Control Theory Meets Software Engineering: The Holonic Perspective

Luca Pazzi
DIEF - University of Modena and Reggio Emilia
Via Pietro Vivarelli 10
Modena, Italy
luca.pazzi@unimore.it

ABSTRACT

One of the main challenges towards a software-based theory of control consists in finding an effective method for decomposing monolithic event-based interactive applications into modules. The task is challenging since this requires in turn to decompose both the invariants to be maintained as well as the main control loop. We present a formalisms for gathering portion of behaviour by special units, called holons, which are both parts and wholes and which can be arranged into part-whole taxonomies. Each holon hosts a state machine and embodies different invariants which give semantics to its states. Control is achieved by both taking autonomously internal actions by the state machine in order to maintain such state invariants, as well as by having the the state machine move from one invariant to another by actions driven by external events. Such an approach requires to introduce non trivial solutions in order to allow communication among such modules, mainly by implementing control loops among couple of holons. The proposed model consists essentially in shaping each module in order to be both a controller and a controllable entity. Each module may control a definite number of modules and is controlled by a single module. Control is exercised by discrete events which travel through a communication medium. Control actions as well as feedback events travel thus from a module to the another, thus achieving local control loops which, taken globally, decompose the main control loop.

Categories and Subject Descriptors

H.1 [**MODELS AND PRINCIPLES**]: General

Keywords

Holons, Part-Whole Statecharts, Compositional verification

1. INTRODUCTION

Time-dependent real-time software systems deal with fast changing environments, to which they adapt in order to maintain a specific and often complex operation task. Changing environments are heavily dynamical, and their changing behaviour is often revealed by events which reach the software system [12]. Software systems react to external incoming events by producing, in turn, other outgoing events towards the environment, thus closing the loop and becoming *de facto* control systems (Figure 1).

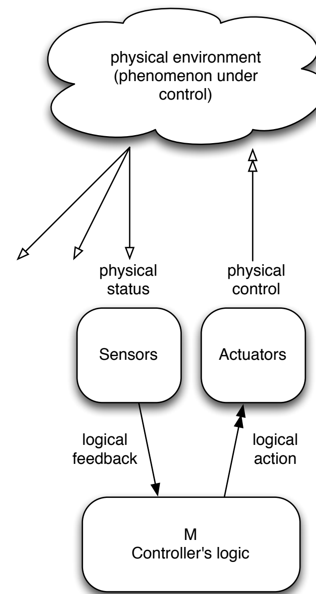

Figure 1: Asymmetry in the control process at first glance. A generic controllable phenomenon does not know its controller M, while the controller is designed specifically for dealing with the phenomenon through actuators and sensors.

Software systems are required to exhibit however other features, mainly effectiveness, robustness, and dependability: when dealing closely with real world, such features have to ensure, in first place, liveness and safety properties. Albeit a control software may be even conceived as a monolithic block modelling a single global behaviour, a really effective software development methodology requires to modularise the overall control software in order to defeat the overall complexity. This requires in turn a coherent criterion for decomposing, as well as composing, the control loop. Though control engineers practiced such a decomposition for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804343

many years, a deeper understanding of the basic software engineering notion of module structure and connectivity would improve both control theory and software engineering. If a modelling paradigm is more efficient then it is presumably more natural, i.e., ontologically sound. By the current view, systems are seen "as transmitting and transforming signals from the input channel to the output channel, and interconnections are viewed as pathways through which outputs of one system are imposed as inputs to another system" [13]. Such a view of systems seen as signal transformers is naively accepted by both software and control engineering communities, but reveals its limitation especially when modelling interconnected systems under a software engineering perspective, typically raising problems in terms of understandability, reusability and maintainability. Even worse, once systems are interconnected by direct links, model checking becomes infeasible due to the exponentially growing complexity of the resulting system. A paradigm shift is therefore needed in order to ensure effective modularity. The paper describes an ongoing research [6][8] in modular decompositions of behaviour of reactive and autonomic systems [7]. In particular this paper will focus on the hierarchic control of distributed invariants and can be seen a continuation of [6]. We believe that the application to control theory of the holonic approach may shed new light and bring further ideas to the original research in software engineering.

As in [9] we are interested in the basic control problem of ensuring that a logical predicate remains invariantly true when it is initially satisfied. By the supervisory control approach [10] the problem bears to state space explosion that is exponential in the number of system components. It becomes thus necessary to explore modular system architectures with the property that the overall complexity can be appropriately defeated. We devise a state-based tree-like structure of modules as in [5]: our composition model and general motivations are similar also to those in [2], but we believe our model is conceptually cleaner and simpler.

Our approach is mainly focused on restricting communication among modules and on labelling states by invariant state propositions as described in detail in [8]. State transitions laid among states have to comply with both arrival and starting state propositions. In other words, each module is "correct by construction" in the sense that it is guaranteed that when control is in a state a given proposition on the states of its components is always satisfied. More complex predicates can be existentially or universally verified by exploring the state diagrams within the modules.

We start by examining (Section 2) how events reveal structural aspects of the domain, and as such may be used in order to shape modelling constructs among which they flow. Events are not only the basic elements of communication in control, they reveal different aspects of entities participating in the scenario being controlled. The way events flow to and from them may in fact help conceiving a method for decomposing control around the participating entities and for shaping control structures. Such control structures may on their turn be centred around behavioural constructs, finite state automata, which indeed prescribe control by parsing and generating event flows. Additionally, since the time of the Fusion methodology [1], finite state automata provide a very effective tool for specifying software behaviour in the development process [11] and notably they can be easily transformed into code or directly executed by a suitable interpreter. Holon inspired control modules are therefore shaped around such behavioural constructs. They provide a connection framework in which events flow from one state machine to another by following a decomposition hierarchy. Section 3 introduces finally holons as control units by a working example adapted from [6] used in order to show the feasibility of the approach.

2. FROM EVENTS TO MODULES

Events taken separately are not meaningful: their sequences instead reveal complex phenomena happening in the real world. Such sequences are part of a language built upon them denoting indeed real-world phenomena. For example it can be easily observed that "*door-open*, *door-close*, *light-green*, *engine-on*" is a string of events belonging to the language describing the interaction of an automated car and a traffic light. Such a car is additionally provided with doors for embarking and disembarking passengers along a track: additionally, it stops and restarts its engine according to signals from traffic lights on its route. Doors are automatically opened when the car has to embark/disembark passengers and closed when it restarts. Doors are locked when the car moves, but the locking mechanism may be bypassed and the doors may be opened at any time, in which case the car has to stop immediately. By the schema of Figure 2 it may be hypothesised that a controller takes care of the mutual interaction of the doors and of the engine.

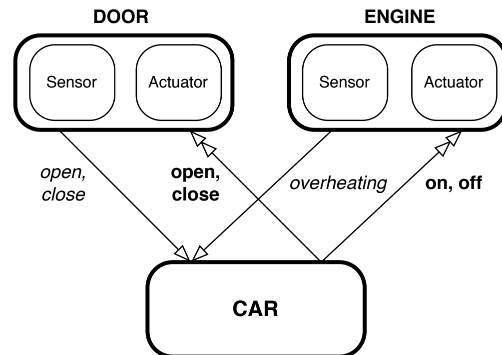

Figure 2: Module Car parses and generates the joint language which denotes the synchronization of its controlled modules. Events may be distinguished between input events (bold) and output events (italic) and travel, respectively, along double and single arrow lines.

Entities belonging to the environment produce a language built from a vocabulary of words, i.e. strings of events coming from their respective event alphabet. Modelling some sort of interaction means restricting the full set of sequences that can be built from the alphabet of events of the participating entities. Consider for example the interaction of the doors of the automated car and its engine. Once the engine and the doors do not interact, that is when they are taken physically apart, any sequence of events from their joint vocabulary may be observed. We call it the *free* alphabet of the doors and the engine, for example "*door-open*, *engine-off*" and "*door-open*, *engine-on*" are both words belonging to their joint free vocabulary. In order to model a useful

behaviour, the second word has to be banned, since a simple safety constraint does not allow the vehicle to move with doors open.

In order to shape a useful joint language, that is a useful behaviour, we therefore insert a module which is able to observe and to prescribe events to both the involved entities. Module Car of Figure 2, for example, is the place where any aspect regarding the mutual interaction of the car's components, doors and engine, should be modelled. It may be observed that there are now two distinct loops, each carrying information from module Car to and from module Door and Engine. An event can be either directed from the controller towards a component (double arrow lines), in this case driving its actuator, or being emitted by the sensor of the component towards the controller (single arrow lines). In the former case the event denotes an action that the component has to undertake, in the latter the event denotes a spontaneous action that already happened within the component.

Module Car may be conceived as hosting an automaton which acts both as parser and generator of the joint language of the two components. The parsing and generating mechanism may be thought as being implemented by transitions of such automaton which are triggered by events belonging to the sublanguage being parsed while, at the same time, forwarding events belonging to the sublanguage being generated (Figure 3).

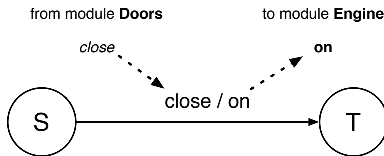

Figure 3: Module Car parses and generates the joint language of modules Doors and Engine by a traditional Statecharts diagram, whose transitions are labelled by events belonging to the modules in the form of triggers and forwarded events

2.1 From Modules to Holons

The car *taken as a whole* interacts with the traffic lights signals, producing its own set of events, for example *start*, *stop* and *fail* which in turn form meaningful words once interleaved with the alphabet of the traffic light, for example "*light-green*, *car-start*" and "*light-red*, *car-stop*".

Such events denote the *external behaviour* of the car, that is the behaviour of the car *taken as a whole* observable from outside. Conversely, the joint behaviour of the components of the car (the doors and the engine in the example) will be referred to as the *internal behaviour* of the car. Both behaviours have to agree, and therefore the related languages have to be parsed and generated accordingly. It is therefore necessary to endow the automaton within the module of additional modelling capabilities, in order to have the two behaviours match in a meaningful way. In other words, the languages of the car components have to agree with the language of the car taken as a whole.

This requires a more elaborated syntax and semantics than traditional Statecharts. Part-Whole Statecharts [8] allow to label a state transition with specific constructs in order to account for *internal* and *external* events. Two different transition typologies result, as shown in Figure 4. Synchronisation amongst the internal and the external language is achieved by allowing both internal and external events to be present in the same state transition with specific operational meanings, which underly two different reactive mechanisms. For example, the transition of Figure 4 (a) models an externally triggered behaviour. The car, seen as a whole, is required to start by receiving event go from an external module, and the doors and the engine are as a result required of being, respectively, closed and turned on. Figure 4 (b) models instead an internally triggered behaviour. In that case, the happening of an event within a component (the opening of a door while moving) requires the engine of being stopped and the car, seen as a whole, to emit an undirected fail event towards an unspecified client module.

The need to take into account, at the same time and by a single behavioural construct, an internal and an external language, requires to introduce a new modular construct, called *holon*, presented in the next Section.

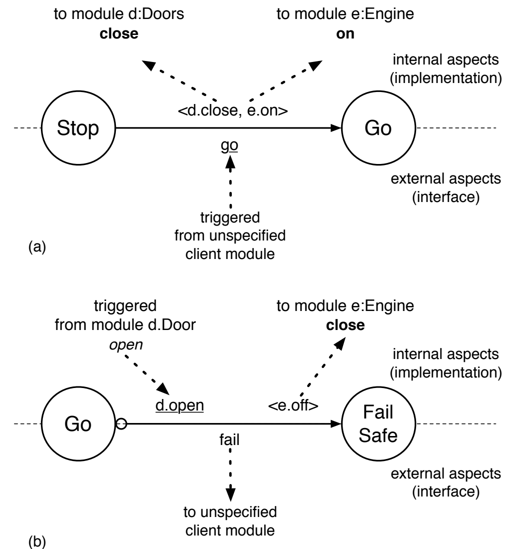

Figure 4: Part-Whole Statecharts is a state-based formalism which is able, amongst other features, to integrate internal and external behavioural aspects. Two main reactive patterns are feasible, that is externally (a) and internally (b) triggered transitions. Events are syntactically distinguished into directed and undirected ones.

3. THE HOLONIC FRAMEWORK

In the preceding sections we argued for constructs which exhibit a "double behavioural nature", that is they should be able to account both for an internal language as well as an external language. What we need are therefore modular units which are able to conform to such a double nature. Each module needs therefore *four* ports, two on the internal and two on the external behavioural side. Such ports are meant to be behavioural connection points to other modules. Modules should communicate one with the other by connecting the internal ports of the module acting as component to the external ports of the module acting as compound

entity. Each module should finally expose only its external features, hiding the internal ones.

Holons, by Arthur Koestler [3][4], have a double, "Janus face", nature, thus being able to host both an implementation and an interface as in Figure 5-(a), referred to as internal and external aspects in the previous sections. The interface allows to view and use the module as "part", the implementation allows instead to view the module as "whole", that to coordinate a number of other holons as parts.

The interest of the holonic approach in the context of a novel software-engineering based theory of control consists indeed in the feasibility of composing holons through partial control loops. Each holon playing the role of whole coordinates n different holons playing the role of parts through n control loops. Holons within a holarchy cooperate in order to achieve a global task by exchanging control events of different typologies which travel along the lattice of control loops established among them.

The ontological rationale behind such a choice is that, as shown in Figure 1, a monolithic control software (the controller) acts upon the environment, aimed at changing its current set of properties, called collectively state. As the state of the environment changes, events are generated (for example temperature changes) and broadcast towards the controller. Vice versa, the controller prescribes directed actions to actuators, for example turning a furnace on in order to raise the temperature of the room. In other words, this marks an asymmetry since the controller knows the controlled entity, but not vice versa, in the sense that trivially the temperature of the room only implicitly and indirectly acts on the temperature controller. The proposed framework mimics such an ontological property of controlled processes by having controllers the capability to observe controlled entities, not vice versa.

This paves the road for an effective software development method, where modular reusability is of primary concern and underlies the general principle of *asymmetry* in control. Each modular artifact is in fact totally reusable since it has to be designed in full generality without having to know the module to which it has to be composed.

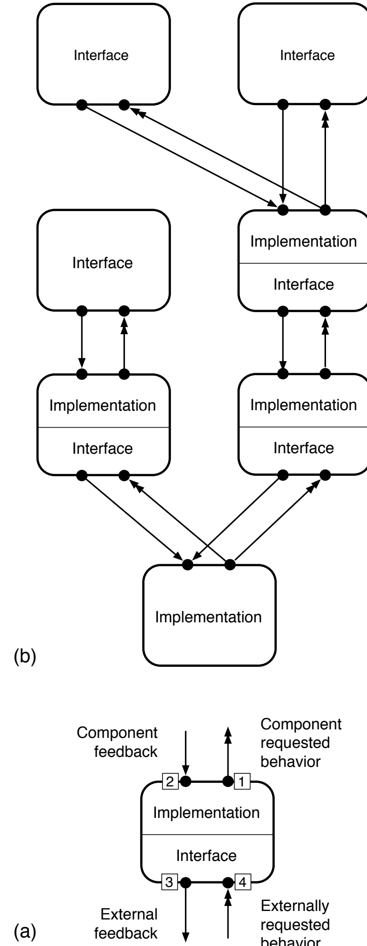

Figure 5: (a) Holon modules present four logical ports and are split into an implementation and an interface part; (b) By connecting iteratively port 2 with port 3 and port 1 with port 4 it is possible to obtain hierarchical structures called "holarchies", consisting of nodes connected by partial control loops.

3.1 Invariant-Based Holonic Control

Monolithic control applications fulfill different tasks, each possibly consisting in maintaining different logical invariants.

Holons have an asymmetrical and complementary nature by which they can be composed into *holarchies*, as in Figure 5-(b). Root modules can be seen as bare implementations, since they do not need to be further composed in the context of a control application. Leaf modules are simple interfaces, since they are the logical view of sensors and actuators. A holarchy can be thus defined as a set of holon modules connected through control loops by which events flow upwards and downwards. Events can be classified into different typologies depending on the direction in which they travel and to the target to which they are directed. The framework presented is aimed at maintaining equivalently control of such state invariants dispersed amongst separate component holons within a hierarchy. Within a specific module of the holarchy such invariants become state invariants of the PW-Statechart governing it. Internal and external aspects of the PW-Statechart hosted by a single holon match the communication ports, by following the schema depicted in Figure 5-(a).

Koestler conceived indeed holons as self-regulatory units [4]. Each holon node is designed for maintaining a limited set of invariants which are not visible to its components. Modules Doors and Engine, in the example, do not *know* that their current joint state must fulfill a limited set of state invariants. Such invariants are encoded in the holon which has them as components. Only module Car indeed *knows* for example that the opening of the door requires the engine to stop.

A holarchy may be seen therefore as a directed acyclic graph of self-regulatory nodes, each node maintaining its own invariants by perturbing its components and being perturbed by other nodes having it as component. A state invariant can be seen indeed as a "configuration" of components.

Module Car in Figure 6 has three invariants associated to its states; state Stop for example is such that when the control is in the state then doors are opened and the engine is stopped (C_1), when in state Go doors are closed and the

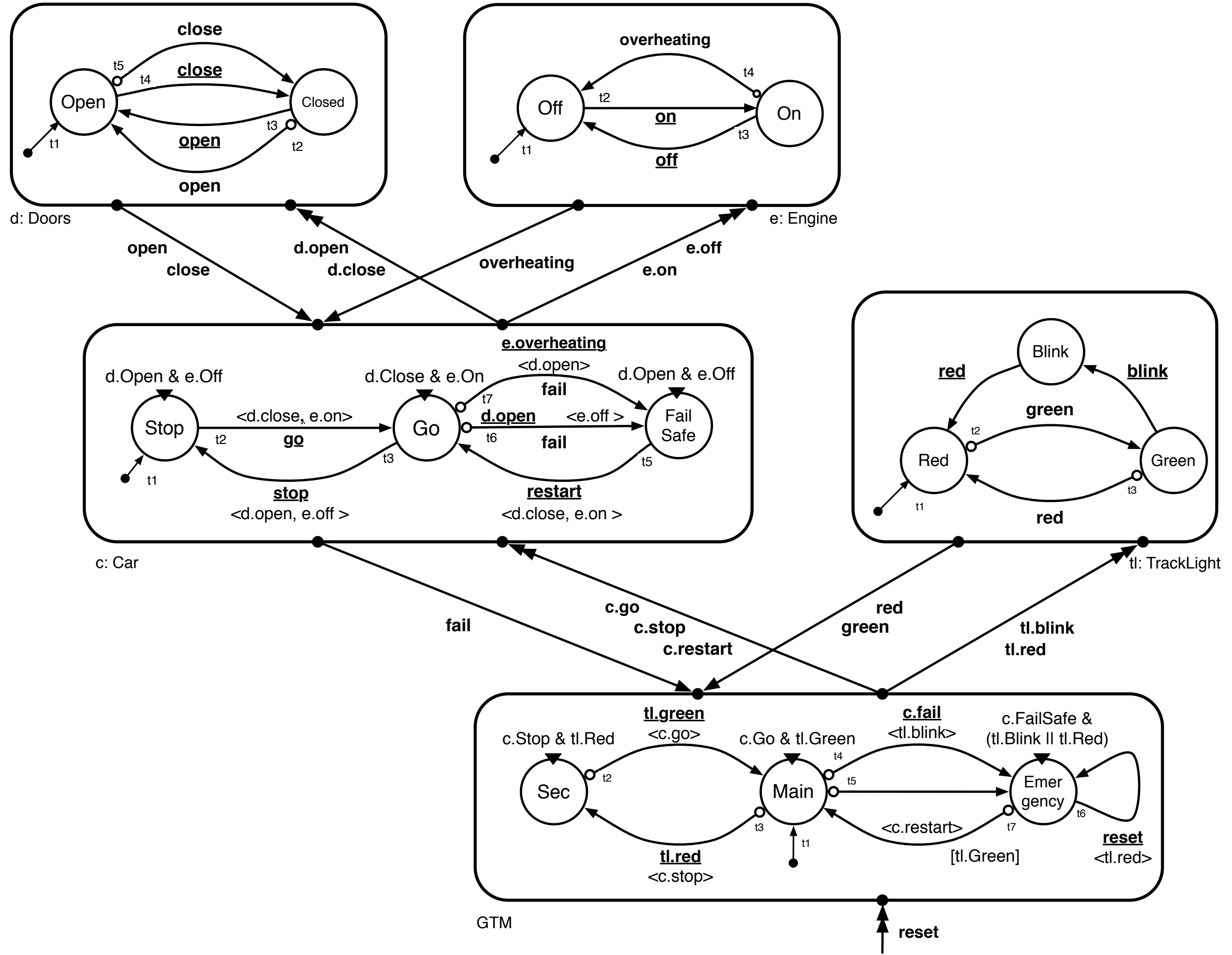

Figure 6: The holarchy implemented by PW-Statecharts modelling the behaviour of the holon Car by its two component holons (Doors and Engine) and of a GTM (Global Track Monitor) holon having Car and TrackLight as components (adapted from [6]).

engine is running (C_2). Finally, when in state FailSafe doors are opened and the engine is stopped (C_3). Observe that the same state invariant may be associated to different states, for example Stop and FailSafe share C_1 and C_3. The holon Car moves along the three state invariants above. Each state invariant may be invalidated by both an external as well as an internal stimulus. External stimuli change the current state and the associated invariant (Figure 7-(a)). Internal stimuli come from autonomous state changes happening within components. Invariants need then to be restored by sending commands to components (Figure 7-(b)).

3.2 Partitioning Control Invariants

A holonic application maintains global state invariants by the recursive composition of control over partial invariants. Each partial invariant is such that it logically constrains the states belonging to the component holons. For example, when the control is in state Main in module GTM (Global Track Monitor) of Figure 6, then invariant condition $C_2^{\mathsf{GTM}} = \mathsf{c.Go} \wedge \mathsf{tl.Green}$ holds, meaning that module car is moving and traffic light is green.

Control consists in comparing, at each clock step, whether condition C_2^{GTM} holds compared to the global state of its components. In this way, only components at the immediate upper level have to be checked in order to verify whether the desired invariant holds. In case a difference is noticed between C_2^{GTM} and the current state of the components, for example by the traffic light changing to red, either

1. current state invariant has to be restored by the global track monitor GTM by sending a command to its *immediate upper level components* Car and TrackLight; this is not possible since the GTM would request the light to switch back to green but the requested traffic light behaviour is modelled through autonomous *non triggerable* transition t_3; or

2. the GTM module has to switch to a different state invariant which complies with the current global state of the *immediate upper level components*, in the example by moving from state Main to state Sec through transition t_3 in module GTM.

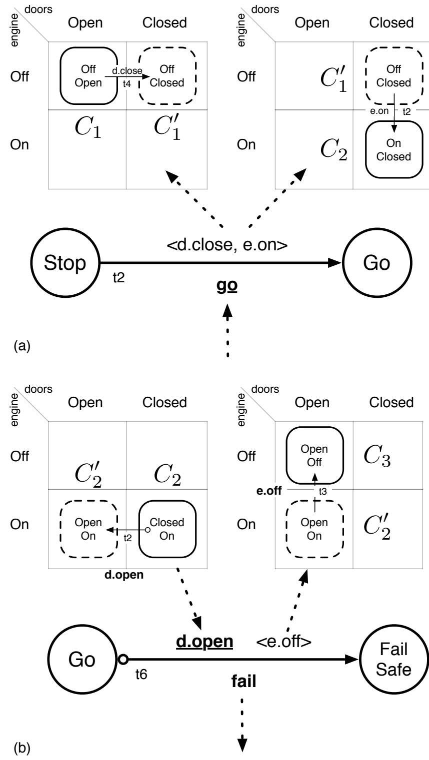

Figure 7: (a) An external stimulus (event go) make the holon change its current state from $C_1 = \mathsf{d.Open} \wedge \mathsf{e.Off}$ to $C_2 = \mathsf{d.Closed} \wedge \mathsf{e.On}$ by forwarding additional event stimuli towards its components, which modify their configuration. (b) An internal stimulus (a door is opened manually by event d.open) makes invariant $C_2 = \mathsf{d.Closed} \wedge \mathsf{e.On}$ invalid and triggers a change of state to the state having associated condition $C_3 = \mathsf{d.Open} \wedge \mathsf{e.Off}$, which becomes the new state invariant. The holon then emits event fail towards the external composition context.

In both cases the only current global state which has to be looked up or modified is the one resulting form the current state of the components at the immediate upper level. Control is thus exercised only by establishing a control loop consisting of a *single level* of composition. By such schema of control, commands travel upward while feedback travels downward and is used to establish, incrementally, the current state of the components. Feedback from components may implicitly trigger state transition, that is, feedback consists in information regarding their current state, not the transitions which have to be triggered in the client holon. For example in Figure 6 a change of state in the doors is revealed by feeding back the event d.open which denotes indeed the change of state which happened. Holon Car reacts to such an event by triggering a state transition since the current state invariant no longer holds (in fact, doors must be closed when car is running). Other client holons (that is holons *using* the doors holon as component) may not need to react to doors opening since they implement a different application policy (for example doors may be both opened or closed while the car is stopped). In other words, the same change of state in a component is dealt with in different ways, not foreseeable at design time.

Observe finally that since:

$\mathsf{c.Go} \implies (\mathsf{d.Close} \wedge \mathsf{e.On})$

We have that

$\mathsf{c.Go} \wedge \mathsf{tl.Green} \implies \mathsf{d.Close} \wedge \mathsf{e.On} \wedge \mathsf{tl.Green}$

In other words each state invariant denotes one or more configuration of leaf holons, as depicted in Figure 8. For example C_3^{GTM} is equal to $C_3^{\mathsf{Car}} \wedge (\mathsf{tl.Red} \vee \mathsf{tl.Blink})$ which in turn is equal to $\mathsf{d.Open} \wedge \mathsf{e.Off} \wedge (\mathsf{tl.Red} \vee \mathsf{tl.Blink})$

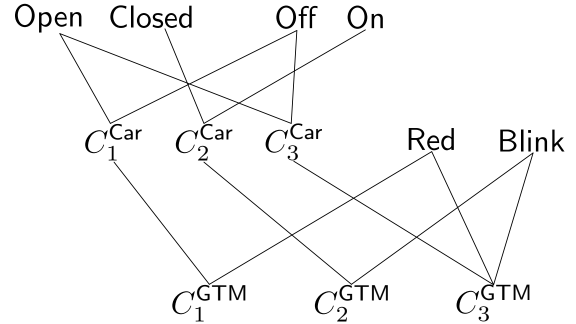

Figure 8: Each invariant in the Car as well as the GTM (Global Track Module) denotes one or more configurations of leaf (i.e. basic, unstructured) holons.

3.3 Discussion and Further Research

Further research is needed in order to answer some noticeable questions. For example: does the holonic perspective provide some guidance for decomposing the control loop as well as the overall behaviour? Which are the general requirements imposed by the holonic perspective? Which are its limits?

Holons do not provide any criterion for decomposing the control loop, rather they support the decomposition or partitioning of control by simply hosting portions of behaviour which is strictly necessary in order to implement the reactive behaviour among a finite and limite number of components. Events are generated as part of such a working behaviour and are therefore dependent from it. In other words there is no criteria for decomposing the main loop, each more complex behavioural level is simply built upon previous less complex levels. Each level emits events towards both more and less complex layers. Event loops simply connect the different layers and give rise to unforeseeable patterns of behaviour, as in Figure 9. Further research is needed in order to understand whether non terminating sequences of events may be occasionally generated.

The overall behaviour of a monolithic control application is devoted at maintaining a number of invariants. Each holon acts as a self regulatory node in order to maintain only some invariants. Once a mutual dependence is detected among two holon entities, a third holon having such entities as "parts" is required in order to host the mutual behaviour and the mutual invariants. Invariants within a holon hence constrain only to its component parts. In the example of Figure 6 holon subsystem Car is devoted at maintaining state invariants which deal with the mutual interaction of its component holons, i.e. doors and engine. Invariants are modelled through state propositions. Each state proposition is a log-

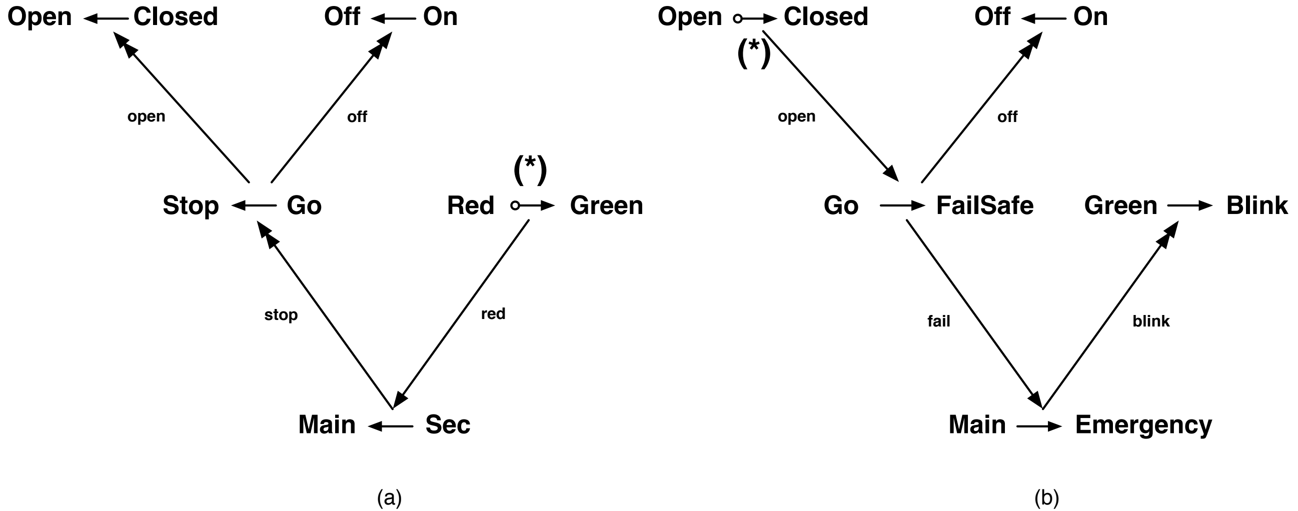

Figure 9: Two execution traces from the example of Figure 6. Observe that, by the proposed approach, events propagate both upwards and downwards. Propagation origin is marked by (*).

ical formula which denotes the current state of the current upper less complex level. At the same time each state of such level is in turn constrained by another formula. As shown in Figure 8 global invariants are maintained by maintaining simpler invariants at each level. Further research is finally needed in order to understand whether any system is decomposable through a holarchy.

4. CONCLUSIONS

Distinguishing between internal and external events requires to adopt new behavioural constructs. Such constructs may in turn be hosted by a suitable module, which provides communication facilities by which modules are connected one with the other. Internal aspects of the module implement the activity of the system by coordinating its components, external aspects model instead the activity of the system seen as a whole. Connecting internal aspects of a module to external aspects of a component module allows to implement a partitioned form of control by establishing control loops which decompose the main loop. Each control loop is simply devoted at controlling its immediate composition level, that is at controlling holons which in turn controls their own component holons, and so on.

One of more invariants coexist within a holon state-based behaviour. Such invariants are state propositions associated to the states of the state machine. Once such propositions are invalidated, regulating events are sent to its components in order to restore them. In case no restoring action is possible, the state machine performs a transition to another state whose invariant is compatible with the internal changes, making them self-regulatory units as in Koestler's vision. Modules can be moreover verified directly at design time by visiting their state invariants along their finite and limited state diagram and composed at any time within a holarchy without having to check them again.

The more appealing benefit of adopting the holonic perspective is therefore related to the capability of reducing the overall complexity by modular units, together with the feasibility of composing off-the-shelf verified modules.

5. REFERENCES

[1] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: the Fusion Method.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[2] G. Delaval, S. Mak-Karé Gueye, E. Rutten, and N. De Palma. Modular Coordination of Multiple Autonomic Managers. In *17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2014)*, page 291, Lille, France, June 2014.

[3] A. Koestler. Beyond atomism and holism - the concept of the holon. *Perspectives in Biology and Medicine*, 13(2):131–154, 1970.

[4] A. Koestler and J. R. Smythies. *Beyond reductionism; new perspectives in the life sciences.* Macmillan New York, 1970.

[5] C. Ma and W. Wonham. Nonblocking supervisory control of state tree structures. *Automatic Control, IEEE Transactions on*, 51(5):782–793, May 2006.

[6] L. Pazzi. Modeling systemic behavior by state-based holonic modular units. In J. Dingel, W. Schulte, I. Ramos, S. AbrahŃo, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 99–115. Springer International Publishing, 2014.

[7] L. Pazzi and M. Pradelli. Part-whole hierarchical modularization of fault-tolerant and goal-based autonomic systems. In *Dependable Control of Discrete Systems, 2009. DCDS '09. 2nd IFAC Symposium on*, pages 175–180, 2009.

[8] L. Pazzi and M. Pradelli. 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*, pages 193 –203, june 2012.

[9] P. J. Ramadge and W. M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal on*

Control and Optimization, 25(5):1202–1218, 1987.

[10] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[12] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[13] J. C. Willems. The behavioral approach to open and interconnected systems. *Control Systems Magazine*, pages 46–99, 2007.