A state-based systemic view of behaviour for safe medical computer applications

Luca Pazzi, Marco Pradelli
University of Modena and Reggio Emilia
Department of Engineering Sciences
Via Vignolese 905, I-41100 Modena, Italy
luca.pazzi@unimore.it

Abstract

The paper addresses safety issues involved in making ad hoc interconnections among medical devices in order to assemble more complex medical systems. The main problem is that the systemic view may be easily concealed by nowadays behavioral modeling tools. Missing such a systemic view does not allow to have a precise view of what is being modeled: we propose instead to adopt novel methodological guidelines in developing assembled medical systems, basically by showing how a a clear and unambiguous semantics may be given for any state of the system being modeled, from specification to test phases. Such a state semantics may then be checked against safety axioms by simply visiting the state diagram without the need of resorting to model checking techniques.

Recent directions in the development of computer systems for medical application [3] show a growing interest towards networking medical devices having embedded computers. The availability of mature interconnecting technologies, typically distributed object middleware, allows indeed a great flexibility in interconnecting control and sensing devices. The result is a system of interconnected medical devices, which may exercise control toward other devices in the network and which are controllable, on their turn, by human operators.

Such a new generation of distributed medical systems presents numerous opportunities as well as challenges. Opportunities are given by the reduced cost of systems, as they should benefit from off-the-shelf components and software. Challenges are given by the development, certification and medical practice-driven models for the high confidence medical software resulting from such a heterogeneous system integration, which has to keep in consideration, in addition, the presence of human operators in the loop. In the medical case, the difficulties inherent system integration are further worsened, since such systems are often assembled in order to support life-critical applications and, in any case, may endanger patient life.

In order to assemble a global behavior by a network of medical devices, the control software on each machine must be aware of the current status of the other machines in the network. Moreover, such software must be able, typically through network signals or messages, to act on the other machines in addition to the machine on which it resides, in order to achieve some global result. For example, the speed of an injecting pump may be raised or lowered due to the sensed status of the patient, alarm devices may be activated and devices may be hot-swapped in case of malfunctioning, in order to increase the reliability of the system. An additional level of complexity is finally given by the fact that machines, due to their internal logic, may refuse to acknowledge an action request coming from another machine.

By a closer analysis, dealing with such an unconstrained architecture of computer based medical devices raises two kinds of distinct, albeit related, problems. On one hand, software related problems are raised by the need of employing software which is not behaviorally self-contained, that is, which depends on the behavior of other machines. In other words, software becomes difficult to specify, design, test, reuse and finally certify, given the need of considering its behavior in the enlarged context drafted above. On the other hand, operational problems are raised, since it is difficult to predict which global status will be reached by the networked system, thus jeopardizing the overall safety and liveness of the assembled system.

1 System Modeling by ordinary Statecharts

The Statecharts state-based formalism [1] is currently used for system behavioral modelling in most of current software development methods, since Rumbaugh's Object Modeling Tool (OMT) [7]. A statechart diagram typically consists of state diagrams, hosted into different interacting parallel sections, which run concurrently and have to synchronize in order to achieve a global, meaningful, systemic behaviour. Modeling a system of interacting devices is typi-

cally achieved by representing the behavior of each component device by a state diagram hosted within one of the parallel sections. Statecharts synchronization primitives, like *event broadcasting* and *mutual condition testing*, have to be embedded into component state diagrams in order to obtain a systemic representation, which is therefore directly incorporated within the interacting system components.

For example consider a simple system constituted by an infusion pump, an antireflux valve and a patient-pressure monitor. The designer wants to enforce a global behavior such that the valve has to be opened before the pump starts and to be closed after it stops: it is further required that the pressure be always under a certain threshold. Such a system may be modelled by Statecharts as shown in Figure 1, where the pump section, upon arrival of start and stop commands propagates, respectively, open and close commands to the valve, which in turn may not take one of the commanded transitions in case pressure is high. In case the pressure level goes up when the pump is working and the valve is open, a stop command is sent from the pressure sensor to the pump and a start command is sent again when low pressure condition is restored.

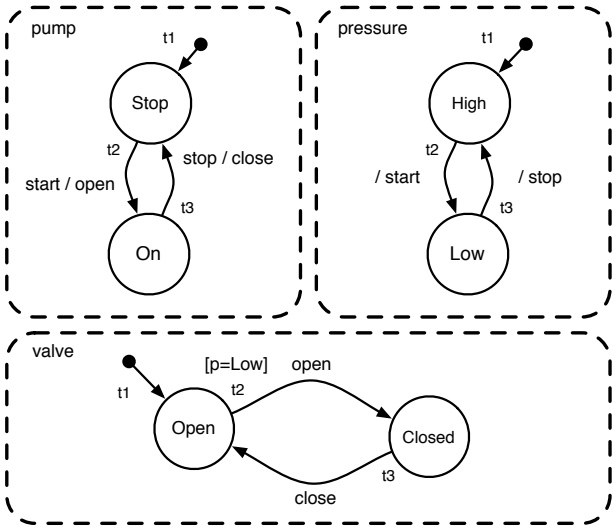

Figure 1. Behavioral modelling of a system made of three components, a pump, a valve and a pressure sensor each hosted within a parallel statecharts section.

Statecharts control models suffer from different drawbacks when modelling a system behavior out of a set of components:

1. the component behavior is barely reusable, understandable as well as manutenable, due to the fact that behavioral references make it tightly bound to the other abstractions. In Figure 1, for example, the pump behavior has effect on the valve behavior, which in turn depends on the pressure sensor behavior, which in turn has again effect on the pump behavior.

2. the system behavior is difficult to understand, since it is difficult for the designer to have a complete view of the whole system behavior scattered into the different component sections. As an example, a deeper analysis of the behavior of Figure 1 reveals details that may not be evident at first glance, in particular (1) the pump may be started even if the valve is not opened and (2) the start command sent from the pressure sensor to the pump may activate the pump at any time, not only if the pump was stopped due to a pressure surge.

3. the system behavior may either deadlock or not terminate. This is due to cross referencing mutual conditions as well as to infinite, circular successions of state transitions and command broadcasting.

It is evident that, if Point 1 impacts on the reusability of the software being built around such abstractions, Points 2 and 3 impact not only on the reusability of the whole behavior, but mainly on safety issues, since it is difficult, at design time, to know in which state a system will be found at a specific time, and therefore it is not possible to assess safety constraint against behavior.

Safety issues in medical systems are nowadays addressed by model checking techniques [2], that is by exhaustively checking all the possible reachable system configurations, in order to verify that the model is deadlock free and that basic modelling assumptions, such as “the pump is off and the valve is closed when pressure is high”, are verified. Although recent model checking techniques alleviate the computational burden [8], model checking is however not feasible for all models and requires the designer to manage complex temporal logic formulae without, as remarked above, having a complete view of the behavior being modeled.

2 System Modeling by PW Statecharts

Networks of interacting components can be also represented by means of a different state based formalism, named Part-Whole Statecharts [4][5] (shortened either as PW Statecharts or PWS), which was created with the aim of allowing an explicit representation of the interaction among the behavior of parallel state based sections. A revised version of the formalism is presented in this paper, complemented by a constraint based specification method [6].

A PWS consists of two main sections, one hosting a set of component state machines, referred collectively to as the

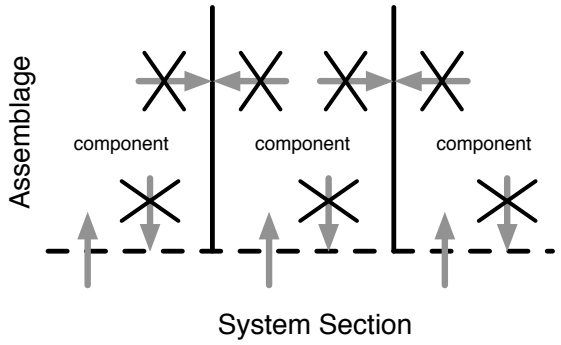

Figure 2. PWS control flow. Gray arrows indicate whether or not control and knowledge is possible among parallel behavioral sections.

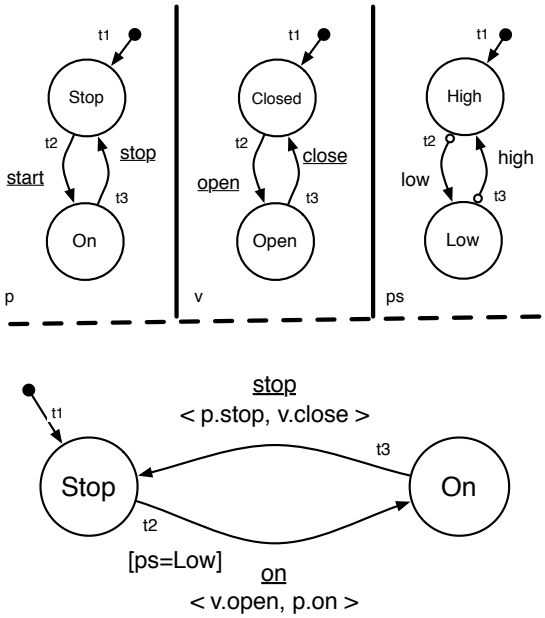

Figure 3. Explicit modelling of a system made of three components, a pump, a valve and a pressure sensor. The state machine under the dashed line is called "system" and depicts the behavior of the whole system.

assemblage, the another a single state machine representing the system behavior, called the *system state machine*. As shown in Figure 2, control as well as any mutual knowledge of current state and behavior is not allowed among component state machines; conversely, the system section state machine is allowed to know, at each time, the current state of each component state machine as well as to send control commands to them. The system machine is finally notified of each state transition happening within the component set.

In order to suggest that neither communication nor knowledge is allowed between component state machines a bold line is drawn among them; conversely, a dashed line separates the components from the system state machine, in order to suggest that the system state machine is able to communicate with the components.

Such communication and knowledge restrictions ensure that behavioral component description are self-contained, since they are not allowed to refer to any of the peer components or to the system state machine. As a side effect, the whole semantics of coordination and communication is gathered within the system state machine.

The example of Figure 1 has been reworked in the PWS of Figure 3, by removing both event forwarding as well as condition testing amongst the state machines in the parallel sections. In other words, components are deprived of the capability of carrying out control towards other components as well as to have their behavior determined by knowing the behavior of other components. We obtain thus components which are self-contained, thus achieving reusability among other software engineering advantages.

The whole behavior of the system is represented explicitly in the system section of Figure 3: it consists of two states, meaning that the whole "pumping system" may be either in a stopped state, named Stop, or in a working state, named On. A revised design according to the methodological guidelines of Section 3, is presented in Figure 4. Both Figure 3 and 4 show also *implementation features* associated with the state transitions, which consist of:

1. a *guard*, consisting of a boolean valued expression enclosed in square brackets about the global state of the assemblage, which will be defined a *state proposition*. For example, transition t_2 is guarded by the state proposition "the pressure is low", while transition t_3 does not have any guard, hence it may be thought of as being guarded by the constant expression which denotes the boolean value true;
2. a *trigger*, that is a symbol (written underlined in the diagram) which denotes that the transition will be activated upon the receipt of:
   (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 labeled by e within the component c of the assemblage, in which case either c.e or c.t is named *internal* trigger (t being the transition name).

For example, transition t_2 and t_3 in Figure 3 may be triggered, respectively, by the external events on and stop, while transition t_4 and t_6 in Figure 4 may be triggered, respectively, by the internal events ps.low and ps.high;

3. a *list of actions*, that is symbols which are used in order to request the activation of the state transitions in the assemblage components. We write c.e in order to require the activation of the triggerable transition labeled by event e in the component c. For example, transition t_2 in Figure 3 has the associated list of action $\langle \mathsf{v.open}, \mathsf{p.on} \rangle$, meaning that, when t_2 is activated, the transition labeled by open has to be activated within component v and the transition labeled by on has to be activated within component p.

Finally, an *automatic* transition, denoted by an hollow dot at the beginning of the transition arrow in the state diagram, is either a transition having no external trigger or no triggers at all: in the latter case it is activated as soon as possible. For example, the pressure sensor may take autonomously state transitions t_3 and t_2 (Figure 3) and the system state diagram state transitions t_4 and t_6 (Figure 4).

3 Modelling safety explicitly

Amongst the different advantages which are feasible with the explicit modeling of the behaviour of a complex system, we show how state propositions may be used to specify, as well as to enforce, the whole system behavior. In other words, when the PWS is within one of the states of the system state machine section, a correspondent state proposition is guaranteed to be *always* verified. It follows that the behavior of the system can be checked against other state propositions, named *safety axioms*, denoting system wide specification of safe behavior, by examining a finite number of states, namely the ones within the system diagram.

For example, consider the two states in the system section of Figure 3. We may say that we want that, (1) when the system is in state Stop, "the pump must be *off* and the valve *closed*"; (2) when the system is in state On, "the pump must be *on* and the valve must be *open* and the pressure level must be *low*". Given such an assignment of state propositions to the system state machine, it can be easily verified that a safety axiom of the kind "when the pressure level is *low* the pump must be *stopped* and the valve must be *closed*" logically follows from the state propositions associated to the states of the system section and hence it is *always* verified.

3.1 Definitions

The set $\mathsf{A} = \{c_1, \ldots, c_N\}$ of component devices is called *assemblage*. $\mathbf{Q}_\mathsf{A} = \{Q_{c_1} \times \ldots \times Q_{c_N}\}$ is the set of global states of the assemblage, where Q_{c_i} is the set of states of the assemblage component c_i. A state proposition P is a function from the powerset built from the set of global states $\mathbf{Q}_\mathsf{A}$ of the assemblage to a boolean value, in symbols $P : 2^{\mathbf{Q}_\mathsf{A}} \rightarrow \{true, false\}$. A state propositions of the form "component c is in state S", with $S \in Q_c$, is said a *basic state proposition* and is written $\mathsf{S^c}$. Composite state propositions can be formed starting from basic ones by means of ordinary logical operators $\odot$, $\oplus$, $\neg$ and form a boolean algebra, hence it exists a partial ordering among state propositions denoted by $\preceq$, which can be read as "is implied by". We say that p_1 is a *subproposition* of p_2 when $p_1 \preceq p_2$. We say that p_1 is equivalent to p_2, written $p_1 \equiv p_2$, when both $p_1 \preceq p_2$ and $p_2 \preceq p_1$ hold.

For example, the proposition "the pump is *off* and the valve is *closed*" may be either true or false depending on the global state $\mathbf{q} \in \mathbf{Q}_\mathsf{A}$ of the assemblage $\mathsf{A} = \{p, v, ps\}$ of Figure 3. It can be easily verified that the global state of the assemblage $\mathbf{q}_1 = (\mathsf{Off}, \mathsf{Closed}, \mathsf{High})$ makes proposition P true, while $\mathbf{q}_2 = (\mathsf{Off}, \mathsf{Open}, \mathsf{High})$ makes it false. Other global states of the assemblage make proposition P true as well, such as $\mathbf{q}_3 = (\mathsf{Off}, \mathsf{Closed}, \mathsf{Low})$.

3.2 State safety

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

Definition 1 (State safety invariant) *When S is the current state of the system section, the current state $\mathbf{q}$ of the assemblage satisfies $\mathsf{C}(S)$.*

In order to have the state invariant above satisfied, the system has to be specified in a consistent manner. By a closer analysis, it can be observed that the requirement of Definition 2 may be broken either by the system section moving to state S with the assemblage global state moving, at the same time, to a state $\dot{\mathbf{q}}$ which does not satisfy the constraint $\mathsf{C}(S)$, or by an uncontrollable *internal* transition happening in the assemblage when the system section is in state S, resulting in a global state $\dot{\mathbf{q}}$ of the assemblage which does not satisfy its constraint $\mathsf{C}(S)$.

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 in Sections 3.2.1 and 3.2.2.

3.2.1 Incoming state transition correctness

It can be observed that any state S of the system section may be reached only through a number of incoming state

transitions, belonging to the set $T_{\mathsf{inc}}(S)$. It can be further shown that, for any transition $t \in T_{\mathsf{inc}}(S)$, a state proposition $\mathsf{post}(t)$ can be effectively computed (as shown in Section 3.2.3), such that the assemblage is in state $\mathbf{q}$ satisfying $\mathsf{post}(t)$ when the transition has been executed. In order to have the state invariant of Definition 2 *always satisfied* for state S it then suffices to check that

$$\mathsf{post}(t) \preceq \mathsf{C}(S) \tag{1}$$

holds for any state transition $t \in T_{\mathsf{inc}}(S)$.

3.2.2 Outgoing state transition correctness

It is possible to identify (Section 3.2.4) a set of pairs (c, t), named *exit zones*, where c is a subproposition of $\mathsf{C}(S)$ and t is a transition of the assemblage, such that, when the assemblage is in a global state state $\mathbf{q}$ satisfying c and transition t happens, state $\mathbf{q}$ is transformed into $\dot{\mathbf{q}}$ which does not satisfy $\mathsf{C}(S)$ anymore. An automatic response from the system section state machine has therefore to be provided in order to move the control to a state T such that Equation 2 is satisfied for state T. We say that a set I of internally triggered automatic state transitions *covers* an exit zone (c, t) iff any transition t_i in I (1) is guarded by a subproposition c_i of c, (2) has t as trigger and (3) the set of guards of the transitions in I form a partition of c.

In order to have the state invariant of Definition 2 *always satisfied* for state S it then suffices to check that any exit zone of S is covered by an adequate set of automatic transitions. Since such transitions will have to satisfy the correctness requirements of Section 3.2.1 for the arrival state, the state invariant is guaranteed to be always satisfied.

3.2.3 State transition pre- and postcondition semantics

Let S and T be two generic states in the system section which are correctly specified according to Definition 2, and let consider adding a transition t joining state S to state T. We show how $\mathsf{pre}(t)$ and $\mathsf{post}(t)$ can be determined for both *regular* and *automatic* transitions.

In the regular case, before t is taken, the current state $\mathbf{q}$ of the system satisfies $\mathsf{C}(T)$. At the same time, in order for t to be executed, $\mathbf{q}$ must satisfy also the transition guard $\mathsf{guard}(t)$. It follows that $\mathbf{q}$ must therefore satisfy the intersection of the two state propositions, that is

$$\mathsf{pre}(t) = \mathsf{C}(T) \wedge \mathsf{guard}(t) \tag{2}$$

In the automatic case, before t is taken, the current state $\mathbf{q}$ of the system satisfies the state constraint $\mathsf{C}(T)$ *after* an internal transition $\mathsf{c}.t$ happened: let $\mathsf{transf}(\mathsf{C}(T), \mathsf{c}.t)$ denote such a modified constraint. As in the case above, in order for the transition to be executed, $\mathbf{q}$ must satisfy also the transition guard $\mathsf{guard}(t)$. It follows that $\mathbf{q}$ must therefore satisfy

$$\mathsf{pre}(t) = \mathsf{transf}(\mathsf{C}(T), \mathsf{c}.t) \wedge \mathsf{guard}(t) \tag{3}$$

In both cases, after the transition is chosen, a (possibly empty) list of actions $\langle a_1, a_2, \ldots, a_N \rangle$ has to be executed. If a generic state $\mathbf{q}_0$ of the assemblage satisfies $P_0 = \mathsf{pre}(t)$ (computed above for the two different cases by Equations 2 and 3) *before* action a_1 is executed, the state of the assemblage $\mathbf{q}_1$ resulting from the execution of such action will satisfy another state proposition, say P_1, *after* action a_1 has been executed. Let $P_1 = \mathsf{transf}(P_0, a_1)$ denote the transformation in the state proposition induced by the execution of the action a_1. Accordingly, it is possible to denote the final state proposition $\mathsf{post}(t) = P_N$ as

$$\begin{array}{lcl} P_1 & = & \mathsf{transf}(\mathsf{pre}(t), a_1) \\ P_2 & = & \mathsf{transf}(P_1, a_2) \\ & \vdots & \\ \mathsf{post}(t) & = & \mathsf{transf}(P_{N-1}, a_N) \end{array} \tag{4}$$

3.2.4 Exit zone computation

Let C be a state proposition about assemblage A. We are interested in computing $E(C)$, the set of *exit zones of* C, consisting of the couples $(p, \mathsf{c.t})$, where p is a subproposition of C and t is a non controllable transition which can be taken by component c *when the assemblage satisfies* p which leads to a global state which do not satisfies C anymore. In [6] we show how the full set of exit zones can be algorithmically determined.

For example, with reference to Figure 3, since $\mathsf{C}(\mathsf{On}) = \mathsf{On}^{\mathsf{p}} \odot \mathsf{Open}^{\mathsf{v}} \odot \mathsf{Low}^{\mathsf{ps}}$, the only non controllable transition t_3 in component ps under $\mathsf{C}(\mathsf{On})$ may lead to a global state $\dot{\mathbf{q}}$ which satisfies $\mathsf{On}^{\mathsf{p}} \odot \mathsf{Open}^{\mathsf{v}} \odot \mathsf{High}^{\mathsf{ps}}$, hence $(\mathsf{C}(\mathsf{On}), \mathsf{ps}.t_3)$ is an exit zone that needs to be covered by an automatic transition in the system section (see Section 3.4).

3.3 State liveness

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

Definition 2 (State liveness invariant) *For any state S in the system section and for any event e, let $T(S, e)$ be the set of transitions departing from S and having event e as trigger. If $T(S, e)$ is not empty, then* exactly one *of such transitions will be executed upon the receipt of e by the PWS.*

In order to ensure the invariant above we have to check that the preconditions of the transitions in $T(S, e)$ form a partition of $\mathsf{C}(S)$.

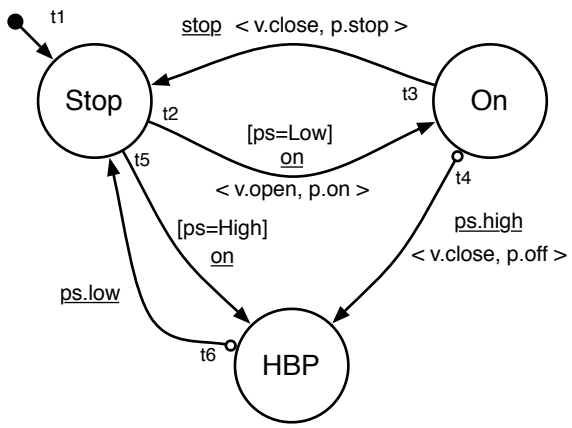

Figure 4. Extending the system section of the PWS of Figure 3 by an explicit exception state HBP and by additional state transitions.

3.4 Final system design

According to the methodology, we find two main flaws in the design of Figure 3; in first place we observe a liveness issue: the whole system may not start on the receipt of the event on if the value of the pressure sensor ps is High; this happens since the outgoing transition t_2 from state Stop constrained by $\mathsf{C(Stop)} = \mathsf{Off^p} \odot \mathsf{Close^v}$ in the system section has precondition $\mathsf{pre}(t_2) = \mathsf{C(Stop)} \odot \mathsf{Low^{ps}} = \mathsf{Off^p} \odot \mathsf{Close^v} \odot \mathsf{Low^{ps}}$ which does not partition $\mathsf{C(Stop)}$. In order to solve the problem, we insert a novel transition t_5 (Figure 4), having the same event on as trigger, guarded by $\mathsf{High^{ps}}$. In such a case, it can be easily verified that $\mathsf{pre}(t_2)$ and $\mathsf{pre}(t_5)$ form now a partition of $\mathsf{C(Stop)}$, hence *exactly one* of the two transitions will be taken.

In second place, we observe a more serious safety issue: the state On, constrained by $\mathsf{C(On)} = \mathsf{On^p} \odot \mathsf{Open^v} \odot \mathsf{Low^{ps}}$ in the system section has exit zone $(\mathsf{C(On)}, \mathsf{ps}.t_3)$ which is not covered by any transition. We insert consequently a new transition t_4 triggered by $\mathsf{ps}.t_3$ or, equivalently, by the assemblage event ps.high.

We have now two dangling transitions, t_4 and t_5, which spring, respectively, from the working and from the stopped state of the system, On and Off. The natural choice is to introduce a novel HBP state, which denotes the global state of the system in which high blood pressure requires to have both valve closed and pump halted, that is $\mathsf{C(HBP)} = \mathsf{Off^p} \odot \mathsf{Closed^v} \odot \mathsf{High^{ps}}$. It can be now observed that $\mathsf{post}(t_4)$ is not compatible with the constraint of HBP:

$$\underbrace{\mathsf{On^p} \odot \mathsf{Open^v} \odot \mathsf{High^{ps}}}_{\mathsf{post}(t_4)} \not\preceq \underbrace{\mathsf{Off^p} \odot \mathsf{Closed^v} \odot \mathsf{High^{ps}}}_{\mathsf{C(HBP)}}$$

In other words, in order to have transition t_4 compatible with the constraint of HBP, we have to turn the pump off and to close the valve by suitable commands sent to the assemblage before entering the state. We thus associate the action sequence $\langle \mathsf{v.close}, \mathsf{p.off} \rangle$ to transition t_4, yielding the new postcondition $\mathsf{post}(t_4) \equiv \mathsf{C(HBP)}$.

Finally, the insertion of the new state HBP requires to "cover" the new exit zone $(\mathsf{C(HBP)}, \mathsf{ps}.t_2)$, corresponding to the system returning to a regular blood pressure. We choose to insert a new transition t_6 in order to bring the system to Stop in such a case: the transition is correct since $\mathsf{post}(t_6) \preceq \mathsf{C(Stop)}$. We observe that other safe design choices were feasible, for example by adding the action sequence $\langle \mathsf{v.open}, \mathsf{p.on} \rangle$ to transition t_6 which might join instead state HBP to state On, in order to automatically restart the whole system when the pressure returns regular.

4 Conclusions

We have shown the basic guidelines of a method for assessing safety axioms into a state-based distributed system. Such a method follows logically from the explicit modeling of the behavior of interacting devices, which yields a system state diagram depicting and enforcing a specific global behavior of the devices making the system. The method lends itself to be easily implemented both as a verification tool and as part of an interactive design tool.

References

[1] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. IEEE Symposium on Logic in Computer Science*, Washington, D.C., 1990. IEEE Computer Society Press.

[3] I. Lee and G. Pappas. Report on the high-confidence medical-device software and systems workshop. Technical Report NSF CNS 0532968, 2005.

[4] L. Pazzi. Extending statecharts for representing parts and wholes. In *Proceedings of the EuroMicro-97 Conference, Budapest, Hungary*, 1997.

[5] L. Pazzi. Part-whole statecharts for the explicit representation of compound behaviors. In *UML 2000 - The Unified Modeling Language. Advancing the Standard.*, volume 1939 of *LNCS*, pages 541–555. Springer, 2000.

[6] L. Pazzi. A method for ensuring safety and liveness rules in a state-based design, http://cris.unimore.it/cris/files/2008-02-01.pdf. Technical Report CRIS-2008-02-01, 2008. Patent pending PCT/EP2008/051300.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[8] J. Staunstrup, H. R. Andersen, H. Hulgaard, J. Lind-Nielsen, K. G. Larsen, G. Behrmann, K. Kristoffersen, A. Skou, H. Leerberg, and N. B. Theilgaard. Practical verification of embedded software. *Computer*, 33(5):68–75, 2000.