Transformation of UML architectures into BIP language models.

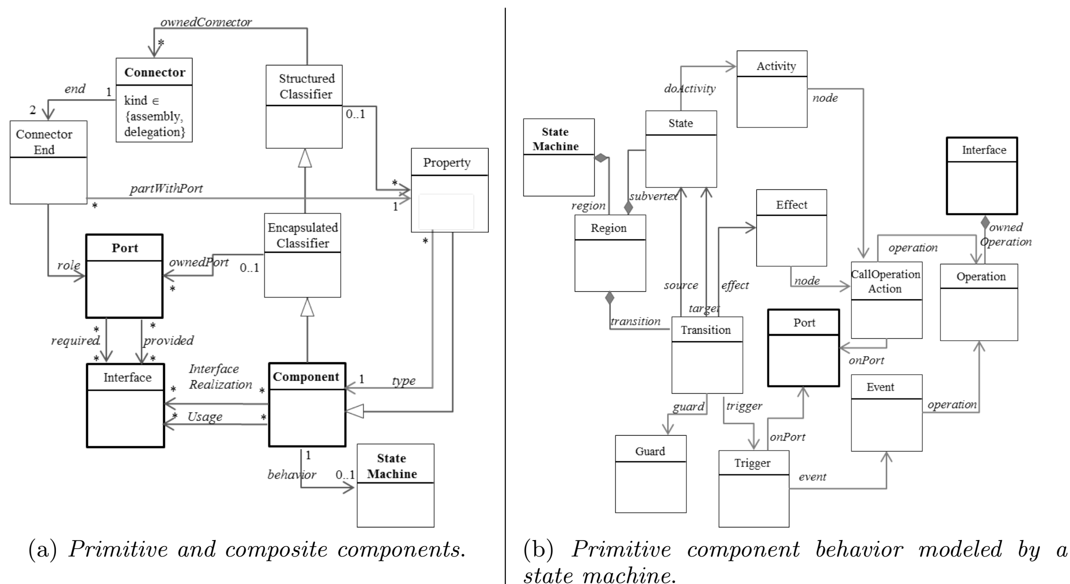
Anne-Lise Courbis[1], Thomas Lambolais[1], and Thanh Hung Nguyen[2]

[1] École des mines d'Alès, 30 035 Nîmes, France,
anne-lise.courbis@mines-ales.fr, thomas.lambolais@mines-ales.fr,
[2] Hanoi University of Science and Techniques, Hanoi, Vietnam
hungnt@soict.hust.edu.vn

Abstract. This document highlights UML concepts that must be used to model architectures in order to match some expected qualities such as loose coupling, design for change and high cohesion. It points out how such models can be transformed into BIP language used to model reactive systems to perform a safety analysis.
Download UMLtoBIP tool: https://idcm.wp.imt.fr/idcm-tool

1 Applying architectural principles in UML

Expected architectural model qualities are loose coupling, design for change and high cohesion [1, 2]. A way to reach these qualities is to apply principles of hierarchy, abstraction, information hiding and separation of concerns ([2], p.111). We are thus concerned by fundamental modeling artifacts of *encapsulation* and *interface specification* which will be implemented in UML through the concepts of *Component*, *Interface* and *Port* (Fig. 1a).



(a) *Primitive and composite components.*

(b) *Primitive component behavior modeled by a state machine.*

Fig. 1: Extract of UML meta-model classes for architectures and their behaviors.

1.1 Modeling primitive UML components

We associate with every primitive component a behavioral specification of the expected interactions of the component with its environment. Our proposal is to enable the designer to specify incomplete and abstract (non deterministic) models, in particular at the beginning

of the design. In UML 2.4, a behavioral specification is a *BehavioredClassifier* that can be a *StateMachine*, an *Activity* or a *UseCase*. We limit the behavioral models of primitive components to *StateMachines* for two main reasons: they are appropriate to model concurrency, activities and events, which are crucial for reactive systems; they can be used at different abstraction levels from high level to more detailed specifications.

Fig. 1b depicts the main classes we have selected to model primitive component behaviors by state machines.

Triggers and effects are respectively modeled by *ReceiveOperationEvent* and *CallOperationAction* allowing their associated *Operation* to be received and called through a defined *Port*. A *State* may have an internal *Activity* modeled by a *CallOperationAction* defined on a *Port* in order to ensure the encapsulation feature. We limit the operations to synchronized ones.

1.2 Modeling composite components

Composite components represent architectures in terms of sub-component assembling. Like primitive components, they are *EncapsulatedClassifier* to which *Ports* and *Interfaces* are associated. The main features of *StructuredClassifier* will be used to define the structural features of composite component. Fig. 1a depicts the main UML classes we have selected to model composite components. The structural view associated with a composite component is defined by a set of *Parts* (instances of Components) whose binding is performed by *Connectors*. *ConnectorEnd* is restricted to *Port* in order to preserve the encapsulation and define the synchronization of sub-components in a proper way. A *SubComponent* can be primitive or, at its turn, composite. This division of components defines a tree whose leaves have to be primitive components in order to be able to deduce the behavior of the composite component.

Ports of *Subcomponents* can be interconnected if they share at least a common *Interface* that must be provided on one side of the *Connection* and required on the other side. Such a *Connection* is of type assembly. *Ports* of a composite component may be connected to *Ports* of its *Subcomponents* when they share a common *Interface* of the same type. Such a *Connection* is of type delegation. Delegation and assembly connectors have a separate transformation rule into BIP.

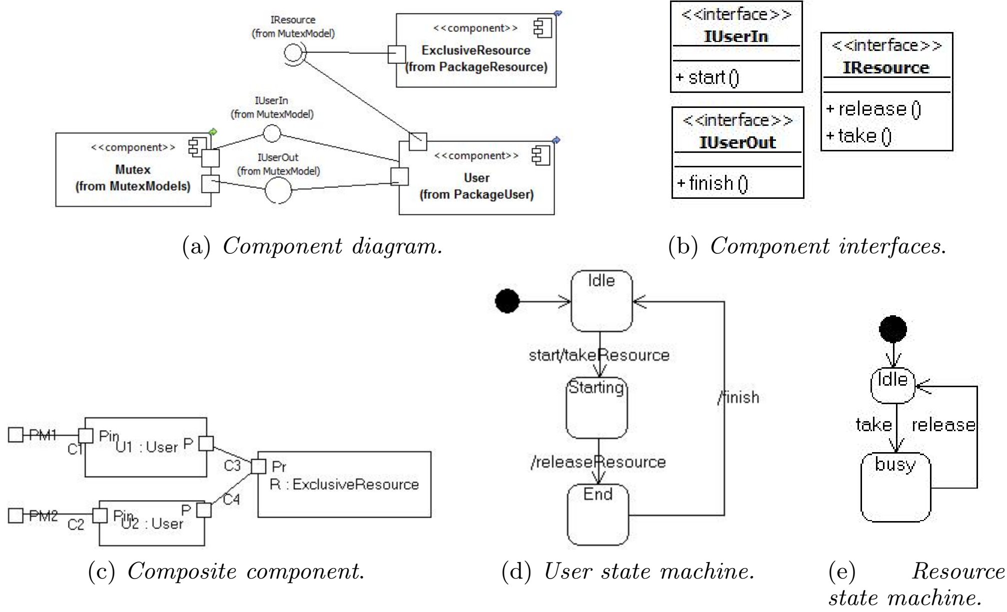
1.3 Illustration

We illustrate the aforementioned UML modeling concepts with a model describing a system named MUTEX. The MUTEX system represents the use of a resource by two users in a mutual exclusion mode. The component diagram (Fig. 2a) points out relations existing between the components implied in the MUTEX system model and their interfaces. Note that it is not an architectural description of the system. The architecture of the MUTEX system is defined by the MUTEX composite component in terms of three component assembly (Fig. 2c). The two sub-components of type *User* and *ExclusiveResource* involved in the architecture are associated with state machines (Fig. 2d and Fig. 2e).

The use of ports and their association with interfaces allow the complete encapsulation of the components. All triggers (resp. call operations) defined on the transitions of state machines are defined without any reference to the transmitter (resp. the provider). For instance, the call operation *takeResource* on the transition between state *Idle* and *Starting* on *User*state machine is specified by the operation *take* and the port *P*. The assembly connectors allow the components to be defined as transmitters and providers.

2 Transformation of UML architectures into BIP architectures

UML and BIP [3] share the same view about component concepts: a UML component whose behavior is defined by a state machine matches the BIP atomic component concept, a UML



(a) *Component diagram.* (b) *Component interfaces.*

(c) *Composite component.* (d) *User state machine.* (e) *Resource state machine.*

Fig. 2: MUTEX model.

composite component matches the BIP compound component. Concept of UML interface does not directly match a BIP concept, and reciprocally, the concept of BIP port is not the same as UML port. However, a matching is possible between the aggregation of UML ports and their corresponding operations defined in their associated interfaces and the BIP ports. For instance, the port *P* of component *User* which is associated with the interface *IResource* will correspond to two BIP ports named *P_TAKE* and *P_RELEASE*. Table 1 summarizes the link between UML and BIP concepts of components, interfaces and ports.

UML Concept	BIP Concept	card.
Component with a behavior	AtomicType	1
Component with parts	CompoundType	1
Port p	Port BIPp	*
for each op $\in$ p.RequiredInterface $\cup$ p.ProvidedInterface	BIPp.ident = p.ident + op.ident	1
Interface	$-$	0

Table 1: Correspondence between UML and BIP modeling concepts

2.1 Transformation of primitive UML components into BIP atomic components

We have given in [4] a LTS semantics to UML state machines. It is important to note that UML is more concrete and precise than LTS language. Hence, this semantics abstracts some concepts of UML: data, events, time, ports, guards, thanks to the intrinsic non-deterministic nature of LTS. The rules allowing the UML atomic transformation into LTS are detailed in [4]. UML triggers and operations are both abstracted by the single notion of LTS actions. UML ports carrying triggers and operations are translated into prefix in the label of LTS actions.

BIP atomic models have a LTS semantics [5]. The mapping from UML to BIP is thus trivial. Table 2 summarizes the matching between concepts of UML atomic components and

BIP components. For reasons of efficiency, the transformation from UML state machine to BIP is realized in two steps: the LTS generation detailed in [4] and the translation of the LTS into a BIP component. Note that transformation of states and transitions from UML into LTS is not a one-to-one function.

UML Concept	BIP/LTS Concept	card.
State	State	1..2
s such that t.source = s and t.effect = null or t.trigger = null	BIPs.ident = UMLs.ident	1
s such that t.source = s and t.effect ≠ null and t.trigger ≠ null	BIPs1.ident = s.ident BIPs2.ident = s.ident + "_0"	2
Transition	Transition	1..2
Event ev	PortExpression = ev.trigger.port.ident + "_" + ev.trigger.operation.ident	1
CallOperationAction cop	PortExpression = cop.onPort.ident + "_" + cop.operation.ident	1

Table 2: Correspondence between concepts of UML state machines and BIP atomic components

2.2 Transformation of composite UML components into BIP compound components

There is a direct correspondence between composite component and BIP compound components. A UML composite component consists of a set of *Parts* which match BIP *Components*. A UML assembly *Connector* matches a set of *BIP connectors*. Indeed, a BIP connector is relative to the synchronization of a single operation shared by the two interconnected ports, while a UML connector is relative to the synchronization of the set of operations shared by the interconnected ports. The *Port* of a *Part* belonging to a delegate connector will be exported and renamed by the name of the port of the compound component. Table 3 summarizes the matching of concepts between UML composite components and BIP compound components.

UML Concept	BIP Concept	card.
Property	Component	1
Assembly connector c	Connector BIPc	*
for each op ∈ c.end[i].role.required.ownedOperation, $i \in \{0, 1\}$	BIPc.actualPort=c.end[i].role.ident + op.ident, *where* $i \in \{0, 1\}$	1
Delegate connector c of compound Comp	export Port BIPp1 is BIPp2	*
let i ∈ {0,1} ⌈ c.end[i].role ∈ Comp.OwnedPort for each op ∈ c.end[i].role.required. ownedOperation ∪ c.end[i].role.provided. ownedOperation	BIPp1.ident = c.end[i]. role.ident + op.ident BIPp2.ident = c.end[j].role.ident + op.ident *where* $j = i + 1 \mod 2$	1

Table 3: Correspondence between concepts of UML composite components and BIP compound components

2.3 Illustration

Let us consider the MUTEX system whose UML description is given in section 1.3. Table 4 gives the BIP code associated with the *User* and *Mutex* components in Fig. 2. Note that BIP ports are labeled by the prefix of the UML ports followed by the name of the UML operation as specified in table 1. Ports labeled by 'i' represent internal operations. The *User* code has three parts:

```
package User
atomic type User
  // 1. declaration of ports
  export port Port PIN_START
  export port Port P_TAKE
  export port Port P_RELEASE
  export port Port PIN_FINISH
  // 2. declaration of states
  port Port i
  place Pseudostate1 , Idle ,Idle_0 ,
        Starting , Starting_0 , End,
        End_0
  initial to Pseudostate1
  // 3. LTS description
  on i from Pseudostate1 to Idle
  on PIN_START from Idle to Idle_0
  on P_TAKE from Idle_0 to Starting
  on i from Starting to Starting_0
  on P_RELEASE from Starting_0 to End
  on i from End to End_0
  on PIN_FINISH from End_0 to Idle
end
end
```

```
model Mutex
include User.bip
include Resource.bip
connector type rendezvous(Port p1, Port p2)
        define [ p1 p2 ]
end

compound type MutexType
  // 1. declaration of sub-components
  component  User U1
  component  User U2
  component  ResourceExclusive R
  // 2. assembly connectors (synchronization)
  connector rendezvous
      C3_release(U1.P_RELEASE, R.PR_RELEASE)
  connector rendezvous
      C3_take(U1.P_TAKE, R.PR_TAKE)
  connector rendezvous
      C4_release(U2.P_RELEASE, R.PR_RELEASE)
  .../...
  // 3. external ports of the compound component
  export port Port PM1_FINISH is U1.PIN_FINISH
  export port Port PM1_START is U1.PIN_START
  .../...
end

component  MutexType Mutex
end
```

Table 4: User and Mutex BIP models.

1. User component port declaration (line 4 to 7),
2. State declaration (lines 9 to 12),
3. State Transition description (lines 14 to 20).

The intermediate state labeled *Idle_0* represents the UML state not explicitly defined between the *start* trigger and the *takeResource* effect. The *Mutex* model code has three parts:

1. Declarations of two sub-components of type *User* and one sub-component of type *ResourceExclusive* (lines 10 to 12).
2. Assembly connectors *C3* and *C4* are modeled by a set of BIP synchronizations (lines 15 to 19) pointing out synchronization of operations *take* and *release* belonging to the interface *IResource*.
3. BIP exported ports representing UML delegate connectors *C1* and *C2* (lines 22 and 23).

3 Conclusion

In previous works [6, 4, 7], we have presented how performing a liveness analysis on UML components for supporting incremental design of reactive systems. The transformation from UML to BIP allows the performing of safety analysis using D-Finder tool [8, 9].

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture: foundations, theory, and practice. Wiley Publishing (2009)
2. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture - A Comprehensive Framework and Guide for Practitioners. Springer (2011)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), Washington, DC, USA, IEEE Computer Society Washington 3–12
4. Lambolais, T., Courbis, A.L., Luong, H.V., Percebois, C.: Idf: A framework for the incremental development and conformance verification of uml active primitive components. Journal of Systems and Software **113** (2016) 275–295
5. Basu, A.: Component-based modeling of heterogeneous real-time systems in bip. PhD thesis, Université Joseph-Fourier-Grenoble I (2008)
6. Luong, H.V., Lambolais, T., Courbis, A.L.: Implementation of the Conformance Relation for Incremental Development of Behavioural Models. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). Volume 5301 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2008) 356–370
7. Lambolais, T., Courbis, A.L., Luong, H.V., Phan, T.L.: Designing and integrating complex systems: Be agile through liveness verification and abstraction. In: Complex Systems Design & Management, Springer (2016) 69–81
8. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: Formal Methods in Computer-Aided Design (FMCAD), 2010, IEEE (2010) 257–256
9. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: International Conference on Computer Aided Verification, Springer (2009) 614–619