A Method for Constraining State Semantics in Part-Whole Statecharts

Luca Pazzi, Marco Pradelli

University of Modena and Reggio Emilia
Department of Information Engineering DII-UNIMORE
Via Vignolese 905, I-41125 Modena, Italy
{luca.pazzi, marco.pradelli}@unimore.it
http://dii.unimore.it/emblab

Abstract. Part-Whole Statecharts were originally proposed with the aim of providing truly modular constructs to traditional Statecharts, in order to allow incremental and fully reusable composition of behavioral abstractions. The hope was that providing an explicit section for enforcing explicitly the coordinated systemic behavior could bring benefits to subsequent modelling and implementation phases. The paper presents recent research results, which go beyond the original improvements observed in software quality factors. It can in fact be shown that Part-Whole Statecharts have a computable semantics, which can be specified through a constraint-driven specification method. Such a method allows to enforce the intended coordination of component state machines directly at design time through propositional logic.

1 Introduction

Part-Whole Statecharts [1][2] (shortened either as PW Statecharts or PWS) were presented, about a decade ago, with the aim of providing Harel's Statecharts formalism [3] with truly modular constructs. Various attempts have been made since then in order to allow the modular, incremental and reusable coordination of behavioral abstractions. The Syntropy [4] Object-Oriented Development Methodology pioneered in the field by encapsulating the state behavior of single entities within modules hosted into parallel Statecharts sections. The Statechart coordination and communication model was then used in order to assemble a global behavior, by letting the state machine residing on each module both aware of the current status of the other machines and able to act on them (in addition to the module on which it resided).

Obtaining a systemic coordinated behavior through such a model has however severe limitations in terms of software quality factors, since the global behavior tends at being represented in a fragmented form, and as such is difficult to understand, reuse and maintain. In addition, the operational semantics of such a coordination model is necessarily ambiguous, since control events bounce from one component to the another depending on the current status of each one. We will refer to such a way of assembling AND decomposed modules as the *implicit* way of coordinating global behaviour.

As an example of implicit coordination, consider the three AND decomposed substates which decompose the state Counting in a typical Statecharts variant (Figure 1-(a)). It can be observed that each substate, representing a single bit in a counter, is different from the others, since each one has to embed some knowledge regarding the global behavior of the aggregation. In particular each bit has to communicate with its right peer in order to set it to zero or to one, and is in turn set by its left peer. This makes each "bit abstraction" not self-contained, since it has to refer to another bit and it has to be referred by another bit in the same fashion, caching the coordination knowledge within the modules being coordinated.

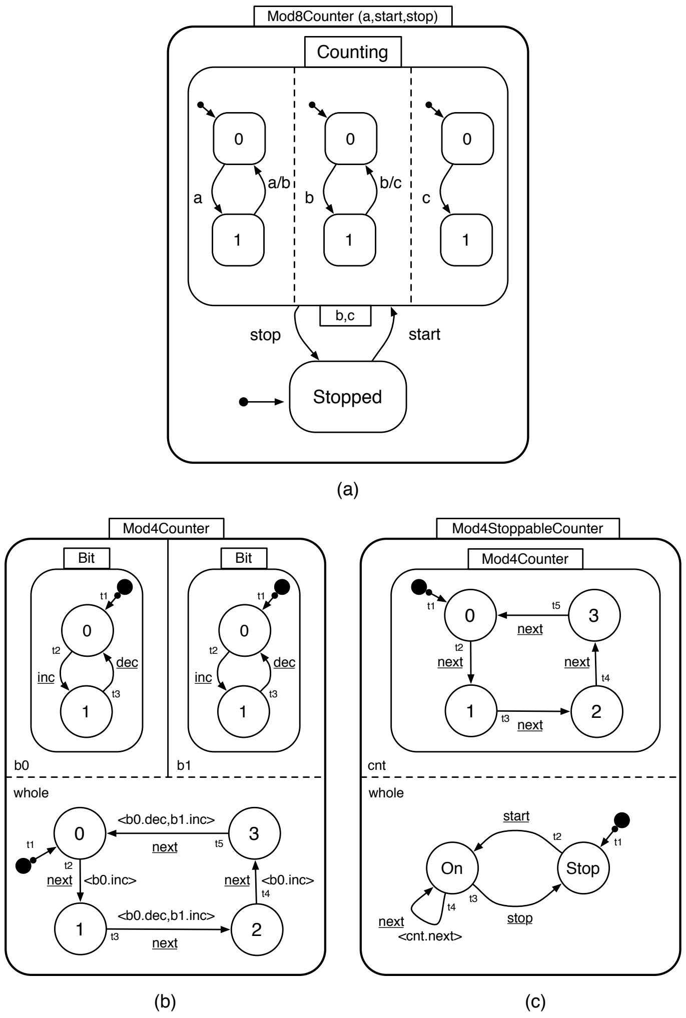

Fig. 1. Comparing Argos (a) and Part-Whole Statecharts (b, c) modelling of bit counters.

Part-Whole Statecharts were conceived with a radical commitment towards modularity. Solutions which departed radically from Harel's framework had therefore to be envisaged. In first place, in order to preserve self-containment, mutual knowledge among component modules had to be forbidden. Secondarily, a specific section, named "whole" (referred to in the rest of the paper as the *whole section* of the PWS) had to be introduced, with the aim of furnishing the necessary coordination among the non-communicating component modules. Such a coordination knowledge is provided by a state machine hosted in the whole section, whose state transition are labelled by events directed towards the components. Each state transitions may react to events coming either from outside the PWS, that is from further level of composition in which the PWS may be employed or from the components. In this way, the aggregation behaviour amongst component state machines is hosted explicitly in the state machine in the whole section of the PWS, and in no other places.

For example, the global coordination knowledge amongst the bits making a bit-counter is drawn explicitly in the whole section of the Part-Whole Statecharts of Figure 1-(b), where each transition is labeled by a set of events directed towards the different bits. It can be observed that the Bit abstractions of Figure 1-(b) are instead totally self-contained and identical. This allows to reuse them in other contexts with no further modifications provided an appropriate whole section specifies how the different components have to interact. It is finally evident that the designed interaction has *four* states of interest, while a similar information would be difficult to infer given an implicit modelling like that of state Counting in Figure 1-(a). Such a feature allows to use the newly assembled PWS as a component in other PWSs.

A key feature of Part-Whole Statecharts is indeed that composition is recursive. Each PWS uses PWSs as components and can be used on its turn as a component in more complex PWS, exposing only an interface derived directly from the state machine in the whole section, obtained by hiding implementation details. For example, Figure 1-(c) employs the counter defined in Figure 1-(b) in order to implement some sort of "stoppable" counter; a less trivial example, concerning timed traffic lights, is developed along the paper. Some PWS modules need to be obviously *primitive*, that is they do not have further component PWSs. A primitive PWS typically acts as a driver to the hardware or to some physical device. Observe that it makes no differences whether the modulo-4 counter employed as component in Figure 1-(c) has additional components or is simply an interface to some hardware device. In both cases the behavioural design of the stoppable counter deals only with a system having four states and four transitions.

The recursive composition feature marks an asymmetry between the whole and the component section of the PWS, since component PWSs have to be ready before the PWS is assembled (this fully complies with COTS software engineering practices). It therefore results that the communication amongst the lower half coordination section and the upper component section is asymmetrical, since the whole section knows the components, but the components need to be totally unaware of the whole section. Such a feature, together with the observation that a PWS denotes always *a single level* composition, is of paramount importance for compositional state semantics computation and verification, as further discussed in the paper. The term "hierarchy" in the context of this work therefore denotes the relationship between the whole section of the PWS and the the behaviour of its components.

Among the different Statecharts variants, Argos [5] first pioneered the modeling opportunities intrinsic in reduced event visibility and asymmetrical communication among enclosing and refined states. Parallel (AND) state decomposition allowed indeed a compositional representation of state machines in Argos, as in Figure 1-(a) representing a modulo 8 bit counter (adapted from [5] and [6]), where state Counting is refined into three independent state machines, each representing one bit of the counter. Local events b and c ensure mutual communication amongst the three decomposed parts of state Counting but are not allowed to influence external enclosing context, thus yielding behavioral encapsulation of the global behavior of the three bits. Asymmetrical, non-blocking communication amongst enclosing and enclosed refined states strengthen such encapsulation capabilities in Argos, and allow external states to act as controllers towards internally decomposed ones. Finally, we remark that both Argos enclosing and enclosed states run in distinct, parallel, processes, thus relieving the formalism of cross level arrows typical of early Statecharts blocking communication, further improving the understandability of state based abstractions.

The method presented in the paper can be framed also as an offspring of the verification technique called Compositional Reachability Analysis [7], which tries to reduce the search space involved in the verification of properties by "compositional minimisation", that is by intermediate simplification of parallel subsystems. Such a simplification takes advantage from structural properties of modules, for example events which are not globally observable or states which are locally forbidden are used in order to prune the global search tree [8]. Our approach allows to check for properties satisfaction by looking only at the immediate upper level of composition, thus involving only a limited number of state machines. Moreover, the presented technique works by reversing the traditional verification approach: properties are not verified *a posteriori*, rather, given the explicit representation of the desired behaviour, transitions can be added to the explicit behaviour only in case they do not violate the state constraint of both the starting and arrival state. Moreover, any uncontrollable event coming from the components has to be handled by a suitable state transition in order to maintain verified the constraint of the current state. Verification is therefore carried out at design time, and state constraints may help the designer in foreseeing non-trivial parts of the behaviour to be built.

The paper is structured as follows: in Section 2 we present a definitive proposal of related modelling constructs. The semantics of Part-Whole Statecharts is presented in Section 3, expressed through operations over a boolean algebra of state-based propositions, an account of which is given in the paper. Section 4 shows finally the method by which such a semantics can be constrained to match user-defined specifications.

2 Part-Whole Modelling Framework

A PWS is a compound state machine aimed at obtaining a systemic behavior out of the disjoint, parallel behavior of other PWS state machines, called *components*, which will be referred to collectively as *assemblage* of components (defined in Section 2.2). Such a systemic behavior is specified through a section of the PWS, called *whole* section (defined in Section 2.3), containing a state machine whose state transitions are labelled with specific symbols and state conditions, which will be specified in detail in Section 2.3. We will refer to such a state diagram simply as the *whole*.

2.1 Transition labelling

Transition labelling is aimed at specifying the allowed interaction between the whole and the assemblage and to make available newly created systemic behavior for further reuse and composition. Such an interaction is double way although not symmetric, in the sense that

1. special compound symbols, named either *actions* or *commands*, may be issued from the whole section state diagram towards components of the the assemblage in order to ask transitions to happen within it, and
2. transitions happening within the assemblage produce other symbols, named *events* towards the whole section and may possibly (i.e., depending on the evaluation of state conditions associated with the transition) trigger state transition within its state diagram.

Neither interaction nor mutual knowledge is possible among the components; moreover, in order to obtain full self-containment, components are not allowed to know anything about the whole. Full knowledge of the interface (defined in Section 2.3) of the PWSs making the assemblage is instead possible from the whole section of the PWS.

Example Figure 2 depicts a PW Statecharts which implements a very common traffic light having three states, R (red), G (green) and Y (yellow). The diagram comprises two main sections, separated by a dotted line, in order to suggest that information exchange is allowed among the two sections.

The upper half section of the diagram hosts the assemblage of *four* component PWSs making the traffic light, that is three identical lamps, which will be referred to by means of the identifiers l1, l2 and l3, and a timer t. The behavior of each component PWS is described through its interface, that is the only part of the component PWSs the modeller is allowed, as well as required, to know. Since the lamp behavior is identical for the three lamps, it is reported only once to save space in the diagram: it shows that each lamp may be either in an Off or in an On state and that it is possible to switch from one state to the another by taking the *input* state transitions t_2 and t_3 labelled, respectively, by the *input* events on and off. The term "input" suggests that necessary (but not sufficient) condition for such a transition to be taken is that the corresponding event has to be issued from the lower PWS section towards the machine containing the transition, representing therefore an *input* for such a machine. The timer component may be instead either in a TOut (timeout) or in a TIn (time-in) state. As in the case of the lamp behavior, two transitions named t_2 and t_3 allow to switch amongst the two states, with the notable difference that transition t_3 is drawn and behaves differently, being an *output* transition, that is a transition which is *not controllable* from the lower section. We adopt the graphical convention of underlining input events and of putting a small white hollow dot at the beginning of a non-controllable output transition. Finally, an initial state q_0 is present in both diagrams, depicted by a black dot as customary. The

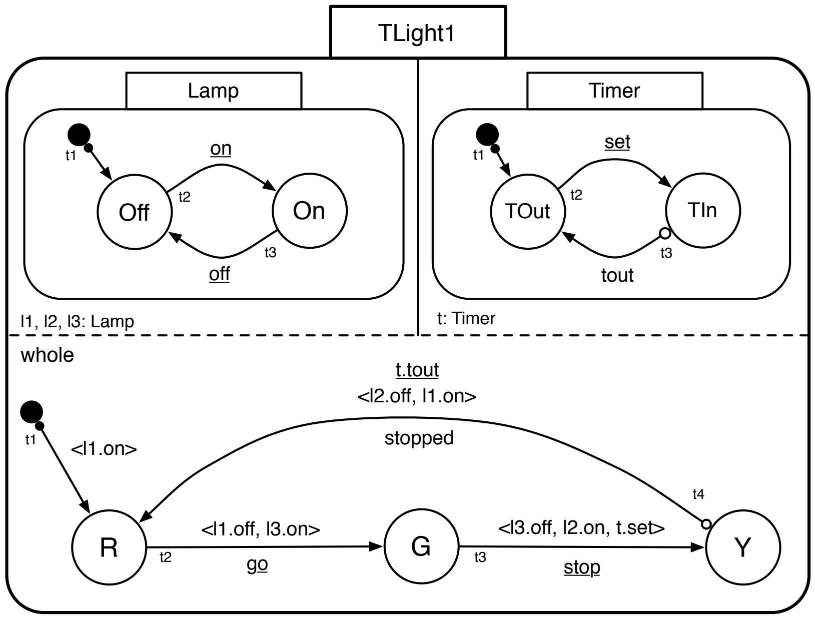

Fig. 2. A Part-Whole Statecharts which describes the coordinated and timed behavior of a traffic light `TLight1` built by assembling three lamps and a timer. The traffic light makes available two controllable input transitions (`go` and `stop`) as well as a non-controllable output timed transition (`stopped`) from yellow to red. Rounded corner rectangles emphasize the recursive compositional flavour of the approach.

initial state is connected by a special state transition (distinguished by a small black dot at the beginning) to the state the component machine will take at startup. We stress that the initial state is a state on its own, and not a distinguished state amongst the other states in the state diagram, for the reasons that will become clear in the following. For example, state Off is the only state in the lamp component state diagram which is reachable from the initial state, but it is *not* the initial state of the lamp diagram.

2.2 Assemblage of Components

Part-Whole Statecharts can be used as components for building more complex ones, exposing simply an interface defined according to the rules given in Section 2.3. A set $\mathsf{A} = \{c_1, \ldots, c_N\}$ of component PWSs is called *assemblage*. The behavior of each component $c \in \mathsf{A}$ is described by a state machine whose transition function is given by $\delta_c : Q_c \times T_c \rightarrow Q_c$, where Q_c and T_c are, respectively, the set of states and the set of transitions of c.

The joint behavior of the components of the assemblage can be correspondingly described by a state machine whose transition function is given by $\delta_{\mathsf{A}} : \mathbf{Q}_{\mathsf{A}} \times 2^{T_{\mathsf{A}}} \rightarrow \mathbf{Q}_{\mathsf{A}}$, where $\mathbf{Q}_{\mathsf{A}} = Q_{c_1} \times \ldots \times Q_{c_N}$, called the set of of global states of the assemblage is given the cartesian set of states of the assemblage components and T_{A} is the union set $T_{\mathsf{A}} = \cup_{c \in \mathsf{A}} T_c$ of the set of transitions of the components of the assemblage. By adopting the operational model given in Section 2.4, T_{A} is a singleton, that is, at most one global transition is processed at a time.

2.3 Whole Section

The systemic behavior of the PWS is described by a state machine whose transition function is given by $\delta_W : Q_W \times T_W \rightarrow Q_W$ where Q_W is a finite set of states and T_W a set of state transitions, which will be characterized in details in the next Section. The whole-section plays different roles within the PWS modeling approach. In first place it denotes explicitly the global behavior the modeler aims to achieve. Beyond such an abstract characterization, the explicit representation of behavior allows the state machine hosted within it to act, at the same time, both as a coordination machine as well as an interface in using the PWS for further composition.

State Transition Implementation Features The PWS approach requires the whole section to be able to ask state transitions to happen within the assemblage of components as a consequence of state transition happening in the whole and vice versa. Such an operational behavior is implemented through special symbols and state conditions associated to the state transitions in the state diagram of the whole section of the PWS, consisting of:

1. a *guard*, that is a boolean valued state proposition about the global state of the assemblage (that is, an expression of the Boolean Algebra of state proposition of Section 2.5) depicted within square brackets in the state diagram;
2. a *trigger*, that is a symbol (written underlined in the diagram) which denotes that the transition will be activated upon the receipt of either:
   (a) an event e sent to the PWS by another PWS having the current PWS as component, in which case $\underline{\mathsf{e}}$ is named *external* trigger;
   (b) an event c.e sent to the PWS by its component c, denoting the happening of a transition t labeled by e within the component c of the assemblage, in which case either $\underline{\mathsf{c.e}}$ or $\underline{\mathsf{c}.t}$ is named *internal* trigger.
3. a *list of commands*, where a command is a compound symbol which is used in order to request the activation of a state transitions in the assemblage of components. A command has the form c.e, meaning that it is required the activation of the externally triggerable transition labeled by event e in component c belonging to the assemblage.
4. an optional *output event*, that is a symbol which will be sent to all the PWSs which have the current PWS as component in order to notify them that the transition to which it belongs happened.

Different state transition typologies are made available within the formalism, depending on the different triggering mechanism:

1. *input* transitions, having an external trigger specified;

2. *automatic/output* transitions, having an internal trigger specified;
3. *automatic/asap* transitions, which have *no* triggers specified and are taken *as soon as possible* according to the operational rules of Section 2.4.

PWS Interface A PWS interface is essentially a state machine obtained from the PWS by hiding its internal implementation details, as described below. 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, obtaining a state machine which contains only the information that may be used by external composition contexts, in other words the externally observable behaviour of the PWS. Given any PWS state diagram, like those of Figures 2 and 3, we obtain their interface, used in the example of Figure 4 by:

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

Example Consider the state diagram in the whole-section of Figure 2. Transition t_1 starts from the initial state of the diagram and is taken automatically at startup. It is labelled by a single action, namely the one *turning on* lamp l1 from state Off to state On. It is possible to have different automatic transitions departing from the initial state, provided guards have been specified for each transition in order to determine which one will be executed at run time. In case no guard is specified, the transition is always chosen for execution. Transition t_2 and t_3 are input transitions, i.e. they are executed upon the receipt of, respectively, the events go and stop. The action list associated to transition t_2 specifies that components l1 and l2 have to be turned, respectively, *on* and *off*; the action list associated with transition t_3 specifies, in addition to similarly turning on and off components l2 and l3, that the timer component t has to begin a time counting session, moving from state Tout to state Tin. After a definite time interval has expired, timer t takes transition t.t_4, moving back to the timeout state Tout sending the timeout output event tout towards the whole, triggering the output transition t_4. It then turns components l2 and l1 *on* and *off* and additionally, at end, sends *outside* the event stopped, in order to notify other PWSs which have the current PWS TLight1 as component that the transition has been completed. For example, Figure 4 shows how the traffic lights of Figure 2 and 3 may be further aggregated in modelling a cross road behaviour. In that case, event stopped from traffic light main is aimed at triggering the automatic state transition t_3. Figure 3 shows a Part-Whole Statechart implementing a different traffic light behavior. The PWS has been in fact designed to start a full traffic light cycle upon the receipt of event go, triggering transition t_2, that is the other two transitions, t_3 and t_4, are taken automatically after two different timeout intervals have been set. Such a cycle is started, in the context of the cross road of Figure 4, by the action farm.go which labels transition t_3.

2.4 Operational Schema

We give here a brief account of the operational mechanism by which Part-Whole Statecharts operate. The whole section operates through a never ending cycle which iterates

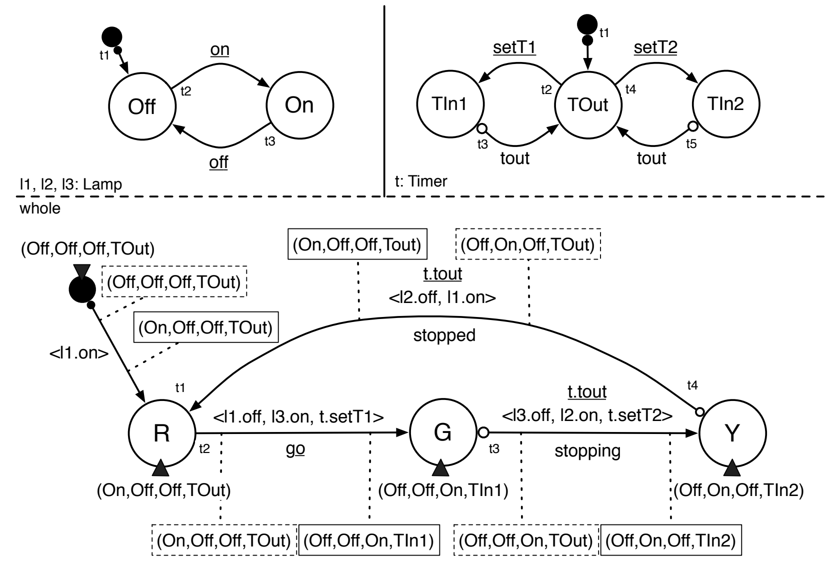

Fig. 3. The Part-Whole Statechart TLight2 implementing a traffic light which makes available a different behavior from the one shown in Figure 2, given by a single go controllable transition from red to green and two non-controllable transitions from green to yellow and from yellow to red. The diagram has been additionally labelled for verification purposes. Rounded corners rectangles have been removed for clarity.

a *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 a (possibly empty) set of state transitions T_S are selected for being executed, where each state transition in the set has the current state of the whole section as departing state, its guard condition is satisfied and either the incoming event matches its transition trigger or the transition has no trigger specified. In case T_S has more than one element, a state transition is chosen arbitrarily for execution. Section 4 present a method for designing the state machine such that, amongst other properties, $|T_S| \leq 1$, that is, there is always *at most one* transition to be executed.

State transition execution consists in delivering the commands (if any) of the command list to the assemblage components through a communication medium; sending the transition output event to all the PWSs which have the current PWS as component; moving the state machine in the whole to the ending state of the transition, which becomes the current state of the machine. The execution and communication tasks operate asynchronously, that is each PWS and the communication medium are driven by distinct threads or processors. The three main entities of the operational model (whole section, assemblage of components and communication medium) are therefore behaviorally independent and synchronize by a typical producer/consumer pattern through *communication ports*, that is mutex or read-write blocks of memory shared among the different processes. Each port can be structured as a FIFO list, in order to have the producer not to stop in case a new control signal is produced before a previously produced control message has been consumed. Some special component PWSs, for example timers, need however to have strict timing constraints, that is to operate synchronously with the processor/task driving the whole section. A suitable implementation technique can therefore be employed in such a case,

2.5 Algebra of State Propositions

A state proposition P is a boolean valued function from the powerset built from the set of assemblage global states $\mathbf{Q}_{\mathsf{A}}$ of the assemblage, in symbols $P : 2^{\mathbf{Q}_{\mathsf{A}}} \rightarrow \{true, false\}$. A *basic*, or *atomic* state proposition, written S^{c}, is true iff the state machine c is in state S. Logical propositional operator *and*, *or* and *not* (written, respectively, $\odot$, $\oplus$ and $\neg$) may be used in order to make complex state propositions out of simpler, basic ones, named *assemblage state propositions* (whose set will be referred to as $\mathcal{E}_{\mathsf{A}}$). Given two assemblage propositions e_1, e_2, $e_1 \preceq e_2$ denotes containment and $e_1 \equiv e_2$ equivalence. State propositions differ from ordinary propositional logic since the same state machine can not be in different states at the same time, that is, for any component c and for any state $\mathsf{S}, \mathsf{T} \in Q_c$, the assemblage state proposition $\mathsf{S}^{\mathsf{c}} \odot \mathsf{T}^{\mathsf{c}}$ is always false. We therefore use special operator symbols in place of the standard ones. It is useful also to have two special symbols, $\mathsf{any}^{\mathsf{c}}$ and $\mathsf{none}^{\mathsf{c}}$ in order to denote the state propositions which, respectively, are always and never true of a specific component state machine c. It can be shown that assemblage state propositions complemented by the two syntactic sugar symbols $\mathsf{ANY}_{\mathsf{A}}$ and $\mathsf{NONE}_{\mathsf{A}}$ which denote the assemblage propositions always true and false, form a Boolean Algebra, thus allowing to employ well known theorems and computability results.

State Proposition Transformation Let P be a state proposition about assemblage A and let t be a transition belonging to component $c \in \mathsf{A}$. We define the state proposition transformation operator $\mathsf{transf}(\cdot, \cdot)$ such that the transformed state proposition $P' = \mathsf{transf}(P, \mathsf{c}.t)$ has the intended meaning "the assemblage satisfied P and transition t happened in component c".

For example, let P be the state proposition "traffic light tl_1 is red *or* yellow". In other words, P denotes *two* states of the world (that is of the assemblage containing only tl_1) in which tl_1 is, respectively, red and yellow. It can be observed that such a set of states of the assemblage is transformed by tl_1 moving from red to green in the set of states in which tl_1 is, respectively, *green* and yellow, since the former state is transformed into the state of the assemblage in which tl_1 is green. P' is then correspondingly given by "tl_1 is green *or* yellow". Observe finally that state proposition P "tl_1 is red *or* green" is transformed in P' "tl_1 is green" since the set of two states of the assemblage in which tl_1 is, respectively, red and green collapse into the single state in which tl_1 is green.

3 PWS State Semantics

PW Statecharts have a computable state semantics, in the sense that it is possible to compute univocally, through the Boolean Algebra defined in Section 2.5, for each state

S belonging to the state diagram of the whole section of the PWS, a definite state proposition, called indeed the *state semantics* $\mathsf{sem}(\mathsf{S})$ of the state. Such a state proposition denotes the set of global states the assemblage will be able to assume at run time. We assume that the state diagram in the whole is a finite directed graph such that any state is reachable from the initial state q_0.

Let S be any state in the whole. In case $\mathsf{S} = \mathsf{q}_0$, that is the initial state of the whole, each component $c \in \mathsf{A}$ of the assemblage will be found in one of the states which are reachable from the initial state of component c, let them be denoted by the set $Q_0(c) \subseteq Q_c$. Then, for each component $c \in \mathsf{A}$ it is possible to form the state proposition $\mathsf{init}(c) = \bigoplus_{q \in Q_0(c)} \mathsf{q}^{\mathsf{c}}$ meaning that c is in one of the states in $Q_0(c)$. The state semantics of state q_0 is then given by

$$\mathsf{sem}(\mathsf{q}_0) = \bigoplus_{c \in \mathsf{A}} \mathsf{init}(c) \tag{1}$$

In the general case, it can be observed that it is possible to be in the generic state $\mathsf{S} \neq \mathsf{q}_0$ only by entering it through a finite number of incoming state transitions, let it be $T_I(\mathsf{S})$. Let us suppose to know in which set of global states the assemblage will be found when each $t \in T_I(\mathsf{S})$ has been completed (that is, when the control is moved to the final state S): let such set of states be denoted by the state proposition $\mathsf{post}(t)$. Since, as observed, it is possible to enter state S only through the transitions in $T_I(\mathsf{S}) = \{t_1, t_2, \ldots, t_N\}$, when the current state of the whole section is S then, necessarily, the assemblage has to be found in one of the global states denoted by $\mathsf{post}(t_1)$ *or* by $\mathsf{post}(t_2)$ *or* by $\mathsf{post}(t_3)$ and so on. In other words we have shown that

$$\mathsf{sem}(\mathsf{S}) = \bigoplus_{t \in T_I(\mathsf{S})} \mathsf{post}(t) \tag{2}$$

We show now how $\mathsf{post}(t)$ can be determined for the the generic transition $t_i \in T_I(\mathsf{S})$. Let R be the departing state of t. If t has been selected for execution then two cases have to be distinguished depending on whether the transition is externally or internally triggered.

In the case of externally triggered transitions, we have that both (a) the state transition guard and (b) the state semantics of the transition departing state R must necessarily hold with respect to the current state of the assemblage, let it be $\dot{\mathbf{q}} \in \mathbf{Q}_{\mathsf{A}}$. Such a state must therefore belong to their intersection $\mathsf{pre}(\cdot)$, let it be named *transition precondition*:

$$\mathsf{pre}(t) = \mathsf{sem}(\mathsf{R}) \odot \mathsf{guard}(t) \tag{3}$$

In the case of internally triggered transitions, we have to take into account that the current state of the assemblage $\dot{\mathbf{q}} \in \mathbf{Q}_{\mathsf{A}}$ may have been moved “outside” the semantics of the transition departing state R *before* the transition takes place. Transition precondition is therefore given by the intersection of (a) the state transition guard and (b) the state semantics of R transformed by the happening of the internal event, that is:

$$\mathsf{pre}(t) = \mathsf{transf}(\mathsf{sem}(\mathsf{R}), \mathsf{c}.e) \odot \mathsf{guard}(t) \tag{4}$$

where e is the assemblage internal event which happened within component c and $\mathsf{transf}(\cdot,\cdot)$ is the state proposition transformation operator described in Section 2.5. Once the transition precondition has been determined for one of the two cases, transition postcondition $\mathsf{post}(t)$ is given by chaining the transformations induced by the generic action list $\langle a_1, a_2, \ldots, a_N\rangle$ associated to transition t:

$$\begin{aligned} P_1 &= \mathsf{transf}(\mathsf{pre}(t), a_1) \\ P_2 &= \mathsf{transf}(P_1, a_2) \\ &\vdots \\ \mathsf{post}(t) &= \mathsf{transf}(P_{N-1}, a_N) \end{aligned} \tag{5}$$

4 Constraining PWS State Semantics

In this Section we analyze the relationship amongst the computation of state semantics, as carried out in Section 3, and a notion of *state correctness*, that is the requirent that a set of user-defined state propositions, called *state semantics constraints*, associated to the states of the machine hosted within the whole section of the Statecharts, will hold *exactly when* the state machine in the whole section moves through the correspondent states.

4.1 State safety

Given a PWS and a state proposition labelling $\mathsf{C}(\cdot)$ we say that the PWS is *safely specified* with respect to $\mathsf{C}(\cdot)$ iff the following state invariant holds:

Definition 1 (State safety invariant). *Let* S *be any state belonging the whole section of a PWS. If* S *is the current state of the whole section, then the current state* $\mathbf{q}$ *of the assemblage satisfies* $\mathsf{C}(\mathsf{S})$*, in symbols*

$$\mathsf{sem}(t) \preceq \mathsf{C}(\mathsf{S}) \tag{6}$$

In order to have the state invariant above satisfied, the system has to be specified in a consistent manner. By the discussion above, it can be observed that the requirement of Definition 1 may be broken either by the system section moving to state S with the assemblage global state moving, at the same time, to a state $\dot{\mathbf{q}}$ which does not satisfy the constraint $\mathsf{C}(\mathsf{S})$, or by an uncontrollable *internal* transition happening in the assemblage when the system section is in state S, resulting in a global state $\dot{\mathbf{q}}$ of the assemblage which does not satisfy its constraint $\mathsf{C}(\mathsf{S})$.

We say consequently that, in order to overcome the two causes above invalidating the state safety invariant, state transitions have to be specified, according to two different notions of correctness, discussed respectively, in Sections 4.2 and 4.3.

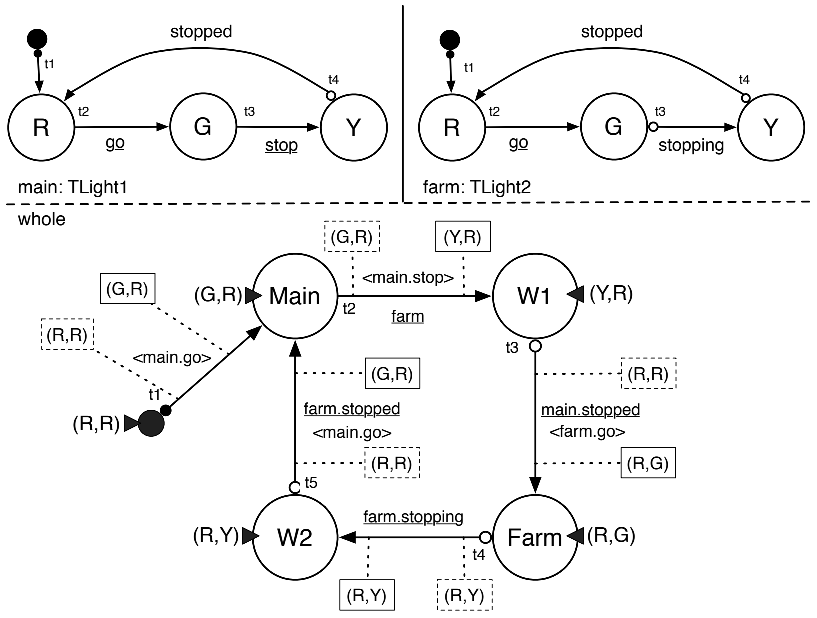

Fig. 4. A PWS aimed at coordinating two traffic lights, which give access to a main road from a secondary farm road by a controllable input transition (farm) followed by three internally triggered ones. The diagram has been additionally labelled for verification purposes. The previously defined PWSs TLight1 and TLight2 are now used as components by exposing only their interface.

4.2 Incoming state transition safety

In order to have the state invariant of Definition 1 *always satisfied* for state S it then suffices to check that, for any incoming transition to state S the state transition postcondition $\mathsf{post}(t)$ is included within the constraint of the state, that is

$$\mathsf{post}(t) \preceq \mathsf{C}(\mathsf{S}) \tag{7}$$

must hold for any state transition $t \in T_I(S)$. We have in fact that, since the containment relationship of Equation 7 holds for any $t \in T_I(S)$, the containment relationship

$$\bigoplus_{t \in T_I(\mathsf{S})} \mathsf{post}(t) \preceq \mathsf{C}(\mathsf{S}) \tag{8}$$

holds as well, since we are dealing with a Boolean Algebra. Since $\bigoplus_{t \in T_I(\mathsf{S})} \mathsf{post}(t)$ can be written equivalently as $\mathsf{sem}(\mathsf{S})$ by Equation 2, we have that $\mathsf{sem}(\mathsf{S}) \preceq \mathsf{C}(\mathsf{S})$, that is the state safety invariant of Definition 1 is satisfied.

Example Specification and verification of state semantics can be carried out with respect to the PWSs in the examples of the timed traffic light of Figure 3 and of the cross road of Figure 4. In the traffic light example we first specify the desired configuration of lamps and timing which is assigned to each state. In doing so we adopt, for clarity and compactness, the convention of writing the tuple (X, Y) standing for the assemblage proposition "the first component of the assemblage is in state X *and* the second component is in state Y". Such a convention can only be applied to atomic assemblage state propositions. We have for example that $\mathsf{C}(\mathsf{R})$ is given by $(\mathsf{On}, \mathsf{Off}, \mathsf{Off}, \mathsf{TOut})$, where the first three components of the tuple denote the state of the three lamps and the fourth the state of the timer. We associate graphically invariants to states by black isosceles triangles pointing towards the states and having the invariant written near their base. States G and Y are specified similarly to state R, except that they have the fourth component constrained to be, respectively, in state $\mathsf{TIn1}$ and $\mathsf{TIn2}$, meaning that the PWS will have to rest in the states for *at most* the definite amount of time measured by the timer: since, in both cases, there is a single outgoing transition triggered by the $\mathsf{timeout}$ event which marks the end of the measured time interval, it can be observed that the PWS will rest in the two states for *exactly* such time intervals. Observe that state R is constrained instead by the timer state TOut, meaning that there is no upper time limit to the permanence of the PWS in such a state. State transitions postconditions (depicted inside a solid boxed rectangle joined by a dashed line connector to the end of the transition) can be easily verified to satisfy Equation 8, as well as their precondition (depicted inside a dashed boxed rectangle joined by a dashed line connector to the begin of the transition) can be deduced by the semantics of each state in the diagram. The semantics of the initial state q_0 is inferred by the starting state of each component and is written as a constraint for the reasons that will become clear in the next Section. The example of the cross road of Figure 4 can be analyzed similarly. We observe that further timing properties can be inferred starting form the timing properties of the component traffic lights. For example, when the cross road in state Main the main road traffic light is in state G and the farm road in state R. Since both states are not timed, the cross road rests in state Main for an indefinite amount of time. When in state Farm the main road traffic light is in state R and the farm road in state G. Since the latter state belongs to PWS $\mathsf{TLight2}$ and we observed it lasts exactly a fixed amount of time, the PWS will rest in the Farm state exactly for the same an amount time. We observe finally that arbitrary safety properties can be checked against the design. For example, in order to check that both traffic lights are never both in the green state at the same time, it suffices to check that, for any $S \in T_W$, $(\mathsf{G}, \mathsf{G}) \not\preceq \mathsf{C}(\mathsf{S})$, which can be show to hold by examining a limited number of state propositions, namely the number of states $|T_W|$ in the whole sections.

4.3 Outgoing state transition safety

In order to have the state invariant of Definition 1 *always satisfied* for every state S of the whole it is further necessary to check that any violation of the constraint $\mathsf{C}(\mathsf{S})$ of the whole results in a response from the system, typically in the form of automatic transitions which originate from the state and react to such violations. Seen the other way, outgoing transitions are correctly specified if they "cover" any violation of the constraint of the state from which they originate. We show that the set of possible constraint violation can be effectively computed at design time, in the form of *exit zones*.

An *exit zone* associated to a state S is a pair (c, t), where c is a subproposition of $\mathsf{C(S)}$ and t is an automatic transition of the assemblage, such that, when the assemblage is in a global state state $\mathbf{q}$ satisfying c and transition t happens, state $\mathbf{q}$ is transformed into $\dot{\mathbf{q}}$ which does not satisfy $\mathsf{C(S)}$.

The entire set of exit zones associated to a state S denote therefore *all and only* the feasible violations of the state constraint, by stating which uncontrollable transitions the assemblage may take autonomously.

Each state S of the whole must therefore provide an automatic response for each associated exit zone (c, t) in order to avoid that control rests within the state while its constraint does not hold anymore. This is achieved by requiring that exit zones associated to a state S are covered by one or more state transitions originating from the state, where a set I of automatic state transitions is said to *cover* an exit zone (c, t) iff any transition t_i in I (1) is triggered by t and (2) the set of guards of the transitions in I form a partition of c.

Exit zone computation In order to compute the set of exit zones associated to a state S we start by defining the *autonomous transitions* associated to a generic state proposition C as the set $\mathcal{AU}(C)$ of state transitions that can be taken autonomously by the component machines of the assemblage when it is found in any global state that satisfies C.

As an example consider the assemblage Cross composed by a pair of a traffic lights $\mathsf{TLight2}$. Since each component state machine has 4 states, the whole assemblage may be found in 16 states, as depicted in Figure 5 and 7. Consider also the state proposition *traffic light 1 is in state Green and traffic light 2 is not in state q0 or traffic lights are both in state Red or both in state Yellow* written as $C = (((\mathsf{G}^{\mathsf{tl1}} \odot \neg \mathsf{q0}^{\mathsf{tl2}}) \oplus (\mathsf{R}^{\mathsf{tl1}} \odot \mathsf{R}^{\mathsf{tl2}})) \oplus (\mathsf{Y}^{\mathsf{tl1}} \odot \mathsf{Y}^{\mathsf{tl2}}))$.

State proposition C can be easily shown to be satisfied by any of the global states depicted as round squares in Figure 5, which also depicts the transitions departing from them. Thicker arrows originating from a white dot distinguish the autonomous state transitions associated to C, that is the set $\mathcal{AU}(C)$.

Such a set can be computed by Algorithm 1, which determines, for any *automatic* state transition in any state machine of the assemblage the respective starting state, let it be state S for the automatic transition t in state machine c. The algorithm then forms the state proposition "state machine c is in state S", written $\mathsf{S^c}$, and verifies whether both $\mathsf{S^c}$ and C hold. In the affirmative case, transition t may happen autonomously when the assemblage is in C and therefore t is added to the output set $\mathcal{AU}(C)$.

Algorithm 2 computes the set of exit zones associated to a state S by taking as input the set $\mathcal{AU}(C)$ from Algorithm 1, with $C = \mathsf{C(S)}$. If $t \in \mathcal{AU}(C)$ is an autonomous transition related to condition C, then it is guaranteed to happen iff the assemblage is in a global state which satisfies both $\mathsf{S^c}$ and C, let it be $\mathsf{pre} = \mathsf{S^c} \odot C$. Moreover, when t happens, it transforms state proposition pre in $\mathsf{post} = \mathsf{transf}(\mathsf{pre}, \mathsf{c}.t)$, meaning that any global state $\mathbf{q}$ of the assemblage which satisfies pre, after t takes place is transformed in the global state $\dot{\mathbf{q}}$ which satisfies post. It should be evident that $\dot{\mathbf{q}}$ may still satisfy C or not. We are interested in finding the subproposition p of pre such that, if $\mathbf{q}$ is satisfied by p then $\dot{\mathbf{q}}$ is not satisfied by C. In order to do so, we perform a sort of

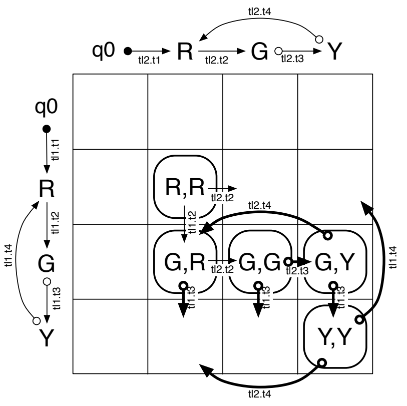

Fig. 5. The set of state transitions which can be taken from the assemblage when found in any of the global states denoted by a rounded rectangle. Autonomous state transitions $\mathcal{AU}(C)$ are drawn as thick arrows originating from a white dot, with C being the union of the global states.

Algorithm 1: Autonomous Transitions Computation Algorithm

```
input  : An assemblage of state machines A and an assemblage proposition $C$
output: The set $\mathcal{AU}$ of autonomous state transitions of A under state proposition $C$
foreach state machine $c$ in the assemblage do
    foreach automatic transition $t$ belonging to state machine $c$ do
        take the start state of the transition, say S;
        form the state proposition $\mathsf{S}^c$ = "state machine $c$ is in state S";
        compute the state proposition $\mathsf{S}^c \odot C$;
        if $\mathsf{S}^c \odot C$ is not empty then
            add transition $t$ to the set $\mathcal{AU}$;
        end
    end
end
```

"trimming" of the state proposition pre, as depicted by Figure 6. In first place the pre and post subpropositions are found as shown in (a); then proposition post is "trimmed", meaning that proposition C is *subtracted from* post, that is $\mathsf{postTrimmed} = \mathsf{post} \odot \neg C$ is computed, as shown in (b); finally in (c) it shown that, when such a proposition is not empty, it transformed into the proposition preTrimmed by reversing state transition t, i.e. $\mathsf{preTrimmed} = \mathsf{transf}(\mathsf{postTrimmed}, \mathsf{c}.t^{-1})$. State proposition preTrimmed can be shown to be the "largest" subproposition of C such that, once t happens, the global state of the assemblage does not satisfy C.

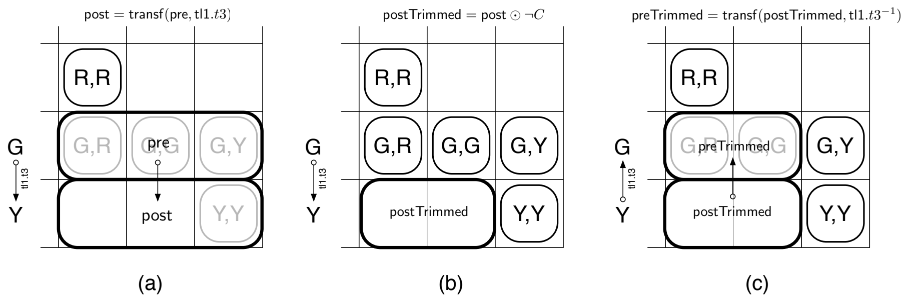

Fig. 6. The trimming steps in the inner block of Algorithm 2 applied to the proposition C of Figure 5 and to transition tl1.t3.

Algorithm 2: Exit Zones Computation Algorithm

```
input  : The set $\mathcal{AU}$ of autonomous state transitions of A under state proposition $C$
output: The set $E(C)$ of exit zones associated to state proposition $C$
foreach transition $t$ in $\mathcal{AU}$ do
    Let $\mathsf{pre} = \mathsf{S}^{\mathsf{c}} \odot C$, where S is the starting state of transition $t$ in state machine c;
    Let $\mathsf{post} = \mathsf{transf}(\mathsf{pre}, \mathsf{c}.t)$;
    Let $\mathsf{postTrim} = \mathsf{post} \odot \neg C$;
    if postTrim is not empty then
        Let $\mathsf{preTrim} = \mathsf{transf}(\mathsf{postTrim}, \mathsf{c}.t^{-1})$;
        Add $(\mathsf{preTrim}, t)$ to the set $E(C)$;
    end
end
```

Example Figure 8 (a) and (c) depict two state propositions, which are related to the assemblage of the two traffic lights tl1 and tl2, having three non-controllable transitions each (we reported the traffic light state diagrams to each axis of the two tables). In the former case, the state proposition, $\mathsf{C(S)} = (\mathsf{R}, \mathsf{G})$, has two exit zones,

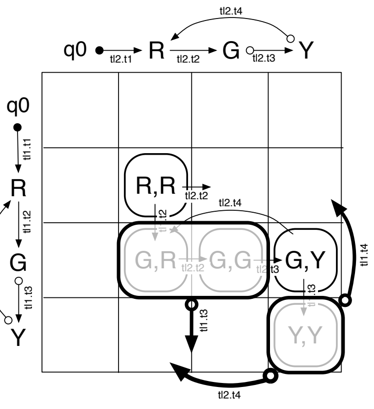

Fig. 7. The exit zones associated to a state constrained by the state proposition of Figure 5.

repectively given by $((\mathsf{R},\mathsf{G}),\mathsf{tl1}.t_2)$ and $((\mathsf{R},\mathsf{G}),\mathsf{tl2}.t_3)$: observe that they share the same subproposition of the state constraint. In the latter case, the state proposition, $\mathsf{C(S)} = (\mathsf{G},\mathsf{R}) \oplus (\mathsf{Y},\mathsf{R})$, has again two partly overlapping exit zones, this time respectively given by $((\mathsf{G},\mathsf{R}) \oplus (\mathsf{Y},\mathsf{R}),\mathsf{tl2}.t_2)$ and $((\mathsf{Y},\mathsf{R}),\mathsf{tl1}.t_4)$: observe that they share the same subproposition of the state constraint, which are, in both cases, coincident. The corresponding states (Figure 8 (b) and (d)) are safely specified since an adequate number of internally triggerable transitions is provided in both cases and the incoming state transition postcondition are comprised within the respective state constraints.

4.4 State constraint completeness

In addition to the safety requirements, it is important that the postconditions of the incoming transition to a state S *fill in completely* its state constraint. It becomes thus possible to use directly the constraint in place of the state semantics, further simplifying the calculation of transition pre- and postconditions. In symbols, if we have that, in addition to Equation 7, for each state $S \in T_W$

$$\bigoplus_{t \in T_I(\mathsf{S})} \mathsf{post}(t) \equiv \mathsf{C(S)} \tag{9}$$

it trivially follows that Equations 3 and 4 may be expressed in a simplified form given, respectively by

$$\mathsf{pre}(t) = \mathsf{C(R)} \odot \mathsf{guard}(t) \tag{10}$$

and by

$$\mathsf{pre}(t) = \mathsf{transf}(\mathsf{C(R)}, \mathsf{c}.e) \odot \mathsf{guard}(t) \tag{11}$$

We observe similarly that, in order to have *at most one* transition selected on the receipt of an external transition request, say be event e when the whole in in state S,

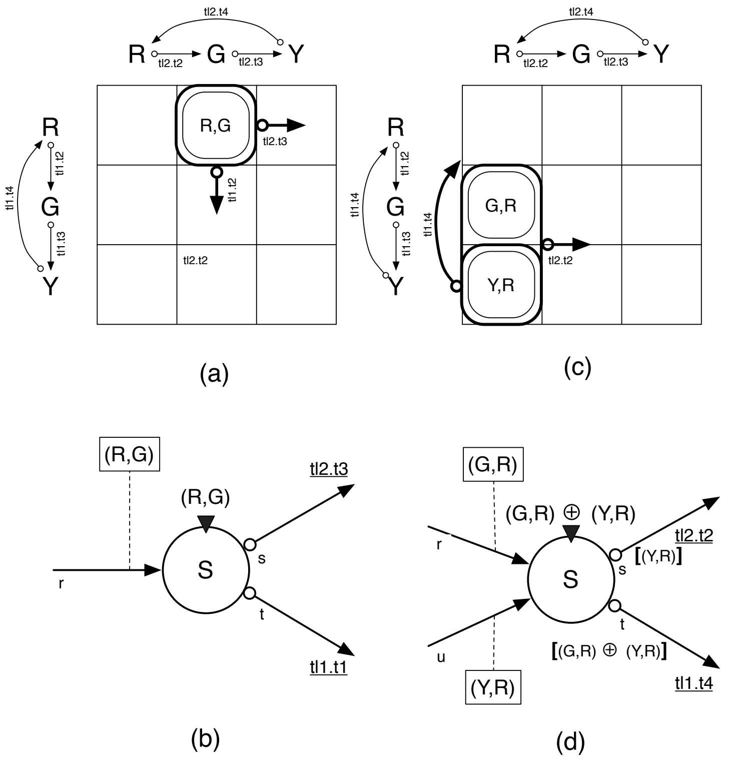

Fig. 8. The representation in the cartesian state space of two state propositions (a, c) together with the related exit zones, and of two state diagram fragments (b, d) implementing a suitable coverage of such zones.

the preconditions of the transition having e as external trigger must (1) fully cover the semantics of their departing state S and at the same time (2) be pairwise disjoint; in other words they must form a partition of the constraint $\mathsf{C(S)}$.

5 Conclusions and further work

The paper surveys the features of the Part-Whole Statechart approach and shows in detail the definitive form of its modeling constructs. On such a basis it shows also that a state semantics can be computed directly at design time: such a semantics tells essentially what happens to the assemblage of components as the PWS moves along the states of its whole section, and vice versa. Such a semantics is however ineffective unless it is contrasted against a well defined specification of a desired behavior. Such a specification can be carried out by constraining *a priori* the state semantics by propositions on the states of the machine. In such a way the modeller gains a robust instrument for the unambiguous specification of the meaning of each state in the designed behaviour, for example by telling that a cross road has the main street enabled when the traffic light is green on such a road and red on the crossing one.

The paper presents then a notion of state correctness with respect to such a specification: in order to keep it satisfied, state transitions must comply with the state constraints of both their initial and arrival states; in second place, since any state constraint may be potentially invalidated by any uncontrollable event happening in the assemblage of components, an automatic reaction by the system must be provided: for example a sensor going out of the desired value, a timer reaching a timeout condition, and so on. The paper shows that by comparing a state proposition with the uncontrollable transitions happening when the whole section of the Part-Whole Statechart is in such a state, it is possible to identify constraint subproposition together with the transition which potentially may invalidate them. By providing a suitable automatic triggered transition for any of such propositions, the system is assured to move to a consistent state when one of its constraints is invalidated. Due to the strong modularity which characterise the approach, such a kind of exception handling mechanism may be show to be practical both in assuring constraint satisfaction and in helping the designer in foreseeing system reactive behaviours which may not be evident at first glance.

It should be observed, finally, that the finite and small number of states in each whole section allows to check easily, by visiting the entire state machine graph and comparing each state constraint with unwanted state propositions, that dangerous situations never happen. As observed in the example, the cross road may be trivially shown to never enter a global state in which the traffic lights are both green (safety) or both red (liveness). Further research is therefore needed in order to fully explore the topic. Other developments may come from the application of the method to dependable systems, especially embedded ones. It should be noted in fact that the proposed approach, which relies on a structured communication medium, fits well with technological evolution, which sees autonomous components connected by programmable field buses, marking a shift of paradigm with respect to early hardware wiring practices.

References

1. Pazzi, L.: Extending statecharts for representing parts and wholes. In: Proceedings of the EuroMicro-97 Conference, Budapest, Hungary. (1997)
2. Pazzi, L.: Part-whole statecharts for the explicit representation of compound behaviors. In: Proceedings of the UML 2000 Conference, York (UK). Volume 1939 of LNCS., Springer (2000) 541–555
3. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
4. Cook, S., Daniels, J.: Designing Object Systems - Object-Oriented Modelling with Syntropy. Prentice-Hall (1994)
5. Maraninchi, F.: Operational and compositional semantics of synchronous automaton compositions. In: In CONCUR. LNCS 630, Springer-Verlag (1992) 550–564
6. Jourdan, M., Maraninchi, F., Lavoisier, M.Z.R.: A modular state/transition approach for programming reactive systems. In: In Worshop on Language, Compiler, and Tool Support for Real-Time Systems, Mouriel.Jourdan@imag.fr (1994)
7. Pezzè, M., Taylor, R.N., Young, M.: Graph models for reachability analysis of concurrent programs. ACM Transactions on Software Engineering Methodologies **4**(2) (1995) 171–213

8. chi Cheung, S., chi Cheung, S., Kramer, J., Kramer, J.: Checking subsystem safety properties in compositional reachability analysis. In: In Proceedings of the 18th International Conference on Software Engineering. (1996) 144–154