

*PauWare*¹ Users' Guide (ver. 1.1), last update: 2008-04-11
The *PauWare* software (www.PauWare.com) is registered under the following *InterDeposit Digital Number*:
IDDN.FR.001.360023.000.S.P.2006.000.10000
Agence pour la Protection des Programmes (APP, <http://app.legalis.net>)
© Franck.Barbier@FranckBarbier.com

Use of this software is subject to the restrictions of the Creative Commons Licenses (<http://creativecommons.org/licenses>) and more precisely ruled by the Creative Commons Legal Code defining the "Attribution-NonCommercial-ShareAlike 2.5" license (<http://creativecommons.org/licenses/by-nc-sa/2.5>)

Contact Address

Franck Barbier
PauWare Research Group
UPPA
BP 1155
64013 Pau CEDEX, France
www.FranckBarbier.com

1 What is *PauWare*? What you can do with *PauWare*

PauWare is a Java API (Application Programming Interface) plus some extra tools that allows you to construct end-user business applications using Model-Driven Development (MDD) technologies. The *PauWare* Java library is divided into two sub-libraries. The first is *PauWare.Composytor* (which stands for *COMPOnent SYstem execuTOR*). It is dedicated to the Java EE platform. The second is *PauWare.Velcro* (which stands for *VELours CROchet*², "Velvet Hook" in English). It is dedicated to the Java ME platform. Both sub-libraries work perfectly well with the Java SE platform version 1.4.x or later ones.

PauWare is also an open source software whose JAR files and Java source files may be downloaded from www.PauWare.com/PauWare_software. However, this software is subject to usage restrictions as stated above. Case studies and extra tools may also be downloaded from the above address as well as this users' guide.

From a software engineering viewpoint, *PauWare* assists and supports Java developers in building intelligible, safe and efficient software components, as well as services and final applications based on the flexible assembly of these components and services when dealing with daily time-to-market constraints. The MDD spirit underlying *PauWare* is that you can visually program by building Harel's *Statecharts* (a sophisticated type of state machine modeling language; see Bibliography for more information). More precisely, you can build *UML 2 State Machine Diagrams* and *Sequence Diagrams*. Web pointers about these modeling techniques may be found at www.PauWare.com/PauWare_software (*Javadoc* link).

Developers may either construct such models by using UML 2 CASE tools (see Section 13.1), which support the XMI (a XML DTD) format for recording models, or they may choose to do so from scratch by directly using the *PauWare* API. *PauWare* makes models persistent at runtime and therefore makes them executable by preventing a break or a gap between the models and their code. As a result, the following activities are favored: rapid prototyping, testing through model-based testing (model simulation), maintenance through state machine diagram correction/evolution management and finally, reuse through well-isolated and well-formed components and services including a composition support.

The rest of this document proposes the easy and progressive discovery of the *PauWare* API through prefabricated models which formalize end-user business requirements. Complex issues are also addressed in this document, like the execution semantics promoted by *PauWare* and its alignment with the UML 2 (see Section 11).

¹ *PauWare* should be pronounced "power" in English.

² *Velcro* is a registered trade mark indicating a fastener for clothes or other items. We use this word as an illustrator of the problem of composition in Component-Based Development.

2 Required environment

PauWare advises using the *NetBeans* IDE (especially the current ver. 6.0 or a forthcoming version) since the *PauWare.Composytor* and *PauWare.Velcro* APIs are provided at www.PauWare.com/PauWare_software as *NetBeans* projects. So, developers may rapidly construct applications because *PauWare* is a standalone product that does not require any third-party library. *PauWare* is also delivered as a *NetBeans* plugin at <http://plugins.netbeans.org/PluginPortal/faces/PluginDetailPage.jsp?pluginid=713>.

In 2008, we investigate the possibility of having *PauWare* as a UML/MDD CASE tool plugin or as a simple complement (see Section 13). This could create an even stronger bond with the *NetBeans* 6.0 version, which includes a dedicated support for UML modeling. This could also be a support for any commercial tool. At this time, *XMI2PauWare* (see again Section 13) is an independent program of *PauWare* that translates the XMI dialect to *PauWare*'s code. Moreover, *XMI2PauWare* has been tested with the *MagicDraw* CASE tool (www.magicdraw.com). The next release of this users' guide will give readers new insights into further research on this topic, and interested readers may contact us about an possible collaboration.

3 Programming with *PauWare*

Within the Java SE framework, one may either use the *Composytor* API (which is mainly dedicated to Java EE) or the *Velcro* API (primarily dedicated to Java ME). The more concise and simple way of programming is by using *Composytor* (see Section 6 for the *Velcro* API).

Figure 1 shows the kernel architecture of *PauWare*. The *AbstractStatechart* Java class is the type of any state, this state being a leaf state of a nested state. In addition, the *AbstractStatechart_monitor* Java class is the type of a state machine. This last class adds event interpretation facilities (by inheritance) to *AbstractStatechart*. These facilities are, among others, expressed by the *run_to_completion* Java method. Therefore, the instantiation of states and state machines have to occur through platform-dependent subclasses of both *AbstractStatechart* and *AbstractStatechart_monitor*. These subclasses are depicted in Figure 1. They either belong to the *_Composytor* package or the *_Velcro* package, depending on the chosen platform. Otherwise, the two UML composition relationships (black diamonds) in Figure 1 simply express the fact that a state machine is composed of states.

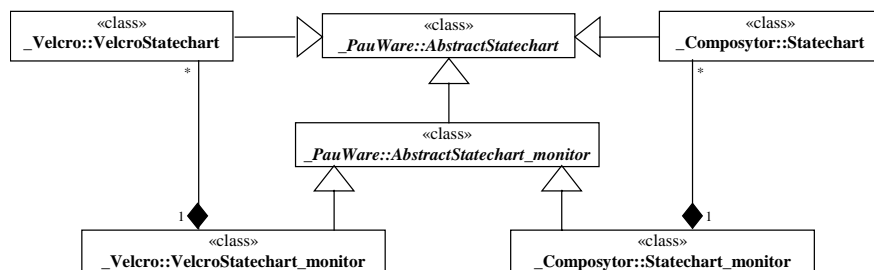


Figure 1. Core of the *PauWare* execution engine.

In order to illustrate the use of the classes appearing in Figure 1, we build the software component named “*PauWare* component” in Figure 2.

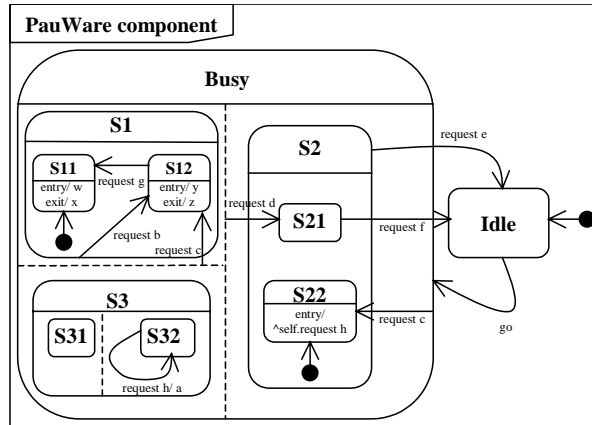


Figure 2. Specification of the behavior of *PauWare component* by means of a UML 2 State Machine Diagram.

In Figure 2, if in *Idle*, the *go* request enables the entry into *Busy*. The *w* action is launched (see inside *S11*) in parallel with the self-sending of *request h* (see inside *S22*), which itself triggers the *a* action (see self-transition in relation with *S32*). Next, the following scenarios may occur, considering that *S11* and *S22* are the default states when entering into *Busy*:

- If *S11* is active and *request b* occurs, then *x;y* is activated and *S12* becomes active
- If *S11* is active and *request c* occurs, then *(x;y)||a* is activated and *S12* and *S22* become active
- If *S11* is active and *request d* occurs, then *S11* (no self-transition) and *S21* become active
- If *S11* is active and *request e* occurs, then *x* is activated and *Idle* becomes active
- If *S11* and *S21* are active and *request f* occurs, then *x* is activated and *Idle* becomes active
- If *S11* and *S22* are active and *request f* occurs, then there is no effect (the event is “lost”)
- If *S11* is active and *request g* occurs, then there is no effect (the event is “lost”)

Performing a similar process for the *S12* state leads to the following:

- If *S12* is active and *request b* occurs, then *z;y* is activated and *S12* and (*S21* or *S22*) become active
- If *S12* is active and *request c* occurs, then *(z;y)||a* is activated and *S12* and *S22* become active
- If *S12* is active and *request d* occurs, then *S12* (no self-transition) and *S21* become active
- If *S12* is active and *request e* occurs, then *z* is activated and *Idle* becomes active
- If *S12* and *S21* are active and *request f* occurs, then *z* is activated and *Idle* becomes active
- If *S12* and *S22* are active and *request f* occurs, then there is no effect (the event is “lost”)
- If *S12* and *request g* occurs, then *z;w* is activated and *S11* becomes active

3.1 State and state machine design

The implementation of the software component depicted in Figure 2 leads to the following state declaration:

```
public class PauWare_component {
    protected AbstractStatechart _Idle;
    protected AbstractStatechart _S11;
    protected AbstractStatechart _S12;
    protected AbstractStatechart _S21;
    protected AbstractStatechart _S22;
    protected AbstractStatechart _S31;
    protected AbstractStatechart _S32;
}
```

```

protected AbstractStatechart _S1;

protected AbstractStatechart _S2;

protected AbstractStatechart _S3;

protected AbstractStatechart _Busy;

```

...

The state machine is next declared as follows:

```

protected AbstractStatechart_monitor _PauWare_component;

```

The instantiation of states, their mutual linking, the definition of their properties and the attachment of actions (*e.g.*, *a*, *x*, *y*, *z* and *w* in Figure 2) have to occur within a dedicated method named “init_behavior”³ (see below). This method is preferably called within the constructor(s) of the *PauWare_component* Java class. The result of an exclusiveness relationship (*xor* method of the *PauWare* API) between two states is a superstate, which may be assigned to a declared state. For example, in the code below, *S1* is defined as the superstate of two direct exclusive substates *S11* and *S12*. This is the same for the orthogonality operator (*and* method of the *PauWare* API).

```

protected void init_behavior() throws Statechart_exception {

    _Idle = new Statechart("Idle");

    _Idle.inputState();

    _S11 = ((new Statechart("S11")).entryAction(this,"w")).exitAction(this,"x");

    _S11.inputState();

    _S12 = ((new Statechart("S12")).entryAction(this,"y")).exitAction(this,"z");

    _S21 = new Statechart("S21");

    _S22 = new Statechart("S22").entryAction(this,"request_h",null,AbstractStatechart.Broadcast);

    _S22.inputState();

    _S31 = new Statechart("S31");

    _S32 = new Statechart("S32");

    _S1 = (_S11.xor(_S12)).name("S1"); // mutual linking through exclusiveness

    _S2 = (_S21.xor(_S22)).name("S2");

    _S3 = (_S31.and(_S32)).name("S3"); // mutual linking through orthogonality

    _Busy = (_S1.and(_S2)).and(_S3).name("Busy");

}

```

To summarize, the code above creates an operational or implementation-centric model which in essence conforms to the conceptual model in Figure 2.

3.1.1 Input and output states

The code of *init_behavior* includes the use of the *inputState* *PauWare* API method which marks a state as an input state (notation is a plain black circle with an outgoing arrow pointing to the input state: see *Idle* for

³ These are just “best modeling practices” in case of one may want to later override the initialization process of this method (see also Section 3.1.7).

instance in Figure 2). A dual *outputState* method for terminating states also exists (this notation is a plain black circle surrounded by an enhanced circle and an incoming arrow coming from the output state). The *outputState* method has no effect for the moment.

The role of input states is important. Any forced or ordinary activation (see also Section 8) of a state *S* having to two or more exclusive direct substates may raise problems, if the *PauWare* engine has no indication about what substate to activate. An exception is thrown, unless one and only one direct substate is marked as input state. In this case, activating *S* is equivalent to activating this specific substate. Most of the time, transitions are defined to avoid such problems by pointing to the substate directly, instead of its immediate superstate. This is always true in order to accept several styles of modeling. In Figure 2, for instance, the transition which has *go* as a label passes from *Idle* to *Busy*. There is nevertheless no risk of a thrown exception, since all existing sets of exclusive states (*i.e.*, $\{S11, S12\}$ and $\{S21, S22\}$), have one and only one input state, (*i.e.*, *S11*, respectively, *S22*).

3.1.2 Actions and activities

Any Java method defined by a software component, here *PauWare_component*, is either:

- An action; for instance, the *w* method:

```
public void w() {
    System.out.println("w activated");
}
```

- Or the processing of an event; for instance, the *request_b* method:

```
public void request_b() throws Statechart_exception {
    _PauWare_component.run_to_completion("request_b");
}
```

By convention, actions' bodies do not have to rely on the *PauWare* API. They deal with internal data and objects owned by the software component which supports these actions. They are in particular executed at key instants of the state machine's evolution. That is why a given method may be viewed as an entry action of a state (*w* and *y* in Figure 2), an exit action of a state (*x* and *z* in Figure 2), an activity of a state (there is no example in Figure 2), or in the most simple case, a reaction to the occurrence of an event (for example, *a* in Figure 2 is launched when *request_h* occurs). **These possible statuses are not exclusive.** So, being an entry action of one state does not preclude being an activity of another one, etc. However, best modeling practices do not encourage such configurations.

When using the *Composytor* API, *PauWare* calls actions by means of the Java reflection mechanism. As a result, **actions must be declared as *public*** (see above, *w* for instance). This is not mandatory for the *Velcro* API, in which actions may be *a priori* declared as *private* or *protected* (see also Section 6).

3.1.3 Setup of entry actions, exit actions and activities

Actions are attached to states as entry actions. One instruments such a modeling possibility through the *PauWare* API *entryAction* method. The *entryAction* method may be called multiple times for the same state within the *init_behavior* method, if more than one action, say *a1*, *a2*..., have to be executed when entering into this state. Moreover, the same action (*e.g.*, *a1*) with, for instance, different parameter values, may also be attached multiple times to the same state. In case of multiple entry actions attached to a state, the order of execution conforms to the order of attachment. The FIFO mode applies, *i.e.*, first attached, first executed. The same rules apply for exit actions (*exitAction* method).

In the code appearing in Section 3.1, *x*, *y*, *z* and *w* are attached to states according the specification in Figure 2. As for *request_h*, it has a double status. It is an action in that it is launched when entering into *S22*. However, it is also an event, which labels a transition from *S32* to *S32*. In contrast, *x*, *y*, *z* and *w* are "internal" and thus do not interfere with the course of the state machine. As a result, *request_h* is attached as an entry action to *S22*, but the *AbstractStatechart.Broadcast* parameter value controls the fact that *request_h* may arrive at any time while the state machine may be active (*i.e.*, while it is processing another event). In short, for a state machine, sending an event to itself requires the use of the *AbstractStatechart.Broadcast* constant class variable. The use of *null* before *AbstractStatechart.Broadcast* just means that *request_h* has no parameters.

Actions may also be attached to states as “activities”. One enables such a modeling possibility through the *PauWare* API *doActivity* method which is associated with the UML *do/* notation. Activities last, while actions are recognized as instantaneous and associated with events. More precisely, activities execute out of the scope of event occurrences, while actions run when events take place. At this time, the *PauWare* engine encapsulates the execution of activities in non-interruptible threads⁴. As a result, re-execution of the same activity (*i.e.*, re-entering into the same state having this activity) is constrained by the fact that the *PauWare* engine waits until this activity finishes (*i.e.*, its immediately prior execution) before launching a new execution of the same activity. In the worst cases, deadlocks may therefore appear, or in the best cases, only some latencies. Logically, according the “best” modeling practices, activities must focus on local data transformation without need for control or for interaction with other components. More generally, no synchronization or blocking mode of data processing is *a priori* required in activities (this also applies to actions) since state machine diagrams intrinsically model and thus take charge of such aspects.

Otherwise, contrary to actions, one activity at the most may be attached to a state. So, calling the *doActivity* method of the *PauWare* API for the same state multiple times has no effect: only the last registered activity is taken into account. Activities are launched after entry actions and before exit actions (see also Section 4.2).

3.1.4 Action and activity parameters

The Java methods of the implementation of a software component to be built may of course have arguments. When modeled in state machine diagrams as activities, entry actions or exit actions, **one may not supply these methods with parameter values by using the parameter values of events**. Event parameters are considered as “external” data. These data may of course have an impact on the evolution of state machines. To do so, event parameter values must be “injected” in state machines **without** using the *doActivity*, *entryAction* and *exitAction* functions of the *PauWare* API. Indeed, activities, entry actions or exit actions by their very nature, only depend upon states and are used to normally process the “internal” data of software components. Once again, this is a “good” modeling principle and it has to be respected to prevent problems at programming time.

For instance, if event *e*, which is labeling a transition *t*, has a parameter *p*; the value of *p* can only be available at the time *e* occurs. Besides, if the target state of *t* is *S* and *S* has *a1* as its entry action (Figure 3); *a1* cannot have *p* as its parameter and, at the same time, a status of entry action (see Figure 3, bottom, left hand side). As a result, a relevant modeling practice is based on activities, entry and exit actions that should not require data coming from events (see Figure 3, top, left hand side). The only derogation is to associate actions (this is not possible for activities) with events (see Figure 3, bottom, right hand side). Thus, they are declared as being launched when events occur, instead of being viewed as entry or exit actions of states. In this case, programming the model at the bottom, right hand side of Figure 3, calls for a different pattern: the concept of runtime transition in *PauWare* (see Section 3.2).

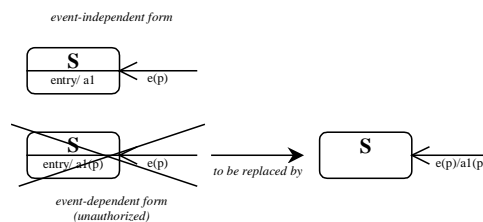


Figure 3. Event parameters and their incompatibility with entry actions, exit actions and activities.

3.1.5 Component kick-off

Returning to *PauWare_component*, the *PauWare*-oriented implementation of the state machine diagram in Figure 2, its *init_behavior* method has to be complemented by a *start* method which is used to construct the global state machine of *PauWare_component* as follows:

```
protected void start(String name) throws Statechart_exception {
```

```
    _PauWare_component = new
    Statechart_monitor(_Idle.xor(_Busy),name,AbstractStatechart_monitor.Show_on_system_out);
```

⁴ This is with respect to the run-to-completion execution model of the UML (see also Section 4.1).

```
...
}
```

The code above shows that an *AbstractStatechart_monitor* (an instance of the type embodying state machines) behaves like an *AbstractStatechart* (an instance of the type embodying states)⁵. The *start* method is mainly dedicated to the instantiation of the state machine. However, **this instantiation starts the execution of all the entry actions of all the default input states**. That is why the instantiation of the state machine must be dissociated from that of its compound states occurring within *init_behavior*.

At this point, one may carefully consider what all the entry actions of all the default input states actually do. Since these actions are supposed to cope with resources (GUIs, ports, connections and so on), these ones must be ready: all the entry actions of all the default input states may request services from them. These ready resources may themselves be the source of events. This means that the state machine must itself be ready to process them. According to the variety of situations, the *start* method may be executed later, instead of executing it as the third statement of the component's constructor as advocated in Section 3.1.7.

3.1.6 State machine tracing

The ability to trace the evolution of a state machine relies on the assignment of the *AbstractStatechart_monitor.Show_on_system_out* constant value (see code above) to the third argument of two offered constructors of the *Statechart_monitor* class (*Composytor* API) or the *VelcroStatechart_monitor* class (*Velcro* API). In such a case, all of the behavior of the state machine will be displayed in a verbose way, on the Java *System.out* stream. Another constructor of these two classes accepts, as a fourth argument, an instance of the *Statechart_monitor_listener* Java interface. This interface is as follows (see also Section 8.3):

```
public interface Statechart_monitor_listener {

    void initialize(AbstractStatechart_monitor monitor);

    void run_to_completion(String text);

    void run_to_completion(java.util.Hashtable execution);

}
```

The more simple way of using this interface is by constructing a class which implements it with an empty body for the *void initialize(AbstractStatechart_monitor monitor)* and *void run_to_completion(java.util.Hashtable execution)* methods⁶. As for the *run_to_completion(String text)* method, the class which implements the *Statechart_monitor_listener* Java interface may display the *text* parameter by using the display resources of its local environment. The content of the *text* parameter is equivalent to what is displayed when we use *AbstractStatechart_monitor.Show_on_system_out* as third parameter of the two dedicated constructors of *Statechart_monitor* (*Composytor*) and *VelcroStatechart_monitor* (*Velcro*).

3.1.7 PauWare's programming patterns

Once the design of the states and the state machine of *PauWare component* in Figure 2 is finished, a Java constructor is required with the following form:

```
public PauWare_component() throws Statechart_exception {

    init_structure();

    init_behavior();

    start("PauWare component");

}
```

⁵ See again Figure 1 for a concise view of *PauWare's* core architecture.

⁶ These methods are used by the *PauWareView* tool (see Section 12) and more generally by management application tiers which aim at monitoring, or even controlling, *PauWare* components (Section 8.3).

Obviously, one may observe in the code above that *init_behavior* is called before *start*. This style of programming is the most common one. However, in some cases, *start* must be called at another appropriate moment. Namely, lots of data and objects will be accessed and used by entry actions. This data and these objects must then be in adequate states when *start* is called. It is therefore sometimes relevant to externalize *start* from the constructor(s) of a software component, so that it may be called in a deferred way.

While the *init_behavior* and *start* methods are concerned with dynamics, it is appropriate to isolate the construction and initialization of data and objects within an *init_structure* method. For instance, a local variable to a state machine may be needed to record some information during the state machine's execution. The *init_structure* method aims therefore at initializing such a variable. In the code above, *init_structure* is of course called before *init_behavior* and *start* because the state machine uses the data and objects constructed and initialized by means of *init_structure*.

3.2 Transition programming

Transitions have to be coded by using the *fires* multi-shape method as follows:

```
_PauWare_component.fires("request_h",_S32,_S32,true,this,"a");
```

In this example, an occurrence of the *request h* event leads to leaving *S32* and re-entering into it (see also Figure 2). Since this transition is not guarded, *true* is used as a default value for the guard declaration. Besides, the *a* action is launched when firing this transition. As said before, *a* will be called via the Java reflection mechanism (*Composytor* sub-library only). Moreover, *a* is called by the *PauWare* engine at an appropriate moment of the state machine's course.

The specificity of *a* is the fact that it has no parameters. This allows us to include the prior statement in the *start* method (see Section 3.1.5) as follows:

```
protected void start(String name) throws Statechart_exception {  
  
    _PauWare_component = new Statechart_monitor(_Idle.xor(_Busy),name,  
    AbstractStatechart_monitor.Show_on_system_out);  
  
    _PauWare_component.fires("request_h",_S32,_S32,true,this,"a");  
  
    ...  
  
}
```

In such a case, the transition is known as cached (see also Section 5.7). It is registered once and for all. The processing of the *request h* will possibly lead to triggering this transition, depending upon the current execution conditions: the transition's source state is active and the transition's guard (pre-condition) is true. An important point here is the naming of *request h* (i.e., the *request_h* string as the first parameter of *fires* in the code above). This name is used later in the code dedicated to event processing, in order to match transition declarations in the *start* method to the events processing methods (see Section 3.4).

PauWare also supports the notion of runtime transition (see also Section 5.7 for more complicated examples). The key difference is as follows: a runtime transition is registered within the Java method (*request_c* below) embodying the processing of its corresponding event: *request c*. In contrast, a cached transition is registered in the *start* predefined method. As an illustration of the former case, the Java method which processes the *request c* event may include a transition declaration (*fires* method) from *Busy* to *S22*:

```
public void request_c() throws Statechart_exception {  
  
    _PauWare_component.fires(_Busy,_S22); // or _PauWare_component.fires("request_c",_Busy,_S22);  
  
    _PauWare_component.run_to_completion("request_c");  
  
}
```

Most of the time, this style of transition programming must be avoided since it is costly. Here, each time *request c* occurs, the *request_c* method is called, which itself in turn calls *fires*. Having the

`_PauWare_component.fires(_Busy,_S22)` statement in the *start* method⁷ would indeed avoid multiple calls. So, we require runtime transitions only when events have parameters, and when these parameters must be forwarded to actions: a typical case is the model at the bottom of Figure 3 (on the right hand side).

Returning to the example of the *request c* event, one may notice that in Figure 2, this event is the label of two different transitions (*Busy -> S22* and *Busy -> S12*). Transitions labeled by the same event may therefore be described both in the *start* method and in the method processing this event. This solution is *a priori* unsatisfactory but it is here illustrated as part of the *PauWare* API. Applying this rule to the *Busy -> S22* and *Busy -> S12* transitions thus lead to viewing *Busy -> S22* as a runtime transition (code above) while *Busy -> S12* is registered (cached) in the *start* method as follows:

```
protected void start(String name) throws Statechart_exception {

    _PauWare_component = new Statechart_monitor(_Idle.xor(_Busy),name,true);

    _PauWare_component.fires("request_h",_S32,_S32,true,this,"a");

    _PauWare_component.fires("request_c",_Busy,_S12);

    ...
}
```

Once again, this style of programming is not satisfactory except when, for a given event, one transition does not require the event's parameter(s), while another requires it(them). Section 5.7 provides a more detailed discussion on this topic.

3.3 Guard programming

Guards (a.k.a. pre-conditions) appear as the third⁸ or fourth parameter of the *fires* method. So, they can be populated accordingly. The *true* value must be used when a transition is not guarded but some extra parameters are required for *fires*. For example, one may want to setup the call of an action (*a* below). Although no guard exists, it is mandatory to use *true* as follows:

```
_PauWare_component.fires("request_h",_S32,_S32,true,this,"a");
```

In this example, the need to declare the *a* action obliges one to use *true* while the corresponding transition in Figure 2 has no guard. Consequently, the possible value of the third or fourth parameter of the *fires* method can then be *false*,⁹ *true*, a logical expression like *i < j* or a method¹⁰ returning a Boolean type (*i.e.*, *java.lang.Boolean* or *boolean*). Any exception or error that would appear when executing this method amounts to evaluating the associated guard to *false*.

So, we may have:

```
_A_state_machine.fires(_From,_To,i < j);
```

Or we may have:

```
_A_state_machine.fires(_From,_To,i_less_than_j());
```

Or:

```
_A_state_machine.fires(_From,_To,this,"i_less_than_j");
```

The second and third forms require the building of the following Java method:

```
public boolean i_less_than_j() {

    return i < j;

}
```

⁷ In such case, the following form is mandatory:

```
_PauWare_component.fires("request_c",_Busy,_S22);
```

⁸ This is the case when the *fires* method does not have, as first parameter, the name of the event labeling it; the concept of runtime transition (see Section 3.2).

⁹ However, one may notice that this is a weird case because the transition will never be triggered by the *PauWare* engine.

¹⁰ This method must be declared as "public" when using the *Composytor* API since it is called by reflection.

Moreover, guards and actions may both have their own parameters. Transitions, which have guards with parameters (and these parameters come from events), cannot be cached, *i.e.*, they cannot be coded in the *start* method:

```
public void an_event(int i) throws Statechart_exception {  
  
    _A_state_machine.fires(_From,_To,i < j); // j is a local data of the component while i is external  
  
    _A_state_machine.run_to_completion("an_event");  
  
}
```

We may also have:

```
public void an_event(int i) throws Statechart_exception {  
  
    Object[] args = new Object[1];  
  
    args[0] = new Integer(i);  
  
    _A_state_machine.fires(_From,_To,this,"i_less_than_j",args);  
  
    _A_state_machine.run_to_completion("an_event");  
  
}
```

This second form itself requires an adapted form of the method computing the guard:

```
public boolean i_less_than_j(Integer i) { // Java SE 1.4-compliant form due to the absence of auto-boxing  
  
    return i.intValue() < j;  
  
}
```

3.4 Event processing

The *request h* event in Figure 2 is the label of one unique transition. Implementing this event processing leads to the following code:

```
public void request_h() throws Statechart_exception {  
  
    // any event post-condition may appear here (see also Section 6 for such examples)  
  
    // runtime transitions, if any  
  
    _PauWare_component.run_to_completion("request_h");  
  
}
```

More generally, each event type¹¹ can be distinguished from the others by its name and its signature in a state machine diagram like the model in Figure 2. This leads to the creation of a Java method with the *void* type as its return type and *public* as its visibility modifier¹². The body of this Java method can consist of some post-conditions¹³, some runtime transitions and the call of the *run_to_completion* method of the *PauWare* API. The latter moves an instantiated (from its model: a state machine diagram) state machine from one **stable consistent context**¹⁴ to another one. The *run_to_completion* method is synchronized, meaning in Java that it cannot be interrupted. The true execution is carried out by means of *run_to_completion* only, since the *fires* method is

¹¹ Here we use the word "type" since the same event may be the label of more than one transition. However, only one Java method is required for this event.

¹² Event processing methods become members of provided interfaces; they must be declared as *public* (see also Figure 4).

¹³ They normally appear in a state machine diagram as OCL constraints or outside the drawing as additional textual specifications.

¹⁴ The word "configuration" is used instead in UML documentation.

fully dedicated to the detailed description of transitions. Applying this approach systematically to the model in Figure 2 yields the UML component diagram in Figure 4.

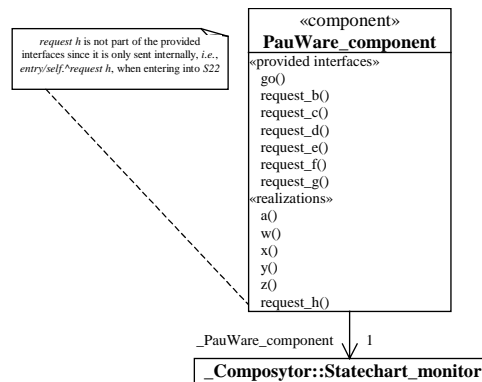


Figure 4. Architectural view of *PauWare* component (use of the *Composytor* sub-library).

So, events become member services of the built component’s provided interfaces. By calling these services, clients of this component simulate event sending. As for actions, they are internal parts named “<<realizations>>” in the UML formalism. The *request h* event gets special treatment. Since it is only sent internally, it is not a member of the component’s <<provided interfaces>> compartment.

We may just note a slight deficiency of the *Composytor* API, which forces the implementation of actions, activities, guard evaluation methods and invariant evaluation methods (see Section 5.8) to be *public* Java methods. This is due to the use of the Java reflection mechanism. The *public* modifier may indeed be viewed as error-prone since actions, activities, guard evaluation methods and invariant evaluation methods do not **have not to be called externally** but internally from the *PauWare* engine. More particularly, calling them from the outside contradicts the spirit of the UML State Machine Diagrams.

3.5 Communication programming

Communication programming consists in sending events that are associated with the ^ symbol of UML¹⁵. In *PauWare*, this amounts to holding a reference to another component or an entry point (for example a queue of messages or a component’s remote interface... see also Section 7). A reference to a remote interface enables the call of a service. This service is modeled by an event which appears on one or more transitions in the state machine diagram of the receiver component. Section 7 comments on a case study with many examples of component communications using the ^ symbol.

The philosophy behind Harel’s *Statecharts* and the *UML State Machine Diagrams* is a broadcast communication mode meaning that the sender does not care about the receiver status at event sending time: “The statechart communication mechanism, on the other hand¹⁶, is based on broadcast, whereby the sender proceeds even if nobody is listening.” (see page 269 of the paper “Statecharts: A Visual Formalism for Complex Systems, Bibliography of this users’ guide). We discuss this point in further details in Section 4.1.

We describe in Section 3.1.3 and in Figure 2 the self-sending of an event (*request h*). Sending an event to oneself is useful and common when a component has parallel substates like *S2* and *S3* in Figure 2 (*request h* is generated from *S22*, a substate of *S2*, and interprets in relation with *S32*, a substate of *S3*). In Section 3.1, in the proposed Java code, we simply use the *this* symbol of Java and the *AbstractStatechart.Broadcast* constant variable, which is mandatory in case of event self-sending. More generally, *AbstractStatechart.Broadcast* can always be used as the last parameter of the *fires* method. There is no failure risk, but some probable performance overheads may appear. For the sender component, using *AbstractStatechart.Broadcast* means that the action¹⁷ within the *fires* method is not called inline (i.e., at the time the *PauWare* engine moves the sender

¹⁵ In UML 2, this symbol seems to have migrated to become part of the Object Constraint Language or OCL (ver. 2.0) instead of the UML 2 itself. In any case, its usage remains.

¹⁶ In his original paper, Harel compares the *Statecharts*’ communication mechanism with that of the *Communicating Sequential Processes* of Hoare.

¹⁷ If this action is not the sending of an event (i.e., the simple processing of local data and objects) using this mechanism is worthless.

component's state machine forward¹⁸). Instead, this action will be called in a deferred way. More precisely, it is executed once the run-to-completion cycle has terminated. *AbstractStatechart.Broadcast* is thus crucial for event self-sending because one must guarantee that the run-to-completion cycles of the same state machine do not overlap. Concerning the implementation of the model in Figure 2, we must ensure that the processing of *request c* is not intertwined with the processing of *request h*. More generally, *AbstractStatechart.Broadcast* manages component reentrance in request/reply communication modes (see Section 4.1 for an in-depth discussion).

3.6 How to learn more about *PauWare*

Before going on, we invite novice readers to study more complicated examples, especially the *Home Automation System* case study in Section 6 of this document. Even if this demonstrator is dedicated to the Java ME platform, it is based on the common expected use of the *PauWare* API and thus may give readers better insights into *PauWare*. For expert readers, Section 4 enters into further details about the underlying modeling theory behind *PauWare*.

4 Execution semantics

This section enters into further details about *PauWare*'s execution semantics. It indeed stresses the fact that any modeler must understand the full effect(s) of his designed state machine diagrams if these are to be executed later. In other words, the effect(s) at execution time, relating to using a given modeling construct or one of its variations, must be unique, non self-contradictory and coherent with other related model pieces. The execution must also be deterministic and thus predictable when models are being simulated with *PauWare*. Keeping this in mind, this section provides a clear and formal interpretation of different model forms that might be confusing or even ambiguous for a modeler. Section 5 completes this section with the execution semantics linked to the notion of "allowed event".

4.1 Run-to-completion cycles

The run-to-completion execution semantics feature is one of the main characteristics of the *UML 2 State Machine Diagrams*. A run-to-completion cycle consists in moving a state machine from one stable consistent context (or configuration) to another stable consistent context. A stable consistent context means being within a modeled state (*i.e.*, this state is active at a given time), or a set of modeled states if these are connected with each other by means of nesting or orthogonality relationships, and no event processing is in progress. An inconsistent context means being in two exclusive states at the same time, or being in a state without knowing which of its direct exclusive substate(s) has to be active (indeterminism), etc. The *run_to_completion* Java method¹⁹ of the *AbstractStatechart_monitor* class of *PauWare* is responsible for passing a state machine from one set of active states to another set of active states. For several reasons (including the fact that the execution semantics of the *UML 2 State Machine Diagrams* are not always well-defined), the *PauWare* execution semantics may slightly differ from that of the UML 2 (see Section 11 for an in-depth discussion). However, it complies to the principle of run-to-completion.

The *run_to_completion* Java method of the *PauWare* API is supposed to raise an instance of the *Statechart_exception* Java class. Handling such an exception is therefore the most common way to analyze why a state machine is not in stable consistent context after a run-to-completion cycle. When simulating state machines, one may thus be able to detect design errors, especially when *Statechart_exception* objects appear.

In *PauWare*, the *run_to_completion* Java method is *synchronized*, meaning that it cannot be interrupted²⁰. This enables the concurrent access of the software components which encapsulate state machines. This is mainly done by client components, which can send requests at the same time (see also Section 5.10 as well as

¹⁸ Here, we analyze a run-to-completion cycle of the sender component's state machine in which a fired transition leads to triggering the sending of an event to itself or a distinct receiver component.

¹⁹ As a complement, one has to notice that the *fires* Java method does not execute anything. It just registers transitions for their late or close triggering within *run_to_completion*.

²⁰ For obvious technical reasons, the *fires* method is also set to *synchronized* in *PauWare* (see also the notion of "runtime transition" in Section 5.7).

the case study in Section 7 for concurrency examples). In this context, the processing of an event can only occur if the immediately prior event processing has already finished. The *run_to_completion* Java method processes events one by one. An arriving event *e*, materialized in *PauWare* by a Java method like *public void e() {... run_to_completion("e");}*, is therefore automatically queued by the Java multithreading service. This guarantees that a stable consistent context is reached before a new event may be processed.

However, we show in Section 3.1.3 and in Section 3.5 the possibility of having *run_to_completion* fortuitously called by itself when a software component sends requests to itself²¹. In such a case, the *AbstractStatechart.Broadcast* parameter value must be used. Remember that in Figure 2, an occurrence of *request h* is formally and strictly processed **after** the processing of an occurrence of *request c* **because** *AbstractStatechart.Broadcast* is used. *request c* enables the entry into *S22*, which leads to sending *request h*. In terms of implementation, *run_to_completion* is first called to process *request c* and next called in cascade to process *request h*, while the processing of *request c* has not been completed. This approach is based on broadcast communication: *request h* is sent to “the” component itself (*i.e.*, the sender is equal to the receiver) without knowledge about the end of the processing of *request c*.

In fact, the main point is to avoid loopback calls²² in *PauWare*. However, if these loopback calls are mandatory because of the business logic (such a case is illustrated in Figure 2), one must manually **detect them in models** and consequently use *AbstractStatechart.Broadcast*. In any case, *AbstractStatechart.Broadcast* can always be used (*i.e.*, generalized) without risk of damage, but it is costly.

To have a rational practice of *PauWare*, the obvious question is thus when should one use the *AbstractStatechart.Broadcast* parameter value? In the case of self-sending an event, as for *request h*, there is no choice: omitting this property leads to an inconsistent context for a state machine. This state machine cannot accept *request h* before the processing of *request c* is terminated. Using *AbstractStatechart.Broadcast* associated with the processing of *request h* ensures a thread creation whose execution is delayed due to the synchronized nature of *run_to_completion*. A more tricky case is when two distinct component instances communicate; the sender generally has to use *AbstractStatechart.Broadcast* if the receiver is subject to reply (directly or indirectly) to the sent request. Excepting this common situation, there is no risk of unanticipated reentrance and *AbstractStatechart.Broadcast* need not be used to keep performance at an acceptable level.

To sum up, one may say that thanks to *PauWare*, run-to-completion cycles are non interruptible. Furthermore, unanticipated reentrance may only occur through a sequence of function calls in which the first call is issued by a component and the last call is concerned with a service provided by this **same component**. The sequence *request c;request h* is an archetype. *UML 2 Sequence Diagrams* may help to detect such reentrance and thus avoid problems which may otherwise have arisen because no safety net is provided. Put differently, in *PauWare*, event sending is by default a (possibly remote) function call. Function call is however in contradiction with broadcast, but most applications do not require such sophisticated facilities like broadcast: point-to-point communication is sufficient. We thus decided to provide *AbstractStatechart.Broadcast* as a tool instead of having a systematic utilization. Moreover, we may observe that in a technological component model like the EJB computing framework, components run in containers which have their own rules about reentrance. In this case, building an application with *PauWare* is an architectural problem (see Section 7 for a case study) rather than a problem which focuses on the single use of *AbstractStatechart.Broadcast*.

4.2 Action/activity execution ordering

For a given state (*S* in Figure 5), entry actions (*b* in Figure 5) are executed after the actions (*a* in Figure 5) launched by the event (*e* in Figure 5) which causes the entry into this state. Next, activities are executed (*c* in Figure 5) and followed by exit actions (*d* in Figure 5) when one leaves, probably later, the state.

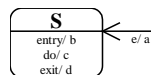


Figure 5. Action/activity execution order.

²¹ Here, we may notice that this form of communication is interesting when a software component has parallel states (see Figure 2 for instance).

²² See also the *Enterprise JavaBeans* (EJBs) component model which does not tolerate such a mechanism.

When more than one entry²³ action for the same state (*a1* and *a2* in Figure 6 for *S*) is defined, the execution order is based on a first-registered, first-served logic. Namely, from a modeling viewpoint, one expects the execution of *a1* before the execution of *a2*.



Figure 6. Entry action execution order.

In *PauWare*, the following declaration order is therefore important (*a1* is registered before *a2*):

```
_S = (new Statechart("S")).entryAction(this, "a1");
```

```
_S.entryAction(this, "a2");
```

4.3 Execution ordering and orthogonality

As sketched in the scenario in the beginning of Section 3, no assumption has to be made, from a modeling viewpoint and thus from an execution viewpoint, about the execution order of actions belonging to concurrent states. For instance, if one enters into two orthogonal states *S1* and *S2*, with *S1* having *a1* as entry action and *S2* having *a2* also as entry action, no order can be established. This means that, from a business logic viewpoint, a specific sequence of execution between *a1* and *a2* is not a strict requirement²⁴: they may occur in parallel, one after the other one, etc. In contrast, if an order is desired, this amounts to reviewing and probably changing the model so that the order in execution is stated.

4.4 Execution ordering and nesting

When entering in states like *S1* (Figure 7), entry actions of superstates are executed first. Here, this means that *a* precedes *d*. When leaving *S1*, in contrast, exit actions of substates are executed first. This means that *f* precedes *c*. In the meantime, *b* and *e* are executed in parallel²⁵.

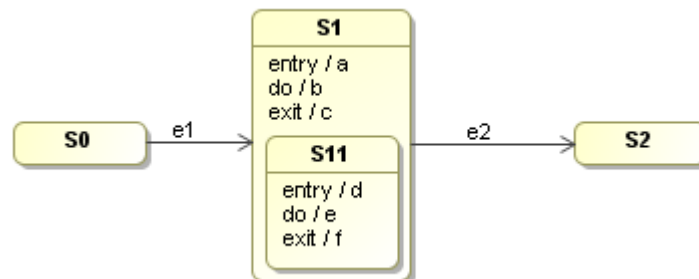


Figure 7. Action/activity execution order when nesting.

In terms of ordering, if we consider the following sequence of event occurrences²⁶: *:e1::e2*, we have the following execution: *a;d;(b||e);f;c*. This order complies with the execution semantics of UML (see also Section 11). Nevertheless, an ambiguity remains. In other words, one may wonder if the following order is tolerable: *a;d;((e:f)||b);c*? This means that if the *b* activity lasts, while *e* does not, then one leaves *S11* and thus activates *f*. However, the execution of *c* is suspended until *b* is complete. Roughly speaking, do we treat the exiting of a composite state like *S1* as a whole or do we have a discrete view of the exiting of each of its substates?

²³ This is equivalent for exit actions.

²⁴ In practice, this often corresponds to independent actions in that they do not use the same resources.

²⁵ From a modeling viewpoint, we do not encourage having activities like *b* in composite states like *S1*. The best is to have activities like *e* only in leaf states like *S11*. The reasons are simplicity and the avoidance of possible deadlocks if, for instance, *b* and *e* access concomitantly to the same resources.

²⁶ A colon preceding an event name embodies an event occurrence of the event type. The colon is used in UML to distinguish instances from types.

In *PauWare*, the second possibility is implemented in the engine (version 1.1) but it is, for the moment, inhibited. It may be later activated as a semantic variation point. This orientation would allow us to choose between the UML semantics $(a;d;(b||e);f;c)$ and this one: $a;d;((e:f)||b);c$.

4.5 Event consumption principle

A run-to-completion cycle²⁷ is in essence bound to the processing of a single event/request occurrence. This event is typed by its name, or better by its signature (if any) in a state machine diagram. **Event occurrences are not shared by state machines** (however, see Section 9 in which a mechanism of state machine composition creates an exception to this principle). Moreover, as many occurrences as needed of a given event type must be sent, if several components have to be informed of (and thus have to process) the event at a given time. For example, two instances of the *PauWare component* type may exist at a given time and thus two state machines (Figure 8) as well. If present, an occurrence of the *go* request is assigned²⁸ to one and only one state machine among the two in Figure 8. In other words, a given event occurrence cannot generate two effects. This means that this occurrence is consumed once and for all by one and only one state machine.

This consumption model enables the distribution of state machines, especially when they are encapsulated in distributable software components, like EJBs for instance (see Section 7), which run on distinct deployment nodes. This rule is also relevant for mobile systems (see Section 6) in which components, and thus state machines, are embedded. Sharing events is conceptually appealing for theoretically computing models but unlikely in practical applications because of the degree of centralization/synchronization that is required.

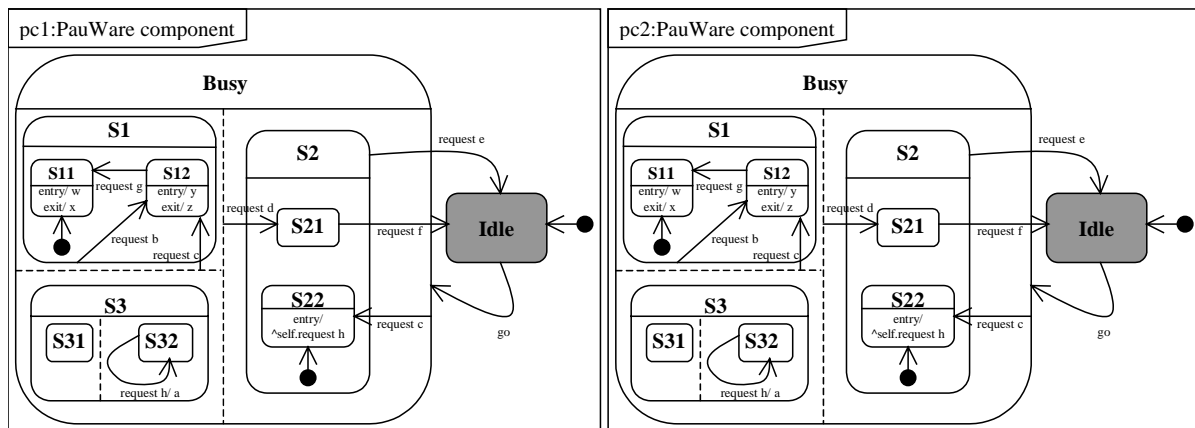


Figure 8. Two instances of *PauWare component* (*pc1* and *pc2*) with both the *Idle* state active.

As a result, in Figure 9, a single occurrence of the *go* event has moved the state machine on the left hand side of the figure to a new set of states, while that on the right hand side has not changed.

²⁷ The word "step" is used in the UML documentation.

²⁸ In general, this happens through the fact that the sender sends the *go* event to one component of the two that are supposed to consume this event.

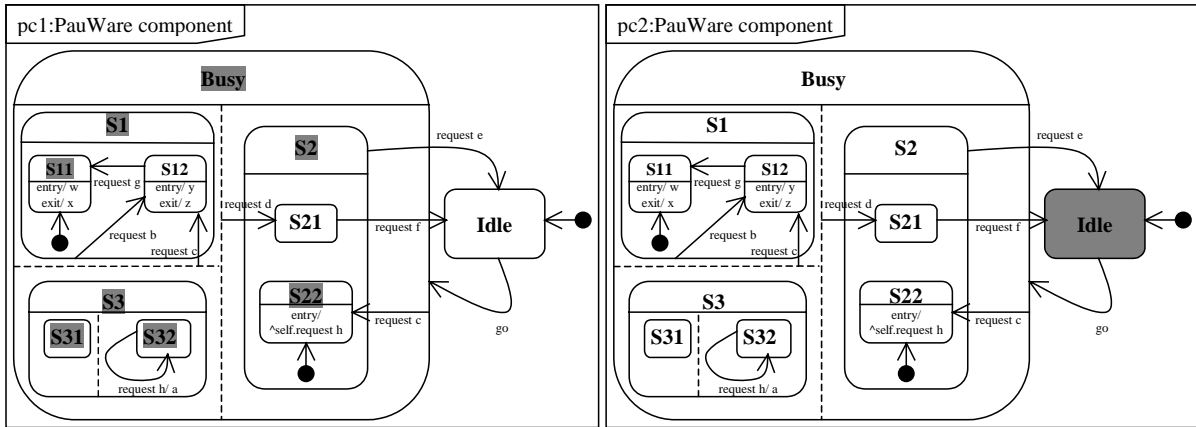


Figure 9. Two instances of *PauWare component* (*pc1* and *pc2*) after the processing of only one occurrence of the *go* event.

The execution semantics are therefore based on the principle that event occurrences are partially defined/identified through the unambiguous identity of their unique expected receiver (*e.g.*, the component which has the state machine on the left hand side of Figure 9). The principle relating to the fact that senders and receivers have identities and that events carry these identities, complies with UML (see Section 11).

4.6 Deferred events

Now, we consider Figure 9 and imagine a second occurrence of the *go* event. Like the first occurrence, it is sent to the state machine on the left hand side of Figure 9. Since the *Idle* state is not active, this occurrence of *go* is considered as “lost”. It produces no effect at all because its destination is the state machine on the left hand side of Figure 9; it cannot generate effects on the state machine on the right hand side of Figure 9.

In fact, within *PauWare*, events cannot be processed in a deferred way. However, an expert reader may notice that events may be received while a state lasts, especially when it has an associated activity (*do/* notation) which is not terminated. This case is **not** event deferring. At the event’s arrival time, if the state still lasts, the received event is not lost and thus processed as soon as the state’s activity is completed. Once again, this is nothing but compliance to the run-to-completion execution model of UML 2.

The default execution semantics of the UML 2 is the proscription of deferred events. However, one may optionally setup states so that they accept deferred events²⁹. This mode is not supported by *PauWare* (see also Section 11).

4.7 Transition conflicts

Within each run-to-completion cycle, *PauWare* may detect conflicting transitions³⁰. A conflict management policy is then provided.

4.7.1 Transition overriding

Within an execution cycle, for an eligible transition in a state machine labeled with a *r* request (see Figure 10) or any lower-level eligible conflicting transition (*i.e.*, a transition labeled with the same *r* request), inhibits the higher one. Some authors often consider this to be a key difference between UML and the original *Statecharts*. This may be summarized by the following rule extracted from the UML documentation: “(...) nested states override enclosing states.”

²⁹ Unfortunately, as far as we know, there is no dedicated notation.

³⁰ There are also potential conflicts between transitions and allowed events (see Section 5.5).

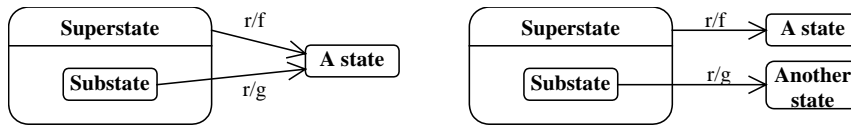


Figure 10. Two a priori conflicting transitions within the same execution cycle.

In terms of execution, if *Substate* is active (and thus *Superstate* is also active) and an occurrence of *r* exists, then only the *g* action is executed. So, ***f* is not launched even if *Superstate* is also active and *r* occurs** because the transition labeled *r/g* hides *r/f*. This mechanism is similar to overriding in object-oriented programming.

For the specific case modeled in Figure 10, the chosen execution semantics of *PauWare* complies with UML (see Section 11 for further details). However, the case in Figure 10 leads to two model variants. Indeed, the target states of the two conflicting transitions may be different (see the right hand side of Figure 10). In this case, the transition labeled *r/g* is triggered and *Another state* is the new active state (a). In contrast, the two end states of the two “conflicting” transitions may be the same state (see the left hand side of Figure 10). If so, one may imagine execution semantics which run both *f* and *g* in an arbitrary order since the computing of the new stable context of the state machine (the state called *A state*) is decidable (b). In other words, the two transitions are, no longer or not “really”, conflicting. In fact, the latter execution semantics (b) are neither supported by UML nor by *PauWare* (see Section 11). So, even if the two end states of the two conflicting transitions are the same (see the left hand side of Figure 10), only one transition among the two is triggered. So, **the overriding principle continues to apply**: the *g* action is executed and *A state* is the new active state for the model on the left hand side of Figure 10. We show more subtle cases of overriding below.

4.7.2 Guard-based conflict

Guard conflicts are quite common situations since logical expressions embodying guards are usually based on the internal data of software components or the external data brought by events when they occur. Model checkers may point out ill-fitted models like that in Figure 11 (the ambiguity is when $i = 0$) but many guard conflicts can only be detected at runtime (see Figure 12, Figure 13, Figure 14, Figure 16 and Figure 18 for instance).

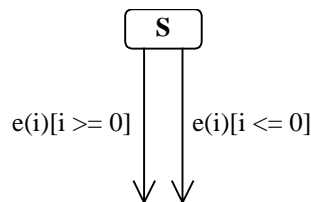


Figure 11. Indeterministic model.

Guard conflict management in *PauWare*

Figure 12 contains a state machine expressing the behavior of a *X* software component. We discuss potential problems resulting from the Boolean values of $g1$, $g2$ and $g3$. These three guards are associated with the same event named *e*.

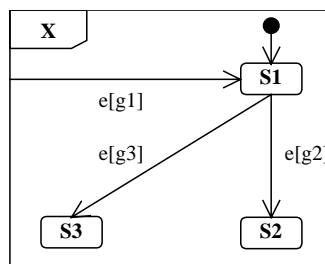


Figure 12. The behavior of a *X* software component.

The implementation in *PauWare* is as follows:

```

public class X {

    protected AbstractStatechart _S1;

    protected AbstractStatechart _S2;

    protected AbstractStatechart _S3;

    protected AbstractStatechart_monitor _X;

    ...

    protected void init_behavior() throws Statechart_exception {

        _S1 = new Statechart("S1");

        _S1.inputState();

        _S2 = new Statechart("S2");

        _S3 = new Statechart("S3");

    }

    protected void start() throws Statechart_exception {

        _X = new Statechart_monitor((_S1.xor(_S2)).xor(_S3),"X",true);

        _X.fires("e",_X,_S1,this,"g1");

        _X.fires("e",_S1,_S2,this,"g2");

        _X.fires("e",_S1,_S3,this,"g3");

    }

    ...
}

```

In the state machine diagram of Figure 12, one only has a conflict if *S1* is active and if *e* occurs while both *g2* and *g3* are true. According to the demonstration provided in the beginning of Section 4.7, if *g1* is true, its concerned transition from *X* to *S1* no longer conflicts with the two ones guarded by *g2* and *g3*: the overriding rule applies. In fact, *X* is the direct superstate of *S1*. Roughly speaking, *S1* is a nested state while *X* is its immediate enclosing state. This leads to the conclusion that, if *S1* is active and *e* occurs while both *g1*, *g2* and *g3* are true, the transition with *e[g2]* “hides” the one with *e[g1]* and the transition with *e[g3]* also “hides” the one with *e[g1]*.

The verbose mode of *PauWare* (see Section 3.1.6) enables the tracing of overridden transitions (they are printed on the screen or in a log file). Overridden transitions do not cause fatal errors within a run-to-completion cycle. However, having overridden transitions may sometimes result from bad modeling, namely if, like in Figure 12, *g1* always implies *g2*. In this case, the state machine diagram is “logically incorrect” but may be nevertheless executed without damage.

In Figure 12, there is therefore a serious conflict if *S1* is active and if *e* occurs while both *g2* and *g3* are true. In *PauWare*, conflict management amounts to raising an instance of a typed exception called *Statechart_transition_based_exception*. The run-to-completion cycle is abruptly interrupted and no transition is triggered at all. The main point is that **there is no side-effect**. In other words, the state machine remains in a consistent stable configuration, which is the immediate configuration before *e* occurs. So, from a programming viewpoint, the raised exception may be caught and the state machine may go on functioning. This depends on if one expects/ensures that *g1* and *g2* return to distinct values the next time *e* occurs.

To go beyond simple conflict management, the fact that both *g2* and *g3* in Figure 12 may be true at the same time is a tricky problem because the state machine is able to reach two exclusive states, which is an insolvable problem. In other situations, the destination states may have relationships which are different from those of exclusiveness.

Transitions with the same origin and the same end

In Figure 13, two different actions ($a1$ and $a2$) are correspondingly attached to the e event. While $a1$ is linked to the $g1$ guard, $a2$ is linked to $g2$. If $(g1 \Rightarrow g2) \vee (g2 \Rightarrow g1)$, this highlights a modeling error. However, this error cannot always be detected, especially outside of a runtime context. From a modeling viewpoint, the $g1$ and $g2$ guards are usually two symbolic expressions (written in OCL for instance) for which one expects/ensures that they are not true at the same time.

Therefore in Figure 13, one has a conflict if $S1$ is active while e occurs and both $g1$ and $g2$ are true. Having this situation at runtime in *PauWare* leads also to throwing a *Statechart_transition_based_exception* instance. A key issue is that neither $a1$ nor $a2$ have been executed. Moreover, contrary to the model in Figure 12, there is no conflict about the state to be reached: for both transitions it is $S2$. Nevertheless, one cannot solve the problem by triggering one of the two transitions (or both) because one should then arbitrarily choose between the execution of $a1$ or (inclusively) $a2$. Besides, UML rejects such options since it advocates the necessity of triggering one and only one transition when they share the same source especially.

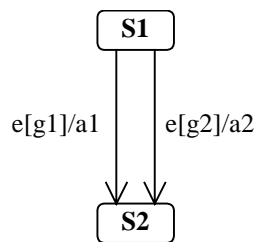


Figure 13. Two possible conflicting transitions.

Transitions with orthogonal states as destinations

The state machine diagram which formalizes the behavior of a software component called Y in Figure 14 shows, once again, a situation in which the $g2$ and $g3$ guards are a potential source of conflict. If $S1$ is active and e occurs while $g2$ and $g3$ are true, the two terminal states ($S2$ and $S3$) of the two enabled transitions can be potentially attained due to the fact that they are orthogonal.

The execution semantics of UML is clear about this issue: this case is a “true” conflict. The conflict does not rely on the two target states which are mutually compatible. In UML, the rule to comply with is that one transition at the most must be triggered among the two that are leaving $S1$ in Figure 14. So, if $g2$ and $g3$ are true at the same time, triggering only one transition among the two conflicting ones is based on expressed priorities. This is an option in UML³¹ which is not supported by *PauWare*. Thus, in the most common case (*i.e.*, no priority is setup), one might interpret the execution semantics of UML as follows: the state machine remains in $S1$. From the UML viewpoint, one indeed cannot execute the b and a actions because choosing between $g2$ and $g3$ is impossible (they are both true, no priority is setup).

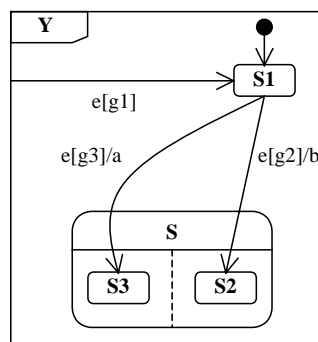


Figure 14. The behavior of a Y software component.

PauWare does not consider the above described situation as a systematic conflict and thus views the execution semantics of UML as controversial. This orientation results from the model piece in Figure 15, which is a common modeling expectation of modelers (see also the *view program* event on the right hand side of

³¹ Once again, as far as we know, noting priorities in diagrams is not yet instrumented (by stereotypes or native modeling constructs).

Figure 34). To sum up, for the model appearing in Figure 14, both transitions are able to be fired ($g2$ and $g3$ being true) leading to the possible concurrent execution of a and b and their arrival in S (i.e., in $S3$ and in $S2$). This process is an alternative execution semantics to that of UML which is here reminded: the state machine remains in $S1$ even if $S1$ is active and e occurs while $g2$ and $g3$ are true.

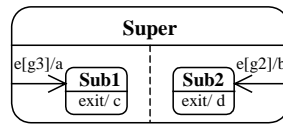


Figure 15. Common modeling practice with Statecharts.

In our opinion, UML restricts all of the modeling capabilities of the original *Statecharts* without relevant justification. In Figure 15, the single significant difference with Figure 14 is the fact that the two “conflicting” transitions are from *Super* to *Sub1* and from *Super* to *Sub2* with *Super* being the superstate (whether direct or indirect is of no importance) of *Sub1* and *Sub2*. Applying the UML execution semantics leads to considering the model in Figure 15 as a source of conflict, if $g2$ and $g3$ are true when e occurs. However, the model pattern in Figure 15 is common. For example, the a and b actions may correspond to the refreshing of distinct display elements within a MVC interaction. In this case, $g2$ and $g3$ may determine if the data to be displayed has changed. So, in *PauWare*, the two execution sequences $c;a$ and $d;b$ are run accordingly if $g2$ and $g3$ hold.

In *PauWare* more generally (see also Figure 34 for a concrete business example), we do not consider the models of Figure 14 and Figure 15 as systematic conflict sources. However, for UML compatibility reasons, the *PauWare* engine implements the following semantic variation point: **the model in Figure 14 is viewed and thus managed as a transition conflict**. Namely, if $S1$ is active and e occurs while $g2$ and $g3$ are true, a *Statechart_transition_based_exception* is raised. This occurs because $S1$ is not the superstate of $S2$ and $S3$. In contrast and in opposition with the current execution semantics of the UML, the model piece in Figure 15 is never viewed as a case of transition conflict when *Super* is active and e occurs while both $g2$ and $g3$ are true.

Transitions with nested states as destinations

In Figure 16, there is a slight variant of the state machine diagram of Figure 12. Moreover, it must not be confused with the state machine diagram in Figure 14. In Figure 16, the target states of the transitions $S1 \rightarrow S2$ and $S1 \rightarrow S3$ are nested and are thus always active at the same time. So, if $S1$ is active and e occurs while $g2$ and $g3$ are true, *PauWare* views these two transitions as incompatible. This results in raising an instance of *Statechart_transition_based_exception* and $S1$ remains active with no side-effect(s).

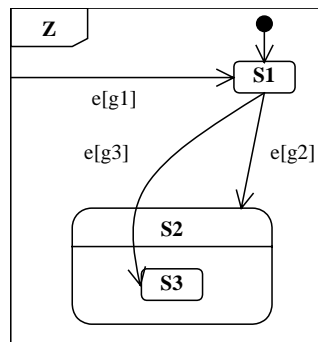


Figure 16. The behavior of a Z software component.

What the models in Figure 14 and Figure 16 have in common is that, in Figure 16, $S2$ and $S3$ do not constitute the source of the conflict. However, the model in Figure 16 can be, with some additional elements, a source of other tricky problems. For instance, non decidable situations may arise if there is another default input state (see $S4$ in Figure 17). In this case, state machines truly become indeterministic at runtime, if $g2$ and $g3$ are both true at the same time.

To sum up, in Figure 16, the two target states of the two conflicting transitions ($S2$ and $S3$) are “compatible” despite UML rejects the possibility to trigger both transitions. In contrast, in Figure 17, $S2$ and $S3$ are “incompatible”: reaching $S2$ means reaching $S4$ which has an exclusiveness relationship with $S3$.

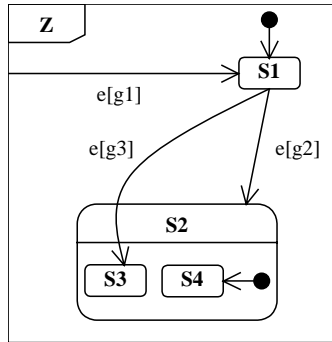


Figure 17. Variant of the behavior of the Z software component in Figure 16.

Transitions with orthogonal states as origins

The state machine diagram in Figure 18 is another subtle example. It has a weak relationship to the prior ones since it illustrates a special case of conflicting transitions. Contrary to what happened before, a “correct” behavior (*i.e.*, the absence of a conflict) can be observed when $g1$ and $g2$ are both true at the same time. This results from the fact that $S2$ and $S3$ are orthogonal states; this means that leaving $S2$ requires leaving $S3$. If S (*i.e.*, the superstate of $S2$ and $S3$) is active and e occurs while $g1$ and $g2$ are both true, then $a1$ and $a2$ are activated in an arbitrary order. This is the most typical situation.

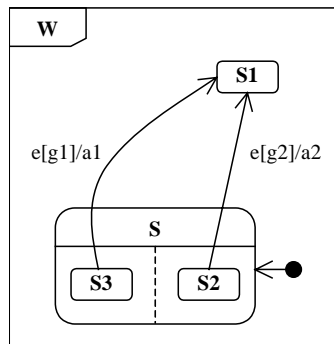


Figure 18. The behavior of a W software component.

A variant is when $g1$ is equal to true and $g2$ is equal to false ($g1 = false$ and $g2 = true$ is a dual problem). In this case, this eventually also leads to reaching $S1$. We have chosen a minimal criterion based on the logical *or*. This means that one eligible transition at the most is necessary for leaving S . However, if $g1$ is equal to true and $g2$ is equal to false then only $a1$ is activated: $a2$ is bypassed. There are no exit actions associated with $S2$ and $S3$ in Figure 18. More generally, if there were any, all would be executed since $S2$ and $S3$ are no longer active.

This execution semantics choice may be considered as controversial, since one may argue that having $g1$ be true and $g2$ be untrue (or the contrary) while an occurrence of e is present, is a runtime error. In practice, if a modeler is aware of the possible effects of her/his models, the risk is low. Furthermore, a good idea is to guarantee that the modeler is not restricted in terms of expressiveness. In this optic, our solution enables the execution exclusion of $a1$ or (inclusively) $a2$. Besides, the state machine diagram in Figure 19 may be used instead. However, one may note that the transition in Figure 19 cannot be concomitantly used with the two other ones in Figure 18. This is due to the policy for transition overriding supplied by UML and *PauWare*.

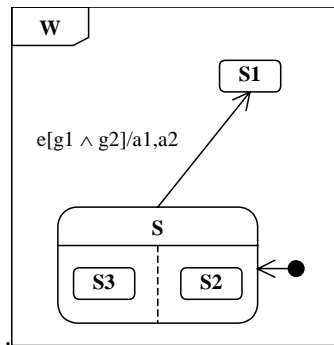


Figure 19. An alternative state machine diagram for the *W* software component in Figure 18.

5 Advanced programming with *PauWare*

This section describes additional services offered by *PauWare*, namely the UML notion of “completion transition” and the use of timer services, which are recurrently used in state machine-based programming. We also provide a support and a description for advanced modeling constructs. This corresponds to multicast communication and the notion of “allowed event”. In this section, we deal with tricky cases of state nesting and provide guidelines about how to manage cached and runtime transitions, including their efficient and optimal usage. We finally close this section by discussing the concurrent use of components that have embedded state machines. We also describe a support for state invariant setup and evaluation which leads to autonomic computing facilities.

5.1 Completion transition

In UML, completion transitions are unlabeled transitions. These are often (but not always) assigned as outgoing transitions of states which have activities (*do/* notation). In Figure 20, the transition from *Working* to *Idle* has no event as a label. So, one moves from *Working* to *Idle* when the *activate device* activity terminates.

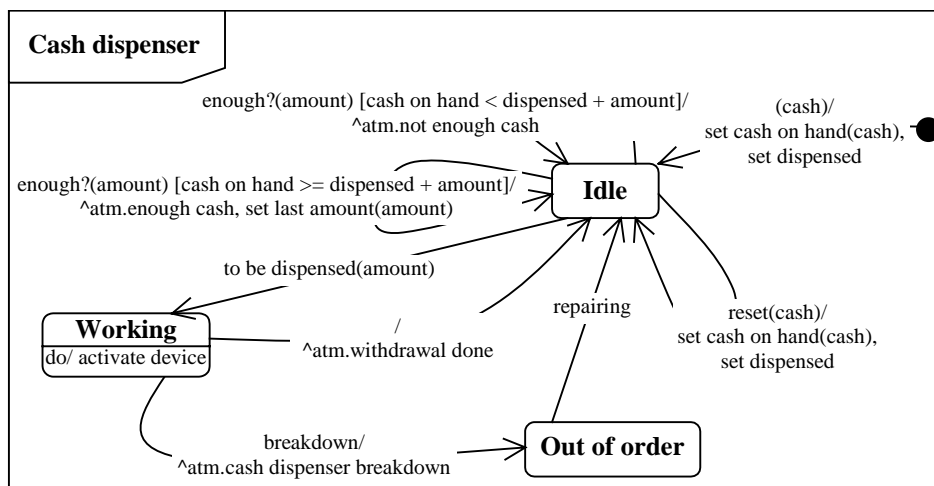


Figure 20. Behavior of a *Cash dispenser* software component (ver. 1).

Since *PauWare* does not support the implicit management³² of completion transitions, a revision of the state machine diagram in Figure 20 is proposed in Figure 21. An explicit completion event named *withdrawal realized* is introduced and sent to the component itself: *^self.withdrawal realized*. The run-to-completion mode adopted by *PauWare* guarantees its deferred processing so that any state machine conforming to the state machine diagram in Figure 21, consistently moves to *Idle* as soon as the *activate device* activity finishes.

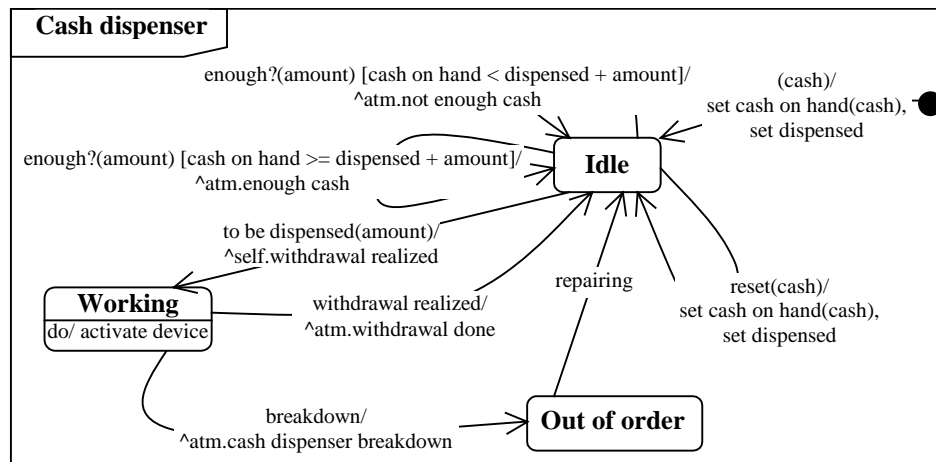


Figure 21. Behavior of a *Cash dispenser* software component (ver. 2).

The code associated with the transition from *Working* to *Idle* absolutely requires the use of the *AbstractStatechart.Broadcast* constant class variable (see Section 4.1) as follows:

```
_Card_dispenser.fires("to_be_dispensed",_Idle,_Working,true,this,"withdrawal_realized",null,AbstractStatechart.Broadcast);
```

5.2 Timer services

When modeling reactive systems, a key event type is “time-out”, meaning that the behavior of a software component is governed by strict time constraints like doing something each second, etc. Distributed component platforms provide appropriate timer services (*e.g.*, the *EJB Timer Service*) but, in modeling, sophisticated timer facilities are especially needed in conjunction with state machine diagrams. Such advanced facilities are rarely available in runtime platforms and often require some extra implementation. For this reason, *PauWare* pre-implements timer services so that models subject to time constraints have direct and obvious mapping in *PauWare*³³.

So, *PauWare* provides a set of common timer services based on the Java ME/Java SE/Java EE-compliant *java.lang.Timer* and *java.lang.TimerTask* predefined classes. However, *PauWare*’s timer services hide these two classes and are adequately correlated with state machines so that component states are able to play an important role in the management of timer events (see examples below).

A typical use of timer services in *PauWare* (Java SE and Java EE) leads to inheriting from the *_Composytor::Timer_monitor* abstract class (Figure 22). Figure 22 illustrates a *Card reader* software component of an banking system that requires timer services in order to eject or swallow cards after fixed periods. Because the *_Composytor::Timer_monitor* class uses the reflection capabilities of Java, using timer services in *PauWare* for building Java ME-compliant applications requires inheriting directly from the *_PauWare::AbstractTimer_monitor* abstract class (see also Section 6.3.6). In fact, *_Composytor::Timer_monitor* is mandatory with Java EE while *_PauWare::AbstractTimer_monitor* is mandatory for Java ME. As for Java SE, both classes may be used indifferently.

³² In UML, completion events are automatically generated by the “state machine control system”, especially at the end of activities, in order to automatically trigger completion transitions.

³³ Here, we do not cover the *UML Timing Diagrams* which are worthless in our opinion.

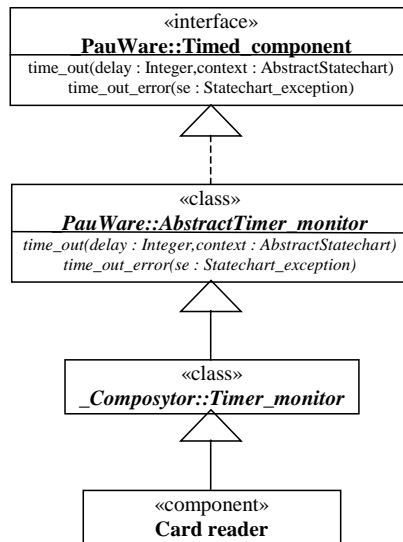


Figure 22. PauWare's timer services and their use.

Based on the structuring principle exposed in Figure 22, the *Card reader* software component is able to ask for timer services (*to_be_set* primitive), to inhibit timer services (*to_be_killed* primitive) and to manage conflicting transitions (see Figure 23 about two conflicting transitions having the "time-out" label). Finally, to lose its abstract nature coming from *_Composytor::Timer_monitor*, *Card reader* must implement the *time_out* and *time_out_error* abstract inherited routines. The code corresponding to the *Card reader* business component may therefore look like:

```

public class Card_reader extends Timer_monitor {
    ...
    protected void init_behavior() throws Statechart_exception {
        ...
        _Ejecting = new Statechart("Ejecting"); // moment when a banking card is ejected
        Object[] args = new Object[2];
        args[0] = _Ejecting;
        args[1] = new Long(30000);
        _Ejecting.entryAction(this,"to_be_set",args); // a timer starts when entering into Ejecting
        args = new Object[1];
        args[0] = _Ejecting;
        _Ejecting.exitAction(this,"to_be_killed",args); // the timer stops when leaving Ejecting
        ... // other state declarations and setups here
    }
    ...
    public void time_out(long delay,AbstractStatechart context) throws Statechart_exception {
        // after 30000 milliseconds, this method is called
    }
}

```

```

// some runtime transitions, if any, here



```

Timers may be setup by stipulating delays, or delays with specific contexts, by means of the following primitives (*Long* instead of *long* may also be used):

```

public void to_be_set(final long delay) throws Statechart_exception; // no context setup

public void to_be_set(final AbstractStatechart context,final long delay) throws Statechart_exception;

```

The second form requires a dual *to_be_killed* primitive:

```

public void to_be_killed(final AbstractStatechart context);

```

Several timers may be used by the same component and each may be associated with a context. In fact, a context corresponds to a state, it is an instance of *AbstractStatechart*. Timers run cyclically until the *to_be_killed* primitive is called. Each context must then be “closed”, if the component wants to stop all of its previously requested timer services. Moreover, a timer associated with a context cannot be reconfigured before it has been stopped via *to_be_killed* or by using the *to_be_reset* primitive. This primitive accepts the same argument types as *to_be_reset* or *to_be_killed*.

5.2.1 Coupling between state machines and timer event services

Software components may ask for timer event services for many distinct reasons in many different ways. Examples are one-shot timer events (like for the above *Card reader* component in the context of the *Ejecting* state), or cyclic events (*e.g.*, “each second” without duration limit, etc). As a result, an emergent problem is when a component receives a time-out event occurrence without knowing why it receives it or what the context was for when a timer was launched and what to do “now” (see as an illustration the model in Figure 23).

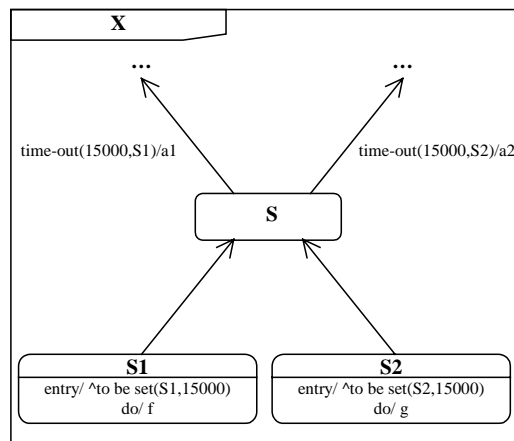


Figure 23. Conflicting transitions based on time-out events.

In Figure 23, two timer services are requested³⁴; the first one when entering into *S1* and the second one when entering into *S2*. So, in the *S* state³⁵, an instance of the *X* software component is subject to receive different variants of time-out occurrences depending upon the contexts in which timer services have been requested. In *PauWare*, the added value is its capacity to test contexts (*i.e.*, states in which timer services are requested). In Figure 23, *S1* and *S2* are parameters of the two received time-out event occurrences. They implicitly play the role of guards, in order to establish where to go when leaving *S*. The implementation is as follows³⁶:

```
public void time_out(long delay,AbstractStatechart context) throws Statechart_exception {
    boolean guard = context == _S1;
    _X.fires(_S,...,guard,this,"a1"); // runtime transition form
    guard = context == _S2;
    _X.fires(_S,...,guard,this,"a2"); // runtime transition form
    _X.run_to_completion("time_out");
}
}
```

Another recurrent problem linked to timer services is the possibility of managing cycles in state machines. For a given active state, say *S* in Figure 24, many timer services may have been requested in the past. Time-out event occurrences may thus be distinguished by their different configured contexts or distinctive configured delays. So, like contexts, delays may be tested by means of guards, but it is insufficient when both contexts and delays are equal (see an example in Figure 24 when leaving *S*).

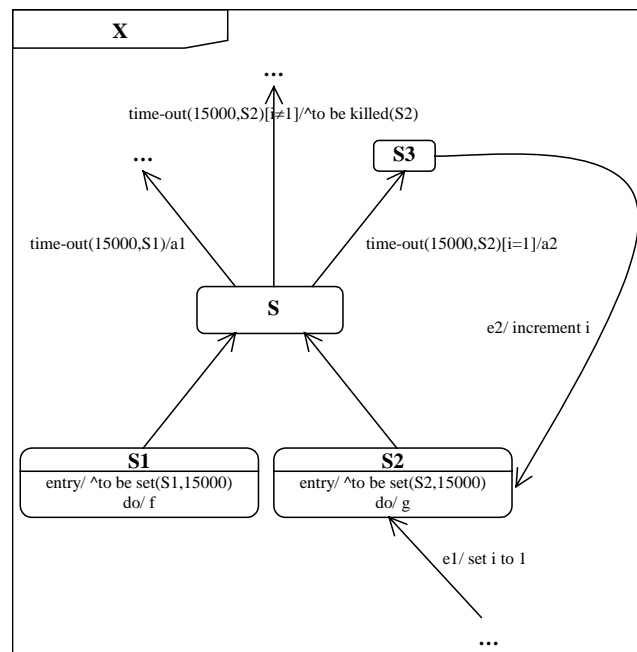


Figure 24. Cycle-oriented conflicting transitions based on time-out events.

³⁴ The *to_be_killed* primitive can accept a state as a parameter (see also Figure 24) in order to kill one required service, while the other (attached by definition to another state) goes on functioning.

³⁵ In the model of Figure 23 and Figure 24, one automatically reaches the *S* state as soon as *f* is completed (in this case, *S1* is active) or as soon as *g* is completed (*S2* is active). This is the concept of "completion transition" in UML (see Section 5.1).

³⁶ Remember that with this implementation and in coherence with the model in Figure 23, one time-out event occurrence among the two ones arriving at *S*, will probably be lost based the event consumption principle exposed in Section 4.5.

In Figure 24, a counter named *i* may be used. This amounts to data to be later tested in guards. Therefore, this data is used in guards to establish the appropriate path in the graph. An example is seen when reaching *S3* and executing the *a2* action if *i* is equal to 1, or reaching another state and killing the timer service associated with the *S2* state, if *i* is not equal to 1. Moreover, when *S3* is active and *e2* occurs, one re-enters into *S2* and thus requests timer services again with the *S2* context. In *PauWare*, **no reset occurs**. This means that the timer associated with the *S2* context goes on working with the initial fixed delay (15000 milliseconds in Figure 24). So, since the count down process is not restarted, the behavior depicted in Figure 24 greatly depends upon the intrinsic duration of the *g* activity in *S2*. So, time-out event occurrences in particular may be lost when in *S2*, but incrementing *i* (by means of the *increment i* action when *e2* occurs) guarantees that the transition labeled *time-out(15000,S2)[i≠1]/^to be killed(S2)* will be at least triggered. The point is that the 15000 delay is here, more indicative than prescriptive.

In fact, this case study shows that tricky situations may exist and the need for special state machine diagram variables like *i* is compulsory if cycles must be managed.

5.2.2 Enhanced timer services

For the moment, the *PauWare* API for timer services is rudimentary, except for its coherent link with state machine-based programming. However, we do offer the possibility of establishing time durations³⁷, dates and so on. Software components must thus manage durations or other equivalent timing constraints in order to kill timers at accurate desired moments. However, such extensions may be easily, straightforwardly and rapidly integrated in the *PauWare* engine and can be a reasonable issue to be addressed in a forthcoming version of *PauWare*.

5.3 State and event naming

States are named at instantiation time (see Section 3.1) but *PauWare* **does not check the uniqueness of a state name** within an overall state machine. Having two or more different instances of the *AbstractStatechart* class with the same name within a state machine may cause the malfunctioning of this state machine. *PauWare* developers must therefore handle state names with great care. Moreover, states cannot be named by using the string value “pseudo-state”. This is the value of the *Pseudo_state* constant and public class variable of the *AbstractStatechart* class. In special cases, this constant may be used (see Section 5.4).

In the same way, events cannot be named by using the string value “pseudo-event”. This is the value of the *Pseudo_event* constant and public class variable of the *AbstractStatechart* class. The name of an event appears as a first parameter of the *fires* method. This means that the declared transition is associated with this event. The name of an event also appears as a first parameter of the *run_to_completion* method to find out and trigger all of the cached transitions which were previously associated with this name by means of *fires*.

In ver. 1.2³⁸ of *PauWare*, the composition of state machines is supported (see Section 9). This means that the *and*, *xor* and *nesting* operators may not only be used for instances of the *AbstractStatechart* class, **but also for instances of the *AbstractStatechart_monitor* class**. In this scope, the constant and public class variable named *State_name_separator* of the *AbstractStatechart* class is equal to “::”. States must therefore not include this string in their name since an assembly of two or more composed state machines will be handled by using “::”, for instance:

```
if(_Assembly.in_state("X::A")) ...;
```

In this code, one supposes that “X” is the name of an instance of the *AbstractStatechart_monitor* class. This state machine has been composed with another one leading to an augmented version of the *_Assembly* state machine (an instance of the *AbstractStatechart_monitor* class as well). Moreover, “A” is the name of a state of the “X” state machine. If “A” is also the name of a pre-existing (*i.e.*, before composition time) state of the *_Assembly* state machine, one needs a means to distinguish the two states that are named “A” in *_Assembly*. In this case, “::” enables the distinction between the two states sharing the same name from an assembly viewpoint.

To close this section, the reader must be aware that the memory-based organization of a state machine (see Section 10) is such that pseudo-states (*i.e.*, states that possess the “pseudo-state” name) are automatically created, but are not intended for direct use by a developer. As a result, such a search makes no sense:

³⁷ Here we mean launching a timer for five minutes. Thus, we need not kill it by sending out time-out events each second.

³⁸ The next release of this users’ guide will deal with ver. 1.2 which is not explained here.

```
if(_A_state_machine.in_state(AbstractStatechart.Pseudo_state)) ...;
```

5.4 Special cases of nesting

This section deals with a situation in which one wants to have only one state as a substate of an immediately surrounding state (Figure 25).

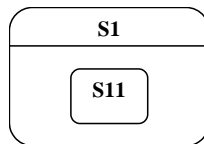


Figure 25. Only one direct substate within a given state.

The required code is as follows:

```
AbstractStatechart _S1;
```

```
AbstractStatechart _S11;
```

```
...
```

```
protected void init_behavior() throws Statechart_exception {
```

```
    _S11 = new Statechart("S11");
```

```
    // a fictitious pseudo-state is required as well as an orthogonality relation between S11 and the fictitious
```

```
    // pseudo-state as follows:
```

```
    _S1 = _S11.and(new Statechart(AbstractStatechart.Pseudo_state)).name("S1");
```

```
}
```

5.5 Allowed event

An allowed event is an event which does not trigger any transition. This concept has been formally defined in a book written by Steve Cook and John Daniels about the Syntropy specification method (see Bibliography). An allowed event is associated with a given state and, **by definition**, bypasses the entry and exit actions of this state. In UML, such a concept is named “internal action”³⁹ since an allowed event in a state (many allowed events may however appear) is connected with an action to be executed when this event occurs and the said state is active. Otherwise, without associated action, an allowed event is useless. Throughout this users’ guide, for clarity we use the expression “allowed event” to the detriment of “internal action”. The latter is both unclear and confusing since actions are in essence internal to components; whereas events are members of the components’ provided interfaces.

In Figure 34, a complex state machine diagram includes a *Setup* state (see the top, right hand side of the figure) which accepts (*i.e.*, “allows”) a *time-out* event. This event is guarded and launches a *set no input* action with a few arguments. The implementation in *PauWare* is as follows (see also Section 6.3.2):

```
_Setup.allowedEvent("time_out",this,"no_input_less_than_ninety",this,"set_no_input",args);
```

This code statement must appear in the Java method which embodies the *time-out* event because the *set no input* action requires contextual data assigned to the *args* variable. Instead, when possible⁴⁰, such a declaration tends to appear in the *init_behavior* method of a component (see Section 3.1).

³⁹ The “internal transition” expression is also used for characterizing the effect of allowed events.

⁴⁰ *i.e.*, no contextual data is required.

The modeling heuristics associated with the notion of “allowed event” deals with its attachment state. If the latter is active at the time the allowed event occurs, it remains active **and no transition is fired**. Having a formalism dedicated to allowed events, like the one used in Figure 34 or Figure 26 (see the inside of *S1* or *S11*), precludes for creating fictitious states and unintelligible transitions that do not correspond to the business logic. In other words, allowed events have to be processed, without impact on the course of a state machine. This is the case of the *time-out* event for the *Setup* state belonging to the model in Figure 34.

However, allowed events may be the source of conflicting transitions (Figure 26): “An internal transition in a state conflicts only with transitions that cause an exit from that state.” [, p. 562]. Indeed, for a given state, an event can be declared both as “allowed” and as a label of an outgoing transition. Moreover, this phenomenon may be generalized for several states, which are or are not nested like *S1* and *S11* in Figure 26.

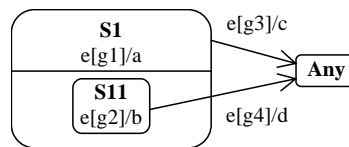


Figure 26. Conflicts linked to the use of allowed events.

Conflict problems in Figure 26 come from the possible values of the *g1*, *g2*, *g3* and *g4* guards. First, one may notice that the *e* event is an allowed event for both *S1* and *S11*. We consider that there is no conflict if *g1* and *g2* are true while *g3* and *g4* are false when *e* occurs. Because *S1* and *S11* are in essence active at the same time, one executes *a* and *b*. Moreover, *S1* and *S11* remain active. To avoid an arbitrary order when executing *a* and *b*, one performs the *b;a* sequence. This order is similar to the way by which exit actions of nested states are called (see Section 4.4). An alternative semantics is executing *b* while *a* is not executed (same conditions: *g1* and *g2* are true while *g3* and *g4* are false when *e* occurs). In our opinion, in the UML documentation, no policy is described even though these two text extracts exist: “For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire.” [, p. 562] and “(...) nested states override enclosing states.” [, p. 552]. In Figure 26, the two internal transitions have no true “origin state” since they have no extremities at all. So, both transitions are fired within the same run-to-completion step. This choice is inspired by the fact that *S1* and *S11* are “compatible”: *S1* and *S11* are always, by definition, both active or both inactive at the same time. Our semantics enables three kinds of execution: *a* ($g1 \wedge \neg g2$), *b* ($\neg g1 \wedge g2$) or *b;a* ($g1 \wedge g2$). The alternative semantics only allows the execution of *a* ($g1 \wedge \neg g2$) or *b* (*g2*). It thus seems poorer in terms of expressiveness.

An extended conflict case is when *e* occurs, *g1* and *g2* are true, and *g3* is true or *g4* is true or both. So, do we enter into *Any*? What is executed? From a model checking viewpoint, if one may statically prove that: $g1 \Rightarrow \neg g3 \wedge \neg g4$, $g2 \Rightarrow \neg g3 \wedge \neg g4$, $g3 \Rightarrow \neg g1 \wedge \neg g2 \wedge \neg g4$ and $g4 \Rightarrow \neg g1 \wedge \neg g2 \wedge \neg g3$, no problem will exist at runtime since one has definitively proven that state machines instantiated from the model in Figure 26 are deterministic. Unfortunately, such a symbolic computation is not always possible and a deeper conflict management policy is required at runtime. So, even if the behavior assigned to substates has priority over (or hides) the behavior assigned to superstates (see Section 4.7 and the second text extract above mentioned), one also needs to establish a priority between internal transitions and “external” transitions. In *PauWare*, for a given state, internal transitions have higher priorities over external transitions. Again, no strategy is provided by UML about this issue. Returning to the example in Figure 26, one may accordingly observe at runtime the following execution semantics:

- $g2 \rightarrow b$ (no situation change: *S1* and *S11* active)
- $\neg g2 \wedge g4 \rightarrow d$ (*Any* active)

The first clause shows that when *g2* is true, the execution of *d* is bypassed (*g4* being true or not). We have a dual situation for the (*g1*,*g3*) pair. This clause illustrates the priority of internal transitions over external transitions. The third clause illustrates the crossing of concerns between internal/external transitions on one side and, enclosing/nested states on the other side. If *g4* is true (*g1* being true or not), *a* is bypassed. This means that an external transition of a nested state has priority over an internal transition of an enclosing state. We adopt this priority type to be closer as possible to UML (*i.e.*, “(...) nested states override enclosing states.”).

In any case, we think that we obtain a robust modeling framework. Even if the choice for internal/external transitions is debatable, it can be easily managed through a stereotype, say «*InternalVersusExternal*», with two tagged values, say *internalOverExternal* (default) and *externalOverInternal*. This stereotype is either an extension of the UML *State* metatype or *StateMachine*. It depends upon the degree of desired precision.

which can be applied to

5.6 Unicast versus multicast

In reaction to events in a state machine, or when entering or exiting states, actions (but also activities) in essence use the internal methods of the component which owns this state machine. In such a context, the *this* Java keyword is used. This keyword is replaced by an appropriate reference, which points to one and only one object when one wants to call⁴¹ another local object or a remote component instance (see for instance Section 7 about remote communication). This is the unicast mode of communication. In contrast, the multicast mode of communication relies on a set of target entities, which have to process the same action with the same contextual arguments (if any). For that, *PauWare* requires the use of instances of Java arrays (*i.e.*, *Object[]*) or of the Java *java.util.Collection* class as follows:

```
protected java.util.LinkedList _a_list;

...

protected void init_structure() throws Statechart_exception {

    _a_list = new java.util.LinkedList();

    ...

}

protected void init_behavior() throws Statechart_exception {

    _Busy = new Statechart("Busy");

    _Busy.entryAction(_a_list,"stop");

    ...

}
```

In this code, it is important to understand that, when entering into the *Busy* state, the *_a_list* instance of *java.util.LinkedList* **does not** execute the *stop* action. Instead, each object in this list will execute the *stop* action. In terms of fault management, the *PauWare* engine will stop the execution as soon as one element in the list raises an exception during execution. As for the result of the global execution, only the execution relating to the last element in the list will be recorded for the tracing mode (see Section 3.1.6).

⁴¹ Here, we do not make a distinction between the fact that we simply call a method and the fact that the called method is itself considered and treated as an event by the receiver.

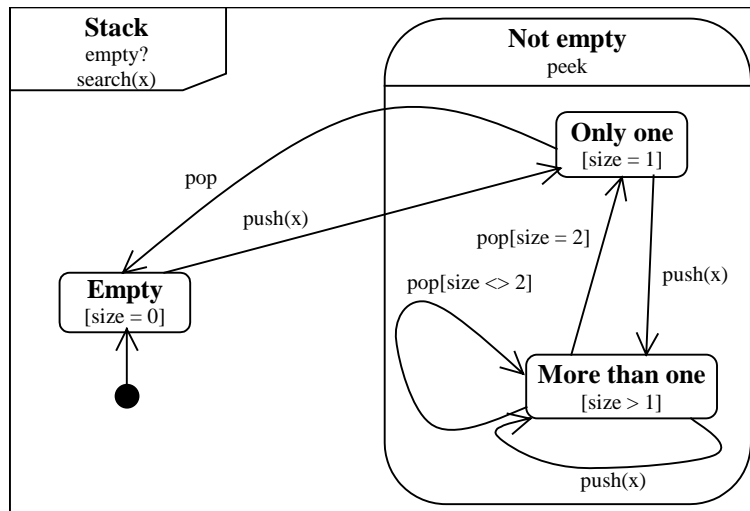


Figure 27. Specification of the behavior of a *Stack* software component.

A remaining problem is when one wants to effectively apply an action or an activity to an instance of the Java *java.util.Collection* class itself. For instance, if one constructs a *Stack* component based on a state machine diagram (Figure 27), one has to proceed as follows (see for example the *pop* function, which moves the state machine from *Only one* to *Empty*, if *Only one* is active when *pop* occurs):

```
protected java.util.Stack _stack;

protected AbstractStatechart _Empty;

protected AbstractStatechart _Only_one;

protected AbstractStatechart _More_than_one;

protected AbstractStatechart _Not_empty;

protected AbstractStatechart_monitor _Stack;

protected void init_structure() throws Statechart_exception {
    _stack = new java.util.Stack();
}

...

protected void start() throws Statechart_exception {
    _Stack = new Statechart_monitor(_Empty.xor(_Not_empty),"Stack",true); // tracing mode activated
    ...
    _Stack.fires("pop",_Only_one,_Empty,true,this,"_pop");
    ...
}

...

public Object _pop() {
    return _stack.pop();
}
```

```
}
```

Simply put, the last Java method named “_pop” is responsible for calling *pop* for the Java *Stack* instance named *_stack* because this code **does not work** with *PauWare*:

```
_Stack.fires("pop",_Only_one,_Empty,true,_stack,"pop");
```

In *PauWare*, this code means, applying *pop* to all elements in *_stack* (multicast), which probably results in a problem.

5.7 Cached transitions versus runtime transitions

Transitions are implemented by means of the *fires* method of the *AbstractStatechart_monitor* class (see also Section 3.2). The most efficient way of calling the *fires* method is to use it within the *start* method of a software component (see also Section 3.1.7). In such a case, the implemented transition is known as “cached”, meaning that it is recorded once and for all and associated with a given event. Example (see also Figure 2):

```
_PauWare_component.fires("request_h",_S32,_S32,true,this,"a");
```

5.7.1 When runtime transitions become mandatory

As written at the beginning of this document, doing so is only possible when the transition’s guard (if any) and the transition’s action(s) (*a* above; if any) do not require dynamical parameters (*i.e.*, parameters that are usually brought by the event that labels the transition). If so, the call of the *fires* method cannot occur within the *start* method. Instead, it must occur within the method embodying the event’s processing. See the example of the *Card reader* component which tends to react to a *card inserted* event:

```
public void card_inserted(Plastic_card plastic_card) throws Statechart_exception {  
  
    /** post-condition(s) */ _plastic_card = plastic_card;  
  
    Object[] args = new Object[1];  
  
    args[0] = new Authorisation(_plastic_card);  
  
    _Card_reader.fires("card_inserted",_Idle,_Busy,read_card(),_atm,"card_read",args);  
  
    _Card_reader.run_to_completion("card_inserted");  
  
}
```

In this code, the *fires* method is only called within the *card inserted* event processing method (and nowhere else) since it requires a special argument (assigned to the first position of the *args* array) whose value depends upon the *plastic_card* parameter of the *card inserted* method. So, the transition is recognized as “runtime”.

Runtime transitions have to be avoided for performance reasons (among others⁴²) because the *fires* method, in the code above, will be called each time the *card inserted* event occurs.

PauWare supports an old style of programming by which runtime transitions may be declared in an anonymous way. This means that they are not associated with an event; see as follows:

```
synchronized public void fan_switch_turned_on() throws Statechart_exception {  
  
    _Programmable_thermostat.fires(_Control,_Control,true,_fan_relay,"run");  
  
    _Programmable_thermostat.run_to_completion();  
  
}
```

The reader must notice that in this code, neither the *fires* nor the *run_to_completion* methods use a string as a first parameter. Calling *fires* as is done in this code is “correct” but improvements are possible. The call of *fires* should indeed occur within the *start* method because the declared transition does not need contextual data (the

⁴² Runtime transitions are also no longer visible by a state machine observer/controller until the event that labels them has occurred at least one time (see also Section 8.3).

guard is always equal to true and the called action, *run*, has no parameters). However, if called within *start*, the transition must have the following shape⁴³:

```
_Programmable_thermostat.fires("fan_switch_turned_on",_Control,_Control,true,_fan_relay,"run");
```

The *fan_switch_turned_on* string is added as the first parameter of *fires*. If this string is not added when calling *fires* within the *start* method, *fires* has no effect. The *run_to_completion* method may also have as the first parameter the name of the event to be processed. Providing this name leads to triggering, within a run-to-completion cycle, the cached transitions associated with this name. As a result, the following code is incorrect:

```
public void fan_switch_turned_on() throws Statechart_exception {  
  
    _Programmable_thermostat.run_to_completion(); // problem here, the cached transition will not be found  
  
}
```

In this case, *PauWare* will not be able to make a link between the processing of the *fan switch turned on* event (*fan_switch_turned_on* method) and the recorded transition within the *start* method. Correcting this leads to having:

```
public void fan_switch_turned_on() throws Statechart_exception {  
  
    _Programmable_thermostat.run_to_completion("fan_switch_turned_on");  
  
}
```

To close, the following style, even if it is not optimized (the transition is not cached) remains correct⁴⁴:

```
synchronized public void fan_switch_turned_on() throws Statechart_exception {  
  
    _Programmable_thermostat.fires(_Control,_Control,true,_fan_relay,"run");  
  
    _Programmable_thermostat.run_to_completion("fan_switch_turned_on");  
  
}
```

In this last case, the transition is known as “runtime”. However, *PauWare* will record it (at the time of the first call of the *fan_switch_turned_on* method) thanks to the fact the *fan_switch_turned_on* string is passed to the *run_to_completion* method. Such a recording is useful for instance, if a viewer displays the transition within a graphical window of a screen. For the next calls of the *fan_switch_turned_on* method, a comparison occurs between the previously recorded transition and the runtime transition in order to update the parameters of guards and/or actions (if any). Since, once again, in this specific case, the *run* action has no parameters, the code just above is not optimized.

5.7.2 Problems raised by runtime guards

As described in Section 3.3, guards may be implemented by means of Java methods, which return Boolean types. Another issue is that such methods may be called normally or through the Java reflection support⁴⁵. In the first case, we have what we call “runtime guards” (form (a)):

```
_Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"please_wait_for_beverage");
```

```
_Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"decrease_total");
```

In the second case, we have (form (b)):

```
_Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,this,"required",this,"please_wait_for_beverage");
```

```
_Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,this,"required",this,"decrease_total");
```

⁴³ In this case, the transition is by definition no longer a runtime transition but a cached transition.

⁴⁴ We comment on the use of the *synchronized* Java keyword in Section 5.10.

⁴⁵ In the Velcro API, since reflection is not possible, one must not forget to take into account guards in the implementation of the *execute* method of the *VelcroExecutor* interface (see also Section 6.3.6).

All of the previous code works in conjunction with the following guard, which tests if a user of a beverage vending-machine provided enough money:

```
public boolean required() {  
  
    return _total >= _beverage_price;  
  
}
```

An important question is: Where to place the two transitions which activate the *please_wait_for_beverage* and *decrease_total* actions? Logically, they have to appear in the *start* method of the built software component since no contextual data is needed. However, between forms (a) and (b) above, only form (b) may be used inside *start* as follows (1):

```
protected void start() throws Statechart_exception {  
  
    _Beverage_vending_machine = new  
    Statechart_monitor(_Coin_insertion.xor(_Service).xor(_Pick_up),"Beverage vending-machine",true);  
  
    /* this code cannot work here:  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"please_wait_for_bev  
    erage"); */  
  
    /* this code cannot work here:  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"decrease_total"); */  
  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,this,"required",this,"please_wait_for  
    _beverage");  
  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,this,"required",this,"decrease_total")  
    ;  
  
    ...  
  
}
```

So, since the *start* method of the *PauWare* API is called one time only (especially at the component's initialization time), calling *required()* in *start* definitely fixes up the value of the guard without any update within the forthcoming run-to-completion cycles. In contrast, the call by reflection, *i.e.*, *this,"required"*, **guarantees the re-evaluation of guards within run-to-completion cycles**. The following is another acceptable form which is an alternative to having the code within *start* (2):

```
public void choose() throws Statechart_exception {  
  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"please_wait_for_bev  
    erage");  
  
    _Beverage_vending_machine.fires("choose",_Coin_insertion,_Service,required(),this,"decrease_total");  
  
    _Beverage_vending_machine.run_to_completion("choose");  
  
}
```

Solution (1) is based on cached transitions, which are more appropriate for performance issues, but reflection is more costly than ordinary method calls. Solution (2) is based on runtime transitions, which are recognized as less interesting, but each guard evaluation bypasses reflection. Within the *Velcro* API, since solution (1) also bypasses reflection, its use is more relevant with respect to performance issues.

5.8 States and business data consistency checking through state invariants

State invariants are usually *Boolean* properties that must hold true when a given state is active. State invariants are mainly intended to "synchronize" the business data of a software component with its embedded state machine. By "synchronize", we here mean checking the consistency of the component's business data in relation to its current active state(s).

State invariants are parts of the UML formalism and are used in Figure 27. For instance, the *Empty* state of a *Stack* software component in Figure 27 matches to $[size = 0]$. In *PauWare*, the idea is to offer the possibility of assigning Java invariant evaluation methods to states. This is dedicated to the computing of these invariants when these states become active.

5.8.1 Basic mechanisms

For instance, if we take the example of the *Stack* component (Figure 27) whose behavior is based on a simple state machine diagram, the following state invariant is useful:

```
protected void init_behavior() throws Statechart_exception {
    ...
    _Only_one = new Statechart("Only one");
    _Only_one.stateInvariant(this,"size_is_equal_to_1");
    ...
}
...
public boolean size_is_equal_to_1() {
    return _stack.size() == 1;
}
}
```

In *PauWare*, using the *stateInvariant* API function⁴⁶ leads to the possibility of executing, within a run-to-completion cycle, a Java method (*size_is_equal_to_1* above) that is connected with a state and recognized as its invariant evaluator. This invariant computing method may return something other than a *Boolean* type. State invariants are thus violated if *false* is returned or if an exception is raised at the method's execution time.

In *PauWare*, state invariants are not systematically executed. Indeed, one has to set the *autonomic* flag to true if they are to be executed. This occurs as follows:

```
public void empty_() throws Statechart_exception {
    _Stack.run_to_completion("empty_",Manageable.Autonomic);
}
}
```

This code means that each time an *empty_* event⁴⁷ occurs, each state invariant, if any, of **each individual active state** of the state machine is computed. Furthermore, since computing invariants is a costly practice and, above all not always necessary, there is no systematization in order to keep performance at an acceptable level. If the tracing option for this state machine is also set to true (see Section 3.1.6), one is able to visualize the results of the state invariant evaluations. For an active state, which does not have an invariant computing method, the evaluation leads to true (*i.e.*, its invariant holds). State invariants are evaluated only one time within a run-to-completion cycle **just after** moving the state machine to its next logically expected state(s). This is what we previously named "a stable consistent context".

A point to be noticed is the fact that invariants are not computed for active states, which have in progress activities. In such a case, invariants are considered neither true nor false but "undefined". In fact, in such a case, an instance of the *Statechart_exception* class is thrown. When invariant computing methods fail, such an object is also raised.

⁴⁶ At most one state invariant may be attached to a given state, meaning that using the *stateInvariant* method twice for the same state erases the previous recorded invariant.

⁴⁷ This event is named *empty?* in Figure 27.

5.8.2 Advanced mechanisms, towards autonomic computing facilities

In the autonomic mode, when an event is processed, a logical “and” is applied to all of the invariants of each active state that has been reached resulting from the fact that this event occurs. The result of this logical “and” is either true, false or “undefined”. The latter case either means that an error occurs when evaluating invariants, or some active states go on running attached activities.

The role of state invariants is to instrument self-managing characteristics or more precisely, self-healing capabilities. This concerns the notion of autonomic computing (see also Section 8). Since actions are activated when entering into or existing in states (as well as when requests labeling transitions are processed), one may wonder “what to do” when actions fail? Do we go back to the starting state(s) or do we reach the next expected state(s)? For illustration purposes, we extend the model in Figure 2 by modifying the specification of the *Idle* state (Figure 28). A state invariant for this state consists in having a *port* object closed. We also add an entry action for the *Busy* state, which forces the port to be open.

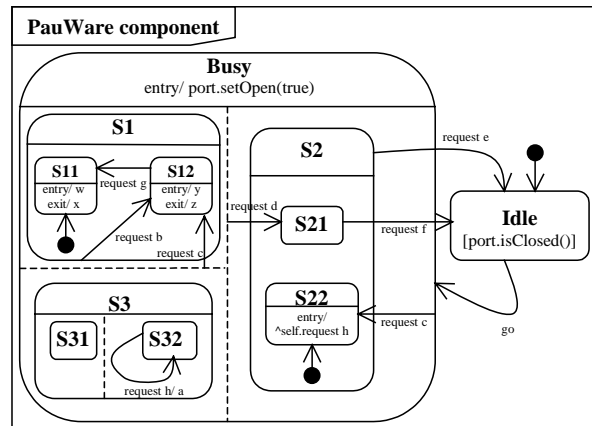


Figure 28. Specification of a state invariant for the *Idle* state of *PauWare* component (see also Figure 2).

Here is the *PauWare* implementation of the `[port.isClosed()]` state invariant:

```
_Idle.stateInvariant(_port,"isClosed");
```

This code allows us to determine whether or not the port is really closed at the time we reach *Idle*. If we observe in Figure 28 that *request e* is the event which makes a state machine arrive at *Idle*, the specification in Figure 28 guarantees that either the *x* action or the *z* action are executed when leaving, respectively, either *S11* or *S12*. One of these two actions, or both, is(are) probably the action(s) in charge of closing the port. A failure when executing them is also probably the reason why the port is not closed. In other words, this port is part of the business data of *PauWare* component. Having a state invariant defined for *Idle* therefore enables a basic verification. For that, we need the activation of the autonomic mode as follows:

```
public void request_e() throws Statechart_exception {
    _PauWare_component.run_to_completion("request_e",Manageable.Autonomic);
}
```

So, returning to *Idle* by means of *request e* (if the logical “and” for all invariants of all the active states: in this case, only *Idle* is active) is equal to false or is “undefined”⁴⁸, the *PauWare* engine **tries to automatically go back** to the prior active state(s). Here, this is a combination of substates of *Busy* which must precisely match to the stable consistent configuration that existed just before the occurrence of *request e*. Either this rollback action (a kind of failure recovery mechanism) succeeds or fails; in the lattercase, one, considers it as a fatal error. After recovering the immediately prior stable consistent situation, invariants are **again** computed. Rollback success amounts to having true as the result the logical “and” for all invariants of all the recovered active states. These are the combination of the substates of *Busy* mentioned above.

In fact, we not only try to cancel the effect of *request e* **but we also measure** if this cancellation is “semantically correct”. Otherwise, when scanning the invariants of all the recovered active states, if false or

⁴⁸ “undefined” is also considered to be a problem in the functioning of a state machine and consequently also leads to putting into practice the self-healing support of *PauWare*.

“undefined” is the computed value, then this globally means that rollback fails. In this situation, the state machine is definitively out of order, apart from the possibility of activating dynamical reconfiguration primitives to reset it to an explicit stable consistent context (see Section 8.1).

5.9 History facilities

PauWare does not support history facilities, which are commonly modeled by the following symbol: a H (shallow history) or a H^* (deep history) within a circle. This symbol is depicted inside a state in order to assign history properties to this state. As a result, *PauWare*'s execution engine uses the default input state when it enters a superstate that has several immediate substates linked with exclusiveness relationships.

5.10 Concurrent use

Software components constructed by means of *PauWare* tolerate concurrent accesses and thus concurrent requests (see also Section 4.1). Most of the time, a *PauWare* developer does not have to take care about how its own components are used by other developers, except when runtime transitions are put into practice (see also Section 3.2). To analyze this problem, we propose a simple state machine diagram in Figure 29. Concurrency simply means for instance that two parallel Java Threads require the execution of $e1$ and/or $e2$ for the same state machine, *i.e.* for the same software component instance.

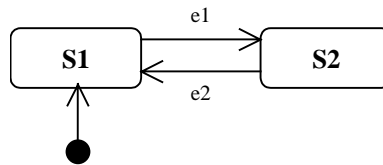


Figure 29. Illustration of concurrent use.

A concurrency-safe implementation of the model in Figure 29 leads to the following:

```
protected void start() throws Statechart_exception {
```

```
    _Concurrent_use = new Statechart_monitor(_S1.xor(_S2),"Concurrent use",true); // tracing is activated
```

```
    _Concurrent_use.fires("e1",_S1,_S2);
```

```
    _Concurrent_use.fires("e2",_S2,_S1);
```

```
}
```

```
...
```

```
public void e1() throws Statechart_exception {
```

```
    _Concurrent_use.run_to_completion("e1");
```

```
}
```

```
public void e2() throws Statechart_exception {
```

```
    _Concurrent_use.run_to_completion("e2");
```

```
}
```

This code shows that runtime transitions are **not** used because all the required usages of the *PauWare fires* method occur within the *start* method (see also Section 3.1.7). So, although this case study does not impose the use of runtime transitions, an alternative implementation using them should be as follows:

```
synchronized public void e1() throws Statechart_exception {
```

```

    _Concurrent_use.fires(_S1,_S2);

    _Concurrent_use.run_to_completion("e1");
}

synchronized public void e2() throws Statechart_exception {

    _Concurrent_use.fires(_S2,_S1);

    _Concurrent_use.run_to_completion("e2");
}

```

In this code, since the *fires* method does not mention its labeling event as the first parameter, the run-to-completion cycle may thus assign (due to the concurrent access of *e1* and *e2*) undedicated runtime transitions to unconcerned events like moving from *S1* to *S2* for *e2*. However, the specification in Figure 29 does not state that. The only way to solve the problem is to declare both *e1* and *e2* as *synchronized* (see above).

To sum it up, Java methods that embody events must be declared *synchronized* when concurrent accesses and requests are probable and runtime transitions are used in the Java body of events. Nevertheless, one may notice that only runtime transitions that do not mention their labeling events as the first parameter, are concerned by the problem. As a result, this form is concurrency-safe (the use of the *synchronized* keyword is no longer required while runtime transitions are used):

```

public void e1() throws Statechart_exception {

    _Concurrent_use.fires("e1" _S1,_S2);

    _Concurrent_use.run_to_completion("e1");
}

public void e2() throws Statechart_exception {

    _Concurrent_use.fires("e2" _S2,_S1);

    _Concurrent_use.run_to_completion("e2");
}

```

6 Using *PauWare* with the Java ME platform

This section is twofold. It first explains how to build a Java ME application with *PauWare*. This section also presents a more significant case study named *Home Automation System* in order to show how a complex state machine may be easily and straightforwardly implemented with *PauWare*. This case study has been initially presented in the book of James Rumbaugh *et al.* about the *Object Modeling Technique* or OMT (see Bibliography). In this users' guide, the case study's requirements have been extended in order to provide a more "realistic" system. From this case study, we build all of the UML models including, of course, state machine diagrams. We also underline how all of these models may lead to software components by using UML Component Diagrams.

In this section, we also address key issues concerning *PauWare*'s programming subtleties. Since the *PauWare* core execution engine is compliant with Java ME (*Velcro* API), programming a Java ME application will require the use of the *VecroStatechart* Java class for instantiating states and *VelcroStatechart_monitor* for instantiating state machines. Another point concerning this case study is the intensive use of runtime transitions instead of cached transitions (see Section 5.7). **This is only a didactical approach** in order to show the broad range of *PauWare*'s programming possibilities. However, such an implementation is by definition not optimized since runtime transitions are recorded at each event occurrence while cached transitions can be

declared once and for all in the *init_behavior* recommended the Java method (see Section 3.1.7). For the reader, a good exercise would then be, to transform runtime transitions into cached transitions, whenever possible.

6.1 General requirements

The *Home Automation System* is based on a programmable thermostat device made up of a table of eight target temperatures. This table is named “the program”. The first four time slots of the table are used for recording target temperatures, which relate to weekdays, while the four remaining slots are intended for weekends. The programmable thermostat is also surrounded by a temperature sensor, which measures room temperature, two switches (a season switch with the *Cool*, *Heat* and *Off* values, and a fan switch with the *Auto* and *On* values), three power relays (a furnace relay, an air conditioner relay and a fan relay), and finally, a run indicator and a graphical user interface. The *Home Automation System* is ruled by strict timing constraints in order to refresh some varied data appearing within the graphical user interface and to periodically retrieve the stored data from the program (*i.e.*, the table of eight target temperatures). For that, *PauWare*’s timer services (see Section 5.2) are used.

6.2 Comprehensive description

The thermostat controls a furnace and an air conditioner (*Control* state in Figure 34) according to data entered by a user by means of a keyboard. During operation the thermostat either controls the furnace (the season switch must be set to *Heat*) or the air conditioner (the season switch must be set to *Cool*), to make the room temperature correspond to the current target temperature. Setting the season switch to *Off* means that no control occurs. A fan relay is added to the furnace and the air conditioner relays, which is also controlled by the thermostat to automatically dispatch air when the fan switch is set to *Auto* (the fan operates only if the furnace is active or the air conditioner is active). As for *On*, the fan operates permanently (*i.e.*, whether or not the furnace or the air conditioner is working).

The thermostat determines the room temperature by using a temperature sensor. The thermostat permanently needs four types of information: the last measured room temperature, the current date and time, the current target temperature and finally *delta*, which is a constant used for stopping the furnace as follows:

ambient temperature > current target temperature + delta

Likewise, the condition for stopping the air conditioner is:

ambient temperature < current target temperature - delta

The thermostat includes a graphical user interface (Figure 30, Swing look & feel⁴⁹) associated with a ten-button keyboard (*Run program*, *F-C*, *Hold temp*, *Set clock*, *Set day*, *View program*, *Time backward*, *Time forward*, *Temp up* and *Temp down*). Note that a slider for setting up the target temperature (Figure 30) is used instead of two “physical” buttons representing *Temp up* and *Temp down*. To that extent, moving the slider right is equivalent to a certain number of *Temp up* event occurrences, while moving the slider left corresponds to several *Temp down* event occurrences. However, in the *PauWare* programming logic, the *Programmable thermostat* software component that supports the interpretation of *Temp up* and *Temp down* has a shape which of course does not depend upon the look & feel and technical features of its graphical user interface.

The user interface is based on a mechanism that alternates the display of the current date and time and the display of the room temperature every two seconds. Figure 30 is the case when the date and time are visible. The current target temperature is also displayed (on the top of the slider), however it is always visible.

In the *Operate* mode, the thermostat checks the program table every second to find out if the current time corresponds to a moment at which the target temperature must change. If true, it is then necessary to replace the current target temperature by the one read in the table and assigned to the aforesaid moment in the table. However, it is possible to ignore the program (to do no table scanning) by pressing *Hold temp*. In this situation, the current target temperature is maintained until pressing *Run program*. In other words, *Run program* is the state when the table of the eight target temperatures is scanned. Both *Hold temp* and *Run program* match the *Operate* mode. The opposite mode is the configuration mode: *Setup*. Most of the ten buttons have a different purpose with regard to the *Operate* or *Setup* modes. Here are these purposes:

⁴⁹ The J2ME look & feel is obviously different especially if MIDP is used.

- *Temp up*: in *Operate*, this event increments the current target temperature. The latter is used within the furnace/air conditioner control loop until any new update occurs. In *Setup*, this event increments the target temperature of a given programming period. One skips from one programming period to another in the table by pressing *View program*. Note that one never must exceed 90°Fahrenheit.
- *Temp down*: same as *Temp up* with “decrements” instead of “increments”. Note that one must never go below 40°Fahrenheit.
- *Time forward*: this key is only usable in *Setup*. It allows us to add 15 seconds to the time associated with a given programming period (one requires the activation of *View program* just before). In contrast, if *Set clock* has been activated just before, *Time forward* also provides the possibility of contextually adding 1 minute or⁵⁰ 1 hour to the time of the active programming period.
- *Time backward*: same as *Time forward* with “subtracting” instead of “adding”.
- *Set clock*: by pressing this key, one activates *Setup* and thus leaves *Operate*. Pressing again leads to switching from minutes (default) to hours, to hours from minutes (a new activation of *Set clock* is required), in order to setup the minutes or hours of the current time.
- *Set day*: by pressing this key, one activates *Setup* and thus leaves *Operate*. One increases (by next pressing *Time forward*) or decreases (by next pressing *Time backward*) the current date by 24 hours.
- *View program*: by pressing this key, one activates *Setup* and thus leaves *Operate*. The first programming period (a weekday) is considered to be accessible, and possibly modifiable (a recorded time and date with its related target temperature). Pressing again *View program* makes the second programming period configurable, etc. *Temp up*, *Temp down*, *Time forward*, *Time backward*, *Set clock* and *Set day* are in essence complementary keys to be used in conjunction with *View program*.
- *Run program*: by pressing this key, one activates *Operate* and thus leaves *Setup*. The system is considered to be in the *Run* state, in which the program is taken into account. This means that the table of the eight target temperatures is consulted every second in order to possibly obtain a new current target temperature.
- *Hold temp*: by pressing this key, one activates *Operate* and thus leaves *Setup*. The system is considered to be in the *Hold* state, in which the program is deliberately ignored. In *Setup*, if a period of 90 seconds elapses without the following keyboard inputs: *View program*, *Temp up*, *Temp down*, *Time forward*, *Time backward*, *Set clock* and *Set day*, it amounts to pressing *Hold temp*, (i.e., one goes back to the *Hold* state).
- *F-C*: this key alternates between Fahrenheit (default) and Celsius. It effects the display of room temperature, the current target temperature, and the temperatures eventually associated with these programming periods.

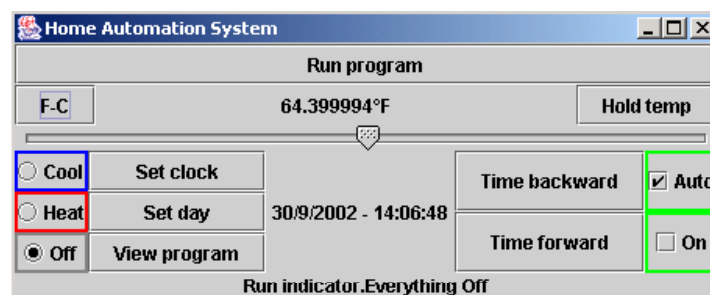


Figure 30. Graphical user interface of the *Home Automation System*.

6.3 Specification

The objective of this section is how to transform the textual requirements of the *Home Automation System* into UML models or a specification. We stress a component-based segmentation of this system, without

⁵⁰ One switches from “minutes” to “hours” by pressing *Set clock* again, “minutes being the default mode.

insisting on the precise design rules⁵¹ which make the proposed models relevant and reliable, from a software quality perspective. We rather emphasize the *PauWare* code, which is necessary to make these models persistent at runtime (*i.e.*, “executable”). The reader may readily imagine that a good code generator may do most of the described work below. This issue is addressed in Section 13. Another issue for the reader is the crucial understanding that the *PauWare* code may serve as a model verification/validation support and also as the “final” code. In the second case, only the internal body of actions, activities and, possibly, guards has to be coded (remember that actions may be viewed as state entry actions, state exit actions or actions launched as a reaction against events). This is in order to achieve an end-user application. In the first case, one aims at simulating models with appropriate displays. For that, *PauWare* advocates the use of JMX, standing for *Java Management eXtensions* (see downloadable examples from: www.PauWare.com/PauWare_software), and/or the *PauWareView* extra tool⁵². From a technical point of view, an instance of a class implementing the *Statechart_monitor_listener* interface may be added to a state machine in order to be informed of its step-by-step evolution (run-to-completion cycles). *PauWareView* intentionally provides the *Statechart_monitor_viewer* class in the *PauWareView* tool.

6.3.1 Class diagram

Figure 31 is the class diagram of the *Home Automation System*. To have a complete specification, it is necessary to add constraints (see Section 6.3.3) to the class diagram in Figure 31. These constraints are written in OCL (Object Constraint Language). OCL is a subset of the UML. To make the *Home Automation System* specification complete, OCL constraints must also be added to state machine diagrams (see Section 6.3.4). Among the different types in the model of Figure 31, only the *Programmable thermostat* type with the two switches, the run indicator and the three relays have a behavior leading to drawing a state machine diagram. Except the thermostat, the state machine of each entity is trivial due to the fact that no more than three states are in general necessary. For instance, the abstract (the notation is in italics in the UML) *Relay* class (see Figure 31) factorizes the behavior of the three sorts of relays and has only two states: *Is on* and *Is off*. Another case is the *Program* class, which represents data without any behavior. More precisely, any instance of this class corresponds to a pair (*i.e.*, target temperature, moment of change) among exactly eight ordered instances linked to the thermostat. Since the *Program* class has no behavior, it has no state machine.

⁵¹ However, the rule “one state machine diagram leads to one component type” is often used for its simplicity and efficiency.

⁵² This tool is downloadable from: www.PauWare.com/PauWare_software and provided “as is”. It is not part of the *PauWare* software and thus not described in this guide.

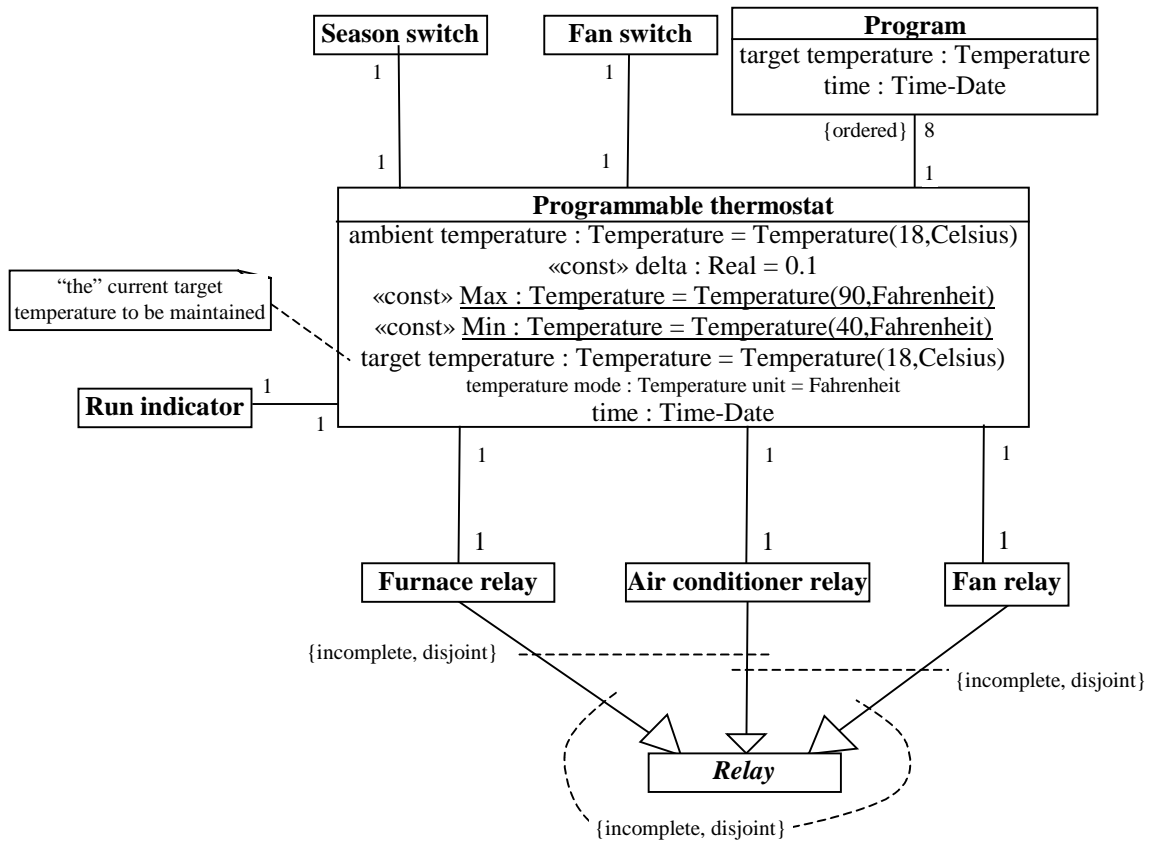


Figure 31. The UML Class Diagram of the Home Automation System.

To summarize, we have types with complex states machines (e.g., the *Programmable thermostat* type), types with trivial states machines (e.g., *Run indicator*, *Relay*), business types (e.g., the *Program* type) that match to business data without explicit behavior and value types that correspond to application-independent types, namely here the *Temperature unit* type, the *Temperature* type and the *Time-Date* type. As an illustration, we provide the specification of *Temperature unit* and *Temperature* in Figure 32 but, like business types, value types do not have an explicit behavior. Value types are specified once and for all and are reused in numerous applications. In UML, they appear inside the boxes embodying classes for typing class attributes, method parameters or method returned values. For instance in Figure 31, the type of the *ambient temperature*, which is an attribute of the *Programmable thermostat* type, is *Temperature*.

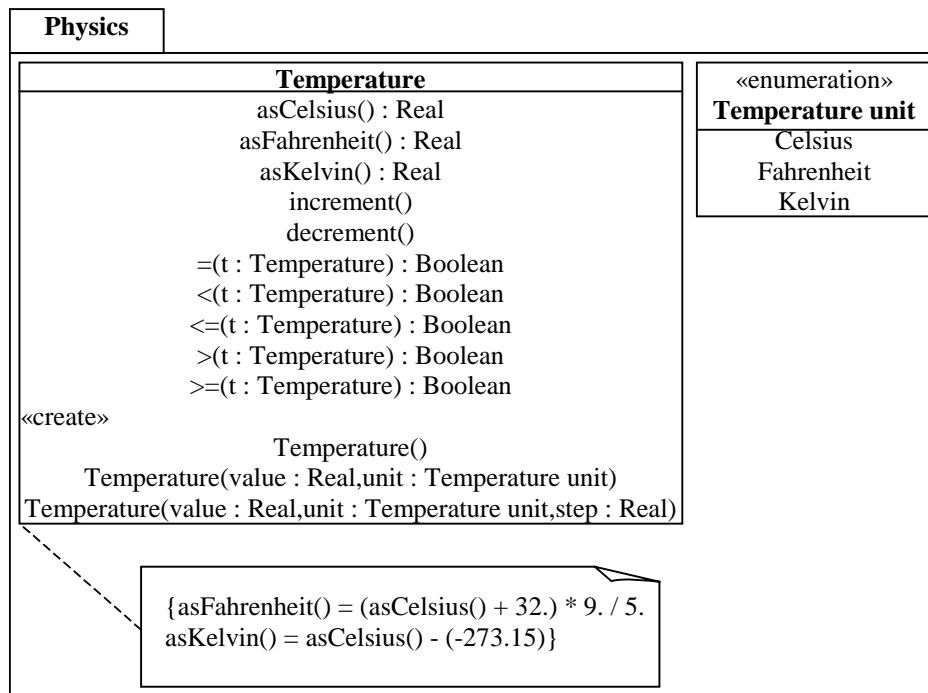


Figure 32. Specification of *Temperature unit* and *Temperature*.

6.3.2 Communication diagram

To complete the class diagram in Figure 31, the communication diagram in Figure 33 provides a view of the system's event flow. This model clearly states how communication occurs in the *Home Automation System*. From a design perspective and a CBD viewpoint, we consider the *Programmable thermostat* type as a software component having three "external"⁵³ clients: a temperature sensor, a user interface and a timer. The UML notation in Figure 33 shows one-to-one interaction scenarios between instances (a ":" before a type name means "an instance" of the type).

⁵³ In UML the boxes framing the temperature sensor, user interface and timer instances have a different look from that of common instances.

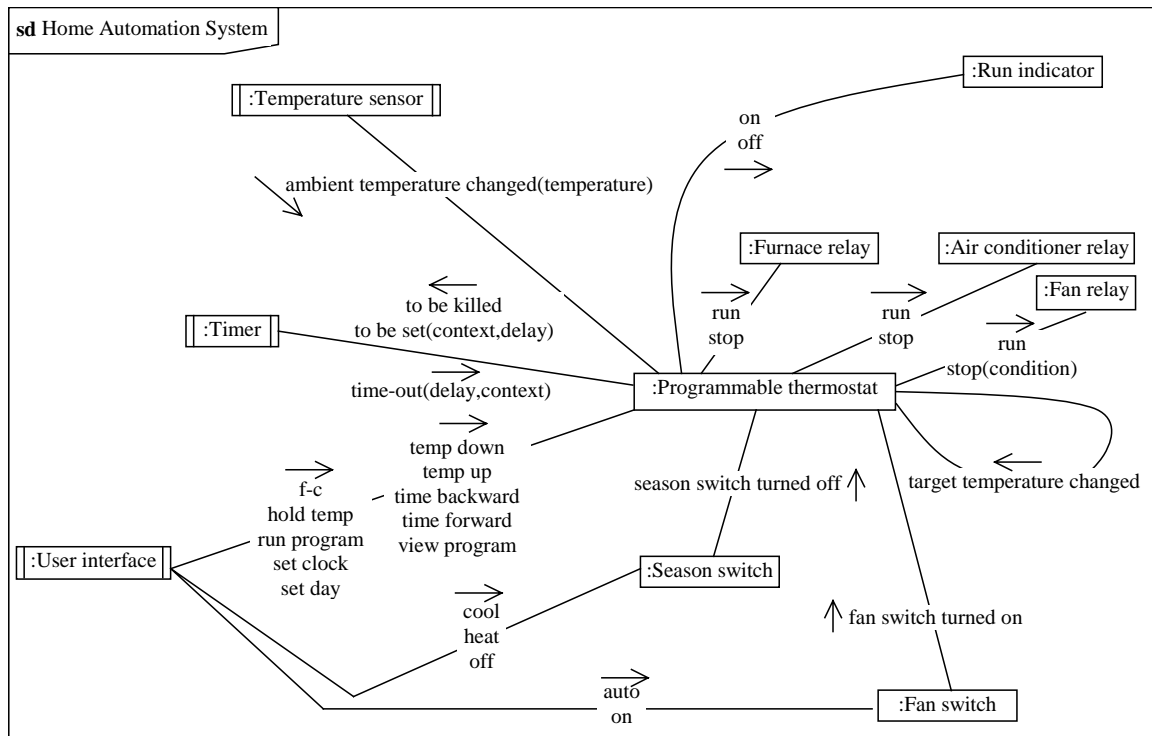


Figure 33. The UML Communication Diagram of the Home Automation System.

Since *PauWare* supports timer services (see Section 5.2), the interaction between *Programmable thermostat* and *Timer* simply requires to first inherit from the *AbstractTimer_monitor*⁵⁴ Java abstract class as follows:

```
public class Programmable_thermostat extends AbstractTimer_monitor implements... { ...
```

Next, an implementation of the Java *time_out* inherited method in compliance with Figure 34 is required:

```
synchronized public void time_out(long delay,AbstractStatechart context) throws Statechart_exception {
    Object[] args = new Object[2];
    args[0] = new Integer(1);
    args[1] = new Integer(4);
    // _Programmable_thermostat.fires(_Run,_Run,! weekend(),this,"set_target_temperature",args);
    _Programmable_thermostat.fires(_Run,_Run,this,"not_weekend",this,"set_target_temperature",args);
    args[0] = new Integer(5);
    args[1] = new Integer(8);
    // _Programmable_thermostat.fires(_Run,_Run,weekend(),this,"set_target_temperature",args);
    _Programmable_thermostat.fires(_Run,_Run,this,"weekend",this,"set_target_temperature",args);
    args = new Object[1];
    args[0] = new Integer(0);
```

⁵⁴ This is specific to J2ME and the *Velcro* API. When using the *Composytor* API, one must rather extend the *Timer_monitor* Java abstract class.

```

    _Programmable_thermostat.fires(_Current_date_and_time_displaying,_Ambient_temperature_displaying,
    _alternately == 2,this,"set_alternately",args);

    _Programmable_thermostat.fires(_Ambient_temperature_displaying,_Current_date_and_time_displaying,
    _alternately == 2,this,"set_alternately",args);

    args[0] = new Integer(_alternately + 1);

    _Programmable_thermostat.fires(_Current_date_and_time_displaying,_Current_date_and_time_displayin
    g,_alternately != 2,this,"set_alternately",args);

    _Programmable_thermostat.fires(_Ambient_temperature_displaying,_Ambient_temperature_displaying,_a
    lternately != 2,this,"set_alternately",args);

    _Programmable_thermostat.fires(_Setup,_Operate,_no_input >= 90);

    args[0] = new Integer(_no_input + 1);

    _Setup.allowedEvent("time_out",this,"no_input_less_than_ninety",this,"set_no_input",args);

    _Programmable_thermostat.run_to_completion("time_out");
}

```

Not only the *time_out* method needs a body but also *time_out_error*, which is often left empty:

```

public void time_out_error(Statechart_exception se) throws Statechart_exception {

    // possible fault recovery here...

}

```

6.3.3 Constraints

OCL constraints are required to enrich both UML Class Diagrams and State Machine Diagrams. Static constraints have no direct counterparts in *PauWare*. The developer must then translate them in appropriate code. For instance, we may specify that the four first slots of the program table are dedicated to weekdays and have to be different⁵⁵:

context **Programmable thermostat** inv: -- weekday

program->subSequence(1,4)->forAll⁵⁶(p1,p2 | p1<>p2 implies p1.time<>p2.time)

context **Programmable thermostat** inv: -- weekend

program->subSequence(5,8)->forAll(p1,p2 | p1<>p2 implies p1.time<>p2.time)

We apply the same approach to the weekend in the second invariant. However, the two invariants above are not essential for the functioning of the thermostat. Indeed, if they are violated, the algorithm which computes the target temperature can, for instance, arbitrarily choose a temperature value. In fact, defense strategies are numerous and more generally their definition can be deferred at implementation time because they are not error-prone.

Here, the constraints linked to UML State Machine Diagrams are more interesting since we have to learn how to cope with them in *PauWare*. For instance, we have to consider an *ambient temperature changed* event sent by the temperature sensor to the thermostat. The effect of this event (among others) is the update of the *ambient temperature* attribute of the *Programmable thermostat* type in Figure 31, as follows:

context **Programmable thermostat::ambient temperature changed(temperature : Temperature)**

⁵⁵ However, the semantics of this difference has to be further stated, like, for instance, two moments of change must be different and their associated temperatures must also be distinct.

⁵⁶ Mathematically, this means: $\forall(p1,p2) \in \text{program} \rightarrow \text{subSequence}(1,4)$ and corresponds to the set of the first four pairs (target temperature,time) in the program table.

post: **ambient temperature = temperature**

This post-condition states that the *ambient temperature* attribute is refreshed in response to an occurrence of the *ambient temperature changed* event. In *PauWare*, this leads to the following implementation:

```
synchronized public void ambient_temperature_changed(Temperature temperature) throws Statechart_exception {  
  
    /** post-condition(s) */  
  
    _ambient_temperature = (Temperature)temperature.clone();  
  
    // runtime transition declarations here, if any  
  
    _Programmable_thermostat.run_to_completion("ambient_temperature_changed");  
  
}
```

The single point to be aware of, is the fact that event post-condition implementation must appear before launching the run-to-completion cycle, which itself precisely corresponds to the event's interpretation; here *ambient temperature changed*. In other words, this cycle moves the state machine forward by possibly using the very last value of the *ambient temperature* attribute. The latter may for instance be used within a guard. Post-conditions may also concern state machine local variables⁵⁷. For instance, *no input* is a variable of the *Programmable thermostat* state machine diagram in Figure 34. It records some elapsed time (in seconds) from the moment at which a key has been pressed in the *Setup* mode. An occurrence of the *set clock* event (this also applies to some other events) makes *no input* equal to 0:

context **Programmable thermostat::set clock()**

post: **no input = 0**

This leads to the following code for the *set clock* event:

```
synchronized public void set_clock() throws Statechart_exception {  
  
    /** post-condition(s) */  
  
    _no_input = 0;  
  
    _Programmable_thermostat.fires(_Programmable_thermostat,_Set_current_minute); /* no event name is here provided meaning that this transition is a runtime transition. Since this transition does not use any contextual data, it may benefit from being declared in the init_behavior Java method */  
  
    _Programmable_thermostat.fires(_Programmable_thermostat,_Current_date_and_time_refreshing); /* same as first transition */  
  
    _Programmable_thermostat.fires(_Set_current_minute,_Set_current_hour); /* same as first transition */  
  
    _Programmable_thermostat.run_to_completion(); /* no event name is here provided meaning that the run-to-completion cycle will not be able to make use of cached transitions */  
  
}
```

Post-conditions have no direct support in *PauWare*. In modeling, the same event type (recognizable by its name and signature in a state machine diagram) may appear at different places of a state machine diagram. An event's post-condition represents a global side-effect resulting from an event occurrence regardless of the transitions labeled by this event. This raises tricky modeling problems. For instance, the *set clock* button is interpretable only in the *Setup* mode of the thermostat. When pressed, in any case, *no input* is reset to 0. In contrast, the *temp up* or *temp down* events have distinct purposes in the *Operate* and *Setup* modes (see Section 6.2 and/or Figure 34). Consequently, it becomes difficult to establish if a post-condition of *temp up* and *temp down* is *no input = 0*. In the *Operate* mode especially, this makes no sense simply because *no input* is useless.

⁵⁷ These variables become attributes at design time. For instance, *no input* (see Figure 34) becomes a private attribute of the *Programmable thermostat* type at design time. At analysis time, it is only a local variable of the *Programmable thermostat* state machine diagram.

In Figure 34, we have decided to associate an action with the *temp up* or *temp down* events. This is a self-transition on top of the *Program target temperature refreshing* state, which is itself an indirect substate of *Setup*. It launches the *set no input* action with the parameter 0 (i.e., *temp down/set no input(0)* and *temp up/set no input(0)*). As an illustration in *PauWare* for *temp down*, one ends up having the following implementation⁵⁸:

```
synchronized public void temp_down() throws Statechart_exception {

    boolean guard = _target_temperature.greaterThan(_Min);

    _Programmable_thermostat.fires(_Target_temperature_displaying,_Target_temperature_displaying,guard
    ,this,"target_temperature_changed",null,AbstractStatechart.Broadcast);

    /* Java ME only */
    _Programmable_thermostat.fires(_Run,_Run,guard,_target_temperature,"com.FranckBarbier.Java._Home
    _automation_system._Programmable_thermostat.Programmable_thermostatActionExecutor.decrement");

    /* Java ME only */
    _Programmable_thermostat.fires(_Hold,_Hold,guard,_target_temperature,"com.FranckBarbier.Java._Hom
    e_automation_system._Programmable_thermostat.Programmable_thermostatActionExecutor.decrement")
    ;

    /**** _Programmable_thermostat.fires(_Run,_Run,guard,_target_temperature,"decrement"); ****/

    /**** _Programmable_thermostat.fires(_Hold,_Hold,guard,_target_temperature,"decrement"); ****/

    guard = _program[_period - 1].target_temperature.greaterThan(_Min);

    /* Java ME only */
    _Programmable_thermostat.fires(_Set_program_target_temperature,_Set_program_target_temperature,g
    uard,_program[_period -
    1].target_temperature,"com.FranckBarbier.Java._Home_automation_system._Programmable_thermostat.
    Programmable_thermostatActionExecutor.decrement");

    /****
    _Programmable_thermostat.fires(_Set_program_target_temperature,_Set_program_target_temperature,g
    uard,_program[_period - 1].target_temperature,"decrement"); ****/

    Object[] args = new Object[1];

    args[0] = new Integer(0);

    _Programmable_thermostat.fires(_Program_target_temperature_refreshing,_Program_target_temperatur
    e_refreshing,true,this,"set_no_input",args);

    _Programmable_thermostat.run_to_completion();

}
```

To sum up constraints in general and event post-conditions in particular, it is tempting to implement them as actions or activities. Sometimes, it may be appropriate like in the *set no input(0)* action above. Sometimes, it is not. UML State Machine Diagrams must be constructed accordingly and OCL constraints must be implemented manually within *PauWare* in coherence with these kinds of models but also with Class Diagrams.

⁵⁸ We comment on this J2ME-specific code in Section 6.3.6.

6.3.4 Programmable thermostat's behavior

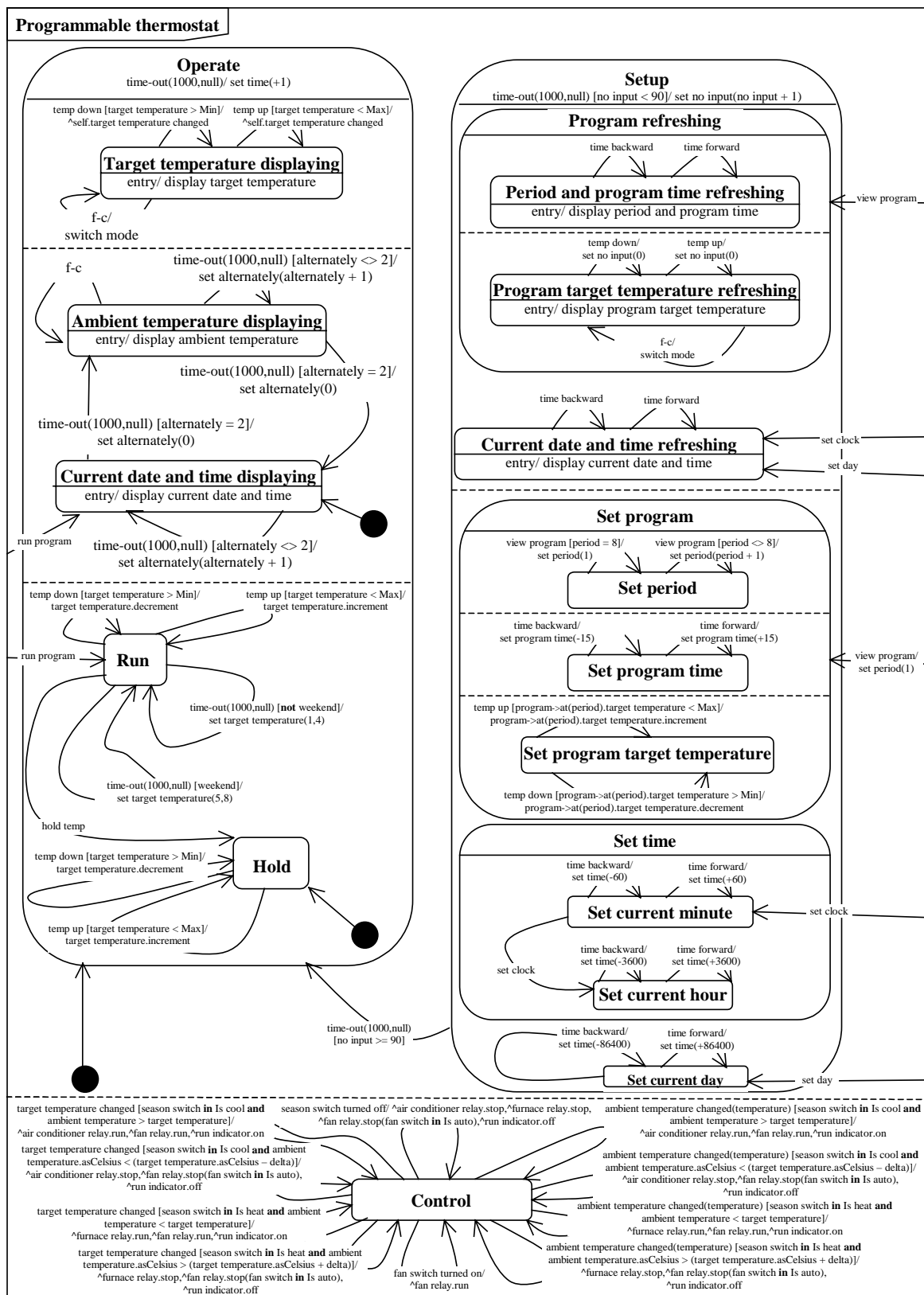


Figure 34. The UML State Machine Diagram of the Programmable thermostat software component.

```

protected void init_structure(Temperature temperature) throws Statechart_exception {

    // business data initialization and setup

}

protected void init_behavior() throws Statechart_exception {

    _Run = new VelcroStatechart("Run");

    _Hold = new VelcroStatechart("Hold");

    _Hold.inputState();

    _Current_date_and_time_displaying = (new VelcroStatechart("Current date and time
displaying")).entryAction(this,"display_current_date_and_time");

    _Current_date_and_time_displaying.inputState();

    _Ambient_temperature_displaying = (new VelcroStatechart("Ambient temperature
displaying")).entryAction(this,"display_ambient_temperature");

    _Target_temperature_displaying = (new VelcroStatechart("Target temperature
displaying")).entryAction(this,"display_target_temperature");

    _Operate =
((( _Run.xor(_Hold)).and(_Current_date_and_time_displaying.xor(_Ambient_temperature_displaying)).an
d(_Target_temperature_displaying)).name("Operate"));

    _Operate.inputState();

    Object[] args = new Object[1];

    args[0] = new Integer(+1);

    _Operate.allowedEvent("time_out",this,"set_time",args);

    _Set_current_minute = new VelcroStatechart("Set current minute");

    _Set_current_hour = new VelcroStatechart("Set current hour");

    _Set_time = (_Set_current_minute.xor(_Set_current_hour)).name("Set time");

    _Set_current_day = new VelcroStatechart("Set current day");

    _Set_period = new VelcroStatechart("Set period");

    _Set_program_time = new VelcroStatechart("Set program time");

    _Set_program_target_temperature = new VelcroStatechart("Set program target temperature");

    _Set_program =
(_Set_period.and(_Set_program_time).and(_Set_program_target_temperature)).name("Set program");

    _Current_date_and_time_refreshing = (new VelcroStatechart("Current date and time
refreshing")).entryAction(this,"display_current_date_and_time");

    _Program_target_temperature_refreshing = (new VelcroStatechart("Program target temperature
refreshing")).entryAction(this,"display_program_target_temperature");

    _Period_and_program_time_refreshing = (new VelcroStatechart("Period and program time
refreshing")).entryAction(this,"display_period_and_program_time");

```

```

    _Program_refreshing =
    (_Program_target_temperature_refreshing.and(_Period_and_program_time_refreshing)).name("Program
refreshing");

    _Setup =
    ((_Set_time.xor(_Set_current_day).xor(_Set_program).and(_Current_date_and_time_refreshing.xor(_Pro
gram_refreshing))).name("Setup"));

    _Control = new VelcroStatechart("Control");
}

public void start() throws Statechart_exception {

    _Programmable_thermostat = new
    VelcroStatechart_monitor((_Operate.xor(_Setup)).and(_Control),"Programmable thermostat",true);

    /** <initialization> post-condition(s) */

    to_be_set(1000);
}

public Programmable_thermostat(Temperature temperature) throws Statechart_exception {

    init_structure(temperature);

    init_behavior();

    // typical case in which the call of start id deferred because some resources used by entry actions of input
states are probably not ready here...

}

protected void finalize() throws Throwable { // finalize process raises problems in Java! Be careful here...

    to_be_killed();

}

```

6.3.5 Component diagram

The component diagram in Figure 35 shows a more operational or implementation-related view of the *Home Automation System* than the communication diagram in Figure 33. It especially stresses the visualization⁵⁹ of all of the provided interfaces and how objects/components have to refer to each other in order to instrument communication. Some design decisions do occur, like for instance having two switches, three relays and a run indicator as “surrounding” objects of the thermostat instead of “true”⁶⁰ software components.

Regarding the model in Figure 34, the thermostat often needs to determine the current state of the season switch and the fan switch. Hence, the *in(name : String) : Boolean* method assigned to the *Season switch* and *Fan switch* types in Figure 35 is based on the following implementation:

```

public class Season_switch implements Season_switch_input, VelcroExecutor {

    // state and other stuff declarations here

    protected AbstractStatechart_monitor _Season_switch;

    // varied methods here including events, actions...

```

⁵⁹ For the sake of clarity, details like cardinalities, attributes or other things have been deliberately omitted.

⁶⁰ Especially From a deployment viewpoint, this means that components, contrary to objects, have their own capabilities like running inside another Java virtual machine for instance.

```

public boolean in(String name) {

    return _Season_switch.in_state(name);

}
}

```

In this code, we simply use the *in(name : String) : Boolean* method borne by the *PauWare AbstractStatechart_monitor* class.

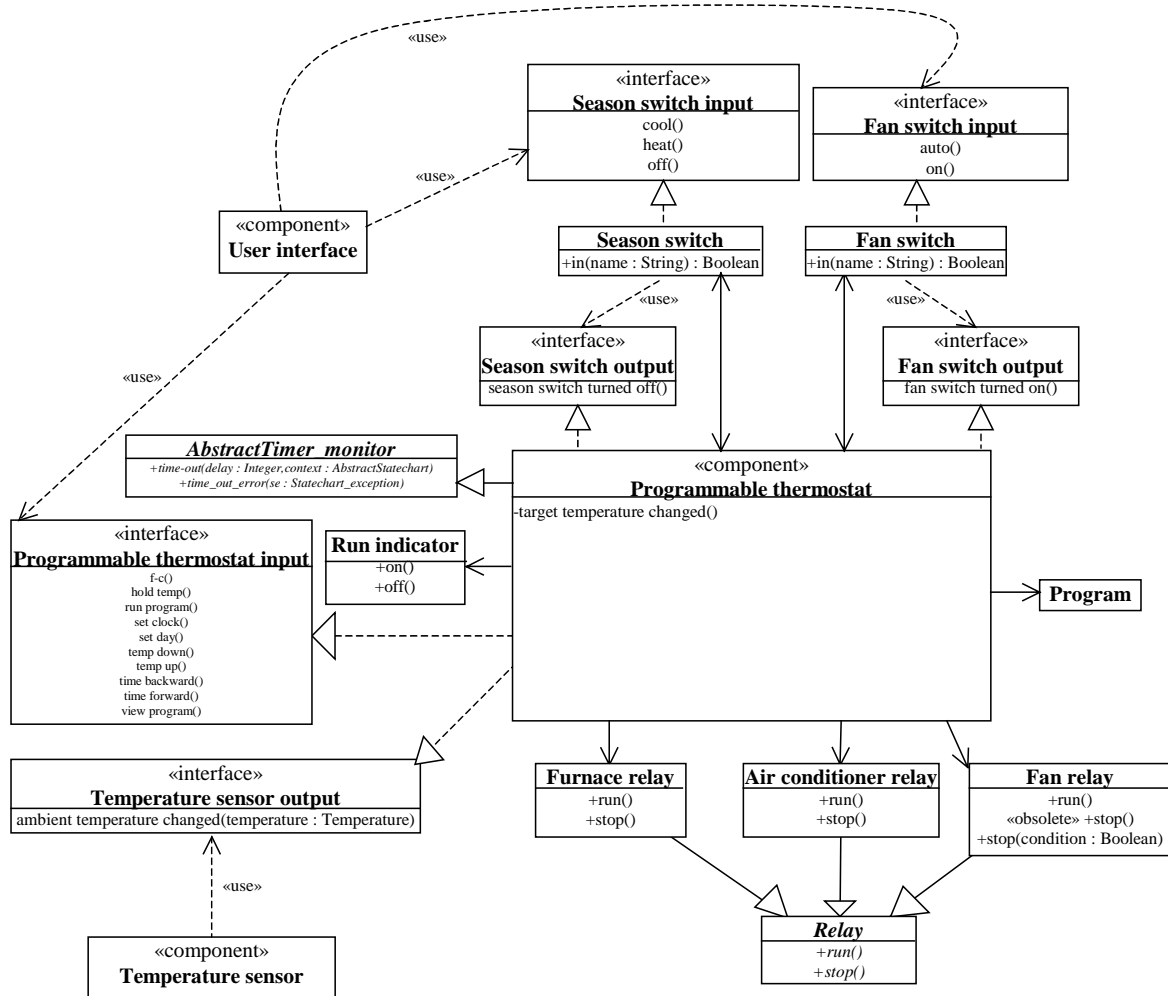


Figure 35. The UML Component Diagram of the Home Automation System.

Fig. 5 also contains other interesting design decisions. Three key interfaces (*Timer client*, *User interface client* and *Temperature sensor client*) are depicted in order to carefully plunge the thermostat component into its environment. In fact, events processed in the thermostat's statechart become public operations. All events are however divided into different *logical* interfaces in compliance with the communication stated in Fig. 4.

Unidirectional communication (e.g. from *Temperature sensor* to *Programmable thermostat*) enhances the provided interface of the thermostat component, like for example the *ambient temperature changed* public (+ symbol in UML) service. Here, this generates low coupling since the thermostat does *not* need to acquire temperature measurement facilities at deployment time.

Bi-directional exchanges (e.g. *Timer/Programmable thermostat*) have impacts on both the provided (*time-out* public operation in *Programmable thermostat*) and on the required (*to be set* and *to be killed* public operations in *Timer*) interfaces of the thermostat component. The deployment of the thermostat component implies the identification of appropriate timer event services like those in CORBA for instance. In Fig. 5, the arrow from

Programmable thermostat to *Timer* demonstrates that the thermostat requires connection with these timer event services. This means that the choice of a suited exchange protocol is linked to implementation concerns.

6.3.6 Java ME specificities

As previously written, the key point with Java ME is the avoidance of the Java reflection mechanism. Actions (including the self-sending of events), activities and guards cannot therefore be called within run-to-completion cycles by means of reflection. There are two ways of dealing with actions, activities and guards in Java ME. For actions, activities and guards that **are member functions⁶¹⁶² of the software component being built**, the latter requires implementing the *VelcroExecutor* interface as follows:

```
public class Programmable_thermostat extends AbstractTimer_monitor implements...,VelcroExecutor {
    ...
    public Object execute(String action,Object[] args) throws Throwable {
        Object result = null;
        if(action != null && action.equals("display_ambient_temperature")) display_ambient_temperature();
        if(action != null && action.equals("display_current_date_and_time"))
            display_current_date_and_time();
        if(action != null && action.equals("display_period_and_program_time"))
            display_period_and_program_time();
        if(action != null && action.equals("display_program_target_temperature"))
            display_program_target_temperature();
        if(action != null && action.equals("display_target_temperature")) display_target_temperature();
        if(action != null && action.equals("set_alternately")) set_alternately((Integer)args[0]);
        if(action != null && action.equals("set_no_input")) set_no_input((Integer)args[0]);
        if(action != null && action.equals("set_period")) set_period((Integer)args[0]);
        if(action != null && action.equals("set_program_time")) set_program_time((Integer)args[0]);
        if(action != null && action.equals("set_target_temperature"))
            set_target_temperature((Integer)args[0],(Integer)args[1]);
        if(action != null && action.equals("set_time")) set_time((Integer)args[0]);
        if(action != null && action.equals("switch_mode")) switch_mode();
        if(action != null && action.equals("target_temperature_changed")) target_temperature_changed();
        if(action != null && action.equals("no_input_less_than_ninety")) result = new
            Boolean(no_input_less_than_ninety());
        if(action != null && action.equals("not_weekend")) result = new Boolean(not_weekend());
        if(action != null && action.equals("weekend")) result = new Boolean(weekend());
        return result;
    }
}
```

61 As written above, within J2ME, actions, activities, guard evaluators and invariant evaluator's member functions may be declared *private* or *protected* since we do not use the Java reflection API.

62 Normally activities, guard evaluators and invariant evaluators cannot be something other than member functions of the software component being built. So, only special actions may be based on this rule.

```

    }
}

```

In this code, the *public Object execute(String action, Object[] args) throws Throwable* Java method has to be populated based on a test of the *action* parameter and, if not enough, the *args* parameter (in the code above, it is not necessary to test *args*). A related member function is next called using, if necessary, the *args* parameter. However, one has to call the reader's attention to the fact that only *args* must be used. For instance, it could be tempting to call the *set_no_input* member function in bypassing *args* as follows:

```
if(action != null && action.equals("set_no_input")) set_no_input(new Integer(_no_input));
```

Since *_no_input* is a local variable of the component being built, one may believe that the two styles are equivalent. In fact, the value of *_no_input* must be equal to the "good one". This is the value which is caught just before starting the run-to-completion cycle. Within a run-to-completion cycle, several actions may alter this variable and, as a result, the line of code above may use an inappropriate value.

Finally, the body of *public Object execute(String action, Object[] args) throws Throwable* is such that the computed result is returned in order to be operated within the in progress run-to-completion cycle.

It remains a case when actions to be launched in response to events are not member functions of the component being built. In such a case, one needs to construct a class which implements the *VelcroActionExecutor* interface as follows:

```
public class Programmable_thermostatActionExecutor implements VelcroActionExecutor {

    /** constructor without arguments is mandatory */

    public Object execute(Object object,String action,Object[] args) throws Throwable {

        Object result = null;

        if(action != null && action.equals("decrement")) ((Temperature)object).decrement();

        if(action != null && action.equals("increment")) ((Temperature)object).increment();

        return result;

    }

}

```

In this code, the *Programmable_thermostatActionExecutor* class requires a constructor without parameters since it is dynamically loaded at the time that actions must be executed within a run-to-completion cycle. As an illustration, the *temp down* event decrements the target temperature of the thermostat when it is in the *Hold* or *Run* states. The *decrement* action is not a member function of the *Programmable thermostat* component. As a result the implementation of the *temp down* event via the *temp_down* Java method requires the following Java ME style:

```
_Programmable_thermostat.fires(_Run,_Run,guard,_target_temperature,"com.FranckBarbier.Java._Home_automation_system._Programmable_thermostat.Programmable_thermostatActionExecutor.decrement");
```

```
_Programmable_thermostat.fires(_Hold,_Hold,guard,_target_temperature,"com.FranckBarbier.Java._Home_automation_system._Programmable_thermostat.Programmable_thermostatActionExecutor.decrement");
```

The *Programmable_thermostatActionExecutor* class is passed as a parameter of the *fires* method instead of having a more concise style of programming (*i.e.*, *Composytor*):

```
_Programmable_thermostat.fires(_Run,_Run,guard,_target_temperature,"decrement");
```

```
_Programmable_thermostat.fires(_Hold,_Hold,guard,_target_temperature,"decrement");
```

The *public Object execute(Object object,String action,Object[] args) throws Throwable* Java method in the *Programmable_thermostatActionExecutor* class therefore simply carries out the execution on behalf of the *object* parameter. The latter is by definition different from the software component which has the state machine.

As a conclusion of this section about Java ME, the reader may first consider the *Velcro* style of programming as inelegant. This is true compared to reflection and thus to *Composytor*. Nevertheless, the reader may be interested in this style since it is less costly than reflection. Within the Java SE platform, it is indeed possible to use the *Velcro* style of programming, but this style does not comply with Java EE. This is especially true concerning communication.

7 Using *PauWare* with the Java EE platform

There are no specific coding rules when using *PauWare* with Java EE (tests have been run with Java EE 1.4) except the fact that the native Java reflection mechanism has been extended in the *PauWare* engine (*Composytor* API) due to its lack of power. More precisely, since in Java EE, EJB components are accessed by means of their remote interface, this interface is used when a component calls for a service of another component. Since a Java EE server replaces remote interfaces by concrete classes, our extension is intended to locate methods by reflection in scanning inheritance and implementation⁶³ graphs. For the average developer, this approach is of course transparent and allows her/him to ground its EJBs in state machines. However, since state machines constitute the inside of software components, only *Stateful Session Beans* are candidate components to be built with *PauWare*. Equipping *Entity Beans* and *Stateless Session Beans* with the *PauWare* code is also possible, but this requires an appropriate style of programming, which is not described in this guide. Moreover, the recent availability of Java EE 1.5 and the EJB 3.0 framework creates new perspectives which are not explored here.

In this section, we focus on the architectural style imposed by the concomitant use of EJB 2.1 and *PauWare*. We especially show how JMS (which stands for *Java Message Service*) enhances *PauWare*'s capabilities by using *Message-Driven Beans* for communicating. In fact, we may use synchronous communication (*i.e.*, *Remote Method Invocation* or RMI) through calls, which relies on remote interfaces. In contrast, *Message-Driven Beans* are appropriate instruments of asynchronous communication. Finally we discuss the configuration of EJB components based on the adequate setup of their deployment properties. For illustration purposes, we discuss a case study named *Railcar control system*.

Finally, one may notice that all of the models have been built using version 2 of the UML.

7.1 Requirements

This case study is an adaptation of another one presented in a paper written by David Harel and Eran Gery in 1997 in IEEE Computer: "Executable Object Modeling with Statecharts". This is a railcar control system that is made up of several contiguous terminals. There are two rail tracks (clockwise and counterclockwise directions) connecting each adjacent terminal pair (Figure 36). In Figure 38, we complementarily provide a *UML Class Diagram*, which models this physical situation.

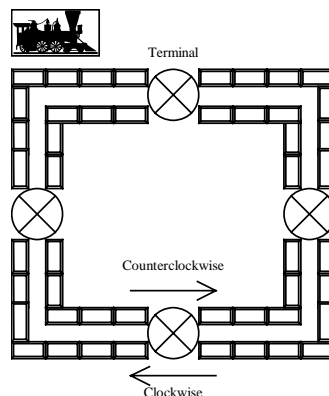


Figure 36. Railcar control system with four terminals.

⁶³ We here mean the Java *implements* relationship.

Infrastructure constraints raise the following problem: at a given instant, only one railcar can be parked at a terminal. There is indeed a parking platform that enables the change of direction for an incoming railcar. However, for the railcar management system to be built, there are no special issues about traffic control. This means that a railcar can know its direction (*i.e.*, clockwise or counterclockwise) but cannot make decisions based on its direction (see also Figure 45).

Moreover, a railcar keeps its direction when going through a terminal or when stopping in order to drop off and/or to board passengers. Thus, we do not include the possibility for changing directions in a primary version of the software application. This is specified by means of the UML and is developed with EJBs and in *PauWare*.

In addition, there is no specific management of the number of railcars riding on the two rail tracks between two given contiguous terminals. As a result, a terminal enables the exit of a railcar which leaves it, without checking the occupation of its two (clockwise and counterclockwise) outgoing railway segments.

Now concerning its two incoming segments, a terminal has to receive events produced by a control center. These event types are: *alert100*, *terminal crossing* and *terminal stopping* (see also Figure 44). A railcar also receives these three types of events, as well as *alert80*, which comes from the aforementioned control center (Figure 37).

For a terminal, each *alert100* event occurrence leads to recording within an ordered set, the identity of the approaching railcar. This set has to be chosen among the *clockwise ingoing vehicle* and *counterclockwise ingoing vehicle* sets (Figure 42). The *terminal stopping* event means for a terminal that the parking platform is going to become busy at the time a railcar arrives at this terminal. The *Busy* state symbolizes the fact that the arriving railcar stops in order to drop off and/or to board passengers. Otherwise, the arriving railcar goes through the terminal without stopping. The latter terminal is accordingly notified by means of a *terminal crossing* event issued from the control center.

When becoming busy, a terminal has the responsibility of sending the *stop* event to all of its remaining arriving railcars (*clockwise ingoing vehicle* and *counterclockwise ingoing vehicle* sets). Later, when becoming non-busy, because its parked railcar leaves it, this same terminal has to send the *go on* event to all of the railcars that are temporarily stopped on the clockwise and counterclockwise incoming portions. One may purposely notice that a parked railcar is released after 30 sec. by means of a *go* order coming from its parking terminal.

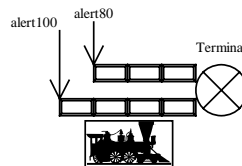


Figure 37. The Position of a railcar between 100 and 80 distance units from a destination terminal.

For a railcar, the *alert100* et *alert80* events lead to slowing down. This action is accomplished by controlling a cruiser. Within this process, a decision of stopping or going through the approached terminal has to be taken according to the value of a *destination board* property (see Figure 42), which is a set of terminals corresponding to the destinations chosen by passengers⁶⁴. Either the current approached terminal is included within this set or it is not. In the first case, the approaching railcar must stop. In the second case, this railcar is supposed to go through the approached terminal. In an equivalent way, the approached terminal also has a destination board feature (see Figure 42).

So, even if the approaching railcar might deduce that it has to go through its approached terminal, this terminal may want this railcar to stop in order to board passengers waiting for other terminals. In terms of business logic, the question then becomes: How to dispatch the decision making process? In other words, “stopping” or “going through” is either a decision from the approaching railcar or its associated approached terminal. As written before, we leave this requirement open to experimentation⁶⁵. So, in order to provide a temporary simplified solution, we propose an interaction protocol in which the control center arbitrarily sends the *terminal stopping* or *terminal crossing* orders: The decision making process is thus temporarily externalized

⁶⁴ This property evolves by means of a GUI and a keyboard available in each railcar. A *new destination(another terminal)* event embodies the adding of a terminal, if not already present, to the destination set. The same support and mechanism is available in each terminal.

⁶⁵ This especially amounts to designing a more complex interaction protocol between the *Terminal* and *Railcar* components (see Figure 45 and Figure 46).

of the offered solution. In other words, we construct a centralized system in which the control center makes all of the critical decisions. This supposes the presence of a sensor network which safely notifies the control center of the “physical” situation of railcars and terminals.

So, an approaching phase starts at the time an *alert100* event occurs. This event is received both by the concerned railcar and terminal. An *alert80* event is then sent to the railcar in order to make it listen to its approaching terminal. For that, the railcar notifies the terminal with an *approaching* event. Finally, the terminal is able to forward the *terminal crossing* and *terminal stopping* events to its approaching railcar. In the same time, it must be able to send the *stop* and *go on* orders to some other arriving railcars. As for the *go* order, it embodies the release of the stopped railcar, if it has previously stopped and thus not gone through the terminal.

7.2 UML models

This section is divided into a static specification (*UML Class Diagrams* and related invariants) and a dynamic specification including *UML State Machine Diagrams*. Invariants and pre/post-conditions associated with events are also parts of the dynamic specification.

7.2.1 Structure

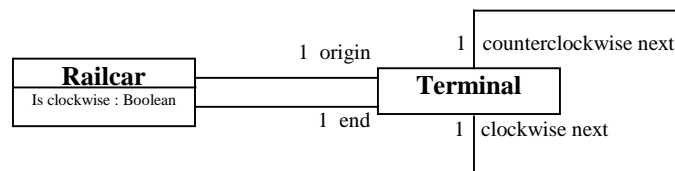


Figure 38. The Railcar control system, *Topology* package.

The UML model in Figure 38 indicates the positioning of terminals and railcars. Since terminals do not move, this especially tells us what the structure of the railcar control system is. For instance, one may imagine the T1, T2, T3 and T4 terminals and how they depend upon each other: T2 follows T1 in the clockwise direction, etc (Figure 39). As for railcars, we may have an initial situation: R1 and R2 between T1 and T2 in the clockwise direction or R3 between T2 and T3 in the counterclockwise direction (Figure 39).

The model in Figure 38 is intended to be enriched with the following invariants:

context **Terminal** inv:

self <> clockwise next

self <> counterclockwise next

self = clockwise next.counterclockwise next

These invariants state that, for instance, a terminal cannot be its successor or its predecessor.

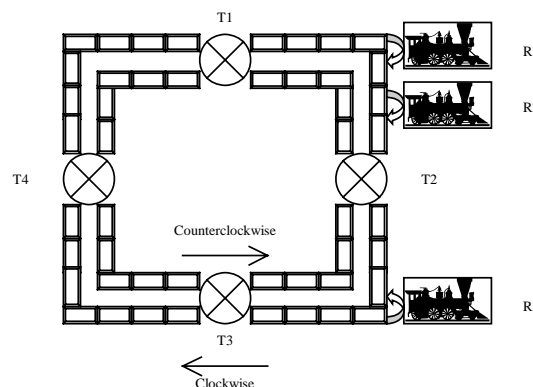


Figure 39. Initial data: 4 terminals, 2 clockwise railcars between T1 and T2 and 1 counterclockwise railcar between T3 and T2.

From a design viewpoint, the following database scheme (Figure 40) results from the model in Figure 38.

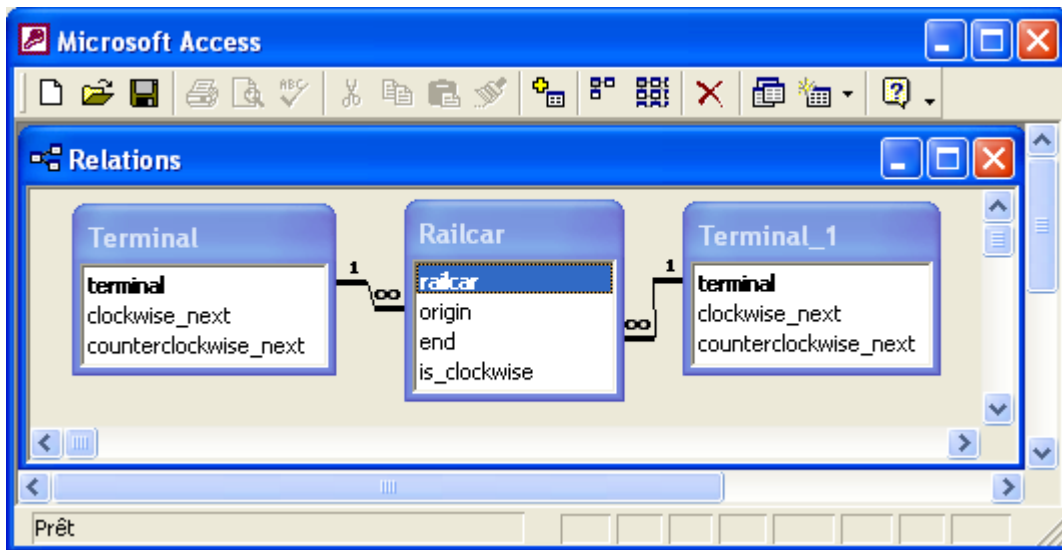


Figure 40. Database scheme of the UML *Topology* package.

The scheme in may be populated with data (Figure 41) in order to strictly match to the context in Figure 39.

terminal	clockwise_next	counterclockwise_next
T1	T2	T4
T2	T3	T1
T3	T4	T2
T4	T1	T3

railcar	origin	end	is_clockwise
R1	T1	T2	-1
R2	T1	T2	-1
R3	T3	T2	0

Figure 41. Data instantiating the model in Figure 38 and corresponding to the context of Figure 39.

Now, we are interested in *Platform-Specific Models* (PSMs), as it happens, the dependencies of EJB components. In Figure 42, we have dynamical features. Namely for terminals and railcars, this is a set of destination terminals (*destination board* UML navigation) which has to evolve frequently according to the *new destination*(*another terminal*) event.

The two other key navigations are the *clockwise ingoing vehicle* and *counterclockwise ingoing vehicle* navigations which enable the recording and the management of a terminal's incoming railcars. For a given terminal, the elements in these two sets often change. This is due the *alert100* event in particular, which makes this terminal aware of the necessity of including a new approaching railcar in the *clockwise ingoing vehicle* set or (exclusively) the *counterclockwise ingoing vehicle* set. This is possible thanks to the Boolean parameter (*is_clockwise*) of the *alert100* event.

Finally, the *railcars* and *terminals* navigations from the *Control center Stateful Session Bean* and the *Railcar* and *Terminal Stateful Session Beans*, are derived from those in Figure 38. In fact, since the control center is responsible for making decisions, it encapsulates and accordingly updates the data in Figure 41. The *railcars* and *terminals* navigations in Figure 42 are therefore supports (*i.e.*, two Java collections of remote interface references) for sending the *alert100*, *terminal crossing*, *terminal stopping* and *alert80* events.

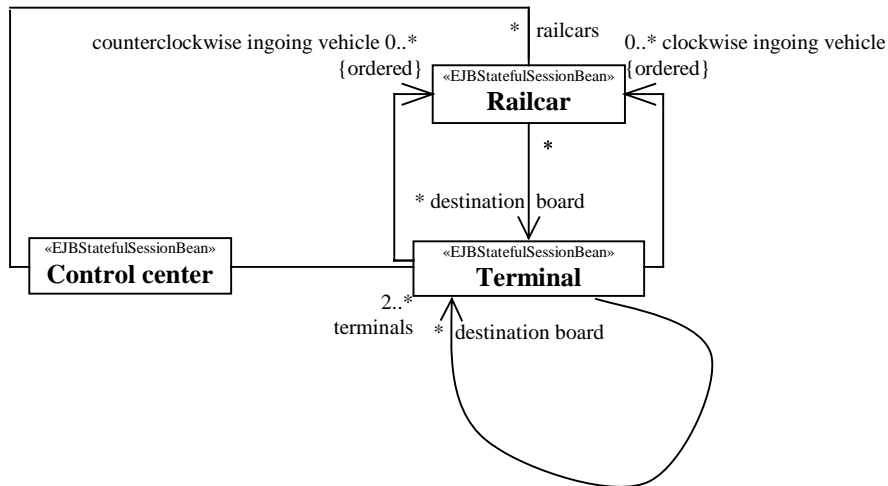


Figure 42. Railcar control system, Situation package

Finally, we supplement the model in Figure 42 with the following invariant:

context **Terminal** inv:

not **destination board**->**includes(self)**

7.2.2 Behavior, UML Interaction Diagrams

UML offers a lot of dynamic diagrams including sequence diagrams like that in Figure 43 and communication diagrams (which is a subtype of sequence diagrams) like that in Figure 44. We use these two styles of modeling for sketching scenarios in general, but their imprecise nature is not a robust design guide, especially when *PauWare* is used.

In Figure 43, we show a *Terminal approach* scenario with alternatives (*alt* operator). This model is limited in scope since only instances are depicted (: symbol before a type name like *Control center* for instance). This leads to only showing one-to-one communication. However, the railcar control system is essentially based on more complex interactions (many-to-many exchanges). Furthermore, the two *alt* blocks in Figure 43 should be related to each other since a terminal forwards the *terminal crossing* event to its approaching railcar if it has previously received this same event with the said railcar as parameter from the control center. This is the same for the *terminal stopping* event. So, the model in Figure 43 is rather informal, but it has the quality of being intuitive. This is the same for the *UML Communication Diagram* in Figure 44: we have a good overview of the *Railcar control system*, but additional material is required to design the software application.

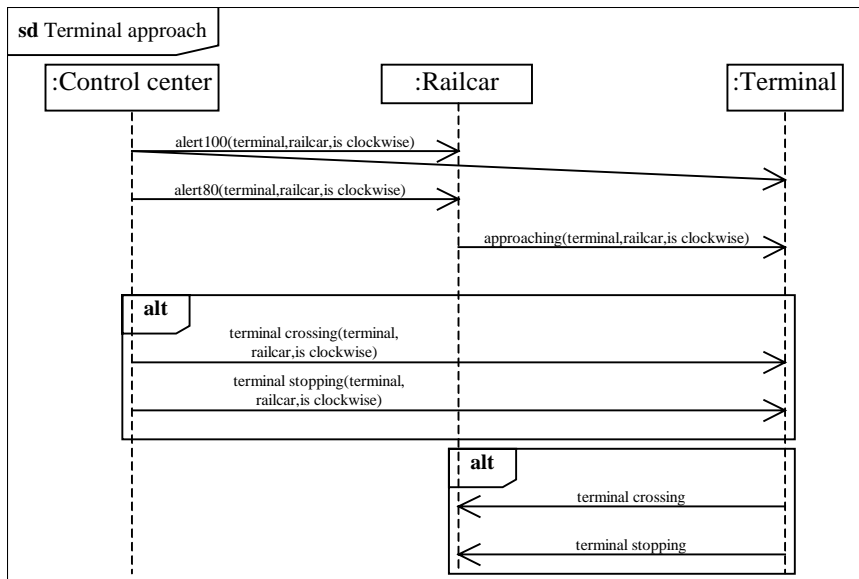


Figure 43. Railcar control system's sequence diagram (partial case).

The communication diagram in Figure 44 has the advantage of showing all of the system's possible control flow regardless of time scale. The reader may notice that, once again, only instances are depicted, preventing the definition of conditions/rules for many-to-many component interactions.

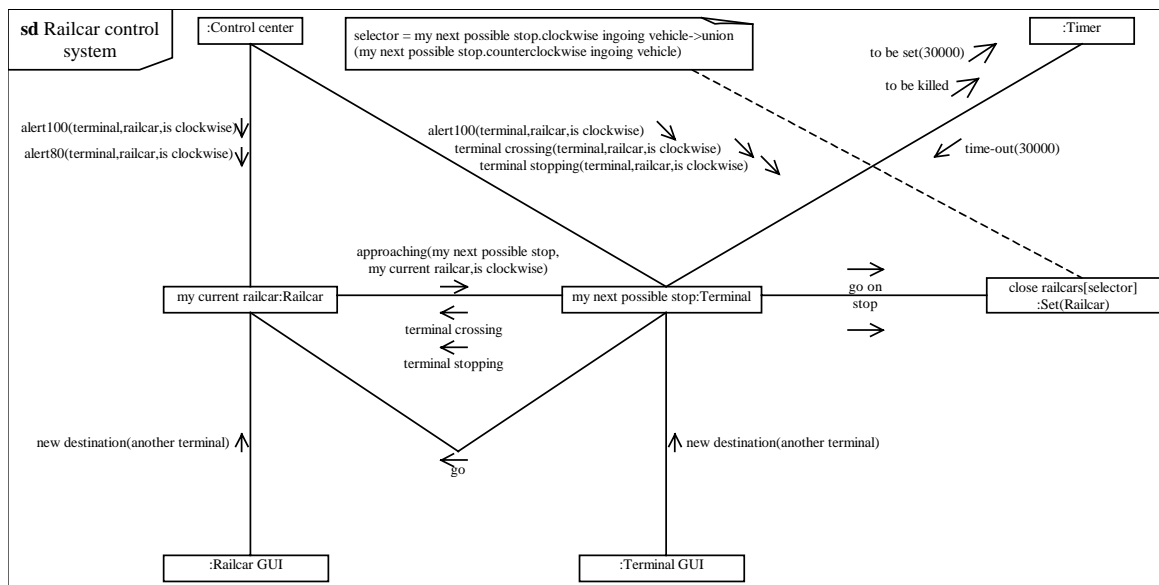


Figure 44. Railcar control system's communication diagram.

7.2.3 Behavior, Railcar component

We here stress *UML State Machine Diagrams* since we are able to fully derive code from them with *PauWare*.

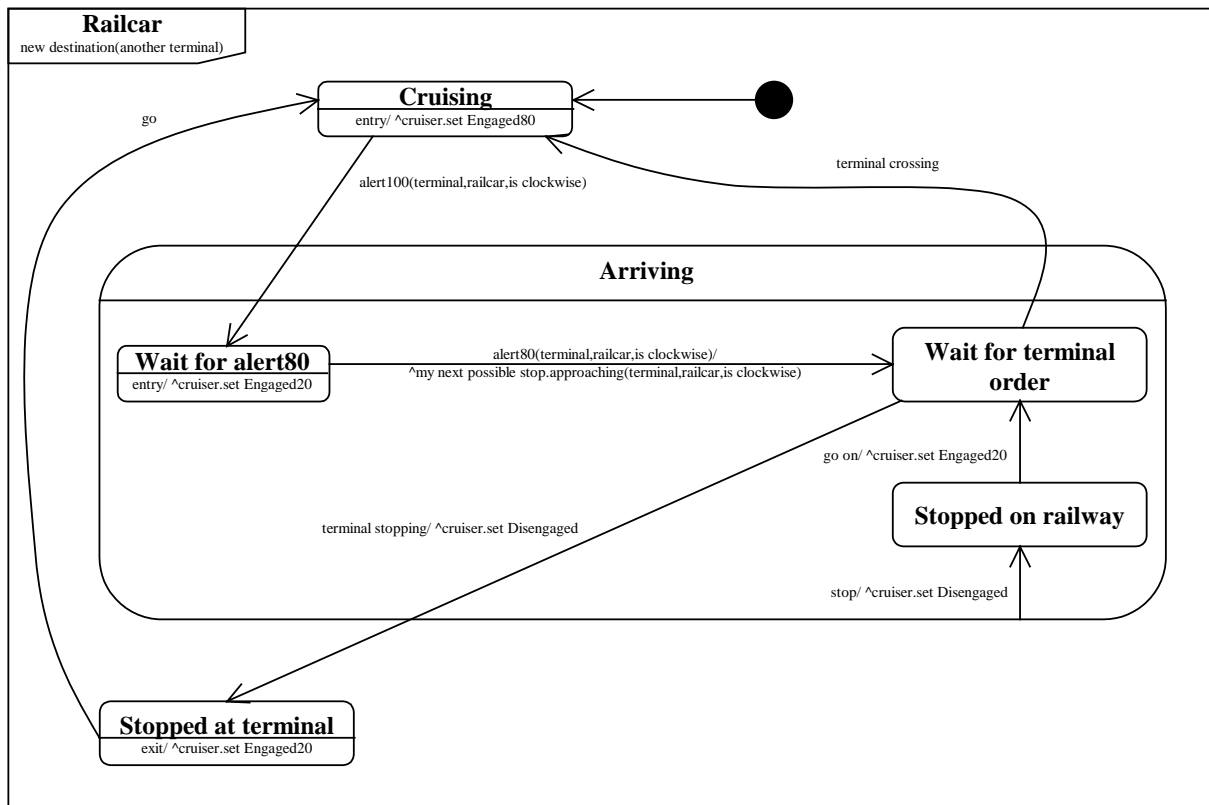


Figure 45. UML State Machine Diagram for the Railcar component.

UML is not sufficiently formal to express the behavior of components. OCL must also be used for underpinning events with pre/post-conditions. There is no special support in *PauWare* for implementing pre-conditions as long as the requester is supposed to ensure pre-conditions. As for event post-conditions, a Java method embodying the processing of an event must first include some code which accomplishes and thus guarantees these event's post-conditions. Taking the example of the *alert100* event received by the *Railcar* component (see Figure 45), the following OCL constraints apply:

context **Railcar::alert100(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)**

pre: self = railcar

post: my next possible stop = terminal -- *my next possible stop* is a local variable of the *Railcar*

-- component's state machine

In terms of implementation, this leads to the following code:

```

public void alert100(TerminalRemote terminal,RailcarRemote railcar,Boolean is_clockwise) throws
Statechart_exception {
    /** pre-condition(s) */
    try {
        if(! this._ejb_context.getEJBOBJECT().isIdentical(railcar)) throw new
Statechart_exception("Railcar/alert100, pre-condition(s) violation");
    } catch(java.rmi.RemoteException re) {
        throw new Statechart_exception("Railcar/alert100, pre-condition(s) violation: " + re.getMessage());
    }
    /** post-condition(s) */
    try {
        _my_next_possible_stop = terminal.getHandle();
    } catch(java.rmi.RemoteException re) {
        throw new Statechart_exception("Railcar/alert100, post-condition(s) violation: " + re.getMessage());
    }
    _Railcar.run_to_completion("alert100");
}
  
```

```
}
```

The transitions concerning the *alert100* event have been setup in the *start* Java method (see also Section 3.1.7 about this method) as follows:

```
protected void start(String railcar) throws Statechart_exception {  
    _Railcar = new Statechart_monitor((_Arriving.xor(_Cruising)).xor(_Stopped_at_terminal),railcar,true);  
    _Railcar.fires("alert100",_Cruising,_Wait_for_alert80);  
    ...  
}
```

The remaining pre/post-conditions for the events received by the *Railcar* component are:

context **Railcar::alert80**(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)

pre: self = railcar

pre: my next possible stop = terminal

context **Railcar::new destination**(another terminal : Terminal)

post: destination board->includes(another terminal)

context **Railcar::terminal crossing**()

post: not destination board->includes(my next possible stop) -- useless, *a priori*

post: my next possible stop.ocllsUndefined() -- no terminal destination must be setup

context **Railcar::terminal stopping**()

post: not destination board->includes(my next possible stop)

post: my next possible stop.ocllsUndefined() -- no terminal destination must be setup

Another possible implementation style for post-conditions is when events are allowed (see also Section 5.5). For instance, the *new destination* event must be accepted at any time by the *Railcar* component (see top left hand side of Figure 45). As result, the processing of this event benefits from being implemented as follows:

```
public void new_destination(String another_terminal) throws Statechart_exception {  
    Object[] args = new Object[1];  
    args[0] = another_terminal;  
    _Railcar.allowedEvent("new_destination",this,"new_destination_post_conditions",args);  
    _Railcar.run_to_completion("new_destination");  
}
```

In this code, the single post-condition (see above) of *new destination* is materialized as a Java method named *new_destination_post_conditions* and called by reflection.

7.2.4 Behavior, *Terminal* component

The *Terminal* component also has accurate behavior specified via the model in Figure 46.

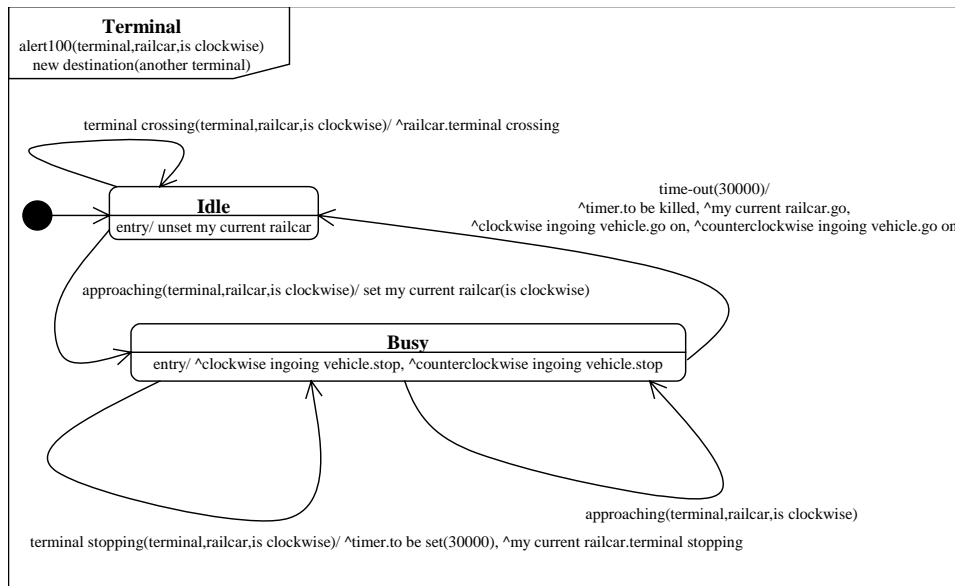


Figure 46. UML State Machine Diagram for the Terminal component.

The pre/post-conditions for the events received by the *Terminal* component are:

context **Terminal::alert100**(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)

pre: **self = terminal**

post: **is clockwise** implies **clockwise ingoing vehicle->last()** = railcar -- arriving railcars are put into a FIFO

-- queue (*{ordered}* constraint in UML), clockwise direction

post: **not is clockwise** implies **counterclockwise ingoing vehicle->last()** = railcar -- arriving railcars are put

-- into a FIFO queue (*{ordered}* constraint in UML), counterclockwise direction

context **Terminal::approaching**(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)

pre: **self = terminal**

context **Terminal::new destination**(another terminal : Terminal)

post: **destination board->includes**(another terminal)

context **Terminal::terminal crossing**(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)

pre: **self = terminal**

pre: **my current railcar = railcar**

context **Terminal::terminal stopping**(terminal : Terminal,railcar : Railcar,is clockwise : Boolean)

pre: **self = terminal**

Two important actions are internally used within the *Terminal* component. These are *set my current railcar* and *unset my current railcar*. They are described by means of OCL (see below) and correspond to the assignment, respectively the release, of a *Terminal* component state machine's local variable named *my current railcar*. The assignment is realized by extracting the first value of either the *clockwise ingoing vehicle* or *counterclockwise ingoing vehicle* ordered sets.

context **Terminal::set my current railcar**(is clockwise : Boolean)

post: is clockwise implies `my current railcar = clockwise ingoing vehicle->first()` and `clockwise ingoing vehicle = clockwise ingoing vehicle@pre->reject(railcar | railcar = my current railcar)`

post: not is clockwise implies `my current railcar = counterclockwise ingoing vehicle->first()` and `counterclockwise ingoing vehicle = counterclockwise ingoing vehicle@pre->reject(railcar | railcar = my current railcar)`

context `Terminal::unset my current railcar()`

post: `my current railcar.ocllsUndefined()`

A final interesting issue is the implementation of communication. As written above, using the remote interfaces of components is easy and straightforward. Sending the *go* order for instance might lead to what follows:

```
protected void start(String terminal) throws Statechart_exception {
    _Terminal = new Statechart_monitor(_Busy.xor(_Idle),terminal,true);
    ... // some transitions here
    _Terminal.fires("time_out",_Busy,_Idle,true,_my_current_railcar.getEJBOBJECT(),"go");66
    ... // some other transitions here
}
```

Even if this code works perfectly well, it raises some architectural problems discussed in the next section. An alternative is the use of JMS to send events asynchronously:

```
protected void start(String terminal) throws Statechart_exception {
    _Terminal = new Statechart_monitor(_Busy.xor(_Idle),terminal,true);
    ... // some transitions here
    _Terminal.fires("time_out",_Busy,_Idle,true,this,"my_current_railcar_go");
    ... // some other transitions here
}
```

In this code, a local Java method named *my_current_railcar_go* is called by reflection. This method is as follows:

```
public void my_current_railcar_go() throws java.rmi.RemoteException,JMSException {
    _Railcar_queue_sender.send(_Railcar_queue_session.createObjectMessage(new
    TerminalRailcarIs_clockwise("go",this._ejb_context.getEJBOBJECT().getHandle(),_my_current_railcar,null)
    );
}
```

This code has the simplicity of concomitantly using *PauWare* and JMS. We elaborate on the interest and deep nature of this code in the next section.

7.3 Architecture

This section discusses the coercive computing framework provided by EJBs. Although *PauWare* has the capacity to process concurrent events, in order to respect the CBD spirit, state machines are encapsulated in software components. So requests arrive with component's remote interfaces playing the role of entry points.

⁶⁶ In conformance with EJBs, the type of *my_current_railcar* is *javax.ejb.Handle*. The call for *getEJBOBJECT()* therefore enables the return of the component's remote interface.

Unfortunately, state machines generate events based on a broadcast mode of communication (that of the original Harel's Statecharts) and thus several requests may attain the same component concurrently. This is a computing model which is not tolerated by EJBs. As a result, the model in Figure 47, which is intuitive in the sense that it complies properly to the way communication operates (see Figure 43 and Figure 44), is a “strong” *Platform-Specific Model*. In this architecture, a given *Railcar* component instance may receive an *alert100* event from the control center and a *stop* event from its approached terminal at the same time.

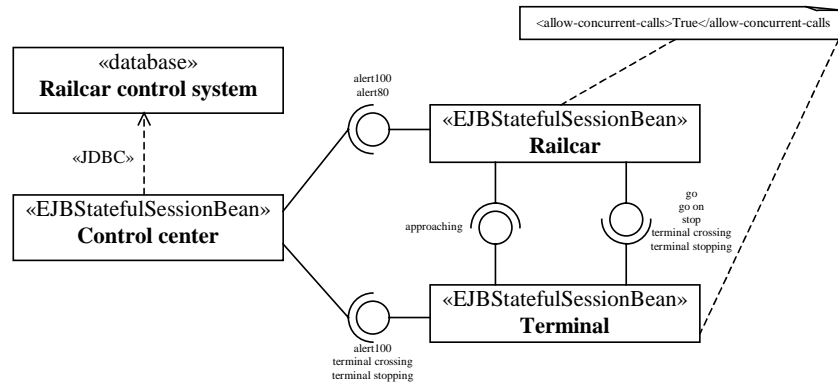


Figure 47. PSM, EJB 2.1-incompatible architecture (BEA WebLogic).

We name the micro-architecture in Figure 47 a strong PSM since it can nevertheless be implemented on top of the *BEA WebLogic* Java EE server. This is subject to the use of the *allow-concurrent-calls* deployment proprietary parameter. This parameter must be set to *true* for both the *Railcar* and *Terminal Stateful Session Beans*. So, this works but this is not compliant with the EJB 2.1 specification⁶⁷.

The problem raised here can be solved by using an appropriate architecture like that in Figure 48. Even if this PSM calls for *Message-Driven Beans* and thus JMS, the main point is that one must have façade components, which make incoming events queue up. These façade components next deliver these events, one by one. This prevents any concurrency for the recipient component.

Of course *Message-Driven Beans* may play the role of façade components (this is the case in Figure 48) but extra tuning is required. Formally, a given recipient component like a *Railcar* component instance requires one and only one façade component. Indeed, having more than one façade component restores the risk of receiving concurrent requests for the recipient component. The problem with *Message-Driven Beans* is that a Java EE server arbitrarily creates and duplicates the instances of this EJB category. Furthermore, it can be difficult to estimate the number of *Railcar* component instances⁶⁸ at a given time; and thus, to know how many façade *Message-Driven Bean* instances are needed.

The solution in Figure 48 is having one façade *Message-Driven Bean* per one *Stateful Session Bean type* (namely the *Railcar* and *Terminal* types) whose instances are likely to be called concurrently. As a result, two *Message-Driven Bean* types are designed: *Railcar event queue* and *Terminal event queue*. **However, at runtime, only one instance of each type is created.** This is due to the fact that the *Max Pool Size* deployment parameter is set to 1.

⁶⁷ No test has been run with the EJB 3.0 computing framework.

⁶⁸ This is not really the case in our case study but, by generalization, this can be a big problem in any other business application.

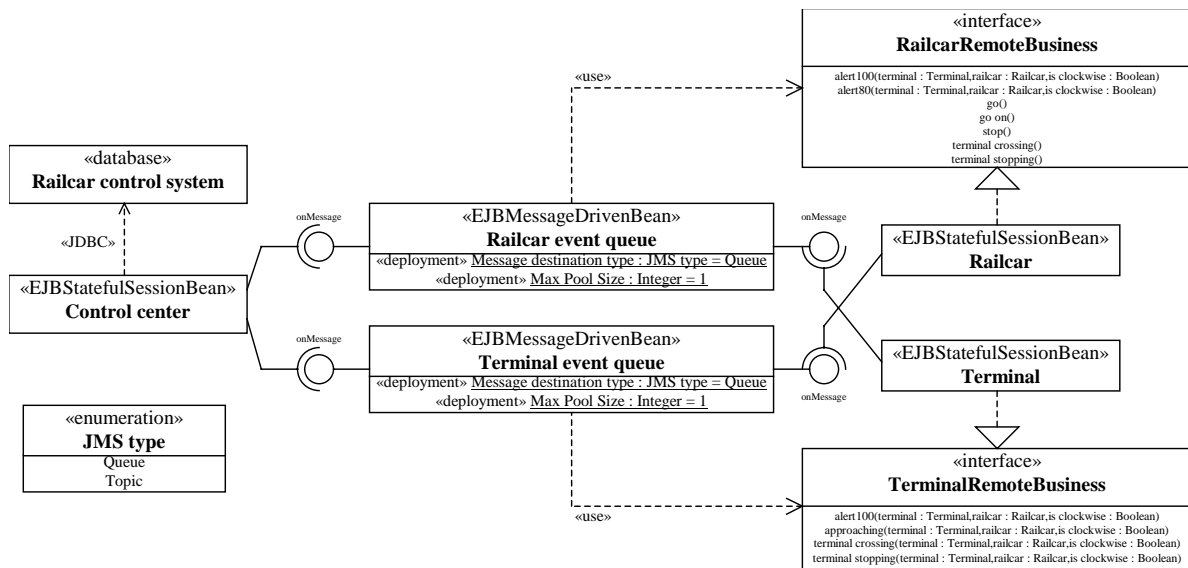


Figure 48. PSM, EJB 2.1-compliant architecture.

So, all of the requests to any railcar and to any terminal are processed by these two instances (respectively, *Railcar event queue* and *Terminal event queue*). The single remaining question is: How the routing to the recipient component is operated? The first method consists in passing JMS *ObjectMessage* instances, which have the following objects⁶⁹ as contents:

```
public class TerminalRailcarIs_clockwise implements java.io.Serializable {

    protected String _event;

    protected javax.ejb.Handle _terminal;

    protected javax.ejb.Handle _railcar;

    protected Boolean _is_clockwise;

    public TerminalRailcarIs_clockwise(String event, javax.ejb.Handle terminal, javax.ejb.Handle railcar, Boolean is_clockwise) {

        _event = event;

        _railcar = railcar;

        _terminal = terminal;

        _is_clockwise = is_clockwise;

    }

}
```

The routing is consequently as follows (example for the *Railcar_event_queue Message-Driven Bean*):

```
public void onMessage(Message message) {

    ObjectMessage object_message;

    if(message instanceof ObjectMessage) {
```

⁶⁹ One may note that the *TerminalRailcarIs_clockwise* Java class implements the *java.io.Serializable* interface so that its instances can be routed via JMS. Furthermore, within the *Composytor* API, the *Statechart_monitor PauWare* class also implements the *java.io.Serializable* interface.

```

object_message = (ObjectMessage)message;

try {

    TerminalRailcarIs_clockwise message_data =
    (TerminalRailcarIs_clockwise)object_message.getObject();

    if(message_data._event != null && message_data._event.equals("alert100"))
    ((RailcarRemote)message_data._railcar.getEJBObject()).alert100(((TerminalRemote)message_data
    a._terminal.getEJBObject()),((RailcarRemote)message_data._railcar.getEJBObject()),message_data
    ta._is_clockwise);

    ...
}

```

8 Software component management

A powerful key utilization of *PauWare* is linked to the possibility of having an explicit, tangible and comprehensive access and control of the behavior of a software component. This may be done remotely and in relation with a discrete time scale (each run-to-completion step). To have manageable components, these must implement the *Manageable* interface (Figure 49).

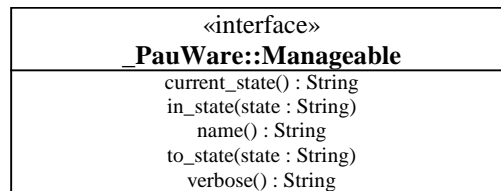


Figure 49. *PauWare Manageable* interface.

Implementing the services at the component level is straightforward and consists in calling the same function offered by the *AbstractStatechart_monitor* class. Taking the example of the *Stack* software component in Figure 27, we have:

```

public String current_state() {

    return _Stack.current_state();

}

```

8.1 Dynamical (re)-configuration

Within the *Manageable* interface, a key service is *to_state*, which forces a state machine to a stable consistent context. This operation of dynamical re-configuration may thus be launched by a manager component that monitors the component which implements the *Manageable* interface. The behavior of the *to_state* function is such that **it bypasses all entry and exit actions**. In other words, it must only be used for management purposes and not for business purposes (*i.e.*, for describing an application's business logic⁷⁰). In the scope of management, *to_state* may especially be used "more intelligently" as follows:

```

public void reset() throws Statechart_exception {

    try {

        while(true) _stack.pop();

    }
}

```

⁷⁰ This advice has to be attenuated by the fact that failure recovery may be a first-class concern of an application, and thus *to_state* may take place in the application's business logic.

```

    } catch(java.util.EmptyStackException ese) {

        to_state("Empty");

    }
}

```

In this code, we make the new forced state of a *Stack* component coherent with this component's business data. This business data is the private *_stack* field. Its content is set to empty through an infinite loop. Afterwards, the state machine of *Stack* is setup accordingly.

The *to_state* PauWare API function may raise a *Statechart_exception* object meaning that forcing a state machine to a stable consistent context is possibly subject to a failure. This process is used within the rollback facility discussed in Section 5.8.2. Self-healing in Section 5.8.2 indeed relies on several calls to *to_state* for all source states which belong to the immediately prior stable consistent context. Most of the time, failures generated by *to_state* come from the impossibility to determine which state to activate when its immediate superstate has several exclusive substates. The *to_state* function uses the default input states which have been setup thanks to *inputState* (see Section 3.1.1) but recovering a stable consistent context is not always possible (see again Section 5.8.2).

8.2 Extended management framework

Managing components in *PauWare* may greatly benefit from the concomitant use of the *Java Management eXtensions* or JMX technology (see also Section Bibliography). Returning to the state machine diagram of the software component named *PauWare component* in Figure 2 or in Figure 28, we may create a JMX-compliant component as follows:

```

public interface PauWare_componentMBean extends Manageable {

    void request_b() throws Statechart_exception;

    ... // others here?
}

```

The *MBean* extension in the code above is just a JMX constraint. The final result is expressed in Figure 50 which is an adapted view of Figure 4.

Figure 50. Mixing *PauWare* for management-centric software components.

Figure 50 (which has, by inheritance, *to_state* as a service of the *PauWare_componentMBean* interface) leads to enabling the dynamical (re)-configuration of any instance of *PauWare component* through a JMX management console or any entry point.

8.3 *Statechart_monitor_listener* interface

To close on software component management, we present in this section, the *Statechart_monitor_listener* interface (Figure 51).

Figure 51. The *PauWare Statechart_monitor_listener* interface.

Managers may implement this interface in order to be online with a state machine. Moreover, the two *run_to_completion* services communicate, when each run-to-completion cycle completes, both a Java *String* and a Java *Hashtable*. The first object is just documentation explaining what happened within a run-to-completion cycle, while the second includes the transitions that were accomplished. These are keys of the *Hashtable* object, while each associated value of a key is an array of actions (*i.e.*, instances of the *AbstractAction PauWare* class). These actions are in essence the actions accomplished since they are associated with the accomplished transitions.

To register a manager to a state machine, first one may use the constructor of the *AbstractStatechart_monitor* class that accepts as a fourth parameter, an object whose type is *Statechart_monitor_listener*. In this case, the *initialize* function in Figure 51 is automatically called back with *this* as a parameter. The other way is to use the *add_listener* and possibly *remove_listener* functions of *AbstractStatechart_monitor*. In this second case, after registering, if the manager wishes to know the structure of the listened state machine, it must be explicitly called *initialize_listener* of *AbstractStatechart_monitor*.

9 State machine composition facilities

To appear in the next release of this user guide. This section discusses a new feature of *PauWare* (ver. 1.2) which consists in composing instances of the *AbstractStatechart_monitor* class with the *and* and *xor* preexisting operators. For the moment, these are only used for instances of the *AbstractStatechart* class (ver. 1.1).

10 *PauWare*'s internal structure

For historical and portability reasons, the core of the *PauWare* engine is Java ME-compliant. As a result, any extension permitted by the openness of *PauWare* must take great care of this attribute. We here give brief insights into the internal structure of the core of *PauWare* for any third-party developer interested in participating in the optimization of this product.

10.1 Organization of states in memory

If we take the example of the state machine diagram in Figure 2, we have in memory a binary tree (Figure 52). Each state has a father state, except the state machine itself⁷¹. Both direct substates of a state have either an orthogonality relationship (“and” in Figure 52) or an exclusive one (“xor” in Figure 52). For a state which has more than two direct nested states in a state machine diagram (*e.g.*, *Busy* which has *S1*, *S2* and *S3* as immediate substates in Figure 2), we may take advantage of the transitivity of the orthogonality and exclusiveness relationships. As a result, fictitious states are sometimes introduced. Their name is the value of the *Pseudo_state* constant class variable (see also Section 5.3). This value is “*pseudo-state*”. In Figure 52, a pseudo-state embodies the orthogonality relationship between *S1* and *S2*. This pseudo-state itself shares an orthogonality relationship with *S3*.

⁷¹ An exception to this rule relies on state machine composition as stated in Section 9.

Figure 52. Binary tree-based structuring of a state machine.

10.2 Organization of transitions in memory

A key field of the *AbstractStatechart_monitor* Java class is a Java *Hashtable* named *_transitions*, which is basically used for recording transitions as shown in Figure 53. The run-to-completion algorithm mainly depends upon this chosen structuring.

Figure 53. Transition structuring.

11 Alignment with UML

This section will be completed in the next release of this users' guide. It aims at strictly positioning *PauWare* with respect to the "declared" execution semantics of UML. For that we use the *Unified Modeling Language: Superstructure*, version 2.0, formal/05-07-04 released in August 2005 (see Bibliography).

State entering

We discuss in Section 3.1.1 some possible incompleteness concerning which, among several exclusive substates, substate to enter if no default input substate is setup.

State exiting

Page 536, paragraph “Exiting non-orthogonal state”.

Communication

Although the absence of the ^ symbol in UML 2.x, XMI distinguishes events in state machine diagrams by their type: Call event, .

Event versus request

On page 409 of the documentation, it is written: “An action representing the invocation of a behavioral feature is executed by a sender object resulting in an invocation event occurring. The invocation event may represent the sending of a signal or the call to an operation. As a result of the invocation event a request is generated. A request is an object capturing the data that was passed to the action causing the invocation event (the arguments that must match the parameters of the invoked behavioral feature); information about the nature of the request (i.e., the behavioral feature that was invoked); the identities of the sender and receiver objects; as well as sufficient information about the behavior execution to enable the return of a reply from the invoked behavior, where appropriate. (In profiles, the request object may include additional information, for example, a time stamp.) While each request is targeted at exactly one receiver object and caused by exactly one sending object, an occurrence of an invocation event may result in a number of requests being generated (as in a signal broadcast). The receiver may be the same object that is the sender, it may be local (i.e., an object held in a slot of the currently executing object, or the currently executing object itself, or the object owning the currently executing object), or it may be remote. The manner of transmitting the request object, the amount of time required to transmit it, the order in which the transmissions reach their receiver objects, and the path for reaching the receiver objects are undefined. Once the generated request arrives at the receiver object, a receiving event will occur.”

Deferring of events

PauWare does not support the deferring of events. This execution semantics **option** is exposed on page 535 of the documentation. However, in UML, events are lost if a state does not specify the event types which have to be deferred in that state by default.

Conflicting transitions

In the documentation, page 547, it is written: “Only transitions that occur in mutually orthogonal regions may be fired simultaneously.” So two enabled transitions (their guards are true) originating from the same active state are by definition in conflict. This may happen **independently of their destination states** and of the deep nature of the relationships between these destination states: exclusiveness, orthogonality or nesting. This point is controversial in that it provides a limited modeling scope (see Section 4.7.2).

Conflicting transitions, nesting and overriding

In the documentation, it is written on page 547: “By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.”

12 List of benchmarking applications

In order to provide concrete material for underpinning the discussion in this users’ guide, *PauWare*-based applications may be downloaded from www.PauWare.com under in the form of *NetBeans* projects. Each of them illustrates one or more technical points discussed during this users’ guide. Here is the list of available applications in alphabetic order:

- *Allowed_event_conflict_example NetBeans* project mainly illustrates Section 5.5;
- *Banking_system NetBeans* project mainly illustrates Sections 5.1, 5.2 and 5.7.1;

- *Beverage_vending_machine* NetBeans project mainly illustrates Section 5.7.2;
- *Concurrent_use* NetBeans project mainly illustrates Section 5.10;
- *Guard_conflicts* NetBeans project mainly illustrates Section 4.7.2;
- *Home_automation_system* and *Home_automation_system_UI* NetBeans projects are based on the Velcro API (Java ME) and mainly illustrate Sections 5.7.1 and 6;
- *JMX_home_automation_system* NetBeans project mainly illustrates Section 8 and the concomitant use of *PauWare* and JMX;
- *MIDP_home_automation_system* and *MIDP_home_automation_system_UI* NetBeans projects are “true” Java ME applications⁷² in the sense that they have a GUI which complies with the MIDP profile;
- *PauWare_component* NetBeans project is based on the Velcro API (Java ME) and mainly illustrates Sections 3, 4.5, 5.8.2, 8 and 10;
- *Railcar_control_system* and *Railcar_control_system_client* NetBeans projects mainly illustrate Section 7;
- *Simple_nesting* NetBeans project mainly illustrates Sections 4.4 and 5.4;
- *Stack* NetBeans project mainly illustrates Sections 5.6, 5.8.1 and 8;

13 XMI2PauWare complementary tool

To appear in the next release of this users’ guide.

13.1 UML 2 CASE tools compliant with *PauWare*

To appear in the next release of this users’ guide including a discussion on the UML *NetBeans* (ver. 6.0) project.

14 *PauWareView* complementary tool

PauWareView enables the visualization of state machines in GUIs. It relies on the *OpenJGraph* third-party library and is provided “as-is”. Some of the benchmarking applications from Section 12 use it.

15 Bibliography

F. Barbier, H. Briand, B. Dano and S. Rideau, “The Executability of Object Oriented Finite State Machines,” *Journal of Object-Oriented Programming*, 11(4), pp. 16-24, 1998.

F. Barbier, *UML 2 and MDE – Model-Driven Engineering with Case Studies*, Dunod, 2005 (in French).

F. Barbier, “An Enhanced Composition Model for Conversational *Enterprise JavaBeans*,” *Proc. of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, LNCS #4063, Springer, pp. 344-351, 2006.

S. Cook and J. Daniels, *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall, 1994.

M. Crane and J. Dingel, UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal, *Proc. of the 8th International Conference on Model Driven Engineering Languages and Systems*, LNCS #3713, Springer, pp. 97-112, 2005.

R. France, S. Ghosh, T. Dinh-Trong and A. Solberg, “Model-Driven Development Using UML 2.0: Promises and Pitfalls,” *IEEE Computer*, 39(2), pp. 59-66, 2006.

D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, 8, pp. 231-274, 1987.

⁷² We remind one here that some provided applications like *Home_automation_system* use the Velcro API, but are really Java SE applications in the sense that they rely on Java SE-oriented GUI features.

- D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *IEEE Computer*, 30(7), pp. 31-42, 1997.
- D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"," *IEEE Computer*, 37(10), pp. 64-72, 2004.
- Y. Jin, R. Esser and J.W. Janneck, "A method for describing the syntax and semantics of UML statecharts," *Software and Systems Modeling*, 3(2), pp. 150-163, 2004.
- H. Kreger, W. Harold and L. Williamson, *Java and JMX – Building Manageable Systems*, Addison-Wesley, 2003.
- S. Mellor and S. Balcer, *Executable UML – A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002.
- Object Management Group, *Unified Modeling Language: Superstructure, version 2.0*, formal/05-07-04, August 2005.
- C. Prehofer, "Plug-and-play composition of features and feature interactions with statechart diagrams," *Software and Systems Modeling*, 3(3), pp. 221-234, 2004.
- L. Pazzi, "Part-Whole Statecharts for the Explicit Representation of Compound Behaviors," *Proc. of the 3rd International Conference on The Unified Modeling Language*, LNCS #1939, Springer, pp. 541-555, 2000.
- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- A. Simons, "On the Compositional Properties of UML Statechart Diagrams," *Proc. 3rd Conf. Rigorous Object-Oriented Methods*, pp. 4.1-4.19, 2000.
- M. von der Beeck, "A structured operational semantics for UML-statecharts," *Software and Systems Modeling*, 1(2), pp. 130-141, 2002.

Appendice: PauWare's Javadoc

See www.PauWare.com/PauWare_software/Javadoc_files/