Concurrent Processes Synchronisation in Statecharts for FPGA implementation

Grzegorz Łabiak and Marian Adamski
Computer Engineering & Electronics Department,
University of Zielona Góra,
Podgórna 50, 65-246 Zielona Góra, Poland.
E-mails: {G.Labiak, M.Adamski}@iie.uz.zgora.pl

Abstract— **In this paper useful methods of concurrent processes synchronization in UML state machine diagrams are presented. It is not easy to transform complex behaviour description into statecharts, even if the formal specification has already been given, for example as a hierarchical Petri net. Both graphical forms of specifications can be used simultaneously, especially when overlapped concurrent processes are considered. The authors specify the behaviour of an industrial chemical controller as a case study to demonstrate the way of transforming the verbal specification through a formal Petri net model into the UML format, which is more and more frequently understood and widely accepted by industry. The diversification of description gives a chance that the obtained design result, which can be seen both from theoretical and practical points of view, will be correct. The specification is considered from a reconfigurable hardware implementation side and a digital design methodology. It is based on hierarchical concurrent state machine implementation through state encoding into mapping into FPGA matrices, supported by hardware description languages, for example VHDL. In the paper some effective process synchronization methods are included: by introducing global variables and using UML graphical synch states accepted by the UML standard. The proposed methodology is fully supported by system HiCoS, developed at the University of Zielona Góra for a rapid prototyping of reconfigurable logic controllers.**

I. Introduction

The reconfigurable hardware implementation of logic controllers is based on a formal precise description of concurrent and hierarchical tasks. Some well known graphical notations are popular among scientists, but unfortunately they are less accepted by industry designers. A new generation of engineers is educated in software engineering and more and more of them frequently use UML [9]. Starting from Harel's statecharts [4], which are considered as a basis for the UML state machine diagram, it is possible to design logic controller as well as from behavioral Petri net description [2], [3], [7], [10].

It should be noted that the UML state machine description is not so precise and easy to be mapped into array logics, in comparison with hierarchical and modular Petri net form of behavioral specification. On the other hand, it is possible to keep the Petri net template style during the creation of nearly equivalent statechart diagram. Two different but related views on the same model can be helpful in keeping mathematical precision of Petri net together with self-evident clarity of hierarchical and concurrent state machine described in UML style.

The purpose of the paper is to demonstrate the applicability of two diverse approaches for preparing one model for digital controller design purpose included into one design methodology. Both considered models can be compared and co-simulated in one environment, developed at the University of Zielona Góra [2] [8]. System PeNCAD [2] makes it possible to transform Petri net graphical specification automatically through VHDL into compact FPGA matrix structures. System HiCoS [5] [6] after hierarchical state assignment, automatically transforms textual form of statecharts onto BDDs, which are also directly translated into VHDL statements on Register Transfer Level for a comfortable timing simulation and rapid prototyping in FPGA.

The paper is organized as follows. Section II presents a case study – the verbal behavioral description of a chemical reactor. Section III informally defines the initial behavioral model of logic controller and clarifies some properties of the Petri net and UML state machine. Section IV shortly presents the three related mutations of one behavioral UML state machine specification, demonstrating some useful ways of coordinating concurrent processes. Finally, section V gives a brief conclusion. Complete statechar diagrams illustrating synchronization techniques are contained in the appendix.

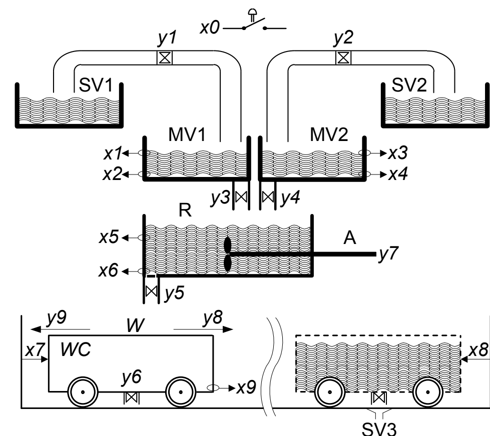

Fig. 1. Industrial chemical reactor – Schematic diagram

II. CHEMICAL REACTOR – CASE STUDY

The industrial reactor, for the first time presented in [1] is a part of hydraulic-mechanical plant, whose functioning is governed by a discrete controller (Fig. 1). The reactor measures out two substances, mixes them together and pours the product into the wagon which transports the outcome to its destination station.

The detailed working of the reactor is as follows. Initially the reacting substances are kept in containers SV1 and SV2. The emptied wagon waits in its initial position on the right. After the signal *x0* a technological cycle starts: valves *y1* and *y2* are opened and scales MV1 and MV2 are poured in, and wagon starts moving to the left (signal *y9*). The pouring of the substrates lasts until sensors *x1* and *x3* in the scales indicate exceeding upper limits. After both sensors indicate exceeding upper limits valves *y3* and *y4* emptying scales MV1 and MV2 become on simultaneously and the main reaction process starts. The agitator A is ready to switch on. After the substance in reactor main container R is over the sensor *x5*, the agitator becomes on. When the substance is beneath again, the agitator becomes off and ready to start again. While emptying scales and pouring main container R, the wagon is moving to its position on the left. Next, when scales MV1 and MV2 are emptied (which is indicated by sensors *x2* and *x4*) and in the meantime the wagon has reached its left position (sensor *x7*), the main container valve *y5* is opened and the container is emptied till the level of substance drops below the reading of the sensor *x6*. Next the wagon starts moving right (signal *y8*). When the wagon reaches the far right position, (sensor *x9*) wagon is emptied (valve *y6*). The end of emptying is signaled by *x9* sensor, and after that the technological cycle is finished and the whole plant is ready to start again.

III. PETRI NET MODEL

Verbal specification, such as given in the previous section, is usually the starting point in designing the digital controller and often can be imprecise and incomplete. Therefore, to formalize the requirements set by the customer, the designer needs more a rigid and strict notation. As far as concurrent processes are concerned, Petri nets formalism is historically first and thoroughly researched formal model which also have a good graphical appeal.

Fig. 2 presents the Petri net model of the discrete controller. In this diagram circles (called places) correspond to local states of the system and solid bars correspond to transitions between them. It is only allowed to connect places with transitions and transitions with places (connections place-place and transition-transition are forbidden). Places can have assigned activities. It means that when a given place is active (formally posses a token – in the diagram depicted as a small black solid circle) the signal assigned to it is asserted. For example, places *1* and *2* have assigned signals *y1* and *y2* and this means that when the places are active valves *2* and *3* in reactor (Fig. 1) are opened. On the other hand transitions can have assigned Boolean predicates. The transition can fire, if both predicate is true and every input places have a token, causing tokens to be removed from the input places and put in the output places. For example transition *t11* will fire if places *8* and *14* individually have a token at the same time and signal *x6=0* (predicate *!x6*). Then tokens are removed from places *8* and *14* and and one token is put in place *15* which becomes active. In terms of reactor operations this translates into a situation when, before the system has changed, the agitator is switched off (place *8* is active), valve *y6* (place *14*) is still opened and the reactor container is emptied (i.e. level of substance is below of sensor *x6*) and after the change: valve *y6* is turned off and wagon starts moving right. Starting global state in the Petri net is a set of places which contains tokens (one token per one place). In the diagram the starting state is place *1*.

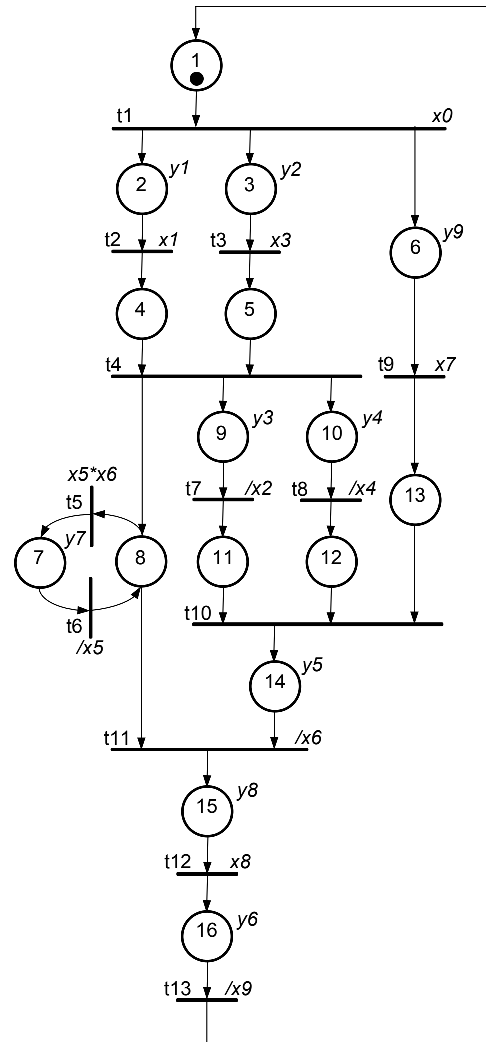

Fig. 2. Industrial reactor – Petri net model

Making a Petri net diagram the designer can freely connect, by means of directed arcs, places with transitions and transitions with places. This freedom allows him to describe very complicated behaviour, and which makes the diagram susceptible to becoming excessively complex and this in turn can leads to faults like traps and deadlocks or be unsafe. The

lack of safeness property, traps and deadlocks are difficult to spot by the designer, hence Petri net models need to be farther formally analyzed by means of very sophisticated methods.

To improve the clarity of the Petri net diagram it is good to put behavior into hierarchy of modules, i.e. to combine primitive behaviors into bigger modules and to combine next these modules into yet bigger ones. Thanks to this abstractions the reactor does not need to pay attention to every detail but can focus instead on the larger structures of activities. The Petri nets formalism in its classical version doesn't support any kind of hierarchy. Some extensions of the Petri net model offer structural hierarchy [2], [10], which consist in grouping places and transitions into macros (called, respectively, macroplaces and macrotransitions). But this type of hierarchy doesn't allow to remove an activity from the module in an easy way, forcing the designer to use tangled arcs and transitions instead. In this regard the statechart diagrams offer truly behavioral hierarchy. But full modularity sometimes prevents from modeling some essential nuisances of behavior – not every behavior can be modeled fully modularly. Then minor changes in original behavior can be investigated. In general, changing original behavior is not a good idea, but sometimes it may be acceptable, especially taking into account the modularity benefits for verification, and clarity of the diagram.

IV. Statechart diagrams

Statechart diagrams offers a truly modular behavioral hierarchy. The statechart diagrams [4] have been devised in order to improve the specification of reactive systems of complex behavior. It is the state-based graphical notation which enhances the traditional finite automata with concurrency, hierarchy and broadcast mechanism. States are connected with arcs with predicates. A complex state can be assigned a group of states (simply or complex), thus creating hierarchy relationships. States can be in a concurrency relationship. An activity can be removed from subordinated states in the exception style through firing transition from their ancestor. The presence of the final state (in the diagram bull's eye) prevents exception transitions, unless the final state is active.

Modularity is a strong weapon in the fight against unclear complex behaviour [4], but it has some drawbacks. In the case of reactor (see Fig. 2) it is very interesting to model some of its component sub-behaviors, namely the agitator control (places *7* and *8*), agents dispensing to main reactor container (consisting of one place *14* and two completely parallel processes: places *9*, *11* and *10*, *12*) and wagon motion to the left (places *6* and *13*). These three concurrent processes neither start nor finish at the same time. They are started with transition *t1* and *t4* and finished with *t10* and *t11*. This means that they overlap and must be synchronized with these four transitions. In the simplest form behaviour of this kind is presented in Fig. 3. Place *4* will become active after place *1* has lost its activity.

Fully modular statechart diagram is a diagram whose compound component behaviors are only defined by its subcomponents. This generally means, that the transitions crossing the state borders are forbidden and broadcast events are also forbidden. The benefits of full modularity are clarity of the diagram and a more efficient verification. This stems from the fact that the dynamic properties of the complex behavior can be obtained from already verified properties of its component sub-behaviors. It turns out that the behavior presented in Fig. 2 can not be modeled in a fully modular way, since the processes are overlapped. To model this behavior same means of inter-process synchronization must be applied. Following subsections present two synchronization method: with a global variable and with a synchronization pseudo state.

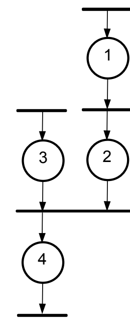

Fig. 3. Petri net simplest case of overlapping synchronized processes

A. Statechart diagram synchronized with global variable

A global variable is a physical implementation of a broadcast event. In the case of a digital controller this is a regular boolean variable. In theory [9] [4] an event is a noteworthy occurrence that may trigger a state transition in any part of the diagram. Events may come from outside world, be a result of state transitions or may be generated on entering or exiting state, or during state activity (respective keywords in diagrams: *entry*, *exit*, *do*; e.g. see Fig. 6, states *WagonLeft* or *WagonWaiting*). Events have global scope and as such impede compositional semantics; through this statechart diagrams lose clarity.

Global variables can be used for synchronizing mutually interacting processes. Fig. 3 presents Petri net model of two simple overlapped processes and Fig. 4 presents corresponding statechart diagrams. Again, state *4* will become active after state *1* has lost its activity. This results from the fact that on transition from state *3* to state *4* predicate *v* is imposed, which will be fulfilled when event *v* is broadcast. This takes place

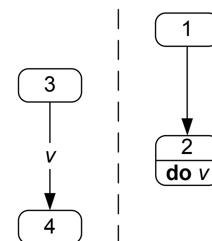

Fig. 4. Simplest statechart diagram with overlapping processes synchronized with global variable

when state *2* is activated. If this happens event *v* is broadcasted as long as state *2* stays active (keyword *do*).

Fig. 6 depicts statechart diagram version whose behavior, in principle, covers behavior modeled by Petri nets (Fig. 2). In this case process *AgentDispensing* doesn't wait for the moment when the wagon reaches its left destination but starts independently instead. Again, the wagon can move to the left while both scales MV1 and MV2 are being emptied and valve (*y5*) of main container won't be on before the wagon has reached its left position. This is achieved through global variable *z1*. State *EmptyingWagon* becomes active after predicate *z1* imposed on transition *t10* is fulfilled, and this will take place after state *WagonWaiting* (switching on valve *y5*) is activated and then event *z1* is being broadcasted.

It is noteworthy that transitions *t9* and *t11*, seemingly in conflict, will never be in conflict although their predicates are not orthogonal [5] [6]. This stems from the fact that these transitions will never be enabled simultaneously. This is achieved through proper synchronization by means of global variable *z1*. Moreover, no pair of transitions from this diagram will ever be in conflict, which was proved symbolically in the author's computer program called *HiCoS* [5] [6] [8]. Computer analysis showed that the system set of global states comprises 32 states.

B. Statechart diagram synchronized with synch states

Synch states provide a means of synchronizing the course of two or more concurrent processes called regions [9]. In diagrams a synch state is put on the border between synchronized regions on the dashed line (see Fig. 7 and 5). A synch state has an incoming arc from a fork transition from the source region, and an outgoing arc to a join transition in the target region. It is used to ensure that one region leaves a particular state before another region can enter a particular state. For example in the diagram from Fig. 5 state *1* will lose its activity before state *4* gets activity. This is equivalent behaviour to the Petri net from Fig. 3.

Fig. 7 presents a diagram which corresponds to the Petri net from Fig. 2. In this case the functioning of the reactor is the same as in statechart diagram from Fig. 6, but instead of the global variable, synchronization has been expressed by means of synch states. Emptying of the main container will start (state *EmptyingReactor*) after the wagon reached its far left position (state *WagonWating*). This is realized by fork *t9*, synch state and join transition *t10*. The motion of the wagon to the right will start after main container is emptied (*x6=0*, transition *t6*, predicate *!x6*). This is realized by by fork transition *t6*, synch state and join transition *t12*.

On the one hand the application of the synch state slightly disturbs the compositional semantics. However, on the other hand, in comparison with the diagram which has a global variable, it makes diagram appear modular and clear.

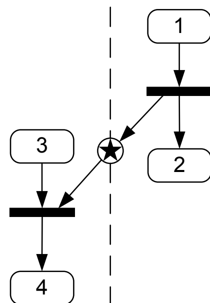

Fig. 5. Simplest statechart diagram with overlapping processes synchronized with synch state

V. Conclusion

The proper behaviour of logic controller can be viewed from two perspectives: the Petri net and hierarchical and concurrent statemachine diagrams. The UML statemachine diagram (statechart) can be so close to the Petri net model that these two descriptions can be related in one VHDL design environment for simulation and implementation purposes. However, it is not self-evident how to combine in UML concurrent overlapping processes (regions) which are easily tied in the Petri net by common transitions. The paper gives simple solutions to this problem. The details are demonstrated by means of a non trivial case study. UML statemachine (statechart) textual description can be automatically mapped into FPGA internal structure: considered logic controllers in the version with global variable consumes 23 flip-flops and 60 4-input LUTs in Virtex device (xcv50-5bg256).

References

[1] M. Adamski, *Parallel Controller Implementation using standard PLD Software*, ed.ed. W. R. Moore and W. Luk, FPGAs, Abingdon EE&CS Books, Abingdon, England, 1991 ss.296-304

[2] M. Adamski, M. Węgrzyn, A. Węgrzyn, *Safe reconfigurable logic controllers design*, Measurements models systems and design, ed. by J. Korbicz, in Wydawnictwa Komunikacji i Łączności, Warsaw, Poland 2007, pp. 343–370, ISBN: 978-83-206-1644-6

[3] L. Gomes, A. Costa, *Petri nets as supporting formalism within Embedded Systems Co-design*, IEEE Symposium on Industrial Embedded Systems, Antibes Juan-Les-Pins, France, 2006, pp. 1-4

[4] D. Harel, *Statecharts: A visual formalism for complex Systems*, Science of Computer Programming, Vol.8, 1987, pp. 231-274

[5] HiCoS Homepage, http://www.uz.zgora.pl/ glabiak/

[6] G. Łabiak, *From UML statecharts to FPGA - the HiCoS approach*, Forum on Spec. & Design Langs - FDL '03, Frankfurt 2003,

[7] Ricardo Jorge Machado, João Miguel Fernandes, Alberto José Proenča, *Specification of Industrial Digital Controllers with Object-Oriented Petri Nets*, Proceeding of the IEEE International Symposium on Industrial Electronics - ISIE'97, Guimarães, Portugal 1997, pp. 78-82

[8] *Design of Embedded Control Systems*, ed.ed. M. A. Adamski, A. Karatkevich and M. Węgrzyn, Springer 2005, pp. 73-83

[9] UML, *Unified Modeling Language Specification. Version 1.4.2*, ISO/IEC 19501, 2005

[10] Marek Węgrzyn, Paweł Wolański, Marian Adamski, *Coloured Petri Net Model of Application Specific Logic Controller Programs*, Proceeding of the IEEE International Symposium on Industrial Electronics - ISIE'97, Guimarães, Portugal 1997, pp. 158-163

VI. Appendix

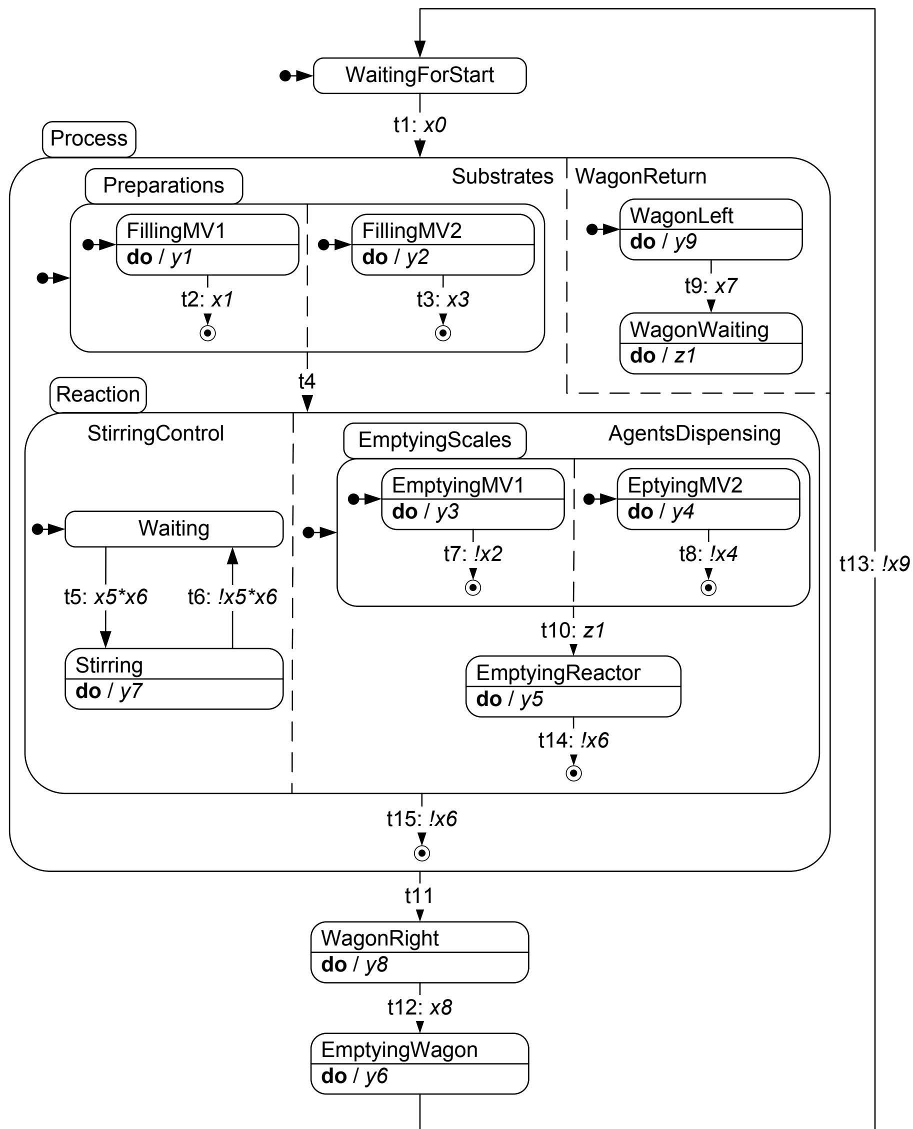

Fig. 6. Statechart diagram synchronized with global variable

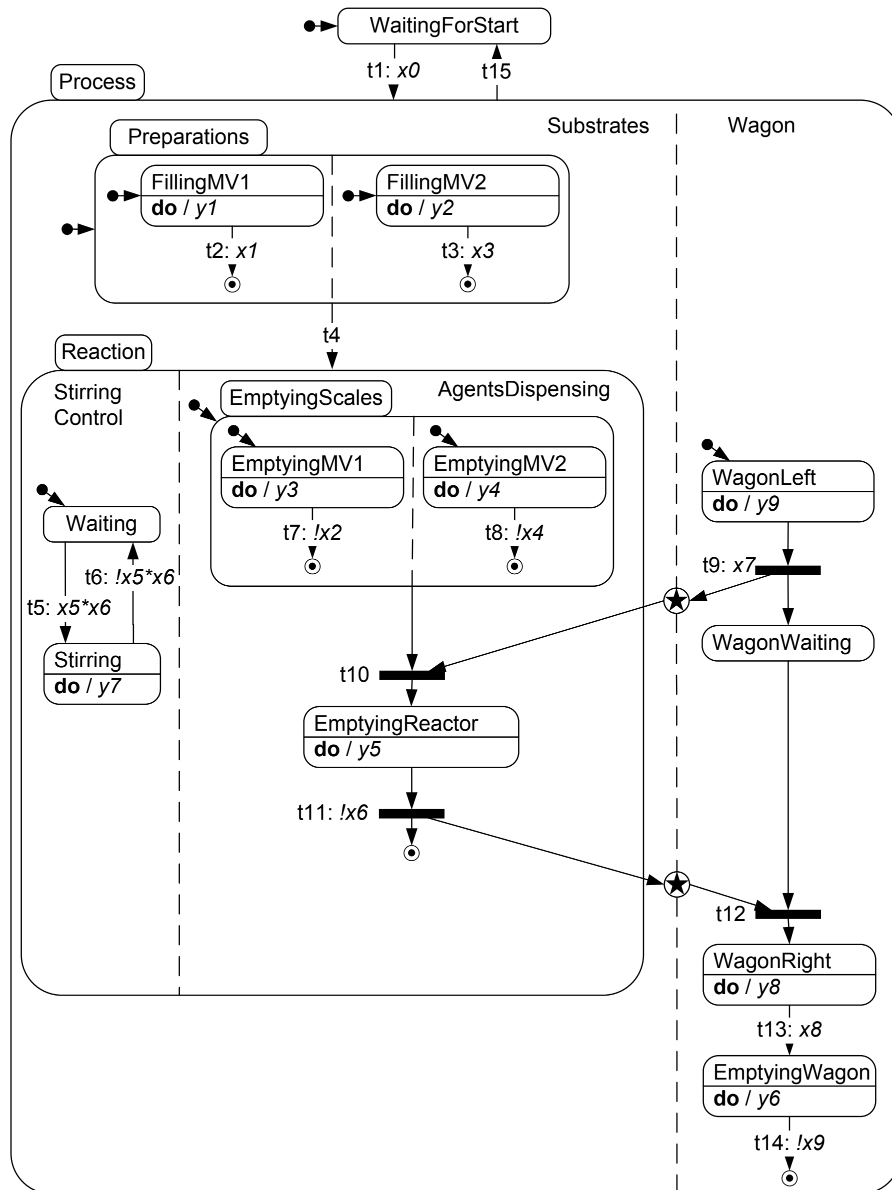

Fig. 7. Statechart diagram synchronized with synch states