Improving Quality Factors in Model-Based Embedded Software

Luca Pazzi and Marco Pradelli

University of Modena and Reggio Emilia
Department of Engineering Sciences DII-UNIMORE
Via Vignolese 905, I-41100 Modena, Italy

{luca.pazzi,marco.pradelli}@unimore.it

Abstract. This paper surveys, mainly through a running example, the most noticeable features of Part-Whole Statecharts, a formalism originally conceived with the aim of introducing modularity within the Statecharts formalism in order to solve software quality issues of Harel's Statecharts.

1 Introduction

Model based software development calls for dynamic software abstractions which are both self-contained and easily composable, due to the diagrammatic and iconic operating modalities of current model-based software development tools.

Although such a shift of paradigm towards self-containedness and composability is on the top of the wish list of software engineering practitioners, few progresses towards a new composable paradigm have been achieved since then. If, on one hand, software modularity principles are well established and accepted, on the other hand the dream of assembling software application starting from modules is still unachieved.

Model based software engineering has to deal with the design and simulation of complex systems. Such systems exhibit a complex systemic interactive behavior and are typically employed in the embedded control of physical entities and processes. Nowadays modeling tools and techniques allow to assemble black-box pieces of code in order to implement functional transformation of data. It should be observed, however, that applications are required to do more than simply transforming data: they have in fact to implement a complex systemic interactive behavior. State diagrams, relying essentially on the concepts of state and state transformation, are a powerful instrument in representing behavior. States and state transitions furnish indeed the natural abstraction for static situations and for changes among such situations. States can be seen also as bunch of properties, state variables, in which the system rests for an amount of time, and therefore state transitions are implementable through functional transformation among such state variables. Behavior is therefore more than functional transformations: its best approximation is through state machines, that is through finite diagrams of states and state transitions.

2 The Behavioral Composition Issue

Composing behavioral abstraction is however still an open issue in model based software engineering. In first place we observe that composition in dynamic context consists essentially in achieving synchronization among different modules. Synchronization issues have been deeply investigated in behavioral formalisms like CCS [1] and CSP [2]: such seminal investigations led to Harel's Statecharts formalism [3], which was in turn adopted by Rumbaugh's software development method OMT [4] and by OMG UML since it appeared, becoming the standard way of achieving behavioral composition among modules.

Despite its widespread success, Harel's Staecharts suffer from different drawbacks which, paradoxically, make it an obstacle towards the achievement of software quality factors, such as reusability, understandability, maintenance-ability and testability.

2.1 Composing Behavior by Statecharts

A statechart diagram typically consists of state diagrams, hosted into different interacting parallel sections, which run concurrently and have to synchronize in order to achieve a global, meaningful, systemic behaviour. For example, modeling a system of interacting devices is typically achieved by representing the behavior of each component device by a state diagram hosted within one of the parallel sections. Statecharts synchronization primitives, like *event broadcasting* and *mutual condition testing*, have to be embedded into component state diagrams in order to obtain a systemic representation, which is therefore directly incorporated within the interacting system components. The use of such synchronization primitives has its roots in hardware composition techniques, which first were employed in the development of embedded control systems: both rely in fact on the concept of wires which have to reach different component.

Example An *ignition device* has to be designed in order to control the ignition of a flame. Different components must exhibit a coordinate behavior in order to accomplish the task. We develop a running example starting from a mechanical device.

The whole ignition process starts by keeping pressed a *start* button. Two devices are connected to the button:

1. a spark lighter, which emits a spark against a gas flow once the button is fully pushed;
2. a spring controlled valve, which turns the gas flow on and off when the button is, respectively, kept pushed and released. Once the flame is ignited, its heating dilates the spring controlling the valve return to the close position, hence the valve remains in its open position.

In other words, the coordinated behavior of the button, the heather and the valve can be summarized as follows: the user pushes the button, opening the valve and emitting a spark against the gas flow coming from the opened valve. If the button is kept pushed for a sufficient amount of time once the flame has ignited, the return spring of the valve is heated and dilated, hence it does not return to its initial, closed position. The flame then remains turned on until either the gas flame stops or it is turned off by accident. In both cases, the metal of the spring, once not heated for a short amount of time, acquires back its elastic property causing the valve to shut.

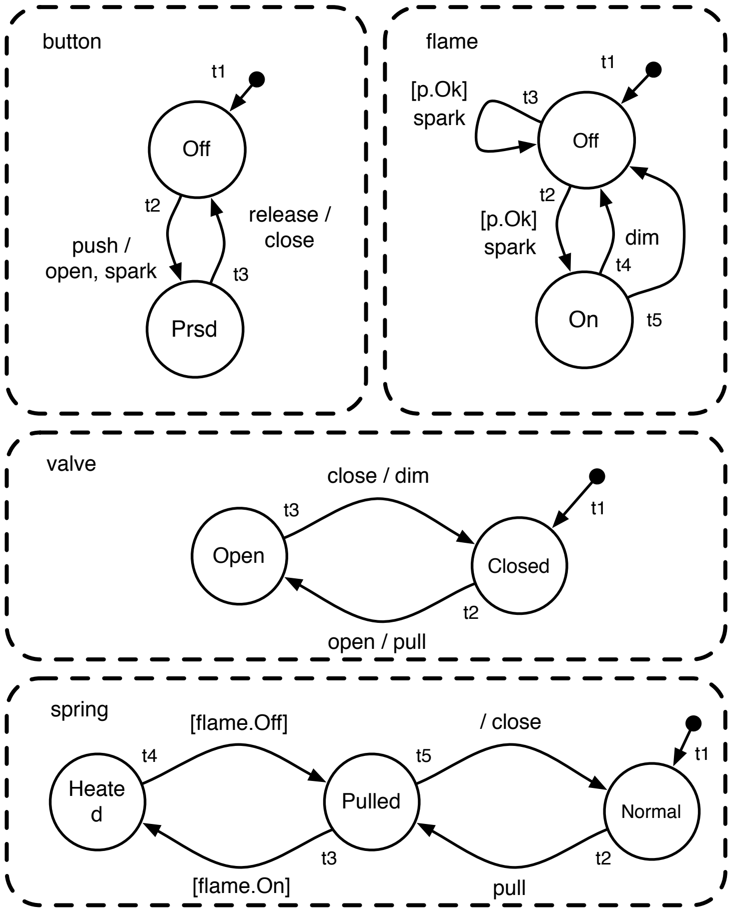


Fig. 1. Statecharts modeling of a manual ignition device.

Figure 1 shows the statechart model representing the full igniter behavior as described above. Each component having a meaningful behavior is depicted by a parallel section in the Statechart. Such sections communicate one with the another by exchanging events and by testing the state of each other. It can be observed that event flow as well as mutual testing of state condition (shown in Figure 1 by gray arrows) follow strictly the causal relationships observed in the description of the mechanical device. For example, once the event *push* reaches the button section, events open and spark are subsequently directed towards the sections representing the valve and the flame. The flame section represents the state of the gas burner, which may have a flame turned on or off: observe that the model accounts for an indefinite number of sparks failing to light the gas burner, as modeled by the looping state transition t_3. Observe moreover that the flame may be turned off either:

1. by an event dim (transition t_4) which models the causal effect of the valve being closed (transition t_3 in the valve section) or
2. by taking at any time state transition t_5 which models the casual extinguishment of the flame.

Analysis Statecharts control model suffers from different drawbacks when modelling a system behavior out of a set of parallel section hosting system components. Such drawbacks can be analysed in terms of software quality factors:

1. the component behavior is barely reusable, due to the fact that behavioral references make it tightly bound to the other abstractions. In Figure 1, for example, the spring behavior has effect on the valve behavior, which in turn depends on the flame behavior, which in turn has again effect on the spring behavior. Figure 2 depicts dependencies among parallel sections by gray arrows;
2. the system behavior is difficult to understand, since it is difficult for the designer to have a complete view of the whole system behavior scattered into the different component sections. This fact becomes evident by observing the gray arrows cluttering of Figure 2, where each arrow denotes a part of the modeled behavior; as a consequence, the system behavior is difficult to extend, since the addition of a single module requires, potentially, the addition of behavioral references from and to each existing module;
3. the system behavior may either deadlock or not terminate. This is due to cross referencing mutual conditions as well as to infinite, circular successions of state transitions and event broadcasting.

It is evident that, if Point 1 impacts on the reusability of the software being built around such abstractions, Points 2 and 3 impact not only on the reusability of the whole behavior, but mainly on safety issues, since it is difficult, at design time, to know in which state a system will be found at a specific time, and therefore it is not possible to assess safety constraint against behavior, such as “when the gas pressure is low or absent the valve must be closed and it must remain closed even if the pressure returns regular”.

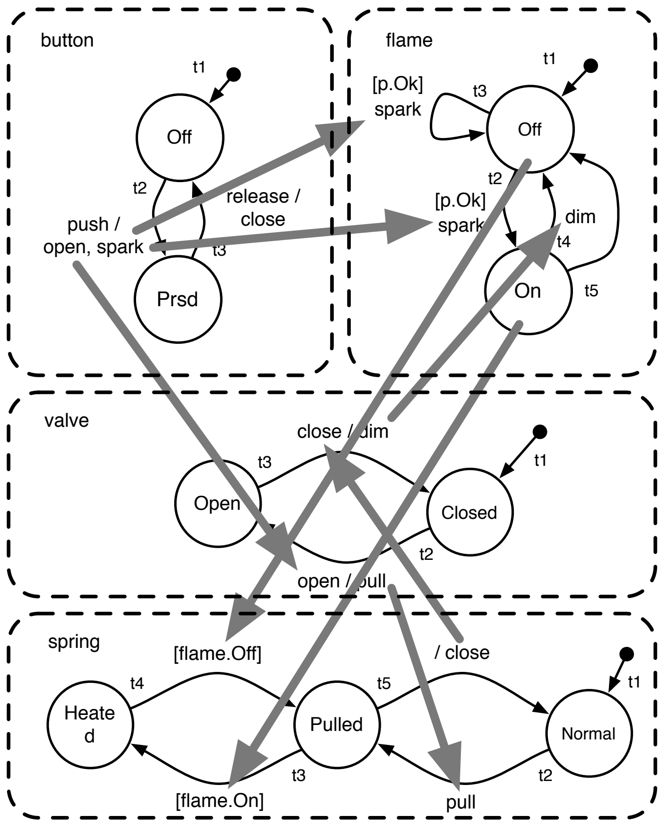


Fig. 2. Behavioral relationships established among parallel sections by event broadcast as well as by mutual condition testing.

3 Explicit Modeling by PW Statecharts

Interacting components can be represented by a different state based formalism, named Part-Whole Statecharts [5][6] (shortened either as PW Statecharts or PWS), which was created with the aim of allowing an *explicit* representation of the interaction among the behavior of parallel state based sections.

A PWS consists of two main sections, one hosting a set of component state machines, referred collectively to as the *assemblage*, the another a single state machine representing the system behavior as a whole, called indeed the *whole* state machine.

As shown by the banned gray arrows Figure 3, control as well as any mutual knowledge of current state and behavior is not allowed among component state machines; conversely, the system section state machine is allowed to know, at each time, the current state of each component state machine as well as to send control commands to them (regular gray arrows). The system machine is finally notified of each state transition happening within the component set.

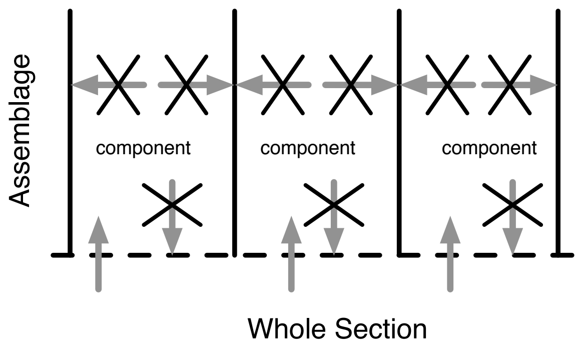


Fig. 3. PWS control flow. Regular and banned gray arrows indicate whether or not *direct* control and knowledge is possible among the assemblage and the whole behavioral sections. Indirect control is however feasible from the assemblage to the whole.

In order to suggest that neither communication nor knowledge is allowed between component state machines a bold line is drawn among them; conversely, a dashed line separates the components from the system state machine, in order to suggest that the system state machine is able to communicate with the components. Note, however, that such a communication is asynchronous, since the whole section state machine is allowed to know each of the component PWSs, but the reverse does not hold. Component PWSs determine however the behavior of the whole by emitting events towards it or by having the whole know their internal state: such form control is therefore better classifiable as*indirect*.

Such communication and knowledge restrictions ensure that behavioral component description are self-contained, since they are not allowed to refer to any of the peer components or to the system state machine. We observe, finally, that the whole semantics of coordination and communication has to be transferred to the system state machine section, in such a way that an additional explicit level of reuse and understandability is made available.

Component Assemblage Section Explicit modeling by the PWS approach requires *components to be deprived of the capability of carrying out direct control towards other components as well as to have their behavior determined by the knowledge of the behavior of other components*. We obtain thus components which are self-contained, thus achieving full reusability as observed above.

For example, the parallel Statecharts' AND sections of Figure 1 have been reworked in Figure 4, by removing both event forwarding as well as condition

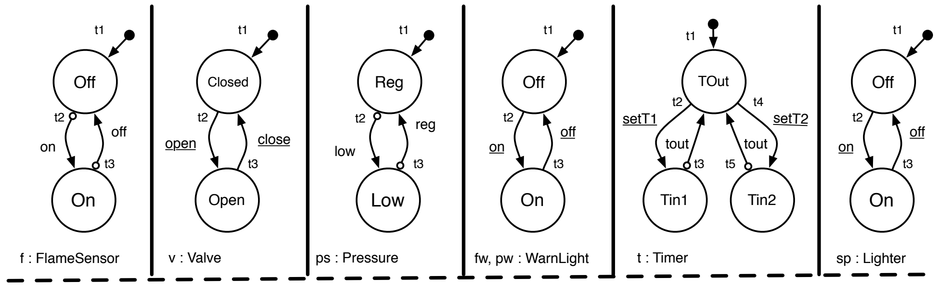


Fig. 4. An assemblage of state machines for the implementation of the ignition system.

testing amongst the state machines in the parallel sections. The modules within the assemblage section expose only *external details*, that is they only show the states in which each component may be found as well as the transitions among them. We distinguish among two kinds of transitions:

- *non controllable* transitions: they often correspond to the automatic behavior of a generic device or to the sensing behavior of a sensor device: For example, device f moves from state Off to state On by transitions t_2 and t_4: each time one of the transitions is automatically taken, the corresponding events on and off are sent to the whole section;
- *controllable* transitions: they often correspond to the behavior of a generic actuator device. For example, the device v, corresponding to an electromechanical valve which moves from state Closed to state Open by transitions t_2 and t_4: the transitions are taken once the corresponding events on and off are sent from the whole section to the component.

Observe that some devices possess both kind of transition: for example, the timer device t in the assemblage.

Whole-Section Behavior. A first draft behavior of the behavior to be enforced by the whole-section of the ignition system is shown in Figure 5: it consists of *five* states, meaning that the whole "ignition system" starts in an initial global state, named Off, then moves to a state, named Pushed, where different ignition attempts are tried until either the button is released, causing the system to return to the initial Off state, or the flame is ignited. In the latter case the system moves to the *flame on* state (FL_On) in which the system rests until the flame is dimmed by any accidental cause. In such a case, the system moves then to *retry* state, in which ignition is retried for a definite amount of time, say T_1 time units. In case retrying ignition is still unsuccessful after T_1 time units, the system moves to a warning state FWarn, from which it is either possible to restart

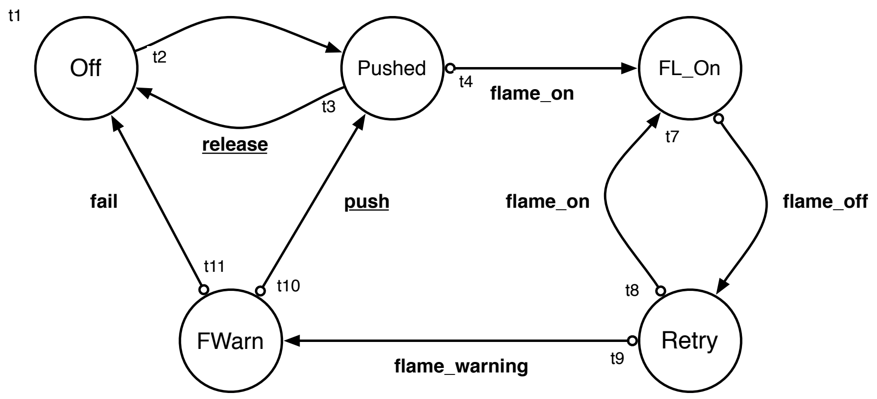


Fig. 5. A first design of the desired ignition system behavior.

the working cycle by pressing the button or the system turns automatically off after an interval lasting T_2 time unit expired.

3.1 Implementing Behavior

In order to make the behavior of Figure 5 effective, we have to specify additional details in order to specify which commands have to be forwarded to the assemblage and how the behavior in the whole section reacts to changes happening in the assemblage. In order words, by labelling each satte transition by additional features, we keep the whole and the assemblage section consistent. In Section 3.2 we give an account of a suggested operational semantics for PW Statecharts, that is we explain how the implementation features are interpreted at runtime.

State transition implementation features The diagram allows to observe *implementation features* associated with the state transitions, which consist of:

1. a *guard*, consisting of a boolean valued expression about the global state of the assemblage, depicted enclosed in square brackets. In case a transition does not have any guard, it may be thought as being guarded by the boolean value *true*;
2. a *trigger*, that is a symbol (written underlined in the diagram) which denotes that the transition will be activated upon the receipt of:
   (a) an event e sent to the PWS by another PWS having the current PWS as component, in which case $\underline{\text{e}}$ is named *external* trigger;
   (b) an event c.e sent to the PWS by its component c, denoting the happening of a transition labeled by e within the component c of the assemblage, in which case either $\underline{\text{c.e}}$ or $\underline{\text{c.}t}$ is named *internal* trigger (t being the transition name).

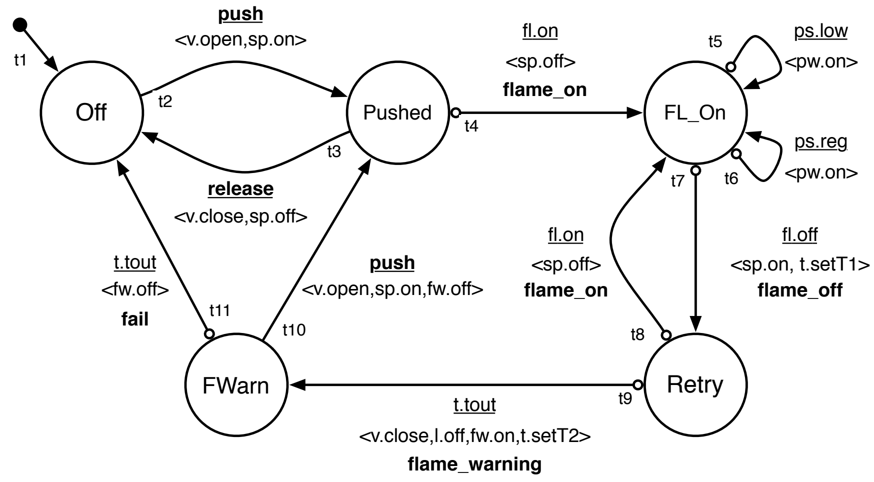


Fig. 6. The implementation of the ignition system behavior of Figure 5.

For example, transition t_2 and t_3 in Figure 6 may be triggered, respectively, by the external events push and release, while transition t_4 may be triggered by the internal event f.on, meaning that the event on happened within component f;

3. a *list of actions*, that is symbols which are used in order to request the activation of the state transitions in the assemblage components. We write c.e in order to require the activation of the triggerable transition labeled by event e in the component c. For example, transition t_2 in Figure 6 has the associated list of action $\langle \mathsf{v.open}, \mathsf{sp.on} \rangle$, meaning that, when t_2 is activated, the transition labeled by open has to be activated within component v and the transition labeled by on has to be activated within component sp.
4. an optional *output event*, that is a symbol which will be sent to sent to all the PWSs which have the current PWS as components in order to notify them that the transition happened. We write c.e in order to require the activation of the triggerable transition labeled by event e in the component c.

3.2 Part-Whole Statecharts Operational Semantics

The way in which the whole section of the PWS behaves and the way in which communication happens among the assemblage and the whole section are deeply interleaved in describing the global operation of PW Statecharts:

1. the whole section operates through a never ending cycle, which iterates a *computation step* during which *communication signals* sent to the machine are evaluated and (a possibly empty) set of state transitions are *selected* for *execution*; one of such transitions is arbitrarily drawn from the set, and

communication signals are sent to other PWSs as well as a new current state is computed;
2. PWSs *communicate* through some communication medium, which again operates through one or more never ending cycles. Such cycles iterate basic communication operations, which consist, essentially, in delivering communication signals from one PWS to the another.

Computation step. The computation step consists in first place in checking whether incoming events (either internal or external) are present for being processed. Given an incoming event either internal or external, a (possibly empty) set of state transitions T_S is selected for being executed iff for each state transition in such a set the following conditions are verified:

1. the transition has state s as departing state; *and*
2. the guard condition is satisfied; *and*
3. the incoming event matches the transition trigger.

In case T_S has more than one element, a state transition is chosen arbitrarily. State transition execution consists in:

1. delivering the commands (if any) of the command list to the assemblage components through the communication medium;
2. sending the transition output event (if specified) to all the PWSs which have the current PWS as component;
3. moving the state machine in the whole to the ending state of the transition, which becomes the current state of the machine.

PWS extendibility and remodelling We stress that the explicit modeling approach brings advantages on the software engineering phase of maintenance and reuse of the global behavior. Given the same set of assemblage components, we show how the global behavior may be modified given *the same set of assemblage components* as shown in Figure 8.

PWS composition Any PWS may be used, in a straightforward way, as a component in higher complexity PWSs. It is in fact possible to extract an *interface* from any PWS, that is the state machine which contains only the information that may be used by external composition context, that is its externally observable behavior. Given an implementation state diagram, like that of Figure 6, we obtain its interface by:

1. hiding the assemblage component set;
2. hiding any internal trigger, guard, action list from any transition.

In the case of the igniter, for example, the interface may be obtained by removing component assemblage and the implementation details from the PWS. Figure 10 shows a fragment of a more complex PWS in which the igniter is employed. In the depicted diagram, the push action is carried out through a command sent to the igniter. By adding a timer to the assemblage it is possible to establish a time limit to the ignition attempts. Such a property was not part of the design of the ignition device, where it is possible to persist in the ignition attempts for an undefinite amount of time (i.e. until a release command is issued). The informal meaning of the behavior fragment in the whole section of Figure 10 is therefore: "when the system is in state Off it is possible to send a start command to the igniter, which either has to start within T time units or fail".

4 Conclusions and Further Work

The paper surveyed, mainly through a running example, the most noticeable features of Part-Whole Statecharts, a formalism originally conceived with the aim of introducing modularity within the Statecharts formalism in order to solve software quality issues in Harel's Statecharts.

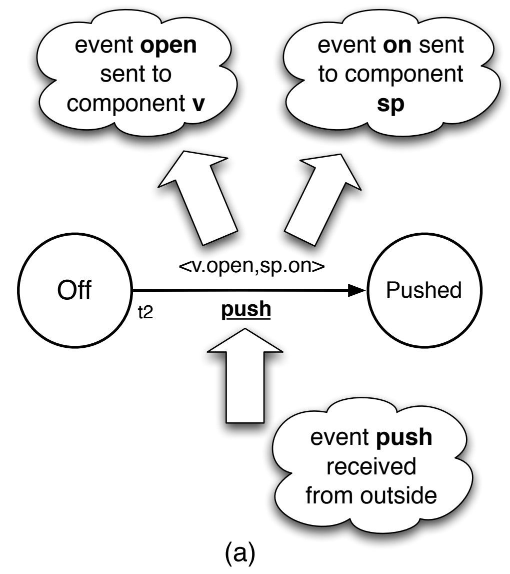
The authors are currently working on a patent pending methodology, which should allow to check, at design time, whether the design complies with state-based propositions. Describing such a methodology was outside the scope of the paper, mainly focused on software quality factors. The reader is referred to [7] for a first report illustrating it.

We give however a final sample account of the rationale of such a method, applied to the running example. Suppose state PWarn of Figure 8, aimed at denoting the situation in which a gas pressure leak is detected and signalled, is specified by the proposition p "the gas pressure is Low *and* the warning light is On *and* the flame is On". The methodology would then be able to detect whether any incoming transition to state PWarn agrees with such a proposition as well as to check whether any uncontrollable event in the assemblage is able to falsify the proposition. In such a case, the designer will be asked to insert the appropriate number of internally triggered transitions in the whole section in order to account for such violations of the state invariant proposition. In case the current state of the whole is PWarn, having p as invariant, it can be easily verified that in case the flame turn off p is not verified anymore. The designer has therefore to insert an ad hoc transition leaving PWarn, triggered by the internal event denoting the flame going off and moving the control to another state. The design correcting such an inconsistency is shown in Figure 11.

References

1. Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, 92. Springer-Verlag (1979)

2. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
3. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
4. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice Hall (1991)
5. Pazzi, L.: Extending statecharts for representing parts and wholes. In: Proceedings of the EuroMicro-97 Conference, Budapest, Hungary. (1997)
6. Pazzi, L.: Part-whole statecharts for the explicit representation of compound behaviors. In: UML 2000 - The Unified Modeling Language. Advancing the Standard. Volume 1939 of LNCS., Springer (2000) 541–555
7. Pazzi, L.: A method for ensuring safety and liveness rules in a state-based design, http://cris.unimore.it/cris/files/2008-02-01.pdf. Technical Report CRIS-2008-02-01 (2008) Patent pending PCT/EP2008/051300.


event **open** sent to component **v**
event **on** sent to component **sp**
<v.open,sp.on>
Off
Pushed
t2
push
event **push** received from outside
(a)

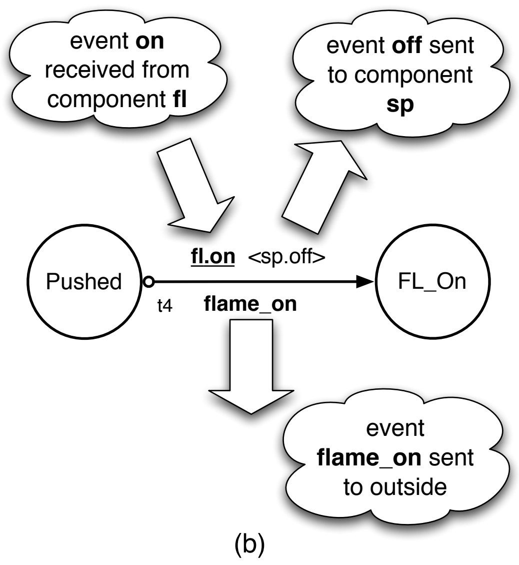

event **on** received from component **fl**
event **off** sent to component **sp**
fl.on <sp.off>
Pushed
FL_On
t4
flame_on
event **flame_on** sent to outside
(b)

Fig. 7. Event flow schema for two different typologies of state transitions, respectively (a) externally and (b) internally triggerable transitions.

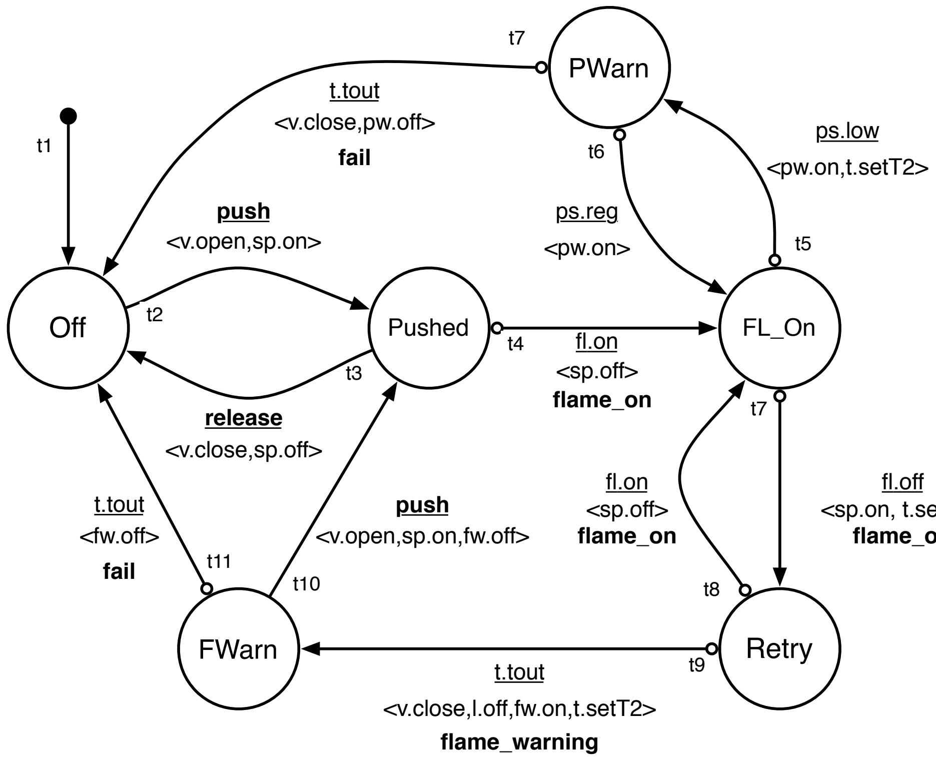


Fig. 8. An exercise in extendibility: a new *pressure warning* state PWarn has been added to the device (compare with Figure 6).

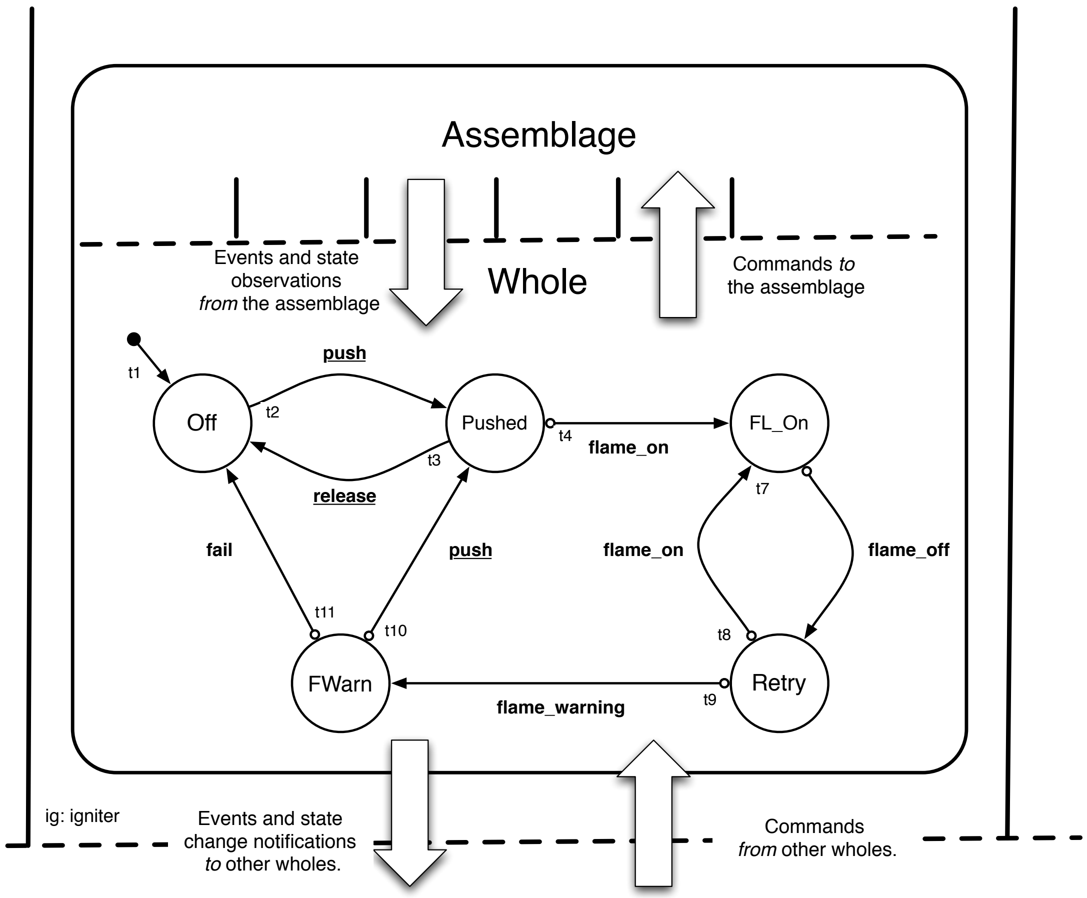

Assemblage
Whole
Events and state observations *from* the assemblage
Commands *to* the assemblage
t1
Off
t2
push
Pushed
t3
release
t4
flame_on
FL_On
t7
flame_on
flame_off
fail
push
t11
t10
t8
FWarn
flame_warning
t9
Retry
ig: igniter
Events and state change notifications *to* other wholes.
Commands *from* other wholes.

Fig. 9. Event flow schema in PWS composition (see also Figure 7).

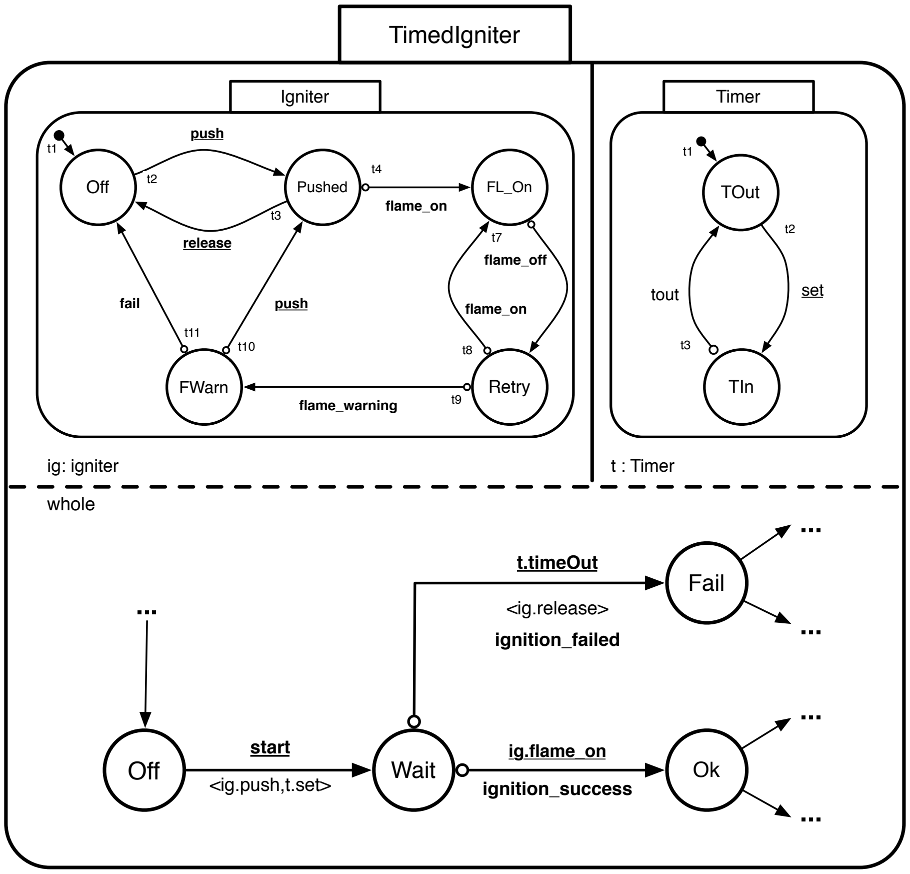


Fig. 10. Further composition of the igniter PWS and of a timer PWS in order to time constrain the ignition attempts. Rounded labelled rectangles emphasize the PWS recursive compositional flavour.

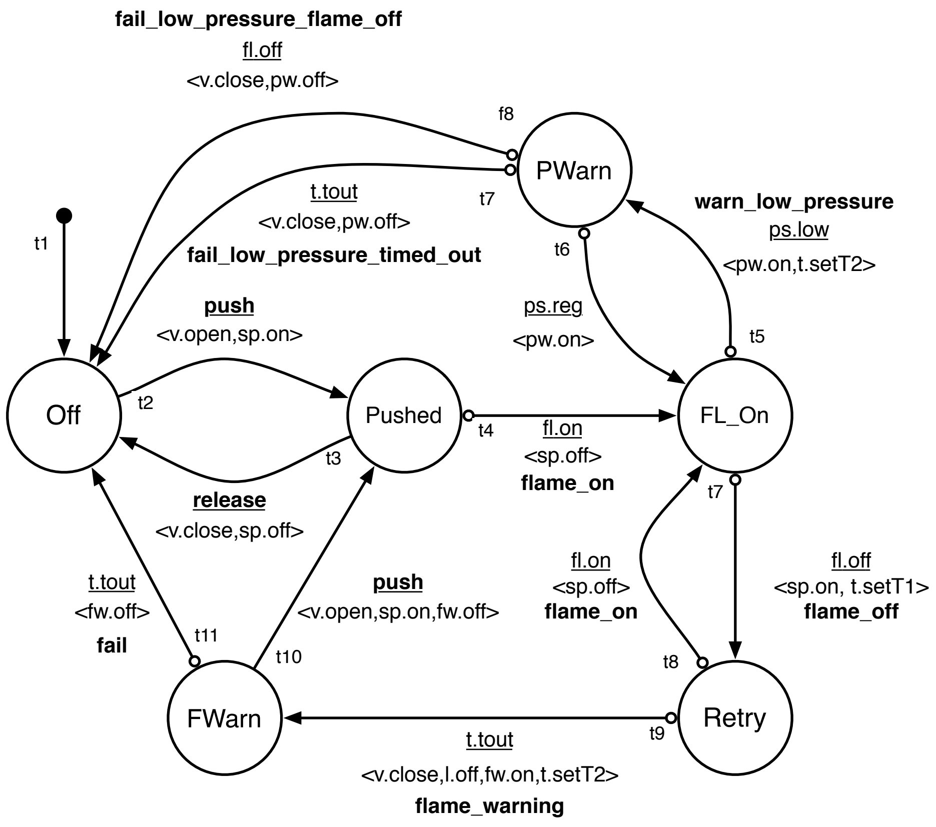


Fig. 11. A revised design accounting for flame dim off when the ignition device is in state PWarn.