Modularity and Part-Whole Compositionality for Computing the State Semantics of Statecharts

Luca Pazzi
University of Modena and Reggio Emilia
Via Vignolese, 905
I-41125 Modena, Italy
luca.pazzi@unimore.it

Marco Pradelli
RE:Lab
Via Tamburini, 5
I-42122 Reggio Emilia, Italy
marco.pradelli@unimore.it

Abstract
The paper discusses modularity and compositionality issues in state-based modeling formalisms and presents related recent research results. Part-Whole Statecharts provide modular constructs to traditional Statecharts in order to allow incremental and fully reusable composition of behavioral abstractions, enforcing explicitly the coordinated systemic behavior and bringing benefits to subsequent modeling and implementation phases. The paper shows that Part-Whole Statecharts have a computable semantics, which can be specified through a constraint-driven specification method. Such a method allows to specify and verify the intended meaning of states directly at design time, thus avoiding to employ less effective verification techniques, such as exhaustive testing or model checking.

1. Introduction
Part-Whole Statecharts [10][12] (shortened either as PW Statecharts or PWS) were presented, about a decade ago, with the aim of providing Harel’s Statecharts formalism [4] with truly modular constructs. Various attempts have been made since Statecharts’ introduction in order to allow the modular encapsulation of behavioral abstractions. The Syntropy [3] and the Fusion [2] object-oriented development methodologies, for example, pioneered the field by, respectively, encapsulating the state behavior of single entities within state modules hosted into parallel Statecharts sections and achieving behavioral aggregation by raising relationships to first class objects.

Both methods (amongst others) lack, however, a clean notion of modular behavioral aggregation. Behavioral aggregation is well understood since CCS [9] and CSP [6] process algebras: both formalisms may be observed to be closed under behavioral aggregation, since the synchronization of two or more processes gives rise to a new process which can be meant as the aggregation of its components.

It can be observed instead that the aggregation of a global behavior from separate modules hosted into parallel Statecharts sections does not enjoy the same property of closure. In other words, by using the Statechart coordination and communication model – that is by letting the state machine residing on each module to be both aware of the current status of the other parallel machines and able, at the same time, to act on them – we do not obtain a novel single behavioral module from the original constituent modules; we get instead the original state behaviors which interact by both exchanging messages and mutual condition testing. Obtaining a systemic coordinated behavior through such a model has 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 [11]. In addition, the operational semantics of such a coordination model is necessarily ambiguous, since control events bounce from one component to another depending on the current status of each one. We will refer to such a way of coordinating AND decomposed modules as the implicit way of assembling global behavior.

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.
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. Secondly, 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 both the necessary coordination among the non-communicating component modules and to act as interface. 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 transition 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 modeling 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 behaviour of its components.

Among the different Statecharts variants, Argos [8] first showed 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 [8] and [7]), 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 [16], 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 [1]. 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 user-defined state constraints assigned to both 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. A final outcome of our work is that, by the proposed approach, “state semantics” can be determined directly at design time. By state semantics we mean that each state in the whole section can be put in correspondence with a set of states belonging to the cartesian product of the states of the components. This marks an advance with respect to traditional Statecharts [17][5][18]. Such a formal denotation of the state semantics is being employed in research in the field of dependable, autonomous and safety-critical systems [13][14][15].

The paper is structured as follows: in Section 2 we present a definitive revision of Part-Whole Statecharts modeling constructs upon which an operational model and the determination of state semantics is based. 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. Modeling 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 2.3.1. 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 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 2.3.2) of the PWSs making the assemblage is instead possible from the whole section of the PWS.

2.1.1 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,
Figure 2. A Part-Whole Statecharts which describes the co-ordinated 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.

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 the lower PWS corresponding event has to be issued from the lower PWS condition for such a transition to be taken is that the PWS is in the only state in the lamp component state diagram which is reachable from the initial state, 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 2.3.2. A set $A = \{c_1, \ldots, c_N\}$ of component PWSs is called assemblage. The behavior of each component $c \in A$ 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 $\delta_A : Q_A \times \delta^{2T_A} \rightarrow Q_A$, where $Q_A = Q_{c_1} \times \ldots \times Q_{c_N}$, called the set of global states of the assemblage, is the cartesian set of states of the assemblage components and $T_A$ is the union set $T_A = \bigcup_{c \in A} T_c$ 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 $\delta_W : Q_W \times \delta_W \rightarrow Q_W$ 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.

2.3.1 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 \( e \) 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 \( c.e \) or \( c.t \) 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.

### 2.3.2 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.

### 2.3.3 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 \( 11 \) 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 \( l_1 \) and \( l_2 \) have to be turned, respectively, off and on; the action list associated with transition \( t_3 \) specifies, in addition to similarly turning on and off components \( l_2 \) and \( l_3 \), 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 \( t_3 \), 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 \( l_2 \) and \( l_1 \) off and on 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 modeling 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 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 set of assemblage global states $Q_A$ of the assemblage, in symbols $P : Q_A \to \{\text{true}, \text{false}\}$. 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, $\circlearrowleft$, $\circlearrowright$ 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 $E_A$). 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 $S, T \in Q_c$, the assemblage state proposition $S^c \circlearrowleft T^c$ is always false. We therefore use special operator symbols in place of the standard ones. It is useful also to have two special symbols, $\forall c$ and $\nexists c$ 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 ANYA and NONEA which denote the assemblage propositions always true and false, form a Boolean Algebra, thus allowing to employ well known theorems and computability results.

### 2.5.1 State Proposition Transformation

Let $P$ be a state proposition about assemblage $A$ and let $t$ be a transition belonging to component $c \in A$. We define the state proposition transformation operator $\text{transf}(\cdot, \cdot)$ such that the transformed state proposition $P' = \text{transf}(P, c, t)$ 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

PWS 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 state proposition, called indeed the state semantics $\text{sem}(S)$ 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 $S = q_0$, that is the initial state of the whole, each component $c \in A$ 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 $c \in A$ it is possible to form the state proposition $\text{init}(c) = \bigoplus_{q \in Q_0(c)} q^c$ meaning that $c$ is in one of the states in $Q_0(c)$. The state semantics of state $q_0$ is then given by

$$\text{sem}(q_0) = \bigcap_{c \in A} \text{init}(c)$$

In the general case, it can be observed that it is possible to be in the generic state $S \neq q_0$ only by entering it through a finite number of incoming state transitions, let it be $T_I(S)$.

Let us suppose to know in which set of global states the assemblage will be found when each $t \in T_I(S)$ 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 $\text{post}(t)$. Since, as observed, it is possible to enter state $S$ only through the transitions in $T_I(S) = \{t_1, t_2, \ldots, t_N \}$, 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 $\text{post}(t_1)$ or by $\text{post}(t_2)$ or by $\text{post}(t_3)$ and so on. In other words we have shown that

$$\text{sem}(S) = \bigoplus_{t \in T_I(S)} \text{post}(t)$$

We show now how $\text{post}(t)$ can be determined for the generic transition $t_i \in T_I(S)$. Let $R$ be the source 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 source state $R$ must necessarily hold with respect to the current state of the assemblage, let it be $q_i \in Q_A$. Such a state must therefore belong to their intersection $\text{pre}(\cdot)$, let it be named transition precondition:

$$\text{pre}(t) = \text{sem}(R) \circ \text{guard}(t)$$

In the case of internally triggered transitions, we have to take into account that the current state of the assemblage $q_i \in Q_A$ may have been moved “outside” the semantics of the transition source 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:

$$\text{pre}(t) = \text{transf}(\text{sem}(R), c, e) \circ \text{guard}(t)$$

where $e$ is the assemblage internal event which happened within component $c$ and $\text{transf}(\cdot, \cdot)$ is the state proposition transformation operator described in Section 2.5.1. Once the transition precondition has been determined for one of the two cases, transition postcondition $\text{post}(t)$ is given by chaining the transformations induced by the generic action list $\langle a_1, a_2, \ldots, a_N \rangle$ associated with transition $t$:

$$P_1 = \text{transf}(\text{pre}(t), a_1)$$
$$P_2 = \text{transf}(P_1, a_2)$$
$$\vdots$$
$$\text{post}(t) = \text{transf}(P_N, a_N)$$

### 4. Constraining 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 requirement that a set of user-defined state propositions, called state semantics
Given a PWS and a state proposition labelling $C(\cdot)$ we say that the PWS is \textit{safely specified} with respect to $C(\cdot)$ iff the following state invariant holds:

\textbf{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 of the assemblage satisfies $C(S)$, in symbols

\begin{equation}
\text{sem}(S) \preceq C(S)
\end{equation}

\section{Discussion}

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 4.1 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 $q$ which does not satisfy the constraint $C(S)$, or by an uncontrollable \textit{internal} transition happening in the assemblage when the system section is in state $S$, resulting in a global state $q$ of the assemblage which does not satisfy its constraint $C(S)$.

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 4.3 and 4.4.

\section{Incoming state transition safety}

In order to have the state invariant of Definition 4.1 \textit{always satisfied} for state $S$ it then suffices to check that, for any incoming transition to state $S$ the state transition postcondition $\text{post}(t)$ is included within the constraint of the state, that is

\begin{equation}
\text{post}(t) \preceq C(S)
\end{equation}

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

\begin{equation}
\bigoplus_{t \in T_I(S)} \text{post}(t) \preceq C(S)
\end{equation}

holds as well, since we are dealing with a Boolean Algebra. Since $\bigoplus_{t \in T_I(S)} \text{post}(t)$ can be written equivalently as $\text{sem}(S)$ by Equation 2, we have that $\text{sem}(S) \preceq C(S)$, that is the state safety invariant of Definition 4.1 is satisfied.

\subsection{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 $C(R)$ is given by $(0, n, o, f, tOu)$, 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 $TIn1$ and $TIn2$, meaning that the PWS will have to rest in the states for \textit{at most} the definite amount of time measured by the timer: since, in both cases, there is a single outgoing transition triggered by the 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 $tOu$, 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 preconditions (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 from the timing properties of the component traffic lights. For example, when the cross road in state $\text{Main}$ 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 $\text{Main}$ for an indefinite amount of time. When in state $\text{Farm}$ the main road traffic light is in state $R$ and the farm road in state $G$. Since the latter state belongs to PWS TLigh2 and we observed it lasts exactly a fixed amount of time, the PWS will rest in the Farm state exactly for the same amount of 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 $S \in T_W$, $(G, G) \not\preceq C(S)$, which can be shown to hold by examining a limited number of state propositions, namely the number of states $|T_W|$ in the whole sections.
In order to have the state invariant of Definition 4.1 always satisfied for every state $S$ of the whole it is further necessary to check that any violation of the constraint $C(S)$ 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 violation. 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 violations can be effectively computed at design time, in the form of automatic transitions which originate from the whole results in a response from the system, typically.

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 $c$. 

### 4.4.1 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 $\mathcal{A}(C)$ 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 TLight2. Since each component machine has 4 states, the whole assemblage may be found in 16 states, as depicted in Figure 5-(a). 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 $C = (((G^{t1} \oplus \neg q0^{t2}) \oplus (R^{t1} \odot R^{t2})) \oplus (Y^{t1} \odot Y^{t2}))$.

State proposition $C$ can be easily shown to be satisfied by any of the global states depicted as round squares in Figure 5-(a), 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 $\mathcal{A}(C)$.

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 $c$. 

---

**Figure 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.
is in state \( S^c \), written \( S^c \), and verifies whether both \( S^c \) 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 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 \( S^c = "\text{state machine } c\text{ is in state } S"; \)
    compute the state proposition \( S^c \circ C \);
    if \( S^c \circ C \) is not empty then
      add transition \( t \) to the set \( \mathcal{AU} \);
  end
end
```

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 \( C = C(S) \). 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 \( S^c \) and \( C \), let it be \( \text{pre} = S^c \circ C \). Moreover, when \( t \) happens, it transforms state proposition \( \text{pre} \) in post = \( \text{transf}(\text{pre}, c.t) \), meaning that any global state \( q \) of the assemblage which satisfies \( \text{pre} \), after \( t \) takes place is transformed in the global state \( q \) which satisfies post. It should be evident that \( q \) may still satisfy \( C \) or not. We are interested in finding the subproposition \( p \) of \( \text{pre} \) such that, if \( q \) is satisfied by \( p \) then \( q \) is not satisfied by \( C \). In order to do so, we perform a sort of “trimming” of the state proposition \( \text{pre} \), as depicted by Figure 6. In first place the \( \text{pre} \) and post subpropositions are found as shown in-(a); then proposition post is “trimmed”, meaning that proposition \( C \) is subtracted from post, that is \( \text{postTrimmed} = \text{post} \circ \neg C \) is computed, as shown in-(b); finally in-(c) it shown that, when such a proposition is not empty, it transformed into the proposition \( \text{preTrimmed} \) by reversing state transition \( t \), i.e. \( \text{preTrimmed} = \text{transf}(\text{postTrimmed}, c.t^{-1}) \). State proposition \( \text{preTrimmed} \) can be shown to be the “largest” subproposition of \( C \) such that, once \( t \) happens, the global state of the assemblage does not satisfy \( C \).

**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 \( \text{pre} = S^c \circ C \), where \( S \) is the starting state of transition \( t \) in state machine \( c \);
  Let post = \( \text{transf}(\text{pre}, c.t) \);
  Let postTrim = post \circ \neg C;
  if postTrim is not empty then
    Let \( \text{preTrim} = \text{transf}(\text{postTrim}, c.t^{-1}) \);
    Add (\( \text{preTrim}, t \)) to the set \( E(C) \);
end
```

4.4.2 Example

Figure 7-(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, \( C(S) = (R, G) \),
has two exit zones, respectively given by \((R, G), tl1.t2\) and
\((R, G), tl2.t3\): observe that they share the same subproposition
of the state constraint. In the latter case, the state propo-
sition, \( C(S) = (G, R) \oplus (Y, R) \), has again two partly
overlapping exit zones, this time respectively given by \((G, R) \oplus
(Y, R), tl2.t2\) and \((Y, R), tl1.t4\): observe that they share
the same subproposition of the state constraint, which are, in
both cases, coincident. The corresponding states (Figure 7-
(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 com-
prised within the respective state constraints.

4.5 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 \( S \in T_W \)

\[
\bigoplus_{t \in T_W(S)} \text{post}(t) \equiv C(S) 
\]  

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

\[
\text{pre}(t) = C(R) \circ \text{guard}(t) 
\]

and by

\[
\text{pre}(t) = \text{transf}(C(R), c.e) \circ \text{guard}(t) 
\]

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

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 mod-
eling constructs. On such a basis it shows also that a state
semantics can be computed directly at design time: such a sem-
antics 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 ineffec-
tive unless it is contrasted against a well defined speci-
fication of a desired behavior. Such a specification can be
carried out by constraining \emph{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. Maintaining the validity of such propositions on the state of the machine requires a notion of state correctness with respect to such a specification: in order to keep it satisfied, state transitions must comply, in first place, 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 research 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 has shown 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 cross road example, it may be easily shown that it never enters a global state in which the traffic lights are both green (safety) or both red (liveness). Our techniques marks therefore an evident advantage with respect to current model checking techniques which, besides the computational burden of having to explore the tree of all feasible behaviors, are essentially \emph{a posteriori} techniques, thus depriving the design phase of the expressive power inherent the specification of desired behaviors.

References


