A Model-Driven Approach to Adapt SysML Blocks

Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir

FEMTO-ST Institute, University of Franche-Comté, Besançon, France
{hamida.bouaziz,schouali,ahammad,hmountas}@femto-st.fr

Abstract. Reusing and adapting existing components is the central topic of component-based development. The major differences between the existing approaches concern the models used to represent the components and the detail given to generate the adapters. In this paper, we present our approach which bases on the hierarchy to generate the adapters. Our components are modelled using SysML blocks and their interaction protocols are modelled using SysML Sequence Diagrams (SDs). We have used coloured Petri nets as formal model to define the adaptation rules, and we have based on meta-modelling and model transformation to implement these rules. We illustrate our approach through a case study.

Keywords: SysML blocks, adaptation rules, meta-modelling, model transformation

1 Introduction

When assembling components developed separately, there is a high probability to confront the problem of mismatches. These mismatches can concern for example the name of services, as well as the order in which the component asks (resp. offers) for environment services (resp. its services). That is what justifies the introduction of third entities or components which are used to solve these mismatches. This kind of components are called *'adapters'*. A big part of the works done in this field start from a formal specification of these components, which makes difficult the communication between the various stakeholder in CBD (Component-Based Development) projects. To tackle this problem and to make the communication easier, system engineering community proposes to use high level languages. This appears clearly through *SysML* [1], a language which is adopted by OMG, it is used to design systems that include software and hardware.

The System Modelling Language (SysML), through its diagrams, fosters the view point that takes the system as a set of components. In SysML, we call them blocks. A block is a modular unit of the system description. It may include both structural and behavioural features, such as properties and operations. To communicate with its environment, a block has a list of ports. These ports are characterised by interfaces that present the offered and required services of the

block. SysML also offers many diagrams to represent the structure, the behaviour and requirements of the blocks.

The privilege given to SysML doesn't mean that it will take the place of formal methods. But it replaces them at a level of system representation, where we need a high level specification of the system. We must mention that SysML lacks formal semantics which makes very interesting the introduction of formal methods in component adaptation domain to compute the adapters and their behaviour semantics. A combination of SysML and formal methods in the same approach is the solution that will tackle the lack of each of them. That is what Model Driven Engineering (MDE) tries to do through the introduction of model transformation approaches. In this paper, we propose to use SysML sequence diagrams to schedule the interactions of each block with its environment and we propose a transformation process to generate their equivalents of coloured Petri nets which are formal models that we consider more suitable for defining and generating adapters for blocks.

The major difference between the existing adaptation approaches concerns the detail given to generate the adapter. In [2], the authors give only an adaptation contract which is resumed in a specification of the correspondences between blocks services. This will have an impact on the generation of the adapter. The adapter will contain all the possible interaction scenarios of executing the reused components, it can contain scenarios which are not necessary for the cooperation of the reused components. However, in [3–5], the authors have enriched their adaptation contract by a specification of the adapter interactions by ordering the vectors of the adaptation contract using regular expressions. This requires that the developer, before making the specification of the adapter, must know very well the interactions of each component with its environment, and he must have an idea about the synchronous execution of the reused components. In this context, we ask the question about the detail that will be enough to generate adapters to make a set of components cooperate with respect of the intention behind their assembling?

In this context, we propose an incremental approach to develop systems by reusing and adapting components modelled using SysML, we base on coloured Petri nets as formal models to compute the adapter interactions. In our process, we do not give only the mapping rules between services like in [2], and we do not give the specification of the adapter as in the works already done in [3–5]. But, we give the interaction protocol of the composite block which will include the reused blocks. The specification of the composite block is built by the architect according to the interaction protocol of the system's part has already been designed, and in function of what the current composite block must perform to the system's part still to develop. The major difference comparing with our previous work [6], is that in this paper, we solve more problems such as the reordering of services calls, and we consider more types of correspondences between services.

In the remainder of this paper, in Section 2, we show our approach to adapt SysML blocks, in Section 3, we present the meta models of sequence diagrams, coloured Petri nets and adaptation contracts. In Section 4, we explain the rules

to generate the adapter. In Section 5, we demonstrate our approach through a case study. Finally, in Section 6, we conclude and we give perspectives.

2 Our Approach

The goal of our approach consists on generating the adapter Ad for a set of blocks $\{B_i\}$ to meet the specification of their future parent block B (a composite block). In the context of our incremental approach, in the next step of adaptation, this block B will be adapted with other blocks to meet the specification of its parent. The interaction of each block with its environment is modelled using a SysML sequence diagram. Because sequence diagrams are not formal models, we have performed a transformation to obtain their equivalents of Coloured Petri Nets (CPNs). The formal semantic of CPNs allows us to reflect easily the adaptation contract on the interactions of the blocks to compute the interactions of the adapter.

Our approach of adaptation is based on meta-modelling and models transformation, where we base on *'ecore'* models to represent the meta-models of formalisms that we use (Sequence diagrams, coloured Petri nets). The transformation is performed through the use of *ATL* [7] rules that allow to define the correspondences between source and target meta-models. To generate the adapter, we have introduced the meta-model of the adaptation contract, and we have based also on ATL rules to reflect the information present at the level of the contract model onto the coloured Petri nets of the reused blocks.

3 Meta-Modelling

3.1 Meta-Model of Sequence Diagrams

By intention to reuse existing modelling tools, we have used the sub-set of Papyrus [8] SysML meta-model and its graphical editor to draw the sequence diagrams. In Figure 1, we represent the meta-model of sequence diagrams. The root is the class *Interaction*. So, each sequence diagrams is an instance of this class. Each interaction can include a set of life lines, a set of messages and a set of interaction fragments. The classes:

- **LifeLine:** represents the set of object which participates in the interaction.
- **Message:** each message has two ends; a send end and a receive end.
- **InteractionFragment:** is the super class of the classes: Interaction, CombinedFragment, InteractionOperand and OccurenceSpecification.
- **CombinedFragment:** each combined fragment includes a set of interaction operands, and it has its own interaction operator. The interaction operator takes a value of this list [alt, opt, break, loop, par, ...]
- **InteractionOperand:** each operand is associated to a combined fragment, and it can have a guard.

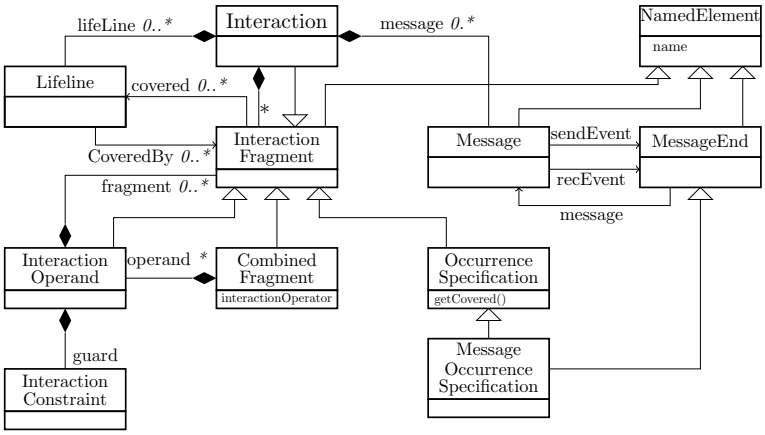


Fig. 1: Papyrus Meta-Model of SysML Sequence Diagram

– **MessageOccurenceSpecification:** Each event associated to the life line is represented as a message occurrence specification. It represents an extremity of a message. We can know the life line, to which the specification is associated, by executing the method *getCovered()* of the super class *OccurenceSpecification*. We can also obtain the message started or finished at this specification, by navigating through the association *message* of the super class *MessageEnd*.
The classes *MessageEnd*, *Message* and *InteractionFragment* inherit the class *NamedElement*.

3.2 Meta-model of Coloured Petri Nets

Coloured Petri Nets (CPNs) preserve useful properties of Petri nets, and at the same time extend initial formalism to allow the distinction between tokens [9]. In CPNs, a token has a data value attached to it. This attached data value is called token colour. The Meta-model of coloured Petri nets is presented in Figure 2. It contains the following classes:

– CPN: represents the root class, each instance of this class will include a set of places, a set of transitions, a set of input arcs (according to the transitions) and finally a set of output arcs (always according to the transitions).
– Place: each place has a name, a *colorSet* attribute which is a list that contains the set of colours of tokens that can be stored in this place. The attribute *nbrTokenPerColor* specifies the number of tokens of each colour.
– Transition: each instance of this class has a name that represents the action done at this stage of behaviour evolution. A transition is fired if the colour and the number of tokens mentioned at the level of each input arc is available in the place situated at the origin of this arc.

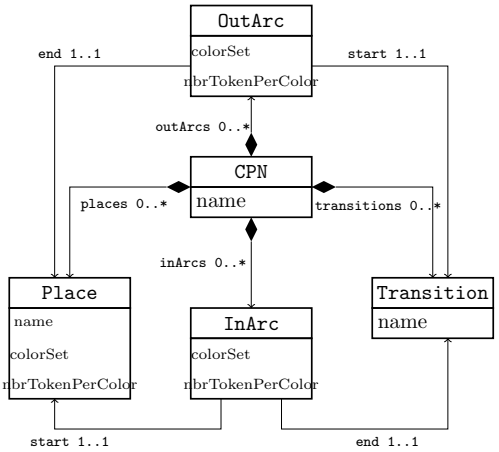


Fig. 2: Meta-Model of coloured Petri nets

- InArc: this class represents the set of arcs that starts at places and ends at transitions. To represent the colour of tokens consumed from the source place to execute the transition located at the end of a given arc, we have defined the attribute *colorSet*. However, to specify the number of tokens of each colour $\in$ *ColorSet*, we have used the attribute *nbrTokenPerColor*.
- OutArc: this class represents the set of arcs that starts at transitions and ends at places. The colours of tokens produced by executing the source transition of a given arc are recorded into the list *ColorSet*. To specify the number of produced tokens of each colour, we use the list *nbrTokenPerColor*.

3.3 Meta-Model of Adaptation Contracts

Our adaptation contract specifies the correspondences between blocks services. Its meta-model (see Figure 3) contains the following classes:

- AdaptationContract: represents the root class. Each instance of this class includes a set of blocks and a set of vectors.
- Block: this class represents all the blocks which are concerned by the adaptation (Child blocks). It includes also the abstract block (Parent block) that represents the container of the reused blocks. Because we are interested by the correspondences between blocks services, we reduce the block features to the set of its required and provided services.
- Service: each service has a name and a type. The attribute *type* can take the value *Prov* or the value *Req*.
- Vector: each mapping vector establishes a link between the services of two blocks. According to its type, a vector can belong to one of these two subsets:
  - **a mapping vector between two reused blocks:** *OneReq2OneProv* (One required service to one provided service), *OneProv2ManyReq* (One

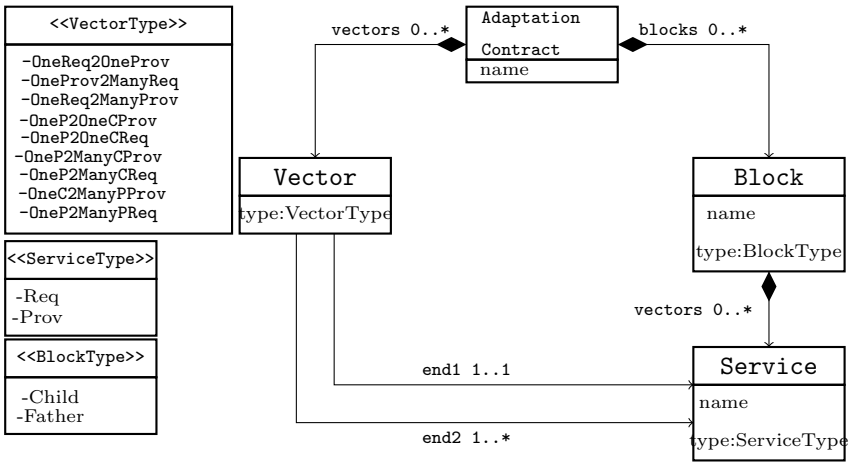


Fig. 3: Meta-Model of adaptation contracts

provided service to many required services), *OneReq2ManyProv* (One required service to many provided services).

- **a mapping vector between a reused block and its parent:** *OneP2OneCProv* (One provided service of the parent to one provided service of a child), *OneP2OneCReq* (One required service of the parent to one required service of a child), *OneP2ManyCProv* (One provided service of the parent to many provided services of a child), ...

So end1 represents the first extremity of the vector and end2 contains the services that correspond to the service end1.

To verify the contract validity, we have defined a set of OCL constraints. In the following, we give an example of an OCL constraint used in the class *Vector* to verify the extremities of a vector of type '*OneReq2OneProv*'.

```
/*cotraints in the class Vector */
invariant verifysizeOfend2andTypeOfServicesOfAVectorOneReq2OneProv:
self.type=VectorType::OneReq2OneProv implies self.end1.type=TypeService::Req
and self.end2->size()=1 and self.end2->first().type=TypeService::Prov ;
```

4 Transformation from SD into CPN

In Figure 4, we present the correspondences between sequence diagrams and coloured Petri nets (*!* represents a request of a service, and *?* represents a reception of this request).

To implement these correspondences, we have based on ATL rules that takes the meta-model of sequence diagrams as source of transformation and the meta-model of coloured Petri net as target. In the following we give the rules to transform the basic interaction of sequence diagrams into coloured Petri nets.

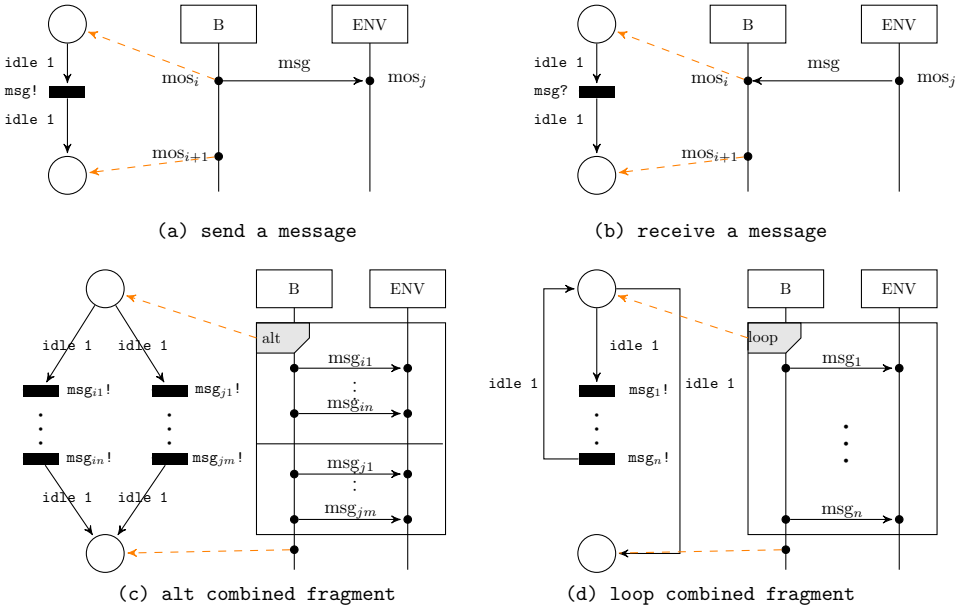


Fig. 4: Transformation SD → CPN

The first rule *createCPN* allows to generate the CPN, the second rule *createStates* allows to create the states of the CPN, where the place associated to the first event on the life line of the block must contain a token *idle* which triggers the execution of the block interactions. Finally, the third rule *createTransitions* allows to create the transitions and the arcs that link the places and transitions.

```
rule createCPN {
from sd : SD!Interaction
to  cpn : CPN!CPN ( name <- 'CPNBlocks',
    places <- CPN!Place.allInstances(), transitions <- CPN!Transition.allInstances(),
    inArcs <- CPN!InArc.allInstances(), outArcs <- CPN!OutArc.allInstances() )}
```

```
rule createStates {
from mos : SD!MessageOccurrenceSpecification
((mos.covered->first().name <> 'ENV') and (mos.MoshavePlace()))
to  place : CPN!Place (
    colorSet <-if (mos.previousMos()=mos) then Sequence{'idle'} else Sequence{} endif,
    nbrTokenPerColor <- if (mos.previousMos()=mos) then Sequence{1} else Sequence{} endif,
    name <- 'p'.concat(mos.covered->first().name).concat(mos.getCovered().coveredBy->
            select(e|e.oclIsTypeOf(SD!MessageOccurrenceSpecification))->indexOf(mos)) )}
```

```
rule createTransitions {
from mes:SD!Message , mos:SD! MessageOccurrenceSpecification
((mos.covered->first().name <> 'ENV') and (mos.message=mes) and (mos.MoshavePlace()))
to
t : CPN!Transition (name<- mes.name.concat(if mes.sendEvent=mos then '!' else '?' endif)),
inArc : CPN!InArc(
    start <- thisModule.resolveTemp(mos, 'place'), end <- t,
    colorSet <-Sequence{'idle'}, nbrTokenPerColor <- Sequence{1} ),
outArc : CPN!OutArc(
    start <- t, end <- mos.nextplace(mos.covered->first()),
    colorSet <-Sequence{'idle'}, nbrTokenPerColor <- Sequence{1} )}
```

- `MoshavePlace()`: is a helper that takes as context a message occurrence specification, and it returns the value *true* if we must associate a place to the current message occurrence specification.
- `previousMos()`: is a helper that takes as context a message occurrence specification, and it returns the previous message occurrence specification.
- `nextplace(ln)`: is the helper that takes as context a message occurrence specification, and it returns the place associated to the next message occurrence specification on the life line *ln*.

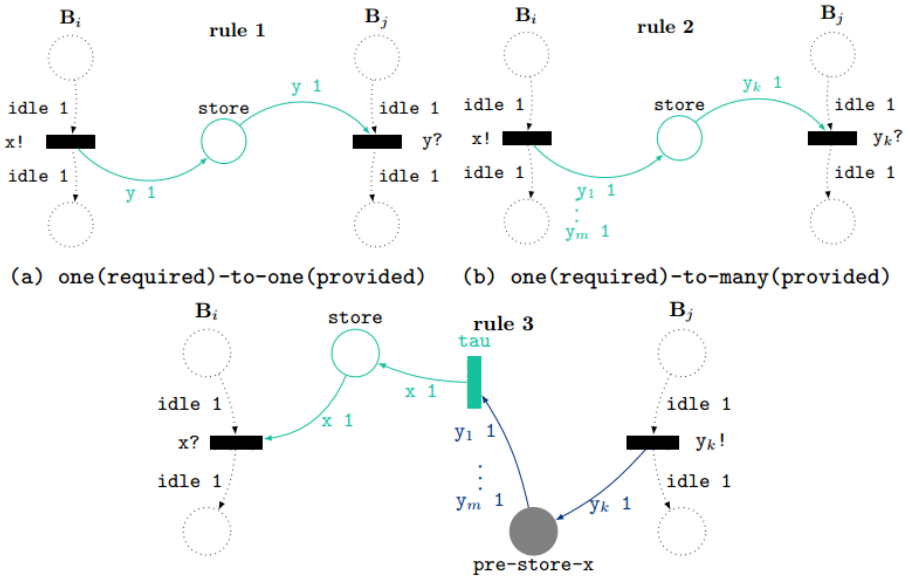
5 Generating the Adaptor

To generate the adaptor, firstly we need to compute the global interaction of the reused blocks by gluing their CPNs according to the adaptation sub-contract that specifies the correspondences between the reused blocks. Thus, the CPNs of the reused blocks are glued using the *store* place and a set of transitions (which translate the adaptation contract). The store place will store the calls for services until the targeted blocks can receive these requests. In the following, through these rules, we explain how we glue them.

Rule 1: one-required-to-one-provided. This rule (presented in Figure 5(a)) is applicable on vectors of type *OneReq2OneProv*: ⟨ e_i, e_j ⟩, where e_i =Bi:x! and e_j =Bj:{y?}. It specifies that the required service x of block B_i corresponds to the provided service y of block B_j. In this case, when the block B_i executes the transition *x!*, it generates the corresponding action y as a token, which will be consumed later by the block B_j, when it tries to execute the action *y?*.

Rule 2: one-required-to-many-provided. This rule (presented in Figure 5(b)) is applicable on vectors of type *OneReq2ManyProv*: ⟨ e_i, e_j ⟩, where e_i = Bi:x! and e_j = Bj:{y_1?,...,y_m?}. It specifies that the required service x of block B_i corresponds to the provided services y_1,...,y_m of block B_j. In this case, when the block B_i executes the transition *x!*, it generates the corresponding actions y_1,...,y_m as tokens, which will be consumed later by the block B_j when it tries to execute an action $y_k \in \{y_1,...,y_m\}$.

Rule 3: one-provided-to-many-required. It means that the block B_i can execute the service mentioned in '*one(provided)*' only after when the block B_j



(a) one(required)-to-one(provided) (b) one(required)-to-many(provided)

(c) one(provided)-to-many(required)

Fig. 5: Rules for synthesizing the reused blocks

sends requests for all services specified in '*many(required)*'. This correspondence can be specified at the level of the adaptation contract by a vector: $\langle$ `e`$_i$`,` `e`$_j$ $\rangle$`, where e`$_i$ `= Bi:x? and e`$_j$ `= Bj:{`y_1`!,...,`y_m`!}`. To represent this vector using CPNs, we apply the rule 3 (see Figure 5(c)). So, we create a place '*pre-store-x*'. This place stores calls for the services that correspond to x. Then, we link all transitions labelled by $y_k!_{k=1..m}$ to the place '*pre-store-x*'. After, we must create a transition *tau* that has an incoming arc which starts from the place '*pre-store-x*'. This arc is labelled by `[`y_k `1]`$_{k=1..m}$. We must also create an arc which starts from transition *tau* and ends at the *store* place, this arc must be labelled by '*x 1*'. Finally, to allow to the block B_i to execute the service x, we link the *store* state with the transition *x?*.

The ATL grammar that implements these adaptation rules takes as entry the meta-model of CPNs and the meta-model of the adaptation contract, and it generates one CPN that represents the global interaction of the reused blocks. The grammar, at the first step, creates the new CPN and copies the places, the transitions, the in-arcs, the out-arcs, and creates the *store* place.

```
rule createStorePlace {
from contract : C!AdaptationContract
to GIR : outPN!CPN ( name <- 'GIR',
    places <- outPN!Place.allInstances(), transitions <- outPN!Transition.allInstances(),
```

```
    inArcs <- outPN!InArc.allInstances(),    outArcs <- outPN!OutArc.allInstances() ),
store:outPN!Place( name <- 'store' ) }
```

In the following we present the ATL rules that allows to reflect the vector of type *one required service to one provided service*. The first one '*oneReq2oneProv1*' creates the incoming arc to the store place (the request of the service), However, the second rule '*oneReq2oneProv2*' creates the arc which starts from the store place to allow the reception of the request by the concerned block.

```
rule oneReq2oneProv1{
from contract:C!AdaptationContract, v : C!Vector, t:InPN!Transition
((v.type=#OneReq2OneProv) and (v.end1.name.concat('!')=t.name) )
to arc : outPN!OutArc (
    start <- t, end <- thisModule.resolveTemp(contract, 'store'),
    colorSet <- Sequence{v.end2->first().name}, nbrTokenPerColor <- Sequence{1} ) }
```

```
 rule oneReq2oneProv2 {
from contract:C!AdaptationContract, v : C!Vector, t:InPN!Transition
((v.type=#OneReq2OneProv) and (v.end2->first().name.concat('?')=t.name) )
to arc : outPN!InArc(
    start <- thisModule.resolveTemp(contract, 'store'), end <- t,
    colorSet <- Sequence{v.end2->first().name}, nbrTokenPerColor <- Sequence{1} ) }
```

To represent the exchange of requests between the blocks and their environment, we base on the mapping vectors that link the reused blocks and their parent block (vectors that represent the delegation relation between the parent and their children blocks), we have defined the correspondences using CPNs and we have implemented them using ATL. But due to the lack of space, we cannot present them in this paper.

Basing on the CPN that represents the interaction of the reused blocks according to the specification of their parent and the mapping between their services, we can generate the adapter. The adapter will play the role of a mirror between the reused sub-blocks $\{B_i\}$. So each call for a service by a sub-block B_i must be intercepted by the adapter, and each reception of a request for a service by a sub-block B_i must be preceded by a call for this service, this call must be emitted by the adapter. Thus, to generate the adapter, we base on the CPN synthesized in the last phase. We take this CPN, and we apply the mirror function on some transitions, we transform each call for a service $x!$ by a reused block B_i into a reception of this call, and each reception of a call for a service $x?$ by a reused block B_i must be transformed to an emission of this call $x!$. Therefore this transformation concerns only the transitions of the reused blocks, because the adapter plays the role of mirror only between the reused sub-blocks. Concerning the relation between the adapter and the parent block, it is a delegation relation. So, the adapter delegates the parent to interact with the environment,

that is why we do not need to inverse the actions done at the level of the parent transitions.

6 Case Study

We give an example of a simple robot which can receive a request to move with a given speed from a station, and it stops after reaching a goal. We consider that the robot that we want to construct and to integrate to our system, will have the interaction protocol given in Figure 6. To build the robot, we have reused two blocks *'Controller'* and *'MovingSystem'*, their interaction protocols modelled using SysML sequence diagrams in Figure 6. To simplify we consider that the corresponding actions have the same name and we differentiate between them by adding the first letter of the block's name to each action. To obtain the interaction protocol of the adapter, we base on the adaptation contract modelled using our graphical editor in Figure 7.

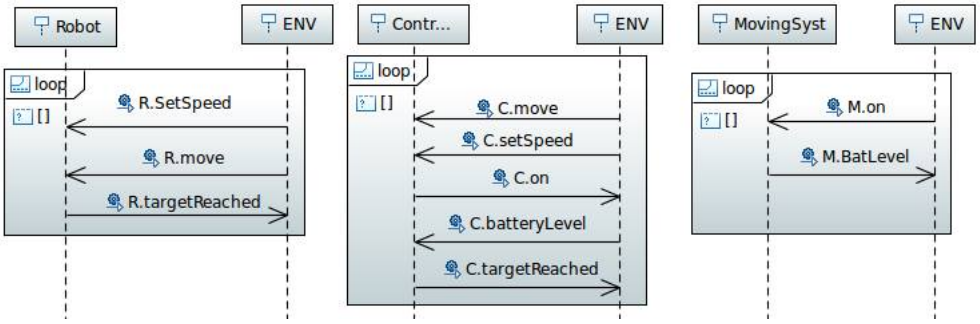


Fig. 6: Sequence diagrams of the robot, the controller and the moving system

The CPN of the adapter is presented in Figure 8. This adapter solves the problem of name mismatches between services of blocks, it restricts the interaction between blocks to the specification of their parent (the robot block) and reorders the call for services (*setSpeed* and *move*). To obtain the interaction protocol of the adapter, we need just to compute the reachability graph of its CPN using for example CPNtool [10]. In Figure 9, we present the new internal structure of the robot block.

7 Conclusion

In this paper, we have presented a bottom-up model-driven approach to adapt SysML blocks. Our adaptation process takes a part of the system to develop, and generates an adapter for the SysML blocks which are reused to meet the

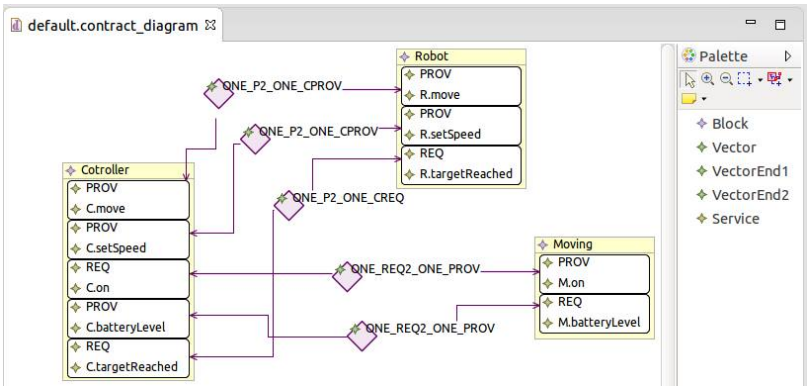


Fig. 7: The adaptation contract C

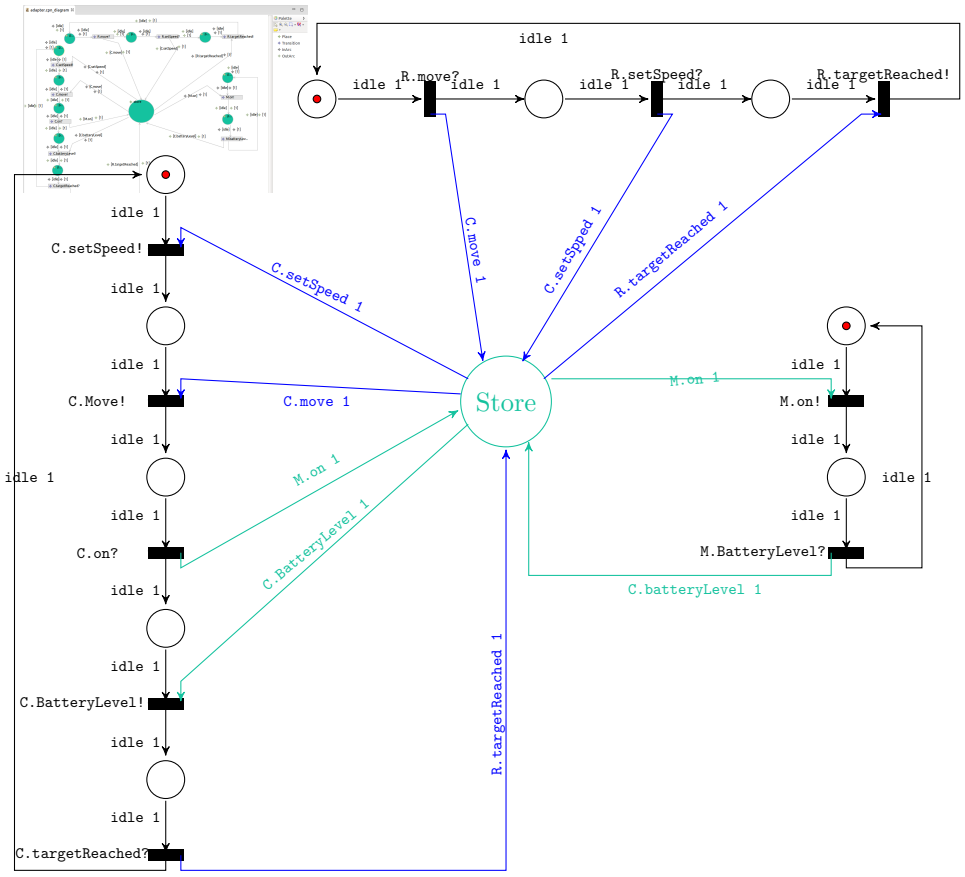


Fig. 8: CPN$_{adapter}$

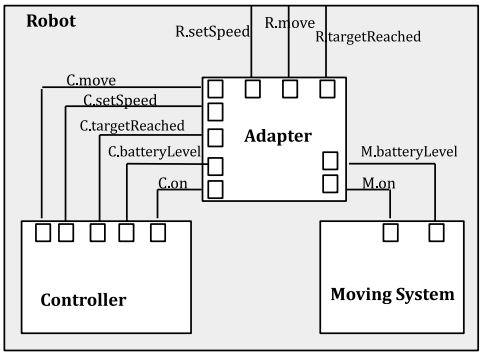


Fig. 9: Internal block diagram of the robot

specification of this part. We have based on SDs of SysML to model the interactions of each block. Due to the informal aspect of SysML, we have proposed to transform the SDs of blocks into CPNs, and we have implemented this transformation using ATL language. Now, we are working on the Acceleo templates that take our adapter and generate the input file for CPN tool, to discharge the user from redrawing the CPN of adapter to compute its reachability graph. In a future work, we plan to deal with the asynchronous aspect in the adaptation context.

References

1. OMG, "OMG Systems Modeling Language (OMG SysML) Version 1.3," 2012. [Online]. Available: http://www.omg.org
2. D. Dahmani, M. C. Boukala, and H. Mountassir, "A petri net approach for reusing and adapting components with atomic and non-atomic synchronisation," in *PNSE, Tunis, Tunisia*, 2014, pp. 129–141.
3. C. Canal, P. Poizat, and G. Salaün, "Adaptation de composants logiciels une approche automatisée basée sur des expressions régulières de vecteurs de synchronisation," in *CAL*, 2006, pp. 31–39.
4. C. Canal, P. Poizat, and G. Salaun, "Model-based adaptation of behavioral mismatching components," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 546–563, July 2008.
5. R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 755–777, 2012.
6. H. Bouaziz, S. Chouali, A. Hammad, and H. Mountassir, "Sysml blocks adaptation," in *ICFEM, Paris, France*, 2015, pp. 417–433.
7. ATL, "ATL: Atlas Transformation Language." [Online]. Available: https://eclipse.org/atl/
8. "Papyrus." [Online]. Available: https://eclipse.org/papyrus/
9. K. Jensen, *Coloured Petri Nets (2Nd Ed.): Basic Concepts, Analysis Methods and Practical Use: Volume 1.* London, UK, UK: Springer-Verlag, 1996.
10. "Cpn tool." [Online]. Available: http://cpntools.org/