-Process/AddToSender(TCase1), Prepare-Send S1Value/sendMessage(TCase5), DoProcessAgain/receiveReply(TCase5), endProcess/RemoveFromSender(TCase3)*	
Machine Event	-	-
State Machine	*ComSM/CommunicationPartSM (SMMatch_1, Seq1)*	-

The state matching rule *SMatch* conditions are preserved for *Sys1ProcessStarted=In_Sender* matching. These states have the same types containing the same instances, all their incoming and outgoing transitions are matched, and all the generated implicit related guards and actions are syntactically matched. *Sys1ValuePreparedSent =Sender_Request_In_Process* compatible matching conditions are preserved and discussed similarly.

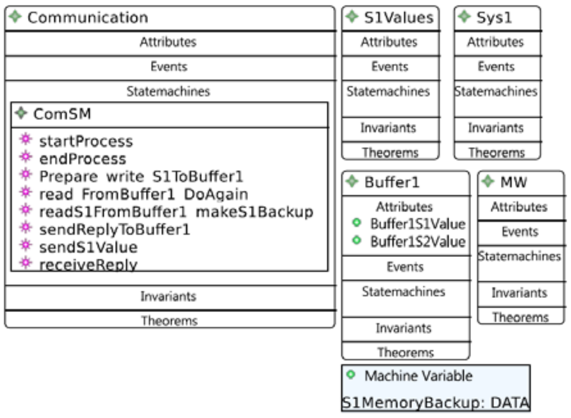
SMMatch_1 is followed to match *ComSM* and *CommunicationPartSM* completely and the matching corresponds to the sequence *Seq1*.

5.2. Second Matching

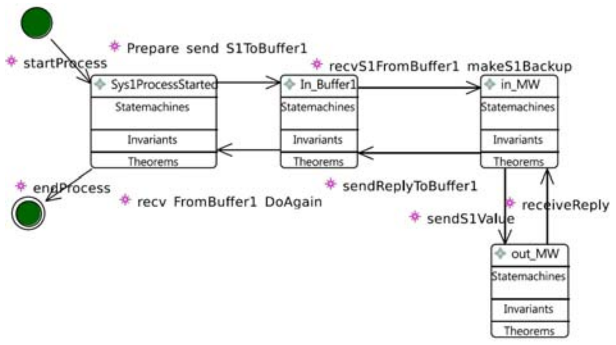
Figure 18(a) and Figure 18(b) show class and state machine diagrams of *ComBasedRef5Match* respectively in the model *Com1*.

At this level, the complete requirements for the communication between *Sys1* and *MW* through *Buffer1* are modelled. It is still required that *MW* to be a sender to *Sys2*. It is interesting to match *ComBasedRef5Match* model with the machine in same model *Com2* in Figure 17. The aim is to show the partial model matching between these machines.

Matching Application: Table 4 shows the matched and non-matched UML-B elements.



(a) Class Diagram



(b) State-machine Diagram

Figure 18. Com1 Model- ComBasedRef5Match

Table 4: Second Possible Matching

Matching Status/ Element Type	**Matched Elements**	**Non-Matched Elements**
Class	*MW=Sender*	*CommunicationPart, Communication, Sys1, S1Values, Buffer1*
Class Attribute	-	*Buffer1S1Value, Buffer1S2Value*
State	*In_MW=In_Sender, Out MW = Sender_Request_ In_Process*	*Sys1ProcessStarted, In_ Buffer1*
Variable	-	-
Class Event	-	-
Transition	Containing classes of the matched transitions are non-matched: *recvS1FromBuffer1_makeS1Backup/ AddToSender(TCase2), sendS1Value/ sendMessage(TCase5), receiveReply/ receiveReply(TCase5), sendReplyTo_Buffer1/ RemoveFromSender(TCase4)*	*startProcess, endProcess, Prepare_send_S1ToBuffer1, recv_FromBuffer1_DoAgain*
Machine Event	-	-
State Machine	*ComSM/CommunicationPartSM (SMMatch_2)*	-

Based on class transitions matching cases which correspond to *SMMatch_2*, the state machines are partially matched and the containing classes *Communication* and *CommunicationPart* cannot be matched. *TCase2* violates the rule *CMatch*, and the construction guard and action (*self* ∈ *COMMUNICATION* ∖ *CommunicationPart*, *CommunicationPart* ≔ *CommunicationPart* ∪ {*self*}) cannot find a corresponding match in *recvS1FromBuffer1_makeS1Backup*. *TCase4* violates the rule *CMatch*, the destruction guard and action (*self* ∈ *CommunicationPart* , *CommunicationPart* ≔ *CommunicationPart* ∖ {*self*}) cannot find a corresponding match in *sendReplyToBuffer1*. *TCase5* in this case study corresponds to the situation that *Communication* and *CommunicationPart* are non-matched, and the containing class guard (*self* ∈ *CommunicationPart*) cannot be matched with (*self* ∈ *Communication*) in *sendS1Value* and *receiveReply*.

States compatible matching conditions in the rule *SMatch* are discussed similarly to the first matching. *In_MW* and *Out_MW* are matched compatibly with *In_Sender* and *Out_Sender* respectively preserving the rule *SMatch*.

6. RELATED-WORK

In [14], pattern matching steps are introduced for Event-B. This assumes that the pattern is matched completely with the problem model. In our work, models can be matched completely or partially considering more possible cases of matching the models *M1* and *M2*. Based on [14], the user should decide the problem variables to be matched with their corresponding in the pattern. In our work, we provide a more concrete guidance to the user defining how to match the class, state as subset of class, and attribute as a relational type. Based on [14], the matching cannot be guaranteed unless the matched events alter the matched variables in the same way through syntactical matching of the events guards and actions. Our work is similar, as the proposed rules for matching classes, states and attributes restrict all their related modifying transitions and class-events to be matched having the same syntactically matched generated actions that modify the matched state-variables depending only on them.

In [17], a matching approach is introduced for Object-Z formal language. Their work considers both syntactical and structural similarity where syntactic similarity is calculated by comparing the elements names strings and that is considered as initial starting point for their matching approach. The structural similarity uses the object oriented aspects (i.e. class elements, relations). In our work, we consider the exact matching of UML-B elements which results in either matched/non-matched since UML-B semantics are based on Event-B and we prefer the matching correctness to be based on the set-theory strict typing for preciseness.

A specification matching method is introduced in [18] to facilitate recognizing and retrieving reusable components that address the objectives of a reuse system. They employ order-sorted predicate logic to specify the software component considering the exact, relaxed and logical matching for methods and components. In our work, we employ first order logic and set-theory considering the exact matching for UML-B individual elements and predicates and the relaxed matching between UML-B machines.

In [19], definitions for matching the formal specifications of components are introduced in the context of object-oriented programming. These are based on the first order logic and theory proving. The definitions consider exact and relaxed matching for functions and modules. Our work is similar, that is based on the first order logic and theory proving considering the UML-B elements exact match and the relaxed matching for machines, however our work considers, in addition to transitions, class-events and machine-events, the UML-B variable-based matching that is related with the set-theory.

In [20], it is proposed to extend the specification matching methods, which are limited to functions and modules, to handle the object-oriented components. This include classes matching and its contained attributes and methods. Our work considers also matching classes and their contained attributes, states, transitions and class events.

Techniques for matching state-based modules are introduced in [21]. These extend the existing specification matching methods that are based on functions specified by pre- and post-conditions. This work considers the data-refinement and the use of state and coupling invariants. Our work is similar in the sense that it considers matching state-variables represented by UML-B class, attribute, state and variable, and events represented by UML-B transition, class event and machine event. In addition, our work considers the state-variables typing invariants for the matching to be compatible.

In [22], a pattern approach which is based on set-theory is introduced for conceptual models. Their approach considers every model as a set of objects and relationships. A collection of functions is proposed and a set of operators is defined to combine the resulting sets from the functions. Their work is similar to our work as both are based generally on set-theory with some differences in terms of elements typing.

7. THE METHOD SIGNIFICANCE, LIMITATIONS AND FUTURE WORKS

In this section, we discuss the method benefits, limitations and some future works explaining in more details the future works that we are working on currently.

The method significance: In this work, we introduce rules for matching UML-B models. The matching is not necessarily complete between the models *M1* and *M2* to provide flexibility. It is necessary to preserve the compatibility conditions mentioned in Section 4 for the method to be correct. In the context of UML-B modelling, the method may serve in the following:

- Managing compared UML-B models on which the matching method is applied. This includes observing compatible and non-compatible UML-B elements among these models.
- Extract and store compatibly matched UML-B model elements, as a common model, for future reuse and integration with other models. The compatibly matched elements, which are: matched state-variables (*M1CASVm, M2CASVm*) and their modifying events (*M1Em*, *M2Em*) including their guards (*M1EmGm*, *M2EmGm*) and actions (*M1EmAm*, *M2EmAm*), may be extracted as a proven correct UML-B model that is suitable for future reuse.
- Reusing compatibly matched UML-B models in constructing larger ones to avoid remodelling and reproving.

The method limitations: In this work, for simplicity, matching state state-machines is performed for flattened state-machines in which no nested state-machines are allowed in the matched states.

Also, this work considers only the state-sets translation, whereas it does not consider the state-function translation that has different semantics of the state-set translation.

Future works: In a special situation in the matching method, the model *M2* may be matched completely with part of the model *M1*. We are currently working on: specializing the matching rules to suit this situation; and employing these special rules in a method for pattern reuse considering *M2* as the pattern model and *M1* as the problem model. In this situation, whenever a state-variable in *M1* is modified under some circumstances, it is indeed that the matched-with state-variable in *M2* is modified in the same way under the same circumstances. This is because all the state-variables and generated events of *M2* are matched with their corresponding in *M1*. The future proposed pattern reuse method will support the modelling in UML-B avoiding remodelling and reproving the pattern model when constructing a larger UML-B model.

The matching method is based on the state-sets translation. Another possible future work is to investigate the state-function translation to explore more matching options when matching UML-B models based on the these semantics that are related with the state-function.

8. CONCLUSIONS

Model matching provides means for the integration, management, and reuse for the matched models in the context of model driven software engineering. In this work, we propose a method to match models in UML-B. The contributions of this paper are as follows:

- Variable-based matching rules.
- Event-based matching rules.
- State-machine matching rules.
- A formalization of the method rules.
- A communication-based case study to illustrate the method applicability.

The variable-based matching provides rules for the compatible matching of UML-B class, attribute, state, and variable. These are based on the elements types, which are given by UML-B semantics, and the compatible behaviour of their modifying transitions, class-events, and machine-events.

The event-based matching provides rules and cases for matching UML-B transitions and class-events. These cases decide the compatibility of the related state-variables.

The state-machine matching provides rules based on the state and transition matching rules. Based on these rules, state-machines and their containing classes are partially or completely matched.

The proposed method considers the partial and complete matching of two UML-B models *M1* and *M2*.

The method rules are formalized by means of the generated corresponding Event-B specifications. The correctness of these rules is justified via preserving the compatibility of the matched state-variables and their corresponding modifying events including their matched guards and actions predicates.

The compatibility of the matched state-variables (class, attribute, state, and variable) is preserved by two conditions. First, all the modifying generated events (transitions, class-events, and machine-events) of the matched state-variables should be matched. Second, in every matched modifying events, the modifying actions of the matched state-variables should be syntactically matched and dependent only on matched state-variables, the guards that depend only on matched state-variables should be enabled simultaneously and syntactically matched, and the guards that depend on non-matched state-variables and cannot be matched should be enabled simultaneously. The compatibility of the matched guards and actions is preserved by the condition that they are only considered matched, when they are dependent only on matched state-variables, and syntactically matched.

To illustrate the applicability of the method, a communication-based case-study is introduced. The matching rules have been applied in this case study showing the practicality of these rules.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the financial assistance from the Malaysian Technical Cooperation Program (MTCP) fund from the Malaysian government reference number KPT.B.600-5/3 JILID 3, and Fundamental

Research Grant Scheme, Project Code: 08-02-13-1368FR.

REFERENCES.

[1] T. Stahl, M. Voelter, and K. Czarnecki, Model-driven software development: technology, engineering, management. John Wiley & Sons, 2006.
[2] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, IEEE Computer Society, 2009, pp. 1–6.
[3] J.-R. Abrial, Modeling in Event-B: system and software engineering. Cambridge University Press, 2010.
[4] J.-R. Abrial and J.-R. Abrial, The B-Book: Assigning programs to meanings. Cambridge University Press, 2005.
[5] R.-J. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," Distributed Computing, vol. 3, no. 2, 1989, pp. 73–87.
[6] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in event-b," International journal on software tools for technology transfer, vol. 12, no. 6, 2010 pp,. 447–466.
[7] Object Management Group, "Unified modeling language UML," 2017.
[8] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no. 1, 2006, pp. 92–122.
[9] C. Snook and M. Butler, "UML-B and Event-B: an integration of languages and tools," 2008.
[10] C. Snook and M. Butler, "UML-B: A plug-in for the event-b tool set," 2008.
[11] C. Snook, I. Oliver, and M. Butler, The UML-B profile for formal systems modelling in UML, Springer, 2004, pp. 69–84.
[12] K. Lano, The B language and method: a guide to practical formal development. Springer Science & Business Media, 2012.
[13] J.-R. Abrial, "From Z to B and then Event-B: Assigning proofs to meaningful programs," in Integrated Formal Methods, Springer, 2013, pp. 1–15.
[14] T. S. Hoang, A. F¨urst, and J.-R. Abrial, "Event-b patterns and their tool support," Software & Systems Modeling, vol. 12, no. 2, 2013, pp. 229–244.
[15] J. Rumbaugh, I. Jacobson, and G. Booch, Unified Modeling Language Reference Manual, The. Pearson Higher Education, 2004.
[16] C. Snook and M. Butler, "U2B-a tool for translating UML-B models into B," 2004.
[17] F. Taibi, F. M. Abbou, and M. J. Alam, "A matching approach for object oriented formal specifications.," Journal of Object Technology, vol. 7, no. 8, 2008, pp. 139–153.
[18] J.-J. Jeng and B. H. Cheng, "Specification matching for software reuse: A foundation," in ACM SIGSOFT Software Engineering Notes, vol. 20, ACM, 1995, pp. 97–105.
[19] A. M. Zaremski and J. M. Wing, "Specification matching of software components," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 6, no. 4, 1997, pp. 333–369.
[20] F. Feiks and D. Hemer, "Specification matching of object-oriented components," in Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on, IEEE, 2003, pp. 182–190.
[21] D. Hemer, "Specification matching of state-based modular components,"in Software Engineering Conference, 2003. Tenth Asia-Pacific, IEEE, 2003, pp. 446–455.
[22] J. Becker, P. Delfmann, S. Herwig, and Ł. Lis, "A generic set theory based pattern matching approach for the analysis of conceptual models,"in International Conference on Conceptual Modeling, Springer, 2009, pp. 41–54.