Extending the UML with a Multicast Synchronisation

Bogumila Hnatkowska

Computer Science Department, Wroclaw University of Technology
Wroclaw, Poland

Zbigniew Huzar

Computer Science Department, Wroclaw University of Technology
Wroclaw, Poland

Abstract

There is many systems, which need synchronisation between their components. Typical examples are concurrent task synchronisation, or synchronisation of multimedia streams during their transmission and presentation.

UML (Unified Modelling Language) is a specification language elaborated for object-oriented modelling. The UML enables explicit specification of peer-to-peer synchronisation only. The specific feature of the UML is its extensibility, allowing adaptation of it to a given domain.

In the paper, a new mechanism for a specification of a multicast synchronisation is presented. First, we extend the UML metamodel by introducing new metaclasses. Next, we define a new stereotype *Synchroniser*. Synchronisers have instances called synchronisation points. Synchronisation points offer synchronisation services to objects that may use them. The paper describes informally semantics of synchronisation points and demonstrates their expressive power by analysis of three examples.

1 Introduction

The Unified Modelling Language (UML) is a language for modelling of complex systems [2,7]. It arose at the beginning of this decade with an attempt initiated by the *Object Management Group* to define a standard object-oriented software development methodology. The UML is more complete than other languages in its support for modelling complex systems. Particularly, it is well suited for modelling real-time, embedded systems [3]. It is also very flexible due to extension mechanisms included in the language. However, the language is still under development. In the result of improvement process, nearly every year a new version of the language has been issued. When comparing all versions of the UML (the last one from the end of 1998 [7]) with other specification languages, e.g. E-Lotos [4], Estelle, Lotos, and SDL [6], we have discovered that a group synchronisation mechanism does not exist in the UML.

To eliminate this drawback, we present a concept of multicast synchronisation and mechanisms of its representation in the UML. We follow rigorous but informal style of presentation that is used in the UML metamodel [7]. Next, we present three examples explaining our extension and illustrating its expressive power.

The introduced extension consists of two stages. In the first stage – Section 2 – three new meta-classes are introduced, and in the second stage – Section 3 – a new stereotype is defined. Next, in Section 4, three examples illustrating application of the proposed extension are presented. The final concluding remarks are included in Section 5.

2 The first stage of the UML extension

The first stage of our extension is more essential because it extends the standard metamodel of the language [7]. In this stage three new meta-classes, called *Synchronisation Action, Synchronisation Signature* and *Synchronisation Event*, are introduced. The new metaclasses are presented in Fig. 1. The figure contains only a fragment of original metamodel contained in the *Core* package, which explains a context of our extension. The figure exhibits also relations between existing and introduced elements.

An instance of the *Synchronisation Action* metaclass – *synchronisation action* – is a new kind of action, different from actions defined already in the standard UML. The synchronisation action is a noninterruplible computational procedure that results in a synchronisation of a group of objects. The synchronisation action generates a synchronisation event. The *Synchronisation action* metaclass inherits standard attributes (*recurrence*, *isAsynchronous*, *target*, and *script*) from the *Action* metaclass. The value of the *recurrence* attribute equal to 1 means that a given synchronisation action may be performed only ones. The *isAsynchronous* attribute is also a constant equal to false which means that dispatched events are synchronous. The *target* attribute may be valuated by any set of object

names. The value of the attribute is interpreted as the set of objects, which take part in a group synchronisation. An *ActionExpression* of the following form may valuate the *script* attribute:

object-set.synchronisation-event-name(*argument*$_{list}$)

where

object-set defines the set of objects, which take part in group synchronisation within the synchronisation action,

synchronisation-event-name is the name of synchronisation event generated during the action,

argument$_{list}$ is a list of values conveyed by the synchronisation event to all participants of group synchronisation.

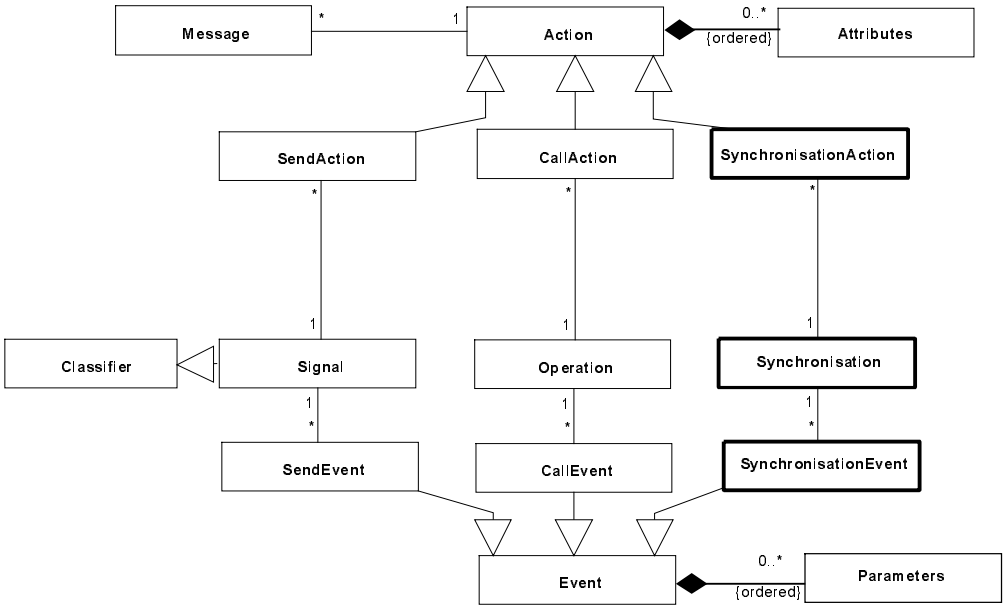

Figure 1: Extension of the UML metamodel

An instance of the *Synchronisation* metaclass is a *signature* of a synchronisation event; i.e. it contains the name and formal parameters of synchronisation event. Syntactically, it is the same as a signature of an operation.

An instance of the *Synchronisation Event* metaclass – *synchronisation event* – is a new kind of event, different from events already defined. The synchronisation event is a specification of a type of observable occurrences generated as a result of a synchronisation action. The synchronisation event may have value parameters. The event is visible to the set of objects defined within a given instance of synchronisation action. Realisation of the event requires a synchronous involvement of all objects from the defined set. The event conveys the same information to all objects in the synchronisation group.

One instance of synchronisation signature may be associated with any number of synchronisation actions, as well as with any number of instances of synchronisation events. One requires that scripts in associated synchronisation actions as well as associated synchronisation events must be consistent with the given signature, i.e. they have the same name of synchronisation event, and types of respective actual and formal parameters should be compatible.

Now, following the style used in official UML documents, we give more details concerning with the introduced metaclasses. The starting point is presented in Fig. 2, which presents the new metaclasses and their relationships to the *Class* metaclass that participate in a group synchronisation.

Synchronisation

Associations

synchronisationPoint	Indicates the class that instances raise a synchronisation event.
participant	Indicates the set of classes prepared to handle an event with a given signature.

Well-formedness rules

No extra well-formedness rules.

Semantics

A synchronisation instance is a signature of synchronisation events. It provides a name of a synchronisation event and a list of its formal parameters. An instance of a synchronisation event with a given signature may be generated by a synchronisation action with the same signature.

Instance notation: *synchronisation-event-name*(*formal-parameter*$_{list}$)

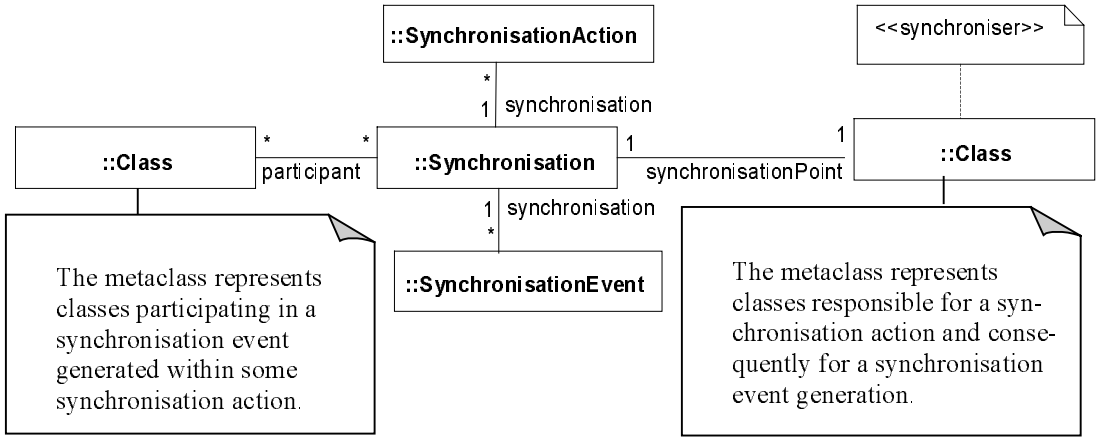

Figure 2: The context of introduced metaclasses

Synchronisation action

Attributes

Inherited from the *Action* metaclass.

Associations

Inherited from the *Action* metaclass.

synchronisation — Indicates a signature of a synchronisation event. It determines description of synchronisation actions.

Well-formedness rules

[1] *self.isAsynchronous* = false

[2] *self.recurrence* = 1

Semantics

A synchronisation action is the action that generates a synchronisation event, which results in a multicast synchronisation. The event is directed simultaneously to a set of receivers, which is dynamically determined when the synchronisation action is instantiated. The synchronisation action is defined by a set of objects that receive the generated synchronisation event, a name and value parameters of this synchronisation event.

Instance notation: *object-set.synchronisation-event-name*(*argumen*$_{tlist}$)

Synchronisation event

Associations

Inherited from the *Event* metaclass.

synchronisation — A signature of a synchronisation event. It determines the way of describing synchronisation actions.

Well-formedness rules

No extra well-formedness rules.

Semantics

A synchronisation event is a multicast synchronous stimulus communicated between objects. A synchronisation event represents the reception of a group synchronisation request. The expected result is a group syn-

chronisation of objects. Values associated with an event instance are conveyed to the participants of the synchronisation, which means that the values are available to all actions directly caused by that event in each participating object.

Instance notation: *synchronisation-event-name*($argument_{list}$)

3 The second stage of the UML extension

The second stage of our approach applies stereotypes as a standard UML extension mechanism. A new stereotype, called *Synchroniser*, is defined. A stereotype is a UML model element that is used to classify other UML elements so that they behave in some respect as if they were instances of new metamodel classes whose form is based on existing classes.

The *Synchroniser* is a specialised template class parameterised by a *TQueue* class. The *Synchroniser* class has an obligatory attribute *queue* – an instance of the *TQueue* class. The *queue* attribute has private visibility and it is used for gathering of synchronisation request calls. The *Synchroniser* class is shown in Fig. 4.

The *TQueue* class is also a template class parametrised by *TElem* class. The class and its instantiation are shown in Fig. 3. *TQueue* organises a queue of instances of *TElem* class and provides four operations. Three of them – *Add*, *Remove*, *Length* – have evident meaning, the fourth – *Target* – returns a set of names of objects gathered in the queue represented by a value of the *queue* attribute.

An actual parameter of *TQueue* instantiation must be any class with *object-name* attribute. In the paper we assume to use the *TStdElem* class or any child of it as an actual parameter of instantiation.

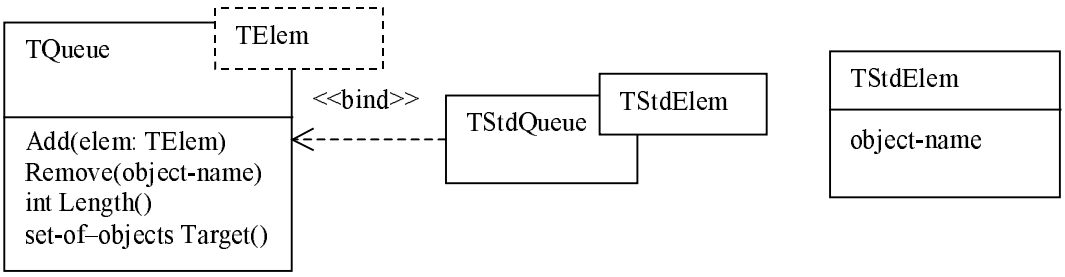

Figure 3: The *TQueue* class and its instantiation

The *Synchroniser* class has three additional compartments named *Synchronisation Request*, *Synchronisation Relinquish* and *Synchronisation*.

The *Synchronisation Request* compartment offers one public operation, which may be invoked by objects requesting synchronisation. The *Synchronisation Relinquish* compartment also offers one public operation, which may be used by objects withdrawing from waiting for synchronisation. The *Synchronisation* compartment contains a signature of a synchronisation event.

An instance of the *Synchroniser*, called a synchronisation point, is responsible for a realisation of a group synchronisation according to a given synchronisation strategy. A synchronisation point is created and destroyed by means of standard operations – *create* and *destroy* call by any object, or *terminate* call by the synchronisation point itself. The synchronisation point offers synchronisation services represented by operations contained in *Synchronisation Request* and *Synchronisation Relinquish* compartments.

Objects may use services of the given synchronisation point by calling these operations. The synchronisation request and release calls are done explicitly within the state machine associated with a given object. A state of the machine which has an exit transition labelled by a synchronisation event is called a *synchronisation state*. An object should call the synchronisation request operation before its entry to the synchronisation state or, at last, during staying in this state. A name of the calling object and actual parameters of the synchronisation request operation are located in the queue of the synchronisation point. The actual parameters of the call are passed explicitly from the object to the synchronisation point, while the name of the calling object is passed implicitly.

An object may retract waiting for synchronisation by calling synchronisation relinquishes operation. This call should be done at the moment when the object leaves its synchronisation state in a result of a transition triggered by an occurrence of some another event before an occurrence of the awaited synchronisation event. The result of the call

is removing the previous synchronisation request from the queue of the synchronisation point. The call has not explicit parameters; a name of the calling object is passed implicitly to the synchronisation point.

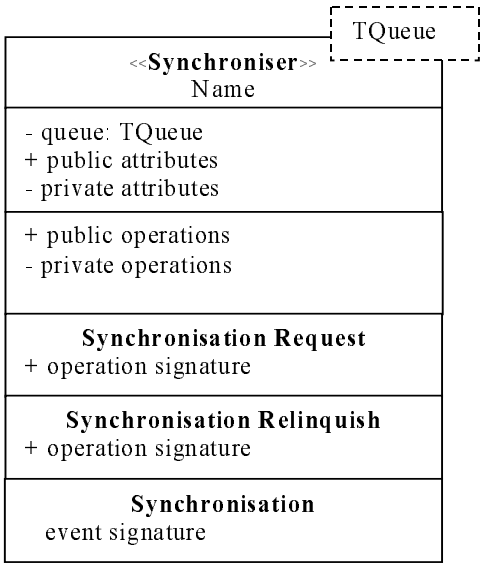

Figure 4: The <<Synchroniser>> stereotype

The problem of a consistent use of the request and release operations is left to an object design.

The static semantics of the *Synchroniser* demands that the actual instantiation parameter of the synchronisation point must be a concrete class which was instantiated with another class, which owns all attributes representing parameters of the synchronisation request operation and the attribute representing an object name.

As a synchronisation point is an instance of a specialised class, objects waiting for synchronisation in the queue of the synchronisation point may communicate with it by calling operations that are defined explicitly for a public use.

The state machine associated with a given synchronisation point has the structure presented in Fig. 5. The *Extended Class State Machine* contains one composite concurrent state. This state has two concurrent regions: *Queue Manipulation* and *Class State Machine*.

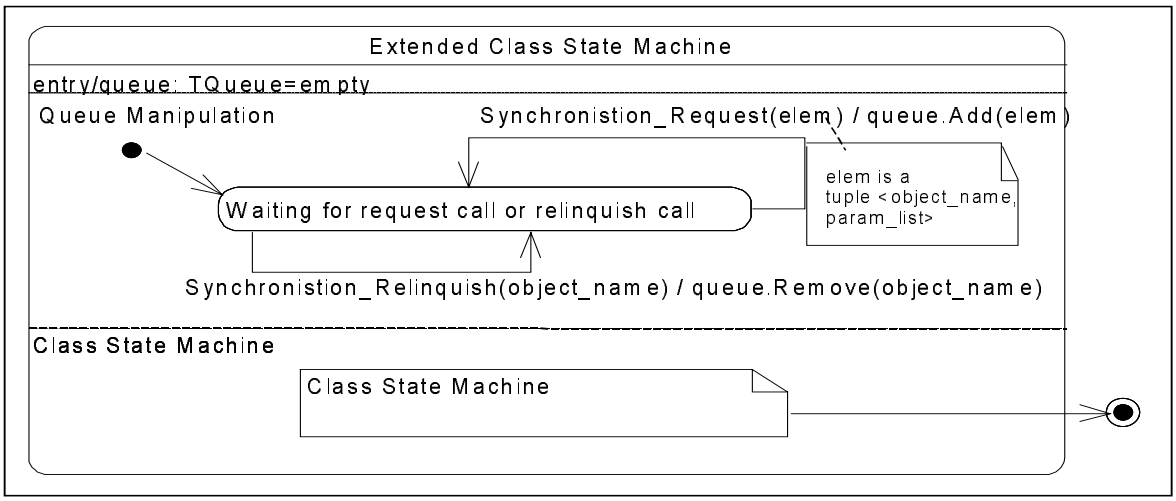

Figure 5: The state machine of a synchronisation point

The *Queue Manipulation* region represents a state machine, which is responsible for operations on the queue owned by the synchronisation point. The region has persistent structure parameterised by signatures of the triggering

events, i.e. *Synchronisation_Request* and *Synchronisation_Relinquish*.

The *Class State Machine* region depends on application and determines presumed strategy of a group synchronisation. The events of synchronisation request and synchronisation relinquish must not be triggering events in this region. The final state of *Class State Machine* region is also the final state of the whole state machine.

Further, to simplify notation the first region will be omitted in presented state diagrams.

4 Examples of group synchronisation

4.1 Safe opening

We model a situation, when to open a safe in a family bank a synchronisation of three persons – the mother, the father and the son – is necessary. Each person, after coming to the safe waits for some period of time for others. If the synchronisation took place, they open the door. In the opposite case each one goes to work, and after then tries to synchronise again.

In our model a person is represented by the *Person* class and the bank by the *Guard* class with «Synchroniser» stereotype. The Fig. 6 presents the class diagram where «synch» dependency relationship means that objects from the *Person* class may use services of synchronisation points which are instances of the *Guard* class.

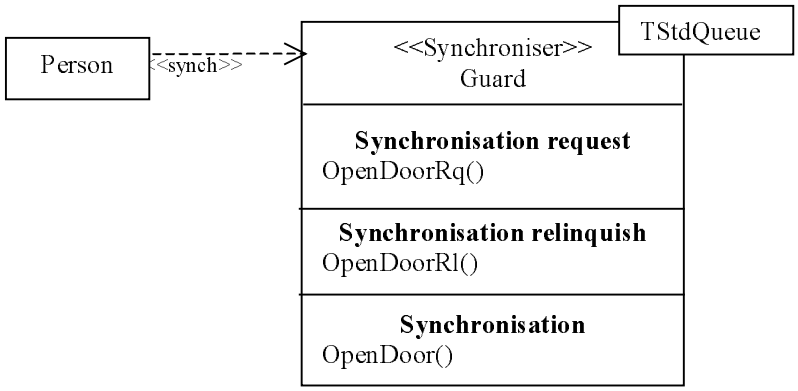

Figure 6: The class diagram of synchronised access to the safe

An example of a scenario of synchronised access to the safe is presented in Fig. 7.

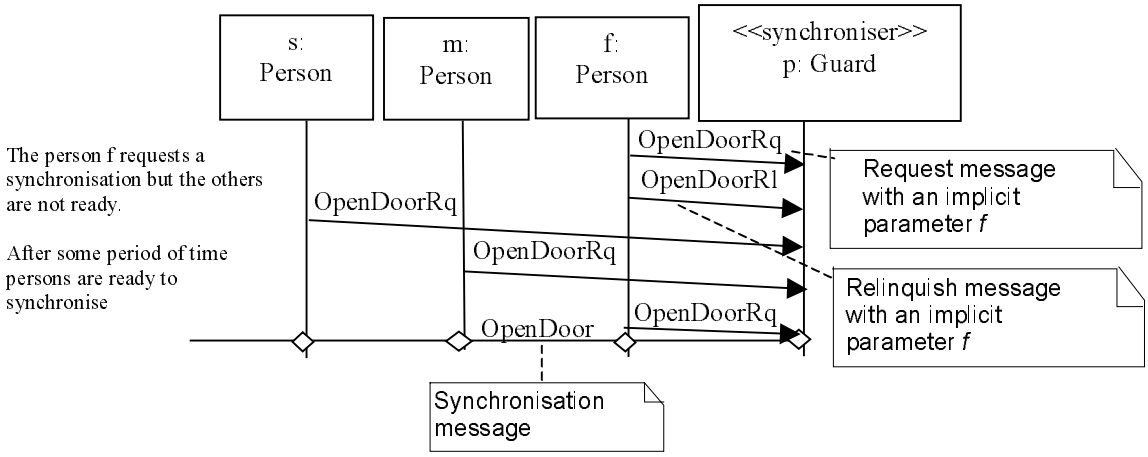

Figure 7: The sequence diagram of a synchronised access to the safe

Person behaviour is presented in Fig.8.

The *Guard* instance *p* is responsible for checking of the synchronisation condition The synchronisation

strategy in this example is very simple and is expressed by the condition: *queue.Target*() = {*f,m,s*} which plays the role of the guard in the transition label (Fig. 9).

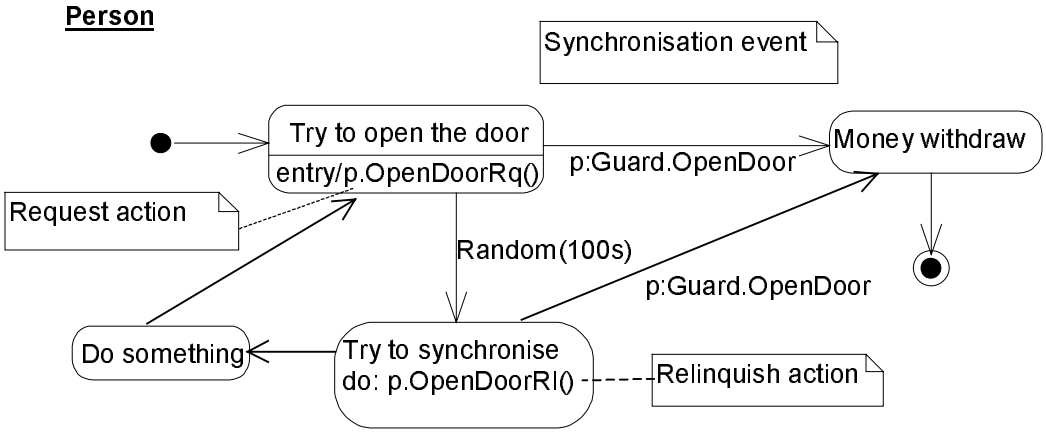

Figure 8: The state diagram of the Person class

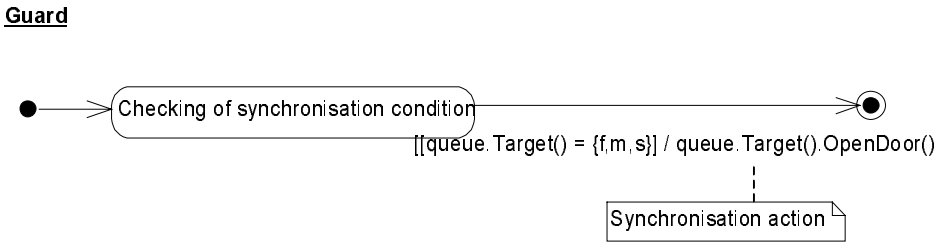

Figure 9: The state diagram of the Guard class

4.2 Data flow computation

We model a data-flow computation that is described by a data flow graph. The graph consists of functional nodes and arcs representing data flow between nodes. Fig. 10 presents a fragment of an exemplary data-flow graph.

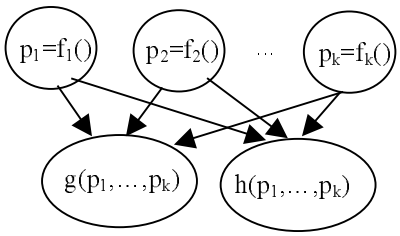

Figure 10: The data-flow graph

Each node produces a value p_i as a result of a function f_i. The value is passed immediately after completion to nodes evaluating *g* and *h* functions. The fragment of a data-flow graph can be transformed into a fragment of the UML activity diagram, Fig. 11. The circles, i.e. activity states represent functional nodes, and transitions represent arcs from the data-flow graph. The bars are join and fork pseudo-states.

The diagram reflects control flow aspect only. The full model, taking into account both data-flow and control-flow aspects, is presented in the extended UML. The model consists of three UML diagrams: class (Fig. 13), sequence (Fig. 14), and state diagrams (Fig. 15 and Fig. 16).

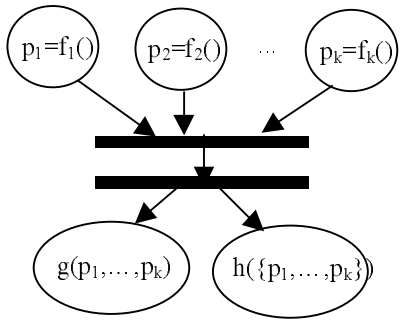

Figure 11: The activity diagram of the data-flow graph

In the class diagram, Fig. 13, the classes $F_1,\ldots,F_k$, G, and H represent computational nodes with operations standing for functions $f_1,\ldots,f_k$, g, and h, respectively. A synchronisation point p is instantiated from *SynchElem* class. The task of the synchronisation point p is first to synchronise functional nodes by generating ready event (acts as a join pseudo state), and next to create two new nodes: based on G and H classes (acts as a fork pseudo state).

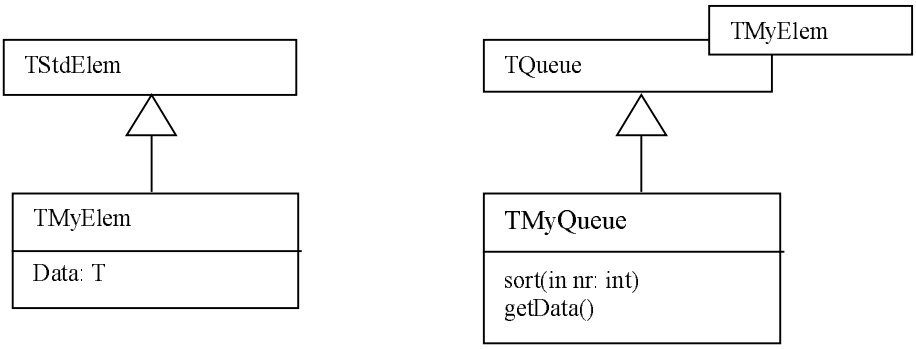

Figure 12: Specialisation of the *TStdElem* and *TQueue* classes

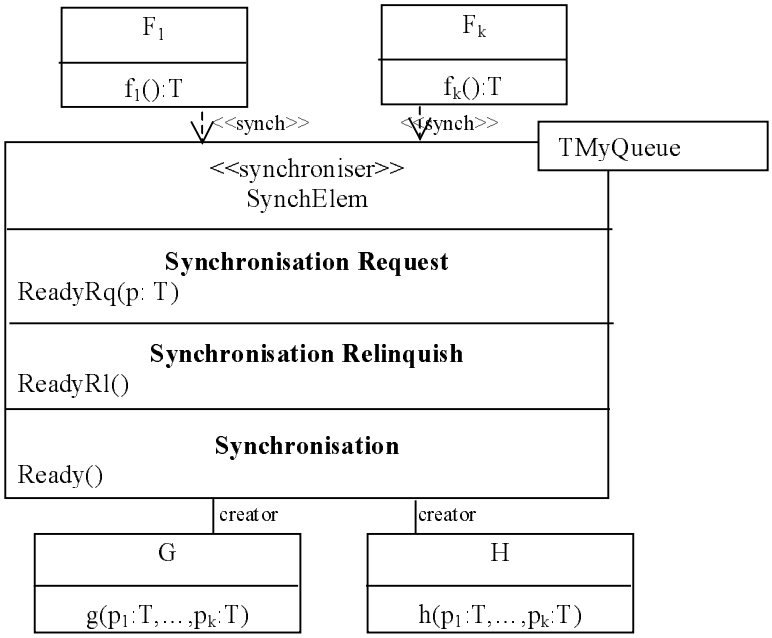

Figure 13: The class diagram of the data-flow graph

The nodes have to pass to the synchronisation point not only their names but also computed values. Therefore, we define the new class *TMyElem* inherited from *TStdElem* class, Fig. 12. The *TMyElem* class is used for instantiation of the *TQueue* class. Next, we derive a new specialised class *TMyQueue* from TQueue<TMyElem> with two additional operations, where:

sort(i) – yields an ordered queue sorted according to *i*-th element of the tuple,

getData – returns a queue of data elements, passed to the synchronisation point by calling the synchronisation request operation.

The attribute queue in the *SynchElem<TMyQueue>* class represents a list of records *<object_name, data>*.

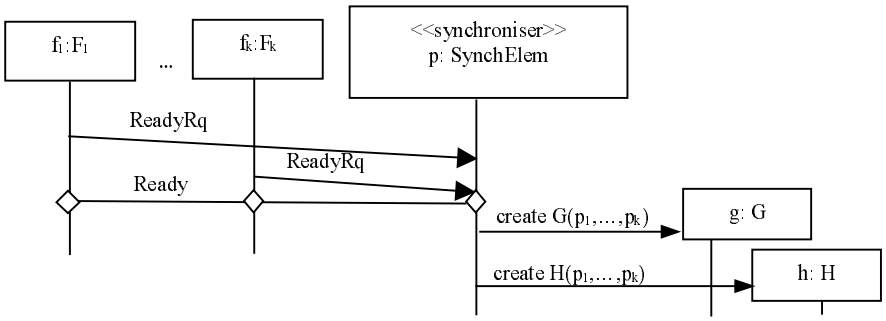

Figure 14: Sequence diagram of the data-flow graph

Fig. 15 and Fig. 16 present a behaviour of the synchronisation point p and the functional nodes F_i for $i = 1,..., k$.

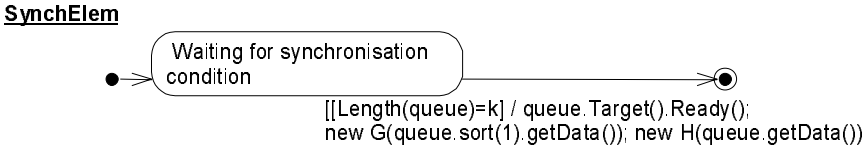

Figure 15: The state diagram of *SynchElem* class

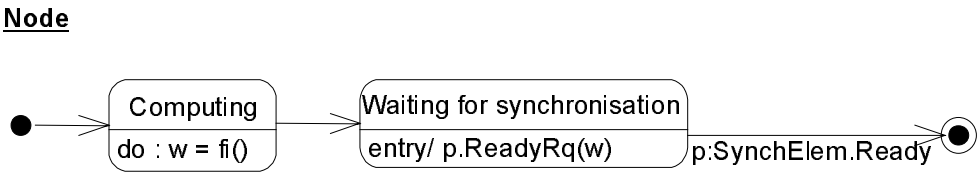

Figure 16: The state diagram of F_i computational node

4.3 Multimedia stream synchronisation

A multimedia stream is a sequence of data units. The data units within a single multimedia stream have to be synchronised, i.e. they have to be processed in subsequent non-overlapping periods of time. For a data unit a triple [x, n, y] of time values is defined, where x denotes the minimal, n – the nominal, and y – the maximal processing time of the data unit. For each data unit there exists an object responsible for its processing.

A multimedia beam is a set of multimedia streams, which have to be processed in parallel. The streams within a multimedia beam must be synchronised each other. Synchronisation between multimedia stream is called an inter-stream synchronisation. There are many synchronisation strategies. For example, many strategies of inter-stream synchronisation are analysed in the paper [5]. Four of them are illustrated in Fig. 17, and explained in the Tab. 1.

We consider an inter-stream synchronisation between different streams. Let assume that there are k different streams in a multimedia beam, and let $p_1, ..., p_k$ be a set of data units, where p_i belongs to *i*-th stream. A triple [x_i, n_i,

y_i] denotes respectively: minimal (x_i), nominal (n_i) and maximal (y_i) processing time of data unit p_i.

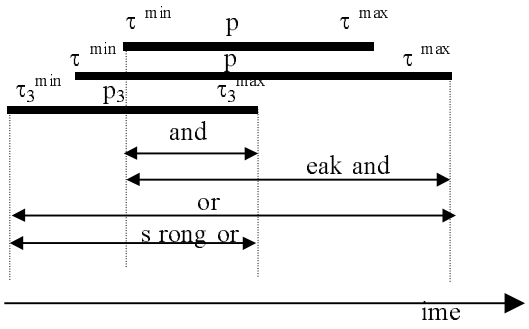

Figure 17: Synchronisation strategies

Synchronisation instants	**Explanation**
τ_i^{start}	When the multimedia object starts to be processed.
τ_i^{min}	When the processing time reaches x_i time units (its minimal time semantics).
τ_i^{end}	When the processing of the multimedia object ends. Ideally this time equals $\tau_i^{start} + n_i$, where n_i is the nominal duration of the multimedia object; the temporal semantics of the multimedia object is preserved if it occurs during the temporal interval $[\tau_i^{start} + x_i, \tau_i^{start} + y_i]$.
τ_i^{max}	When the processing time reaches y_i time units (its maximal time semantics).
τ^{tra}	When the processing of all the multimedia objects associated with a synchronisation point has begun.
τ^{syn}	When the synchronisation point is met.

Table 1. Important time instants for synchronisation control

The synchronisation time instant related to the *and* synchronisation strategy is defined as [5]:

$\tau^{syn} = \min \{ \tau \mid \forall i \in I.(\tau \geq \tau_i^{min}) \wedge (\tau \geq \tau_i^{end})) \vee ((\forall i \in I.\ \tau \geq \tau_i^{min}) \wedge (\exists i \in I.(\tau \geq \tau_i^{max}))\}$ where I is a set of multimedia streams (all streams reach τ^{min}, and either all of them reach τ^{end} or first of them reach τ^{max}).

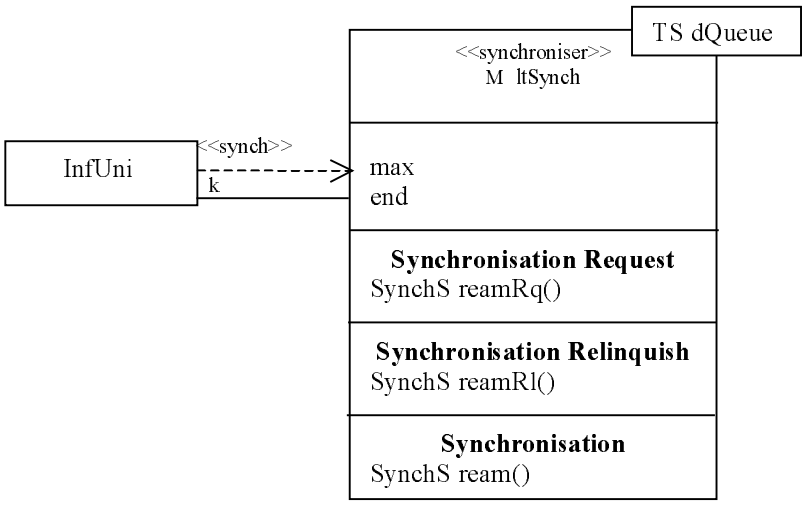

Figure 18: Synchronisation of multimedia streams – class diagram

In the example, we present a model of the *and* strategy in the extended UML. We assume, that objects of *MultStream* class represent data units, which co-operate with a synchronisation point SM. Each object informs dynamically the synchronisation point SM about time instant it has reached, i.e. τ^{min}, τ^{max}, and τ^{end}, by calling *SynchStreamRq*, *max*, and *end* operation of the synchronisation point respectively, Fig. 18 and Fig. 19.

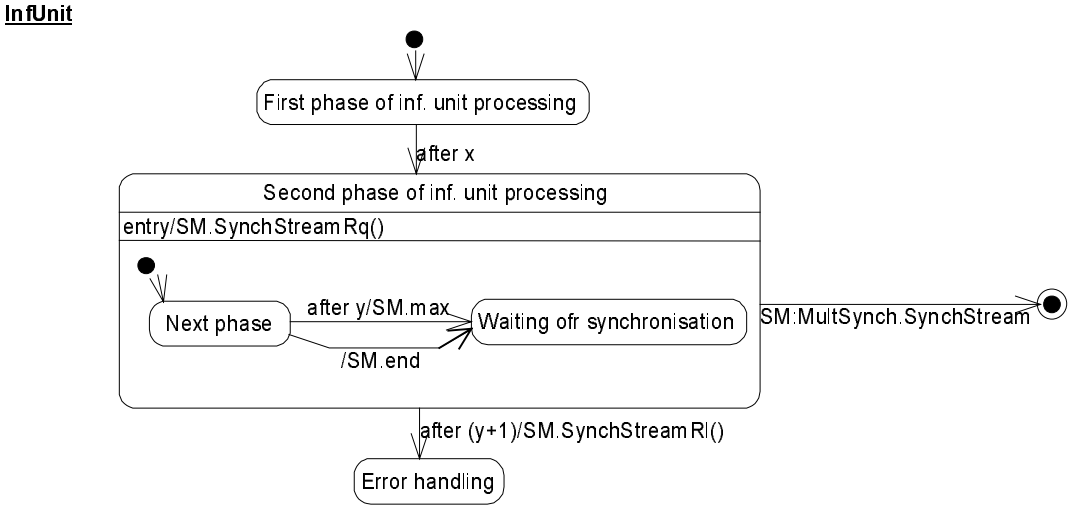

Figure 19: Synchronisation of multimedia streams – state diagram of *MultStream* class

The synchronisation point SM evaluates the synchronisation condition (*SynchStreamRq* operation *k* times called and *end* operation *k* times called or *max* operation ones called), Fig. 20. After synchronisation the synchronisation point is ready to be involved in the next synchronisation.

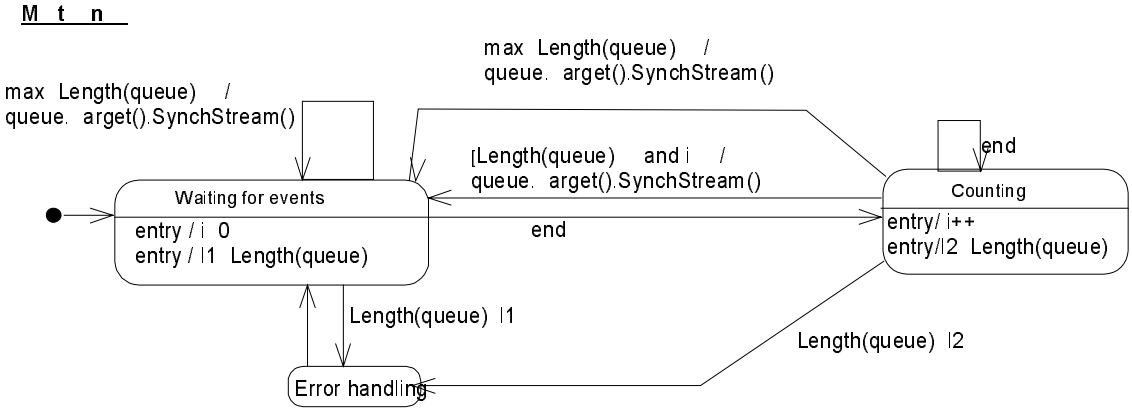

Figure 20: Synchronisation of multimedia streams – state diagram of *MultSynch* class

5 Conclusions

The paper presents an extension of the UML basing on the concept of multicast synchronisation.

The extended UML enables defining of synchronisation rules and a data exchange schema for a group of objects. The synchronisation without data exchange (see examples 4.1 and 4.3) uses well-known broadcast communication as an underlying mechanism, but data exchange (see example 4.2) exploits a more sophisticated pattern matching mechanism known, for example, from Lotos [6].

The proposed synchronisation mechanism is more general than mechanisms offered by formal specification techniques Estelle, SDL or Lotos [6], and enables specification of complex systems at a high level of abstraction. The mechanism may be characterised by the following main features:

- a synchronisation point may implement any multicast synchronisation strategy,
- objects may call a synchronisation point for a multicast synchronisation,
- objects may retract waiting for synchronisation.

Additionally, while waiting for synchronisation, objects may communicate with a synchronisation point.

With respect to the standard UML, our extension is essential. A multicast synchronisation might be modelled in the standard UML as a sequence of operation calls dispatched sequentially from a synchronisation point to objects from a given group. The events resulting from call receptions model a synchronisation event. In the case of precise modelling of real-time systems, these events should occur at the same time instant what, in general, is not possible to guarantee. Another possibility to model a multicast synchronisation is sending a signal to a collection containing a set of objects [1], but this solution is even worse because signals are received asynchronously.

We follow rigorous but informal style of presentation that is used in the UML metamodel [7]. In particular, semantics of the proposed extension was presented informally. Formal definition of the proposed extension as well as the formal definition of the whole UML is still open question, and does not belong to the scope of the paper. Many papers demonstrate that formal definition even of some diagrams, e.g. statechart diagrams [1], is very complex and needs further efforts.

6 References

1. Babczynski T., Huzar Z., Magott J. Algebraic Semantics for Markovian Statecharts. In: Proc. 15th Annual UK Performance Engineering Workshop, 105-119, Bristol, July, 1999

2. Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide, Addison-Wesley, 1998

3. Douglass B. P. Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, 1999

4. ISO/IEC JTC1/SC21 WG7. Enhacements to LOTOS, Project WI 1.21.20.2.3, November 1997

5. Senac P, Diaz M, de Saqi-Sannes P. Toward a formal specification of multimedia synchronisation scenarios. Annales des Telecommunication, Vol. 49, 1994, pp 297-314

6. Turner K.J. (ed.) Using Formal Description Techniques, Wiley, 1993

7. Unified Modelling Language, UML Semantics v. 1.3, Rational Software Corporation, October 1998