Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping, Geneva, Switzerland, 28-30 June 2004

TLCharts: Armor-plating Harel Statecharts with Temporal Logic Conditions

Doron Drusinsky and Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
{ddrusins, shing}@nps.edu

Abstract

This paper addresses the need for armor-plating Harel Statechart design specifications of real-time systems with safety requirements (which are commonly written in temporal logic) using a new visual specification language named TLCharts. TLCharts combine the visual and intuitive appeal of non-deterministic Harel Statecharts with formal specifications written in Linear-time (Metric) Temporal Logic. We demonstrate such armor-plating with a specification of the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump.

1 Introduction

The Harel Statecharts notation is a graphical specification language that extends finite state diagrams with hierarchy (state nesting), concurrence, and history states [Ha]. Harel Statecharts are commonly used for design analysis and implementation; for example, Brugge suggests using statecharts in the design analysis phase of an object oriented UML based design methodology to specify the dynamic behavior complex reactive systems [Br]. Statecharts are typically used in their deterministic form. However, theoretical results [DH] show that non-deterministic statecharts are exponentially more succinct than deterministic Harel Statecharts.

While statecharts can effectively specify what a system *should do* (positive information), they tend to be less effective for the specification of safety requirements (i.e., negative information about what a system *must not do*). Hence, researchers have attempted to augment statechart specifications with other formalisms like process algebra [PB], symbolic timing diagrams [LN] and temporal logic [GH], and demonstrated formal proofs for certain properties of the Statechart design. This is typically done using two separate formalisms (e.g., statecharts and temporal logic) joint by tool cooperation or proof techniques. The major problem with such approaches is that the lack of a unified formalism requires users to work with two models, with no guarantee that the correctness of one implies the correctness of the other.

This paper describes TLCharts, a hybrid visual specification language that combines the visual and intuitive appeal of non-deterministic Harel Statecharts with formal specifications written in Linear-time (Metric) Temporal Logic. TLCharts differ from the augmentation approach described earlier in that they combine two formalisms into a single coherent language with simple semantics. The interlingua semantics of TLCharts have been published in [D3, D4].

The new demand for high performance and intelligent automobiles, aircrafts and autonomous robots has pushed the complexities of embedded systems to a new level. These systems now need to interact closely with other embedded systems and function under much tighter timing and control constraints. The need to interact closely with their environment also makes the understanding and satisfaction of their safety requirements a number one priority. We need to increase the robustness of these software by *armor-plating* them against any unexpected behaviors. In [D1], Drusinsky proposed one form of armor-plating that fortifies the software's exception-handling ability via runtime monitoring of temporal assertions, where formal specifications are translated by a code generator into C, C++, or Java statements to be deployed for catching exceptions in the final product during runtime. In this paper, we describe another form of armor-plating via over specification of design. We propose using TLCharts to armor-plate Harel Statechart design specifications of real-time systems with safety requirements, where an existing statechart, possibly correct and complete, is augmented with additional, possibly redundant, transitions annotated with temporal logic conditions. These additional transitions induce over specification with multiple computations potentially enabled by a given input sequence. While such redundancy is undesirable for implementation purposes, a TLChart is a formal specification. As such, this over specification is a form of armor-plating which increases the level of assurance in the correctness of the specification. These TLChart specifications can then be translated into target code to support the run-time monitoring of the resultant software.

The rest of the paper is organized as follows. Section 2 provides an overview of temporal logic followed by an introduction to TLCharts. Section 3 presents a statechart design example of the safety-critical computer assisted resuscitation algorithm (CARA) software for a casualty intravenous fluid infusion pump. Section 4 illustrates armor-plating the specification using TLCharts. Section 5 presents a discussion on the approach and draws the conclusion.

2 Temporal Logic and TLCharts

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. Linear-time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the well-known propositional logic operators, there are four future-time operators (◊-*Eventually*, □-*Always*, *U-Until*, O-*Next*) and four dual-past time operators. Pnueli [Pn] suggested using LTL for reasoning about concurrent programs. Since then, several researchers have used LTL to state and measure correctness of concurrent programs, protocols, and hardware (e.g., [MP, Pn]). Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [CP]. MTL extends LTL by supporting the specification of relative-time and real-time constraints. With MTL, all four LTL future-time operators can be characterized by relative-time and real-time constraints specifying the duration of the temporal operator. Temporal Logic with Time Series constraints (MTLS) was suggested by Drusinsky as an extension of MTL which enables temporal specifications that assert about time-series properties such as stability, monotonicity, and min-max values [D2]. In [DS], we showed that the use of run-time monitoring and verification of MTLS assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors earlier in the design process.

Drusinsky recently suggested TLCharts as hybrid of Harel Statecharts and temporal logic [D3, D4]. TLCharts visually and intuitively resemble Harel Statecharts while enabling non-determinism, negation and temporal-logic conditioned transitions. This is useful for specifying abstract non-deterministic temporal properties inside a statechart specification.

TLCharts enable temporal conditions (guards) along Harel Statechart transitions. Like a statechart, a TLChart processes an input sequence one symbol at a time and performs one or more state transitions per concurrent statechart thread every cycle. In addition, a TLChart considers the remaining input sequence as a model for the temporal logic guard. Similarly, when using past-time temporal logic, the input sequence already processed is considered as a model for the past. The Boolean evaluation of this guard is then imposed (via conjunction) onto the ordinary Harel Statechart condition thereby augmenting the statechart's behavior. Using a similar approach regular expressions can be used instead of temporal logic. TLCharts specify good (or bad) behavior using the notion of *Good* and *Error* states. Computations that end in Good states are *accepted* while those that end in Error states are *rejected*. TLCharts semantics enable non-deterministic specifications. A priority scheme performs resolution of accepted vs. rejected computation in case conflicts due to non-determinism.

Figure 3 shows a TLChart that is the armor-plated version of the statechart of Figure 2, which will be explained in details in Section 4. We refer the reader to [D3, D4] for a more complete discussion of TLCharts.

3 The CARA Software

CARA is a safety-critical software developed by the Walter Reed Army Institute of Research to improve life support for trauma cases and military casualties [W1, W2, W3]; it has been used as a case study by several software engineering research groups [AA, LS]. CARA's mission is to monitor a patient's blood pressure and to automatically administer intravenous (IV) fluids via computer-controlled pump at levels required to restore intravascular volume and blood pressure.

The main responsibilities of the CARA system include:

1. To monitor a patient's blood pressure.
2. To control a high-output patient resuscitation infusion pump.
3. To display (to a human caregiver) vital information about the patient and the system.
4. To log all data.
5. To alarm the caregiver during emergency situations.

We will use CARA to demonstrate an application of TLChart as a vehicle for armor-plating Harel Statechart design specifications of safety critical real-time systems. We have extracted from [W3] a subset of requirements that corresponds to: (i) infusion pump functionality and (ii) blood pressure monitoring. We have developed a Harel Statechart design for the expected behavior of the CARA software. For space and simplicity reasons we have excluded those requirements pertaining to message display and logging.

Figure 1 shows the top level of the statechart for CARA software. It consists of two simple states, *Start* and *PwrOn*, as well as a composite state named *PumpControl*, illustrated in Figure 2, which consists of five concurrent control threads.

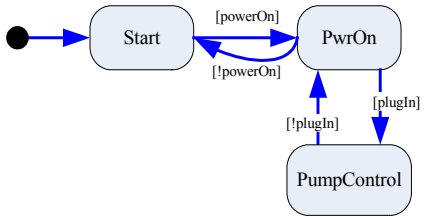

Figure 1. Top Level Statechart of the CARA software

CARA monitors the pump connector on the Life Support for Trauma and Transport (LSTAT) to determine when a pump is powered on and plugged in. Once the pump is plugged in, CARA continuously checks for continuity on all wires going to the pump, as shown in the *Line* component in Figure 2. Logic level input signals (representing continuity, occlusion, air in IV line) occur as interrupt signals whenever the state of a corresponding signal changes; they are represented in Figure 2 as guarded conditions *discon*, *occ* and *!airOk*. If *discon* or *occ* becomes true or if *!airOk* remains true for 10 seconds, CARA will sound a level 1 alarm and terminate auto-control if it is in auto-control mode.

After the pump is plugged in, a clock interrupt triggers an event at precise five-second intervals signaling the necessity to check the back EMF, display the updated flow rate, and check the impedance; these activities are shown in the *CheckFlow* component in Figure 2. The guarded condition *badBackEmf* is set whenever the back EMF reading is zero or cannot be obtained, and the condition *badImp* is set whenever the impedance of the IV fluid is outside the tolerance. CARA will sound a level 1 alarm if *badBackEmf* or *badImp* becomes true, and will terminate auto-control if it is in auto-control mode.

When CARA determines that (i) the pump is plugged in and not stopped, (ii) an IV fluid with an impedance within tolerance is in place and (iii) the occlusion line is clear, it shows a "CARA Status OK" message indicating that CARA is ready to start auto-control and displays the "*Start Auto-control*" button. It then transitions into the *AutoReady* state of the *Mode* component in Figure 2.

When the "*Start Auto-control*" button is pressed, CARA concurrently enters the *Auto* state of the *Mode* component and the *InitialCuffBp* state of the *Control* component in Figure 2. Once inside the *Auto* state, CARA will attempt to use blood pressure from various sources as the input for the CARA algorithm to control the pump. For simplicity, we use only two blood pressure sources in this version of the prototype - an arterial line sensor and a cuff pressure sensor. The arterial line sensor is an active device that updates its value once every second. The cuff pressure sensor is a passive device that is inflated on demand under the control of CARA to measure the patient's blood pressure. CARA is required to prefer the arterial line blood pressure over the cuff pressure for purposes of pump control.

As CARA enters the *InitialCuffBp* state, it initializes the pump at a default flow rate of 4 liters/hr and inflates the cuff attempting to obtain the patient's blood pressure. If cuff pressures are not available, it will sound a level 1 alarm, display the override "*yes*" and "*no*" choice buttons, and enter the *FirstOverride* state. Pressing the "*alarm reset*" button will reset the alarm and reattempt to inflate the cuff. Pressing the "*no*" button will reset the alarm and return to manual mode. Pressing the override "*yes*" button will reset the alarm, force CARA to enter the *UncorroborateALine* state and use the uncorroborated arterial line pressure for control. If cuff pressure is not available and there are no other blood pressures sources available, CARA will revert to manual mode and sound a level 1 alarm.

Whenever the cuff pressure is available while CARA is in the *InitialCuffBp* state, CARA enters the *Corroborating* state and begins blood pressure source corroboration. CARA uses cuff pressure for control during arterial line corroboration. The arterial line pressure is compared to a corresponding cuff pressure. If arterial line pressure is within 10% of the corresponding cuff pressure then CARA enters the *CorroboratedALine* state and the arterial line is corroborated and will be used for control. If the arterial line pressure is *not* within 10% of the corresponding cuff pressure then two more cuff readings must be taken and compared against the corresponding arterial line readings, as performed in the states *SecondAttempt* and *ThirdAttempt*. If both arterial line readings are within 10% of the corresponding cuff readings, the arterial line is considered as corroborated and should be used for control. If one or more of the arterial line pressure readings are not within 10% of the corresponding cuff readings then an override dialog box is displayed as CARA enters the *SecondOverride* state. If the user presses the "*yes*" override button then the uncorroborated arterial line pressure is used for control. If the override "*no*" button is pressed, CARA enters the *CuffControl* state and uses the cuff pressure for control. While under the cuff pressure control, uncorroborated arterial line pressures will be compared to each cuff reading in the *OnGoingCor* state. If the arterial line readings are within 10% of the cuff readings, CARA will automatically switch over to the arterial line for control. Override dialog boxes are not displayed during these subsequent corroboration attempts.

While under arterial line pressure control, CARA recorroborates the blood pressure control source against the cuff every 30 minutes. Any active corroboration attempt must be completed before the periodic 30-minute recorroboration can begin. If the cuff pressure is not available for re-calibration, CARA will sound a level 1 alarm.

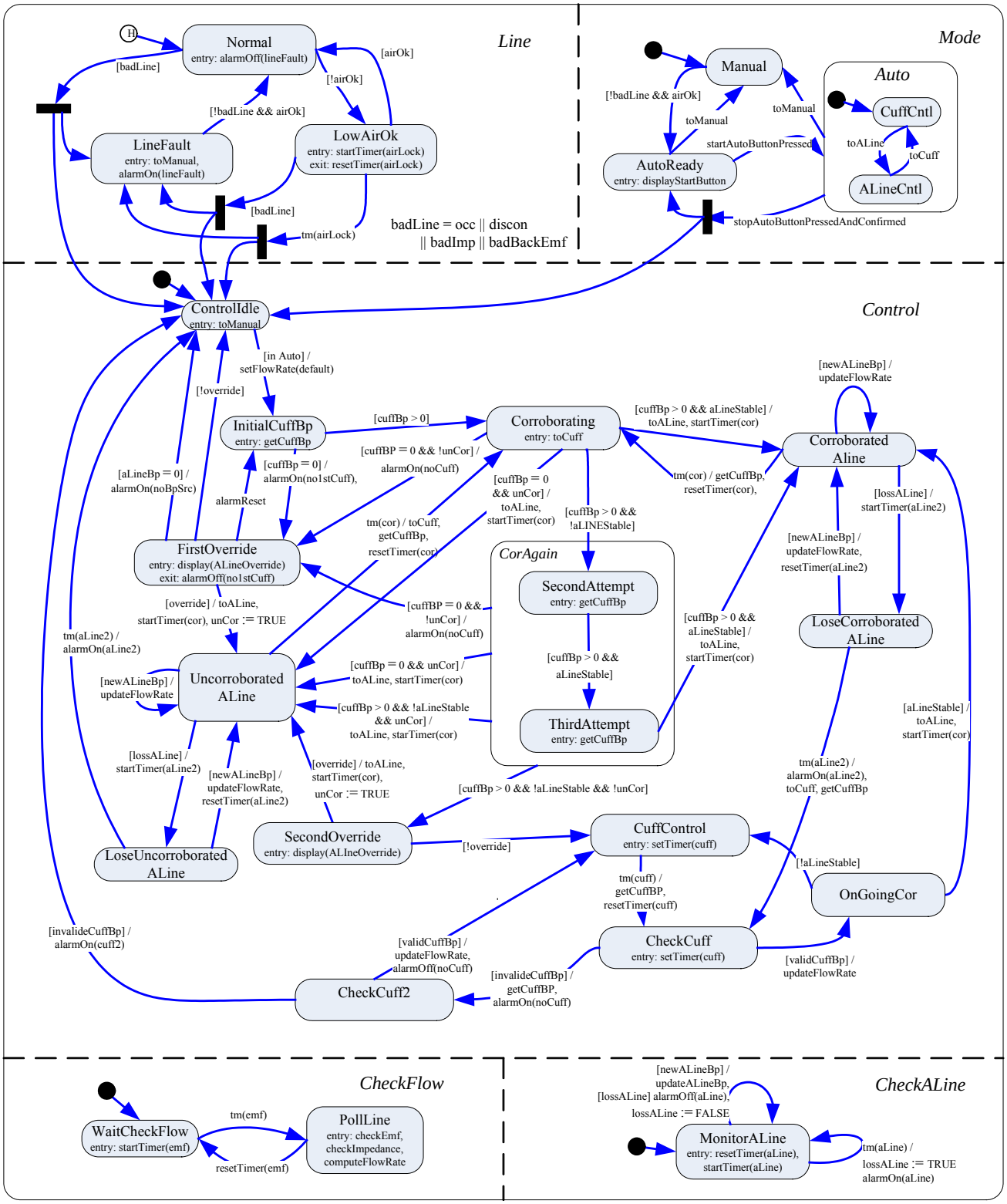

Figure 2. Components of the PumpControl state

Once a valid blood pressure has been established, CARA calculates a voltage to drive the pump and will readjust the voltage after each blood pressure reading. When the cuff pressure is being used for control, CARA sets a cuff reading frequency based on a table below. If the cuff is already inflating for some other reason when the time arrives for another reading, an additional cuff reading does not need to be requested.

BP	≤ 60	(60, 70]	(70, 90]	> 90
Freq	1 min	2 min	5 min	10 min

While under the arterial line pressure control, if the arterial line signal is lost for more than 1 minute, CARA sounds a level 1 alarm and enters the *LoseCorroboratedALine* state. If the arterial line signal is lost for more than 3 minutes, a level 2 alarm sounds. If only the cuff pressure is being used and an expected blood pressure reading is invalid (i.e. outside the 40-150 mmHg range), CARA sounds a level 1 alarm and enters the *CheckCuff2* state. It then initiates another request for a cuff pressure. If this pressure is invalid, CARA sounds a level 2 alarm and reverts back to manual mode.

While in auto-control mode, a "*Terminate Auto-control*" button will be made available. Pressing this button will activates "*yes*" and "*no*" confirmation buttons. The "*yes*" button causes CARA to revert back to manual mode. The "*no*" button causes CARA to return to the auto-control mode.

4 Armor-plated TLChart for CARA

The safe operation of CARA depends on the timely detection of abnormal situations in the pump lines and the patient's blood pressure. In addition to deciding on automatic or manual pump control according to the run time changing of the patient's blood pressure value, CARA must alert the caregivers to intervene in emergency situations via a set of alarms. The assumption is that the caregivers know what they are doing and will correct the situations (like loss of pump line signals or patient's blood pressure) before restarting the auto-control process. The fact that "*CARA should not start the auto-control process before the emergency situation is corrected"* is an example of the safety requirements that should be included in the specification of the system's statechart to ensure the safe operation of the software. In this section, we show how this can be done using TLCharts.

Figure 3 is an armor-plated TLChart version of the statecharts of Figure 2. In this TLChart, all states other than the *Error* state are good states. Computations that end in the *Error* state indicate undesirable behavior, such as the transitions labeled as Type 1 or Type 2. In Figure 3, the gray transitions are those that exist in the statechart shown in Figure 2 while the black transitions are the newly added armor-plating transitions, which are annotated with temporal logic conditions (denoted as *temporal guards*). Temporal guards are marked by curly braces while conventional propositional guards are marked by square brackets. Note that a transition can be annotated by both temporal and propositional (conventional) guards.

Consider armor-plating transition

$$UncorroboratedALine \rightarrow ControlIdle;$$

it is annotated with the temporal guard $\{\Box_{>3\ min} lossALine\}$. This transition reads as follows: if the present state at time *t* is *UncorroboratedALine* and *lossALine* is true for more than three minutes into the future then transition (on the cycle following *t*) to state *ControlIdle.* In other words, temporal guards make present time decisions by looking into the future and the past; see the interlingua, automata-based semantics in [D4]. While not directly useful as a model for a real-time controller, such a model makes sense in a specification and verification context and provides the basis for run-time model checking of the software [D5].

Armor-plating is achieved by superimposing new Error states as well as additional armor-plating transitions on top of the statechart specification. In Figure 3, we have added three types of armor-plating transitions, as follows:

- Type 1 transitions behave like conventional temporal logic assertions; they make a high-level temporal logic assertion about the statechart (or most of it), such as

  $$\Box_{>10sec} !air\_ok \Rightarrow !(\Diamond_{[10sec,12sec]}\ lineFaultAlarmOn)$$

  in Figure 3, which states that it is an error if *!airOk* is true for more than ten seconds in any sub-state of *PumpControl* and *lineFaultAlarmOn* does not become true within the next two seconds.

- Type 2 transitions are anchored in specific states (e.g., the *Manual* state in Figure 3) and therefore make conditional assertions about expected behavior when the statechart visits that particular state. The transition from the *Manual* state with the assertion

  $$\{(in\ Auto)\ Before\ (allAlarmsOff\ \&\&\ airOk\ \&\&!badLine)\}$$

  states that it is an error to enter the *Auto* state before all alarms are off, *badLine* is reset to false and *airOk* is set to true; while the transition from the *LoseCorroboratedALine* state with the assertion

  $$\{(in\ OnGoingCor)\ Before\ ((in\ Manual)\ \&\&\ aLine2AlarmOff)\}$$

  states that once CARA enters the *LoseCorroboratedALine* state, it is an error to corroborate the arterial line blood pressure in the *OnGoingCor* state without first going into the *Manual* state and turning off the *aLine2Alarm*.

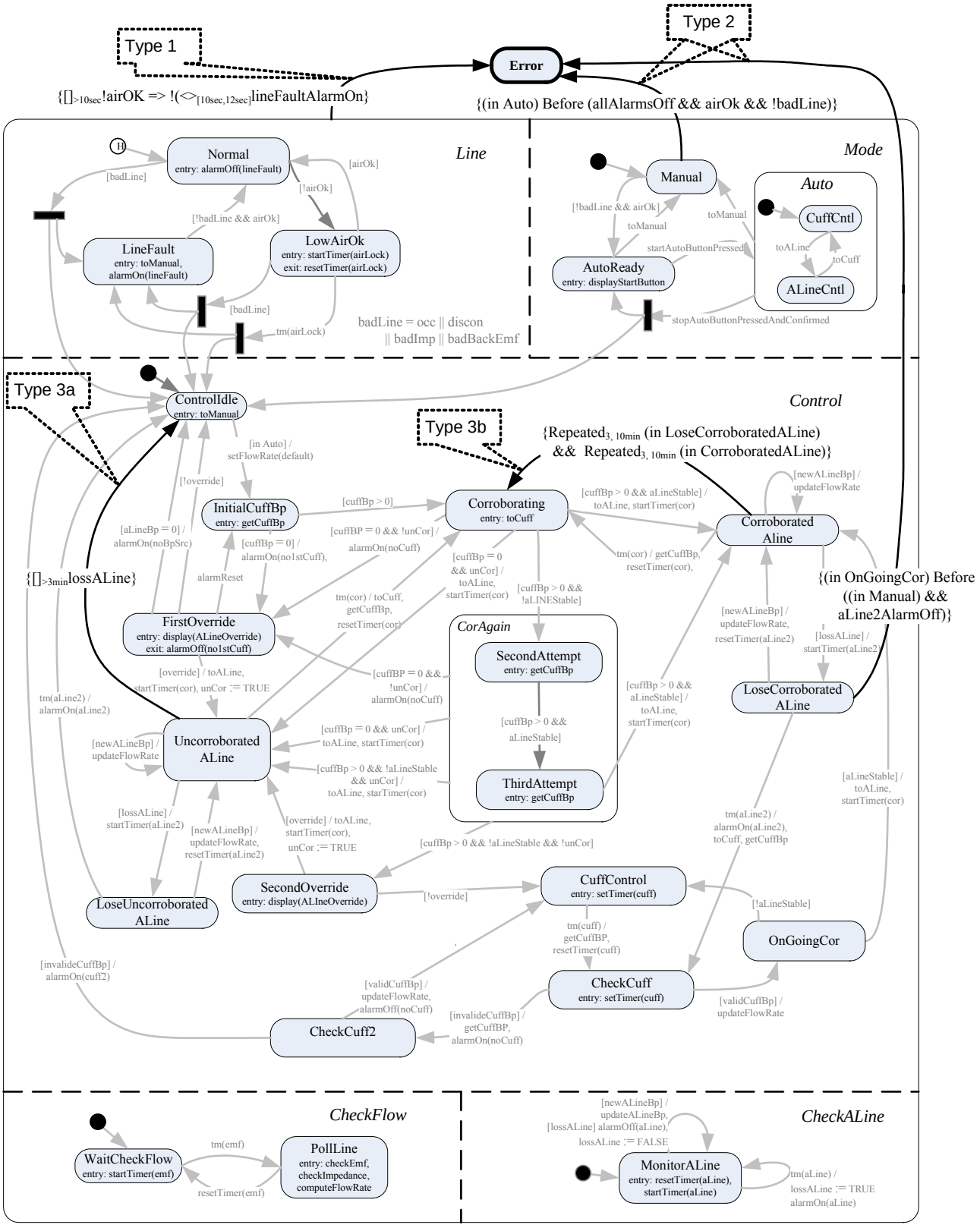

Figure 3. Armor-plated TLChart

Type 3 transitions are completely integrated into the statechart's transition function, i.e., they do not lead to an Error state, such as forcing a transition into state *ControlIdle* whenever *lossALine* is true for more than three minutes. Note how, in Figure 3, we distinguish between Type 3a and Type 3b transitions. Type 3b transitions are similar to Type 3a but use the LTL *until* operator, nested temporal logic, or counting operators as guards. (Note that *p Before q* is an abbreviation for the LTL formula *!q Until p*.) The transition from *CorroboratedALine* → *Corroborating* with the assertion

$$Repeated_{\geq 3,\ \leq 10min}\ (in\ LoseCorroboratedALine)$$
$$\&\&\ Repeated_{\geq 3,\ \leq 10min}\ (in\ CorroboratedALine)$$

forces CARA to re-corroborate the arterial line blood pressure whenever it loses the arterial line pressure 3 or more times within a 10-minute interval while using corroborated arterial line pressure to control the pump. The assertion uses the counting operators that were described in [D5]. These operators are more readable than their equivalent pure-LTL representation that uses a nesting of until operators. Moreover, as discussed in Section 5, the translation of the notation LTL *until* operator into a pure statechart notation is non-trivial.

5 Discussion and Conclusion

The inclusion of safety requirements in design specifications helps highlight what the system must not do, which if overlooked, will lead to unsafe operations of the software. TLCharts offer an opportunity for armor-plating specifications using *over-specification*, namely by adding temporal conditions to an otherwise fully specified design. The TLCharts also provide the basis for armor-plating run-time applications. Following ideas suggested by Drusinsky in [D1], armor plating is accomplished using run-time monitoring of LTL and MTL assertions combined with exception handling. This was done with tools like Temporal-Rover code generator, that converts temporal into code snippets to be embedded in the target software for run-time verification [D5], thus providing further protection against unsafe behaviors of the reactive systems. TLCharts offer an opportunity for better run-time armor plating because TLChart armor plating transitions (such as the transitions of Figure 3) are tightly integrated with the statechart transition function. Currently, we have to translate TLChart specifications to temporal assertions manually and then use the DBRover's TLChart support to generate the run-time monitor code. We are working on the development of TLChart interactive development environment (IDE), code generator, and model checking related tools. The code generator will integrate known code generation techniques for deterministic statecharts [D6] with tools for run-time monitoring of temporal logic (DBRover and Temporal Rover) [D5].

TLCharts enable a coherent uniform formalism for a hybrid of statecharts and temporal logic. A similar hybrid can support statecharts and regular expressions. We believe that such a hybrid language is preferable over having two separate languages (temporal logic and statecharts) with assertions written in one language and used for armor-plating the other language. The former integration is a language level integration while the latter is a tool level integration.

TLCharts are expressive extension of statecharts. While simple temporal logic formulae like $\Box p$ and $\Diamond p$, where p is propositional (i.e., non-temporal), have simple equivalent statechart representations, the following two kinds of temporal logic guards are harder to express as statecharts:

1. Temporal logic guards that contain the *until* operator. An example of such a guard is the transition marked as Type 3b, in Figure 3. We also refer the user to [D3] for other examples.
2. Temporal logic guards with nested temporal operators, such as $\Box_{<10}\ (p \Rightarrow \Diamond_{<5} q)$. Typically, every level of nesting requires a concurrent thread in the equivalent statechart.

Acknowledgements and Disclaimer

The research reported in this article was funded by a grant from the U.S. Missile Defense Agency under contract number BMDO01342923172. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

[AA] R. Alur, D. Arney, E. Gunter, I. Lee, W. Nam and J. Zhou, "Formal Specifications and Analysis of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump Control System", *Proc. Integrated Design and Process Technology (IDPT)*, 2002.

[Br] B. Bruegge, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, ISBN 0-13-489725-0.

[CP] E. Chang, A. Pnueli and Z. Manna, "Compositional Verification of Real-Time Systems", *Proc. 9th IEEE Symp. on Logic in Computer Science*, 1994, pp. 458-465.

[D1] D. Drusinsky. Specs Can Handle Exceptions. *Embedded Developers Journal*, November 2001, pp. 10-14. (http://eet.com/embedsub/archive.html).

[D2] D. Drusinsky, "Monitoring Temporal Rules Combined with Time Series", *Proc. 2003 Computer Aided Verification Conference (CAV 2003)*, 8-12 July 2003, pp. 114-117.

[D3] D. Drusinsky, "Visual Formal Specification using (N)TLCharts: Statechart Automata with Temporal Logic and Natural Language Conditioned Transitions", *Proc. International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2004. Invited paper.

[D4] D. Drusinsky, "Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions", *Proc. 4th Runtime Verification Workshop (RV'04)*, 2004, Invited paper.

[D5] D. Drusinsky, "The Temporal Rover and ATG Rover", *Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science*, 1885, pp. 323-329.

[D6] D. Drusinsky, "A State Assignment for Single-block Implementation of Statecharts", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10(12), 1991, pp. 1569-1575.

[DH] D. Drusinsky and David Harel, "On the power of bounded concurrency I: Finite Automata", *J. ACM*, 41(3), 1994, pp. 517-539.

[DS] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping", *Proc.14th IEEE International Workshop in Rapid Systems Prototyping*, 9-11 June 2003, pp. 47-53.

[GH] G. Graw, P. Herrmann, and H. Krumm, "Verification of UML-Based Real-Time System Design by Means of cTLA", *Proc. 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, 15-17 March 2000, pp.86-95.

[Ha] D. Harel, "A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8, pp. 231-274, 1987.

[LN] K. Lüth, J. Niehaus and T. Peikenkamp, "HW/SW Co-synthesis using Statecharts and Symbolic Timing Diagrams", *Proc. 9th International Workshop on Rapid System Prototyping*, 3-5 June 1998, pp.212-217.

[LS] Luqi, M. Shing, J. Puett, V. Berzins, Z. Guan, Y. Qiao, L. Zhang, N. Chaki, X. Liang, W. Ray, M. Brown, and D. Floodeen, "Comparative Rapid Prototyping, A Case Study", Proc. *14th IEEE International Workshop in Rapid Systems Prototyping*, 9-11 June 2003, pp. 210-217.

[MP] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles", *Proc. Workshop on Logics of Programs*, Springer LNCS, 1981 pp. 200-252.

[PB] M.H. Park, K.S. Bang, J.Y. Choi and I. Kang, "Equivalence Checking of Two Statechart Specifications", *Proc. 11th International Workshop on Rapid System Prototyping*, 21-23 June 2000, pp.46-51.

[Pn] A. Pnueli, "The Temporal Logic of Programs", *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, pp. 46-57.

[W1] WRAIR Dept. of Resuscitative Medicine, *Narrative Description of the CARA software, Proprietary Document*, WRAIR, Silver Spring, MD, Jan 2001.

[W2] WRAIR Dept. of Resuscitative Medicine, *CARA Pump Control Software Questions (Version 6.1)*, Proprietary Document, WRAIR, Silver Spring, MD, Jan 2001.

[W3] WRAIR Dept. of Resuscitative Medicine, *CARA Tagged Requirements, Increment 3 (Version 1.2)*, Proprietary Document, WRAIR, Silver Spring, MD, March 2001.