Systems of Systems Modeled by a Hierarchical Part-Whole State-Based Formalism

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

The paper presents an explicit state-based modeling approach aimed at modeling Systems of Systems behavior. The approach allows to specify and verify incrementally safety and liveness rules without using model checking techniques. The state-based approach allows moreover to use the system behavior directly as an interface, greatly improving the effectiveness of the recursive composition needed when assembling Systems of Systems.

1 Introduction

While traditional systems engineering focuses on systems made of simple constituent parts, Systems of Systems (SoS) comprise multiple autonomous systems which can be very different in technology, context, operation, geography and conceptual frame [5]. The coordinated behavior of such systems constitutes the primary behavior of the Sos itself. Finally, Sos have a recursive nature, each component of a Sos being possibly a SoS itself.

Although the difference can be at first sight very loose, since constituent parts in traditional system engineering are often system themselves, engineering Systems of Systems poses very specific challenges due to the heterogeneous nature and role of the systems participating in the whole assembly. In other words, the focus shifts from choosing the right system to choosing the system, or even multiple systems, able to satisfy the right specific behavioral and functional requirements. Component systems in SoS need therefore to be easily interchangeable both in the design and in the operation phases.

Such an heterogeneous diversity and interchangeability context calls for a unifying language for describing and prescribing the behavior of both the components and the assembled system. Such a language should be general enough for the sake of taking into account system diversity, but, at the same time, it should be able to express modal and logical properties of the global system being engineered, that is what the system behavior should or should not be allowed to do. Such a language should moreover take into account architectural issues.

The recursive nature of the SoS approach, and the need for interchangeability of component parts while still satisfying requirements, calls in fact for thinking system architecture in a modular way. A system must be modeled by a module which should be able to play different roles in different compound systems: this in turn requires that a system should not be allowed to know any detail of the compound systems that will contribute to form. On the other hand, the compound system should be able to know in detail the behavior of the single systems by which it is composed by. The relationships among component and compound system is therefore asymmetrical, distinguishing and clearly identifying the part from the whole. The whole is required to know its parts, the parts are forbidden to know the whole in order to be interchangeable among different compound systems. It is finally remarked that both roles must be played by the same module, since, as observed, each system is a SoS itself.

Larsen, Legay, Nyman (Eds.): 1st Workshop on
Advances in Systems of Systems (AiSoS 2013)
EPTCS 133, 2013, pp. 24–34, doi:10.4204/EPTCS.133.3

The asymmetric part-whole composition framework suggests consequently, an asymmetric communication framework. The compound whole should be able to control *directly* the component parts: the parts, on their turn, should not control directly the whole, but be allowed to influence it only *indirectly*. Consider for example an Air Traffic Control, and suppose to model it as a SoS where airplanes are, among the others, the principal system components. While an ATC may issue commands to the different planes being under its control, airplanes may only *notify* ATC of their position, altitude, possible failures as well as requests for landing, approaching, takeoff, and so on. ATC will consider each notification or request from one of its subsystems (planes, runaways, safety ground systems, weather stations) and issue back commands to them taking into account the *global state* of the compound ATC system, resulting from the different planes position and altitude, weather conditions, runaway free or in use by other planes, safety ground systems, and so on.

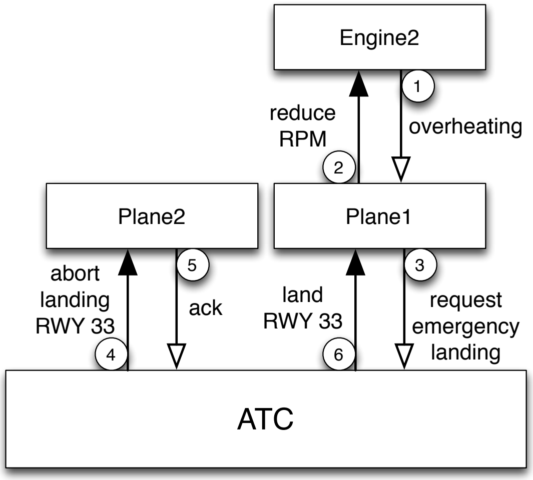

Figure 1: Event flow exchanged amongst four systems belonging to an Air Traffic Control (ATC) scenario. Command and feedback among systems are represented, respectively, by black and white arrows.

Figure 1 shows a typical flow of events from different systems arranged hierarchically relating to an ATC scenario. An engine (`Engine2`) notifies the control system of the airplane to which it belongs an overheating failure (1). The airplane (`Plane1`) reacts to the failure by (2) commanding `Engine2` to reduce power and by (3) notifying the `ATC` of the problem. Observe that the system `Plane1` has the engine under its direct control, but it can not send commands directly to the `ATC` system, but only notifications. The `ATC` system, on its turn, has `Plane1` and `Plane2` under its control, and gives a command to the second plane to abort landing (4) on specified runaway. Once the `Plane2` acknowledges to abort its landing (5), the runaway is free for `Plane1`, which is given the command (6) to undertake its emergency landing.

1.1 Structure of the work

By the approach proposed in this paper, a System of Systems (i) has control over other systems and (ii) is in turn controlled by other systems. According to such a view, its behavior has to play seamlessly both roles, that is, it has to be be, *at the same time*, a controller and a controlled behavior. The system behavior has moreover to agree within the architectural and event flow framework depicted above.

We discuss the different aspects in the rest of the paper. In Section 2 we argue that interacting systems can be modeled equivalently by introducing an explicit additional system having the original systems as components, which encapsulates the dynamical aspects regarding interaction among the original systems. We argue moreover that such a modeling brings advantages in software quality terms. (It is therefore implied that any set of interacting processes can be modeled by a more effective SoS having the original interacting systems as components.) In Section 3 we we show the feasibility of the approach hypothesized in the previous Section by adopting the PW-Statecharts state-based formalism, which allows to represent, by a single construct, the behavior of a system acting both as whole and as part of more complex wholes. We show moreover that the state semantics of the system acting as whole is computable and that it is possible to check its correctness against safety and liveness rules by exploring a finite state diagram.

2 Implicit approach in system modeling

We propose to use state diagrams for expressing behavioral specifications of Systems of Systems since state-based modeling is clear, realistic, formal and rigorous [6] for describing and prescribing the behavior of both the components and the assembled system. Such a language is general enough for taking into account system diversity and for enforcing and verifying, through model-checking techniques, modal and logical properties of the global system being engineered. In this section we explore the relationship between state-based behavioral descriptions and architectural issues.

Modular encapsulation of state-based behavioral abstractions is still an open issue. Object-oriented development methodologies, such as Real-time UML [3], encapsulate the state behavior of single systems within state modules hosted into parallel Statecharts [7] sections.

A system is thus modeled by a set of interacting parallel state machines (each state machine hosted within an AND-decomposed state, each single state of the machine being and XOR-decomposed state), which synchronize through message exchange and mutual condition testing. Statecharts state decomposition mechanism furnishes thus a straightforward way of representing single entities, which compose into more complex systems through synchronization. In other words, process synchronization denotes system aggregation.

Example. Figure 2 shows two interacting state machines, `farm` and `main`, each hosted within a Statecharts' parallel sections. Each can be seen as a state-based process. The farm road is normally stopped, while the main road is normally open. States `R`, `G` and `Y` stand for lights read, green and yellow. A car arriving at the crossroad from the farm road is sensed, trough some device not in the example, by the `farm` traffic light, which asks the `main` to block the main road by sending it an "open" request (`reqGoF`), which in turn sends a "stop" request to the main-troad traffic light (`reqStopM`). The farm traffic light moves then to a special wait state `w1`, aimed at modeling the fact that we have to wait for the main road traffic light to go to the `R` state before moving to the `G` state.

—

The two state-based processes of the example in Figure 2, once synchronized, become a single process, which in turn denotes a single system, namely *the crossroad controller system*. It can be observed that process synchronization can be achieved by two different approaches: by direct communication among system components, as in the two traffic lights example of Figure 2 or through an explicit additional entity representing the system being modeled, which has the system components as parts and hosts

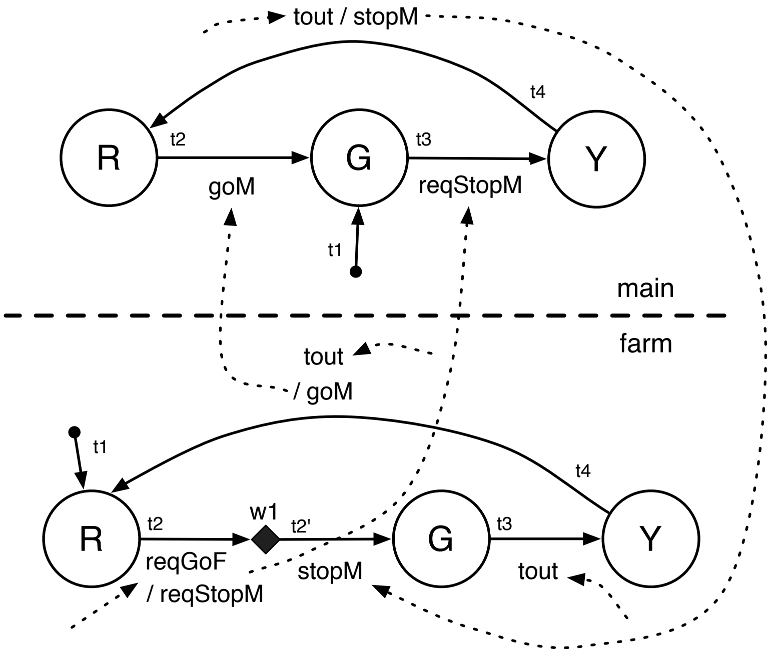

Figure 2: Two mutually interacting parallel state-based processes, each associated to a traffic light regulating the access from a farm to a main road. Dotted arrows show mutual interactions among the two state machine by direct event forwarding. Timeout (`tout`) events come from timer state machines (not shown).

the system behavior as a whole. The two approaches have been named respectively *implicit* and *explicit* system modeling [12]. The explicit approach will be discussed in Section 3.

Focusing on system components, a practice inspired by real-world observation and experience, may be misleading at the system level. A physical system is in fact assembled from a set of physical components, which exercise physical control one upon another. For example, a set of mutually related devices may globally exhibit a systemic behavior through direct physical interactions, which cause, in turn, state changes in related components. However, a different view is possible, since the global state changes resulting from a chain of causally induced state changes at the component level can be seen as a *single* state change at the system level. Consequently, a number of state transitions at the component level may be represented by a single state transition at the system level. In the same way, the global system behavior which implicitly results from direct interactions may be explicitly represented in the model.

Most programming and modeling paradigms are committed towards the implicit approach in modeling system behavior, since they mimic the physical interactions among components by direct event messages, as observed. Such a commitment towards the implicit modeling of systems has major drawbacks. For example mutually interacting processes lack clarity and understandability, since they have to embed synchronization details in the code, as shown in the two traffic light example of Figure 2. Resulting modular abstractions are therefore not self-contained and tightly-coupled [9]. Moreover, it may be the case that they have to embed behavioral details which pertain to the overall systemic behavior, as in the traffic light example where the farm traffic light has to introduce a wait state $w1$ which, in addition to its own state Y(ellow), order to model the switchover timing.

Consequently, the implicit global behavior is difficult to understand, modify, reuse, extend, and so on. State machines, in the Statecharts variant, lack moreover a definite and precise state semantics [1], that is, it is not possible to establish in advance if and when, and in which global state, interacting state machines stop.

3 Explicit modeling: Part-Whole Statecharts

In order to overcome the problem observed, we propose a model of concurrent autonomous systems where composition is restricted by a part-whole hierarchy: a System of Systems is modeled by a central controller system, referred to in the following of the paper as the "whole", which has one or more controlled system as its components, called "parts". The behavior of each system is specified by extended state machines through a special state-based language, Part-Whole Statecharts [11] which is able to represent, by a unique state diagram, the behavior of the system seen both as whole having other systems as parts, and as a system being part of other wholes. Such extended state machines are able to process both the different kinds of events (commands and notifications exchanged between the system acting as whole and the systems acting as components).

Although we do not report here the complete syntax of PWSs in this paper (the reader may refer to [11] for full details), we illustrate the main features of the approach by the following example.

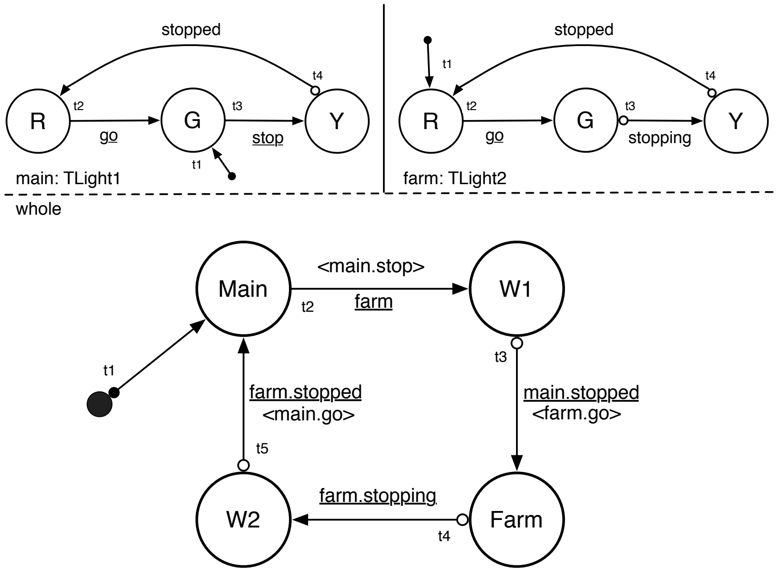

Figure 3: Explicit synchronization by Part-Whole Statecharts (adapted from [11]).

Example. The two synchronized traffic lights of the previous Section may be equivalently modeled by the Part-Whole Statechart of Figure 3 representing the crossroad system as a whole. A PWS is basically constituted by two communicating sections, the "whole" and the "assembly" sections, separated by the dotted horizontal line. The "whole" section hosts a state diagram which *explicitly* coordinates a set of state machines hosted in the "assembly" section and allows to view the semantics of interactions amongst participating systems at a glance. Any interaction among the state machines in the assembly section is forbidden, in order to force the designer to make explicit the synchronization semantics, which is indeed “shifted” to the whole section. State diagrams in the upper section of the PWS are called *state interfaces*. The lower section diagram embeds synchronization details which bind the whole to the components' interfaces. The whole section of the PWS may use only what is present in the such interfaces, that is states, event and triggers labeling transitions. State transition t_3 from G to Y of state machine main, for example, may be triggered by event stop: consequently, transition t_2 in the whole section will trigger such a transition by having the command event main.stop in its own command list. The “whole” state diagram becomes, in turn, the state interface of the modeled crossroad system which has the two original state machines as parts. State changes and event coming from the assembly trigger state transitions in the whole section of the PWS, which in turn send back action events to the state machines in the assembly. For example, transitions t_3 reacts to the notification event stopped coming from the components traffic light main through the trigger main.stopped, which triggers the transition and which sends, in turn, command event go to component farm through the action farm.go. It can be observed that, while the “whole” moves from state W1 to state Farm the assembly of state machines moves, accordingly, from the global state (Y,R) to the global state (R,G) thus furnishing a first base towards the computation of state semantics discussed in Section 3.1.

3.1 State semantics

It is possible to determine *at design time* the state configurations the set of system components, referred to int he rest of the paper as *assembly* of components, will assume when the control is in a given state of the state machine which controls the behavior of the compound system [11].

By *state configuration* it is meant a tuple $\pi = \langle q_1, q_2, \dots, q_N \rangle$, where N is the number of systems in the assembly of components, and $q_i \in Q_i$, the set of states of the i-th system in the assembly, with $i \in N$. Let each state configuration π denote trivially the *basic* proposition “the assembly of systems is in configuration π” about the global state of the assembly of systems.

A *state proposition s* is a disjunction of basic state propositions $s = \pi_1 \vee \pi_2 \vee \dots \vee \pi_k$ Alternately, a state proposition can be seen as a set of possible configurations of the assembly of system components, i.e. $s = \{\pi_1, \pi_2, \dots, \pi_k\}$.

Let A be one of the states of the state machine W which controls the behavior of the compound system. Let $\mathsf{sem}(A)$ denote a state proposition, called the *state semantics* of A. The state semantics of each state $S \in Q_W$ in the state machine W can be computed inductively by following the state diagram structure.

Let us suppose a state transition t links state A and B (as in Figure 4) and that, by the induction hypothesis, the state semantic of the starting state A of the transition is known. Let $l = \langle a_1, a_2, \dots, a_k \rangle$ be a list of action commands directed towards the N systems $c_1, c_2, \dots, c_N$ making the assembly. Each command action $a \in l$ is of the form $c_i.e$ meaning that system c_i has a state transition which can be triggered by event e. Finally, state proposition G within square brackets acts like a guard condition in ordinary Statecharts, that is it must hold in order for the transition to be taken.

Let us suppose the current configuration of the assembly of component systems be π_c when the current state of the controller is A. Then either $\pi_c \in G$ or not. Since, only in the former case, the transition is triggered, and since, by the inductive hypothesis $\mathsf{sem}(A)$ is known and holds of the current global state of the assembly, then π_c is such that transition t is triggered if and only if it belongs to the set of configurations $\mathsf{pre} = \mathsf{sem}(A) \cap G$.

Let $I(l)$ be the set of indexes such that $i \in I(l)$ iff $c_i.e \in l$. In case transition t is triggered, the action commands $c_i.e$ in l prescribe state transitions $q_i = \delta(q_i, e)$ in component c_i of the assembly, with $i \in I(l)$.

Let $\pi_c = \langle q_1, q_2, \ldots, q_N \rangle$ be the current state configuration. Then π_c is transformed into the tuple π_c' in such a way that each q_i in tuple π_c is replaced by q_i'. We denote the transformation induced by the command list l on the assembly configuration π_c by function transf, such that $\pi_c' = \mathsf{transf}(\pi_c, l)$. We define transf equivalently for sets of assembly configurations Π, meaning that $\Pi' = \mathsf{transf}(\Pi, l)$ iff for any $\pi \in \Pi$ we have that $\mathsf{transf}(\pi, l) \in \Pi'$.

Given a set of state configurations which hold when the system controller is in state A, the set of configurations which hold for the arrival state of the transition t is then given by:

$$\mathsf{post}(t) = \mathsf{transf}(\mathsf{sem}(A) \cap G, l) \tag{1}$$

which can be meant as the state semantics of state B induced by state transition t. Since state B may have different incoming transitions, its full semantics, that is the entire set of configurations the assembly may assume when the controller is in state B, is given by the "union" of the different incoming state transition semantics $\mathsf{post}(t)$:

$$\mathsf{sem}(B) = \bigcup_{t \in i(B)} \mathsf{post}(t) \tag{2}$$

where $i(B)$ denotes the set of state transitions which have B as arrival state. Finally, the base case on which the inductive hypothesis is grounded is that the semantics of the initial state of the whole is given by a configuration which contain the tuple of the initial states of each components in the assembly.

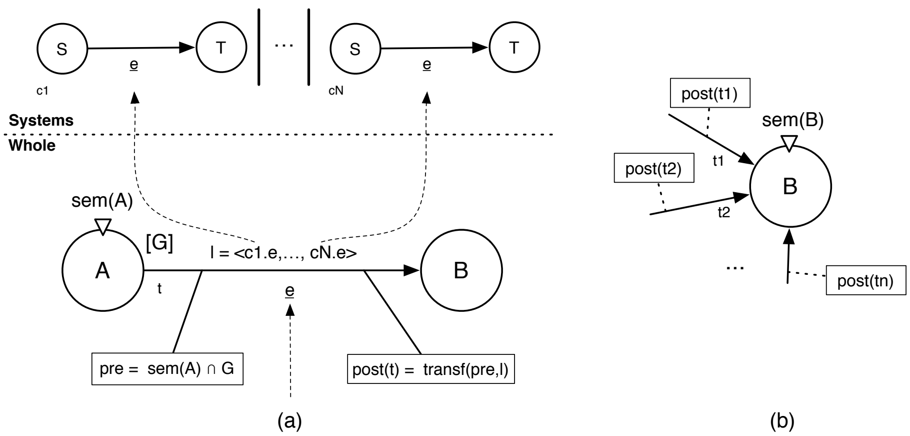

Figure 4: State semantics determination: basic state transition case. In (a) it is shown the how the set of allowed assembly configurations are determined for a single transition, in (b) it is shown how the full state semantics is determined for a state having N incoming state transitions.

3.2 Formal safety verification

Safety and liveness issues are raised by systems competing for a mutually exclusive resource. Each behavioral process associated with the system has to check whether the resource is free, and in case it is not, it has to ask the other processes to release it. On its turn, a process holding a resource should release it eventually. The two airplanes in Figure 1 compete for the same resource, that is the runaway. It may

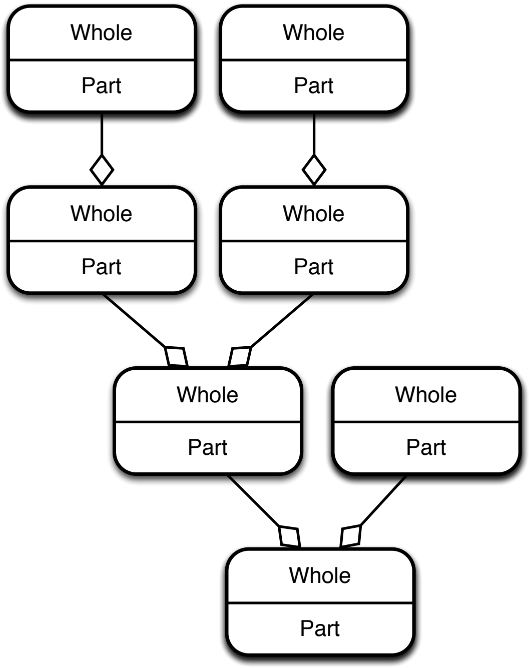

Figure 5: A holarchy, i.e. a part-whole hierarchy of holons.

be observed that the ATC task is to ensure that no two airplanes have the same runaway in use at the same time; it may be observed also that different airplanes may also communicate directly one with the another, as normally happens in small airfields where no central control is available.

The implicit, traditional approach, uses model checking techniques in order to explore all the feasible mutual behavior in order to check whether rules are satisfied, for example:

1. the system starts in the global state (G,R) such that only one process is in the critical section (i.e., only one road has access to the crossroad);
2. it is always guaranteed that it is never the case that both processes are in the critical section, (i.e, is the global state (G,G) is not reachable);
3. each process is guaranteed to release the critical section (i.e., each traffic lights moves from G to Y then R).

The main advantage in having an explicit representation of behavior is that the system will always be in a finite set of states *and in no other state*. Each state of the whole section can be put in correspondence with a set of allowed configurations of states of the components' assembly, as shown in Section 3.1. It is then possible to trivially visit the finite state diagram in the whole section in order to check whether safety and liveness rules are satisfied. In the crossroad example we have for example:

$$\mathsf{sem}(\mathsf{Main}) = \{(\mathsf{G},\mathsf{R})\} \tag{3}$$
$$\mathsf{sem}(\mathsf{W1}) = \{(\mathsf{Y},\mathsf{R})\} \tag{4}$$
$$\mathsf{sem}(\mathsf{Farm}) = \{(\mathsf{R},\mathsf{G})\} \tag{5}$$
$$\mathsf{sem}(\mathsf{W2}) = \{(\mathsf{R},\mathsf{Y})\} \tag{6}$$

Hence rules 1, 2 and 3 can be trivially verified to hold for the crossroad PWS. Due to the overall compositionality of the approach, the verification process is moreover incremental and fully compositional. Once the crossroad is *safe*, it can be composed into further systems having it as part, without the need to reconsider its internal safety.

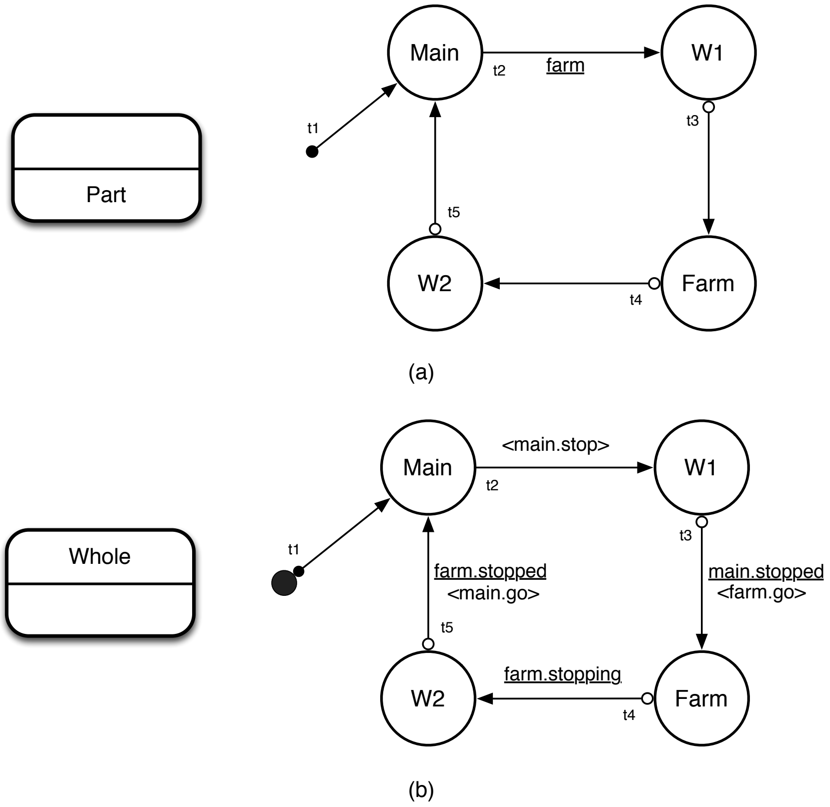

Figure 6: The interface section of an holon (a) can be obtained from the whole automaton (b) by stripping implementation details.

4 Conclusions

Interacting state machines synchronize system behavior by message exchange. Such messages, however, denote different kinds of information. Typically, systems communicate either by “peer to peer” or “part to whole” message exchange, the latter case pertaining to systems composed of other systems. The problem consists, at the ontological level, in determining whether two systems stand in the former or in the latter relationship. Statecharts, for example, do not distinguish amongst the two cases.

As observed, vertical, part-whole, system composition is asymmetrical in nature and preserves model reusability. On the other hand, horizontal, peer to peer message exchange hinders model reusability, since it forces system modelers to introduce exogenous details within systems being modeled, bringing severe limitations to the overall software quality of the modeled systems.

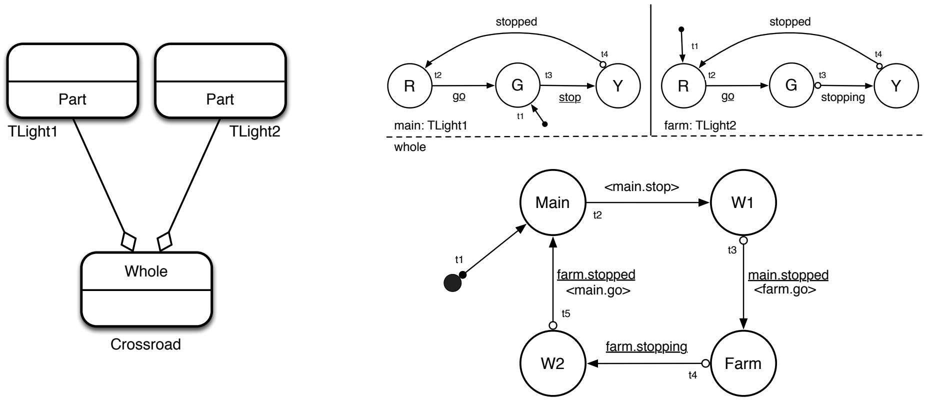

Figure 7: Basic holonic part-whole composition from a single PWS.

Physical interactions in physical systems denote in fact less evident conceptual structures, which host the overall interaction and synchronization knowledge among the component parts. By introducing additional system entities with the aim of hosting such knowledge in a localized and compact manner, we obtain a part-whole hierarchy of systems, called holarchy [8][4][2], as in Figure 5. Such systems are, at the same time, both parts and wholes within a holarchy, thus giving a formal characterization to the notion of Holon (Figure 6 and 7).

The paper presents an explicit approach for the recursive modeling of systems. The approach forces the modeler to expressing the behavior of composition by a single state machine, called whole. Such a state machine plays the double role of being both an executable specification of the behavior of the system, and to be an interface for further composition of the entire assembled system. This double side, "Janus"-like feature makes such kind of systems suitable for modeling, as observed, the behavior of Holons.

The explicit approach may be used in order to partition safety tasks into hierarchically arranged modules, each checked incrementally. Real-time critical systems, for example, may benefit from the approach since it allows to decompose a single, monolithic, control program into smaller, safe, reusable and composable systems. It is for example possible to defeat the overall complexity issues given by the concurrent modeling of operating modes and failure management policies. For example, fail silently sub-devices may be used as components for assembling a device behavior, which is able, at the higher level to *reduce* the fail silent behavior to a more tractable fail explicit behavior. The latter, in turn, may be used, at the next composition level, to obtain a fail safe or fail operational behavior. An example of such hierarchical arrangement of failure modes is given in [10].

References

[1] Michael Beeck (1994): *A comparison of Statecharts variants*. In Hans Langmaack, Willem-Paul Roever & Jan Vytopil, editors: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, *Lecture Notes in Computer Science* 863, Springer Berlin Heidelberg, pp. 128–148, doi:10.1007/3-540-58468-4_163.

[2] Massimo Cossentino, Stephane Galland, Nicolas Gaud, Vincent Hilaire & Abderrafiaa Koukam (2010): *An organisational approach to engineer emergence within holarchies*. *Int. J. Agent-Oriented Softw. Eng.* 4(3), pp. 304–329, doi:10.1504/IJAOSE.2010.036986.

[3] Bruce Powel Douglass (1998): *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley.

[4] Fengjuan Guo, An Zhang, Chang Li & Linghui Qi (2012): *Research on Formation Air-to-Ground Attack's C2 Holarchy and Mission Planning*. In: *Proceedings of the 2012 Second International Conference on Electric Information and Control Engineering - Volume 03*, ICEICE '12, IEEE Computer Society, Washington, DC, USA, pp. 629–632, doi:10.1109/ICEICE.2012.1043.

[5] Martin Hall-May & Tim Kelly (2006): *Using Agent-Based Modelling Approaches to Support the Development of Safety Policy for Systems of Systems*. In: *Computer Safety, Reliability, and Security*, *Lecture Notes in Computer Science* 4166, Springer Berlin Heidelberg, pp. 330–343, doi:10.1007/11875567_25.

[6] D. Harel & Gery E. (1994): *Executable Object Modeling with Statecharts*. In: *ICSE18*.

[7] David Harel (1987): *Statecharts: a visual formalism for complex systems*. *Science of Computer Programming* 8(3), pp. 231 – 274, doi:10.1016/0167-6423(87)90035-9. Available at `http://www.sciencedirect.com/science/article/pii/0167642387900359`.

[8] Fu-Shiung Hsieh (2008): *Holarchy formation and optimization in holonic manufacturing systems with contract net*. *Automatica* 44(4), pp. 959–970, doi:10.1016/j.automatica.2007.09.006.

[9] B. Meyer (1988): *Object-oriented Software Construction*. Prentice Hall.

[10] L. Pazzi & M. Pradelli (2010): *Using Part-Whole Statecharts for the safe modeling of clinical guidelines*. In: *Health Care Management (WHCM), 2010 IEEE Workshop on*, doi:10.1109/WHCM.2010.5441269.

[11] L. Pazzi & M. Pradelli (2012): *Modularity and Part-Whole Compositionality for Computing the State Semantics of Statecharts*. In: *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pp. 193 –203, doi:10.1109/ACSD.2012.22.

[12] Luca Pazzi (1999): *Implicit versus explicit characterization of complex entities and events*. *Data Knowl. Eng.* 31(2), pp. 115–134, doi:10.1016/S0169-023X(99)00020-8.