Modeling and Verification of Safety-Critical Systems Using Safecharts

Pao-Ann Hsiung and Yen-Hung Lin

Department of Computer Science and Information Engineering,
National Chung Cheng University, Chiayi, Taiwan−621, ROC
hpa@computer.org

Abstract. With rapid development in science and technology, we now see the ubiquitous use of different types of *safety-critical systems* in our daily lives such as in avionics, consumer electronics, and medical systems. In such systems, unintentional design faults might result in injury or even death to human beings. To make sure that safety-critical systems are really safe, there is need to verify them formally. However, the verification of such systems is getting more and more difficult, because the designs are becoming very complex. To cope with high design complexity, currently model-driven architecture design is becoming a well-accepted trend. However, conventional methods of code testing and standards conformance do not fit very well with such model-based approaches. To bridge this gap, we propose a model-based formal verification technique for safety-critical systems. In this work, the *model checking paradigm* is applied to the *Safecharts model* which was used for modeling, but not yet used for verification. Our contributions are five folds. Firstly, the safety constraints in Safecharts are mapped to semantic equivalents in timed automata for verification. Secondly, the theory for safety constraint verification is proved and implemented in a compositional model checker (SGM). Thirdly, prioritized transitions are implemented in SGM to model the risk semantics in Safecharts. Fourthly, it is shown how the original Safecharts lacked synchronization semantics which could lead to safety hazards. A solution to this issue is also proposed. Finally, it is shown that priority-based approach to mutual exclusion of resource usage in the original Safecharts is unsafe and corresponding solutions are proposed here. Application examples show the feasibility and benefits of the proposed model-driven verification of safety-critical systems.

1 Introduction

Safety-critical systems are systems whose failure most probably results in the tragic loss of human life or damage to human property. There are numerous examples of these mishaps. The accident at the Three Mile Island nuclear power plant in Pennsylvania on 28^{th} March, 1979 is just one unfortunate example. Moreover, as time goes on, there are more and more cars, airplanes, rapid transit systems, medical facilities, and consumer electronics, which are all safety-critical systems appearing in our daily lives. When some of them malfunction or fault, a tragedy is inevitable. The natural question that comes to mind is that can we use these systems without 100% warranty? Obviously,

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 290–304, 2005.

the answer is negative. That's why we need some methodology to exhaustively verify safety-critical systems.

Traditional verification methods such as simulation and testing can only prove the presence of faults and not their absence. Simulation and testing [11] both involve making experiments before deploying the system in the field. While simulation is performed on an abstract model of a system, testing is performed on the actual product. In the case of hardware circuits, simulation is performed on the design of the circuit, whereas testing is performed on the fabricated circuit itself. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. For software, simulation and testing usually involve providing certain inputs and observing the corresponding outputs. These methods can be a cost-efficient way to find many errors. However, checking all of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible. Conventionally, safety-critical systems are validated through standards conformance and code testing. Using such verification methods for safety-critical systems cannot provide the desired 100% confidence on system correctness.

In contrast to the traditional verification methods, formal verification is exhaustive and provides 100% guarantee. Further, unlike simulation, formal verification does not require any testbenches or stimuli for triggering a system. More precisely, formal verification is a mathematical way of proving a system satisfies a set of properties. *Formal verification* methods such model checking [4] are being taken seriously in the recent few years by several large hardware and software design companies such as Intel, IBM, Motorola, and Microsoft, which goes to show the importance and practicality of such methods for real-time embedded systems and SoC designs. For the above reasons, we will thus employ a widely popular formal verification method called *model checking* for the verification of safety-critical systems that are formally modeled.

In the course of developing a model-based verification method for safety-critical systems, several issues are encountered as detailed in the following. First and foremost, we need to decide how to model safety-critical systems. Our decision is to adopt Safecharts [4] as our models. Safecharts are a variant of Statecharts, especially for use in the specification and the design of safety-critical systems. The objective of the model is to provide a sharper focus on safety issues and a systematic approach to deal with them. This is achieved in Safecharts by making a clear separation between functional and safety requirements. Other issues encountered in designing the formal verification methodology for model-based safety-critical systems are as follows:

1. How to transform Safecharts into a semantically equivalent *Extended Timed Automata* (ETA) model that can be accepted by traditional model checkers? How can the transformation preserve the safety semantics in Safecharts?
2. What are the properties that must be specified for model checking Safecharts?
3. Basic states in Safecharts have a risk relation with each other specifying the comparative risk/safety levels. How do we represent such information in ETA for model checking?
4. Safecharts have safety loopholes due to the lack of synchronization mechanisms. A motivational example will be given in Section 4.4.

5. The current semantics of Safecharts states that mutual exclusion of resource usages can be achieved through priority. This is clearly insufficient as priorities cannot ensure mutual exclusion.

The remaining portion is organized as follows. Section 2 describes the background form our model including a comparison between conventional validation, such as simulation and testing, and formal verification. Basic definitions used in our work are given in Section 3. Section 4 will formulate each of our solutions to solving the above described problems in formally verifying safety-critical systems modelled by Safecharts. The article is concluded and future research directions are given in Section 6.

2 Related Work

A commonly-used method to demonstrate the safety of a system is *proof by contradiction* [13]. In this method, we assume that the unsafe states, identified by hazard analysis, can be reached by executing the program. We then systematically analyze the code and show that the pre-conditions for a hazardous state are contradicted by the post-conditions of all program paths leading to that state. If this is the case, the initial assumption of an unsafe state is incorrect. If this is repeated for all identified hazards, then the system is safe. However, to find and list all possible hazards of safety-critical systems is difficult. For example, a system may fail due to an unpredicted hazard that may lead to a serious tragedy. This is not allowed, and that's why we propose a more formal method to verify safety-critical systems that are modeled by Safecharts and verified by model checking as introduced in the rest of this Section.

Safecharts [4] is a variant of Statecharts intended exclusively for safety-critical systems design. With two separate representations for functional and safety requirements, Safecharts brings the distinctions and dependencies between them into sharper focus, helping both designers and auditors alike in modeling and reviewing safety features. Safecharts incorporates ways to represent equipment failures and failure handling mechanisms and uses a safety-oriented classification of transitions and a safety-oriented scheme for resolving any unpredictable nondeterministic pattern of behavior. It achieves these through an explicit representation of risks posed by hazardous states by means of an ordering of states and a concept called *risk band*. Recognizing the possibility of gaps and inaccuracies in safety analysis, Safecharts do not permit transitions between states with unknown relative risk levels. However, in order to limit the number of transitions excluded in this manner, Safecharts provides a default interpretation for relative risk levels between states not covered by the risk ordering relation, requiring the designer to clarify the risk levels in the event of a disagreement and thus improving the risk assessment process.

Timed Computation Tree Logic (TCTL) is a *timed* extension of the well-known temporal logic called *Computation Tree Logic* (CTL) which was proposed by Clarke and Emerson in 1981. We will use TCTL to specify system properties that are required to be satisfied.

Model checking [2,3,12] is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically. The

procedure normally uses an exhaustive search of the state space of a system to determine if some specification is true or not. Given sufficient resources, the procedure will always terminate with a *yes/no* answer. Moreover, it can be implemented by algorithms with reasonable efficiency, which can be run on moderate-sized machines. The process of model checking includes three parts: modeling, specification, and verification. *Modeling* is to convert a design into a formalism accepted by a model checking tool. Before verification, *specification*, which is usually given in some logical formalism, is necessary to state the properties that the design must satisfy. The *verification* is completely automated. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and a reapplication of the model checking algorithm.

Our safety-critical system model and its model checking procedures are implemented in the *State-Graph Manipulators* (SGM) model checker [14], which is a high-level model checker for both real-time systems as well as systems-on-chip modeled by a set of timed automata.

3 System Model, Specification, and Model Checking

Before going into the details of how Safecharts are used to model and verify safety-critical systems, some basic definitions and formalizations are required as given in this Section. Both Safecharts and their translated ETA models will be defined. TCTL and model checking will also be formally described.

Definition 1. Statechart
Statecharts are a tuple $\mathcal{F} = (\mathcal{S}, \mathcal{T}, \mathcal{E}, \Theta, \mathcal{V}, \Phi)$, where $\mathcal{S}$ is a set of all states, $\mathcal{T}$ is a set of all possible transitions, $\mathcal{E}$ is a set of all events, Θ is the set of possible types of states in Statecharts, that is, $\Theta = \{AND, OR, BASIC\}$, $\mathcal{V}$ is a set of integer variables, and $\Phi ::= v \sim c \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi_1$, in which $v \in \mathcal{V}$, $\sim \in \{<, \leq, =, \geq, >\}$, c is an integer, and Φ_1 and Φ_2 are predicates. Let $\mathcal{F}_i$ be an arbitrary state in $\mathcal{S}$. It has the general form:

$$\mathcal{F}_i = (\theta_i, C_i, d_i, T_i, E_i, l_i)$$

where:

- θ_i : the type of the state $\mathcal{F}_i$; $\theta_i \in \Theta$.
- C_i : a finite set of direct substates of $\mathcal{F}_i$, referred to as *child states* of $\mathcal{F}_i$, $C_i \subseteq \mathcal{S}$.
- d_i : $d_i \in C_i$ and is referred to as the *default state* of $\mathcal{F}_i$. It applies only to OR states.
- T_i : a finite subset of $\mathcal{T}$, referred to as explicitly *specified transitions* in $\mathcal{F}_i$.
- E_i : the finite set of events relevant to the specified transitions in T_i; $E_i \subseteq \mathcal{E}$.
- l_i : a function $T_i \rightarrow \mathcal{E} \times \Phi \times 2^{E_i}$, labelling each and every specified transition in T_i with a triple, 2^{E_i} denoting the set of all finite subsets of E_i. □

Given a transition $t \in \mathcal{T}$, its label is denoted by $l(t) = (e, fcond, a)$, written conventionally as $e[fcond]/a$. $e, fcond$ and a in the latter, denoted also as $trg(t) =$

$e, con(t) = fcond$, and $gen(t) = a$, represent respectively the triggering event, the guarding condition and the set of generated actions.

Definition 2. **Safechart**
Safecharts $\mathcal{Z}$ extend Statecharts by adding a safety-layer. States are extended with a risk ordering relation and transitions are extended with safety conditions. Given two comparable states s_1 and s_2, a risk ordering relation $\sqsubseteq$ specifies their relative risk levels, that is $s_1 \sqsubseteq s_2$ specifies s_1 is safer then s_2. Transition labels in Safecharts have an extended form:

$$e[fcond]/a[l,u)\Psi[G]$$

where e, $fcond$, and a are the same as in Statecharts. The time interval $[l, u)$ is a real-time constraint on a transition t and imposes the condition that t does not execute until at least l time units have elapsed since it most recently became enabled and must execute strictly within u time units. The expression $\Psi[G]$ is a safety enforcement on the transition execution and is determined by the safety clause G. The safety clause G is a predicate, which specifies the conditions under which a given transition t must, or must not, execute. Ψ is a binary valued constant, signifying one of the following enforcement values:

- *Prohibition* enforcement value, denoted by $\nwarrow$. Given a transition label of the form $\nwarrow [G]$, it signifies that the transition is forbidden to execute as long as G holds.
- *Mandatory* enforcement value, denoted by $\nearrow$. Given a transition label of the form $[l, u) \nearrow [G]$, it indicates that whenever G holds the transition is forced to execute within the time interval $[l, u)$, even in the absence of a triggering event. □

The Safecharts model is used for modeling safety-critical systems, however the model checker SGM can understand only a flattened model called *Extended Timed Automata* [6] as defined in the following.

Definition 3. **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$,$c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. □

Let $B(C, D)$ be the set of all mode predicates over C and D.

Definition 4. **Extended Timed Automaton**
An *Extended Timed Automaton* (ETA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, L_i, \chi_i, T_i, \psi_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M_i$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, L_i is a set of synchronization labels, and $\epsilon \in L_i$ is a special label that represents asynchronous behavior (i.e. no need of synchronization), $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode, $T_i \subseteq M_i \times M_i$ is a set of transitions, $\lambda_i : T_i \mapsto L_i$ associates a synchronization label with a transition, $\tau_i : T_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and $\rho_i : T_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values. □

A system state space is represented by a *system state graph* as defined in Definition 5.

Definition 5. System State Graph
Given a system $\mathcal{S}$ with n components modelled by $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, L_i, \chi_i, T_i, \psi_i, \tau_i, \rho_i)$, $1 \leq i \leq n$, the system model is defined as a state graph represented by $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n = \mathcal{A}_\mathcal{S} = (M, m^0, C, D, L, \chi, T, \psi, \tau, \rho)$, where:

- $M = M_1 \times M_2 \times \ldots \times M_n$ is a finite set of system modes, $m = m_1.m_2.\ldots.m_n \in M$,
- $m^0 = m_1^0.m_2^0.\ldots.m_n^0 \in M$ is the initial system mode,
- $C = \bigcup_i C_i$ is the union of all sets of clock variables in the system,
- $D = \bigcup_i D_i$ is the union of all sets of discrete variables in the system,
- $L = \bigcup_i L_i$ is the union of all sets of synchronization labels in the system,
- $\chi : M \mapsto B(\bigcup_i C_i, \bigcup_i D_i)$, $\chi(m) = \wedge_i \chi_i(m_i)$, where $m = m_1.m_2.\ldots.m_n \in M$.
- $T \subseteq M \times M$ is a set of system transitions which consists of two types of transitions:
  - *Asynchronous transitions*: for each $e \in T$, $\exists i, 1 \leq i \leq n, e_i \in T_i$ such that $e_i = e$
  - *Synchronized transitions*: $\exists i, j, 1 \leq i \neq j \leq n, e_i \in T_i, e_j \in T_j$ such that $\psi_i(e_i) = (l, in)$, $\psi_j(e_j) = (l, out)$, $l \in L_i \cap L_j \neq \emptyset$, $e \in T$ is synchronization of e_i and e_j with conjuncted triggering conditions and union of all transitions assignments (defined later in this definition)
- $\psi : T \mapsto L \times \{in, out\}$ associates a synchronization label and a direction of communication with a transition, which represents a blocking signal that was synchronized, except for $\epsilon \in L$, ϵ is a special label that represents asynchronous behavior (i.e. no need of synchronization),
- $\tau : T \mapsto B(\bigcup_i C_i, \bigcup_i D_i)$, $\tau(e) = \tau_i(e_i)$ for an asynchronous transition and $\tau(e) = \tau_i(e_i) \wedge \tau_j(e_j)$ for a synchronous transition, and
- $\rho : T \mapsto 2^{\bigcup_i C_i \cup (\bigcup_i D_i \times \mathcal{N})}$, $\rho(e) = \rho_i(e_i)$ for an asynchronous transition and $\rho(e) = \rho_i(e_i) \cup \rho_j(e_j)$ for a synchronous transition. □

Definition 6. Safety-Critical System
A *safety-critical system* is defined as a set of *resource* components and *consumer* components. Each component is modeled by one or more Safecharts. If a safety-critical system $\mathcal{H}$ has a set of resource components $\{R_1, R_2, \ldots, R_m\}$ and a set of consumer components $\{C_1, C_2, \ldots, C_n\}$, $\mathcal{H}$ is modeled by $\{\mathcal{Z}_{R_1}, \mathcal{Z}_{R_2}, \ldots, \mathcal{Z}_{R_m}, \mathcal{Z}_{C_1}, \mathcal{Z}_{C_2}, \ldots, \mathcal{Z}_{C_n}\}$, where $\mathcal{Z}_X$ is a Safechart model for component X. Safecharts $\mathcal{Z}_{R_i}$ and $\mathcal{Z}_{C_j}$ are transformed into corresponding ETA $\mathcal{A}_{R_i}$ and $\mathcal{A}_{C_j}$, respectively. Therefore, $\mathcal{H}$ is semantically modeled by the state graph $A_{R_1} \times \ldots \times A_{R_m} \times A_{C_1} \times \ldots \times A_{C_n}$ as defined in Definition 5. □

For both hardware and software systems, a property or requirement can be specified in some *temporal logic*. The SGM model checker chooses TCTL as its logical formalism, as defined below.

Definition 7. Timed Computation Tree Logic (TCTL)
A *timed computation tree logic* formula has the following syntax:

$$\phi ::= \eta \mid EG\phi' \mid E\phi' U_{\sim c}\phi'' \mid \neg\phi' \mid \phi' \vee \phi'',$$

where η is a mode predicate, ϕ' and ϕ'' are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$, and c $\in \mathcal{N}$. $EG\phi'$ means there is a computation from the current state, along which ϕ' is always true. $E\phi' U_{\sim c}\phi''$ means there exists a computation from the current state, along which ϕ' is true until ϕ'' becomes true, within the time constraint of $\sim c$. Shorthands like EF, AF, AG, AU, $\wedge$, and $\rightarrow$ can all be defined [5]. □

Definition 8. Model Checking
Given a Safechart $\mathcal{Z}$ that represents a safety-critical system and a TCTL formula, ϕ, expressing some desired specification, model checking [2,3,12] verifies if $\mathcal{Z}$ satisfies ϕ, denoted by $\mathcal{Z} \models \phi$.

Model checking can be either explicit using a labeling algorithm or symbolic using a fixpoint algorithm. *Binary Decision Diagram* (BDD) and *Difference Bound Matrices* (DBM) are data structures used for Boolean formulas and clock zones [3], respectively. □

4 Model Checking Safecharts

Safecharts have been used to model safety-critical systems, but the models have never been verified. In this work, we propose a method to verify safety-critical systems modelled by Safecharts. Our target model checker is *State Graph Manipulators* (SGM) [14,6], which is a high-level model checker for both real-time systems, as well as, Systems-on-Chip modelled by a set of extended timed automata. As mentioned in Section 1, there are several issues to be resolved in model checking Safecharts.

Basically, a system designer models a safety-critical system using a set of Safecharts. After accepting the Safecharts, we transform them into ETA, while taking care of the safety characterizations in Safecharts, and then automatically generate properties corresponding to the safety constraints. The SGM model checker is enhanced with transition priority, synchronization, and urgency. Resource access mechanisms in Safecharts are also checked for satisfaction of modeling restrictions that prevent violation of mutual exclusion. Finally, we input the translated ETA to SGM to verify the safety-critical system satisfies functional and safety properties. Each of the issues encountered during implementation and the corresponding solutions are detailed in the rest of this section.

4.1 Flattening Safecharts and Safety Semantics

Our primary goal is to model check Safecharts, a variant of Statecharts. However, Safecharts cannot be accepted as system model input by most model checkers, which can accept only flat automata models such as the extended timed automata (ETA) accepted by SGM. As a result, the state hierarchy and concurrency in Safecharts must be transformed into semantically equivalent constructs in ETA. Further, besides the functional layer, Safecharts have an extra safety layer, which must be transformed into equivalent modeling constructs in ETA and specified as properties for verification.

There are three categories of states in Safechart: OR, AND, and $BASIC$. An OR-state, or an AND-state, consists generally of two or more substates. Being in an AND-state means being in all of its substates simultaneously, while being in an OR-state means being in exactly one of its substates. A $BASIC$-state is translated into an ETA *mode*. The translations for OR-states and AND-states are performed as described in [8].

Safety Semantics . The syntax for the triggering condition and action of a transition in Safecharts is:

$$e[fcond]/a[l,u)\Psi[G],$$

where e is the set of triggering events, $fcond$ is the set of guard conditions, a is the set of broadcast events, $[l,u)$ is the time interval specifying the time constraint, Ψ means the execution conditions for safety constraints, and G is the set of safety-layer's guards. In Safecharts, $e[fcond]/a$ appears in the *functional* layer, while $[l,u)\Psi[G]$ may appear in the *safety* layer. The two layers of Safecharts can be integrated into one in ETA as described in the following. However, we need to design three different types of transitions [1]:

- *Eager Evaluation* (ϵ) : Execute the action as soon as possible, i.e. as soon as a guard is enabled. Time cannot progress when a guard is enabled.
- *Delayable Evaluation* (δ) : Can put off execution until the last moment the guard is true. So time cannot progress beyond a *falling edge* of guard.
- *Lazy Evaluation* (λ) : You may or may not perform the action.

The transition condition and assignment $e[fcond]/a[l,u)\Psi[G]$ can be classified into three types as follows:

1. $e[fcond]/a$
   There is no safety clause on a transition in Safechart, thus we can simply transform it to the one in ETA. We give the translated transition a *lazy* evaluation (λ).
2. $e[fcond]/a \upharpoonleft [G]$
   There is *prohibition* enforcement value on a transition *t*. It signifies that the transition *t* is forbidden to execute as long as G holds. During translation, we combine them as $e[fcond \wedge \neg G]/a$. We give the translated transition a *lazy* evaluation (λ). The transformation is shown in Fig. 1.
3. $e[fcond]/a[l,u) \upharpoonright [G]$
   There is *mandatory* enforcement value on a transition *t*. Given a transition label of the form $e[fcond]/a[l,u) \upharpoonright [G]$, it signifies that the transition is forced to execute within $[l,u)$ whenever G holds. We translate functional and safety layers into a transition t_1 and a path t_2, respectively. t_1 represents $e[fcond]/a$, which means t_1 is enabled if the triggering event *e* occurs and its functional conditional $fcond$ is true. We give t_1 a *lazy* evaluation (λ). Path t_2 is combined by two transitions, t_ϵ and t_δ. Transition t_ϵ is labeled $[G]/timer := 0$, where *timer* is a clock variable used for the time constraint, and we give t_ϵ an *eager* evaluation (ϵ). When G holds, t_ϵ executes as soon as possible, and t_ϵ's destination is a newly added mode, named *translator(t)*. t_δ's source is *translator(t)*, and its destination is *t*'s destination. t_δ's guard is $[timer \geq l \wedge timer < u]$. However, we give t_δ a *delayable* evaluation (δ), which means it can put off execution until the last moment the guard is true. The procedure of translation is shown in Fig. 2.

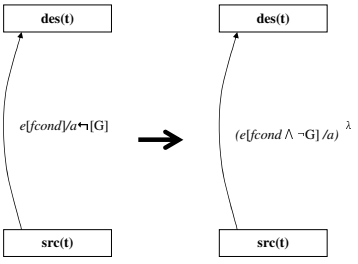

Fig. 1. Transformation of prohibition evaluation

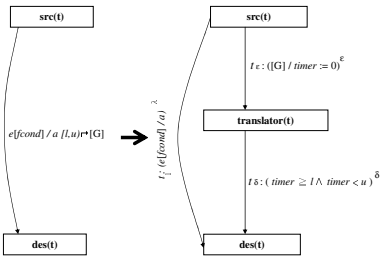

Fig. 2. Transformation of mandatory evaluation

4.2 Property Specification for Safecharts

In the safety-layer of Safecharts, there are two types of safety conditions on a transition, one is *prohibition* and the other is *mandatory*. After parsing the Safechart models of a safety-critical system, corresponding properties are automatically generated without requiring the user to specify again. Such properties are used to verify if the safety-layers work or not. As described in the following, to ensure that the safety constraints are working, two categories of properties are generated automatically for model checking.

1. $AG((src(t) \wedge G) \rightarrow \neg EX(des(t)))$
   If a transition *t* in Safechart has prohibition condition $\upharpoonleft [G]$ in its safety-layer, it means that such transition is forbidden to execute as long as G holds. As shown in Fig. 1, *t*'s source is *src(t)*, and its destination is *des(t)*. Due to $\upharpoonleft [G]$, *src(t)* is not allowed to translate to *des(t)* as long as G holds. If such property is tenable in our system state graph, which means that there is no transition from *src(t)* to *des(t)* executing whenever G holds, then we can know that the safety-critical system won't become dangerous while G holds.
2. $AG((src(t) \wedge G \rightarrow \neg EX(\neg translator(t)))$ and $AG(translator(t) \wedge timer < u)$
   If a transition *t* in Safechart has $[l, u) \upharpoonright [G]$ in its safety-layer, it means that such transition is enabled and forced to execute within $[l, u)$ whenever G holds. As mentioned in former sections, we add two transitions for the safety-layer's behavior, namely t_ϵ and t_δ, and a mode, *translator(t)* between them.
   From Fig. 2, when G holds, t_ϵ must be executed as soon as possible due to its eager evaluation and the next active mode must be $translator(t)$. Moreover, we know that if the mode $translator(t)$ is active, then the next active state must be $des(t)$ within the time limit $timer \geq l \wedge timer < u$. If this constraint is violated, then the safety condition will not be satisfied.

4.3 Transition Priority

When modeling safety-critical systems, it is important to eliminate any non-deterministic behavior patterns of the system. Non-determinism arises if the triggering expressions of two transitions starting from a common state are simultaneously fulfilled. Because of its concern with safety-critical systems, Safecharts remove non-determinism in all cases except when there is no safety implication. In a Safechart model, a list of *risk*

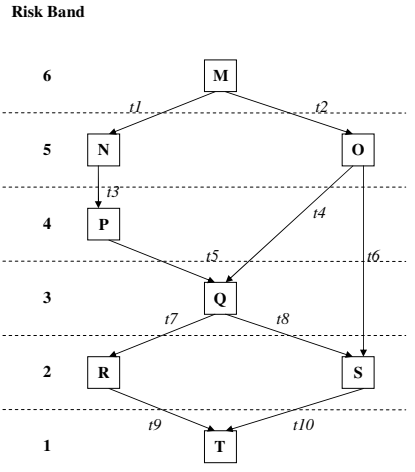

Fig. 3. Risk graph with risk band

relation tuples is used to represent a *risk graph* [10]. Non-comparable conditions may still exist in a risk graph. An example [9] is given in Fig. 3, where, relative to other states, the state O may have received less attention in the risk assessment, resulting in it becoming non-comparable with other states in the graph, namely, the states N and P. Consequently, Safecharts do not allow any transition between them, for instance, a transition such as $O \rightsquigarrow P$.

As a solution to the above problem, the authors of Safecharts proposed *risk band* [9], which can be used to enumerate all states in a risk graph to make precise their relative risk relations that were not explicitly described. To adopt this method, we implemented transition priorities based on the risk bands of a transition's source and destination modes. According to a list of risk relations, we can give modes different risk bands, as depicted in Fig. 3, where the maximum risk band, max_{rb}, is 6. We assign each transition a priority as follows:

$$pri(t) = max_{rb} - (rb_{src(t)} - rb_{des(t)}),$$

where $pri(t)$ is the priority assigned to transition t, $rb_{src(t)}$ is the risk band of transition t's source mode, and $rb_{des(t)}$ is the risk band of transition t's destination mode. Moreover, the smaller the value of $pri(t)$ is, the higher is the priority of transition t. In Fig. 3, $pri(t4)$ is 4, and $pri(t6)$ is 3. Obviously, when $t4$ and $t6$ are both enabled, $t6$ will be executed in preference to $t4$. With risk bands, we can give a transition leading to a lower risk band state a higher priority.

For implementing transition priorities into the SGM model checker, the triggering guards of a transition are modified as follows [1].

$$\tau'(t_i) = \tau(t_i) \wedge \bigwedge_{j \geq i} \neg\tau(t_j),$$

where $\tau(t_i)$ and $\tau(t_j)$ are the guard conditions of transitions t_i and t_j, $j \geq i$ means that t_j's priority is higher than or equal to t_i's, and $\tau'(t_i)$ is the modified guard condition of

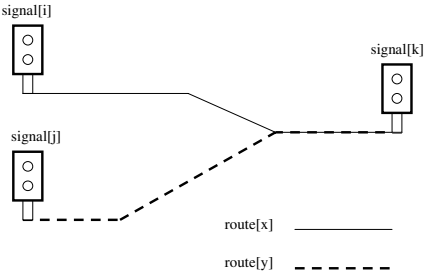

Fig. 4. Routes and signals

t_i. This application results in allowing t_i executed only if there is no enabled transition t_j which has priority over t_i.

4.4 Transition Urgency and Synchronization

Safecharts have a safety/security loophole due to the lack of synchronization mechanisms. A motivation example is the railway signal system illustrated in Fig. 4, where a route can be requested, evaluated, and set when the required signals on a route are operating without faults and are in the free state. The Safecharts for route[x] and signal[i] are given in Fig. 5 and Fig. 6, respectively. A signal breaks down when either its lamp or its sensor fails. The signal mode is changed from OPR to FAULTY upon receiving either ε_l (lamp fail event) or ε_s (sensor fail event). However, this mode change is not synchronized with ε_l or with ε_s, thus in-between these two actions, a route could have been evaluated and set, although the signal is faulty which is not detected because the signal's mode has not been changed as yet. Due to this lack of synchronization, safety loopholes exists in Safecharts. The route once set could allow a train to pass through a faulty signal endangering human lives as well as damaging properties. Safety-based resolution of non-determinism as proposed in [9,10,11] also does not solve this synchronization issue because non-determinism is resolved only among transition of the same Safechart and not among different Safecharts.

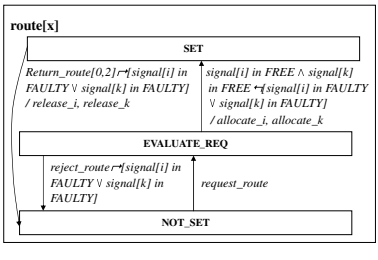

Fig. 5. Safechart for route[x]

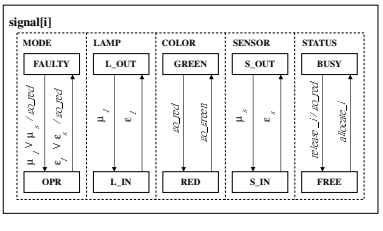

Fig. 6. Safechart for signal[i]

To solve the above problem, we propose the use of *transition urgency* as detailed in the following.

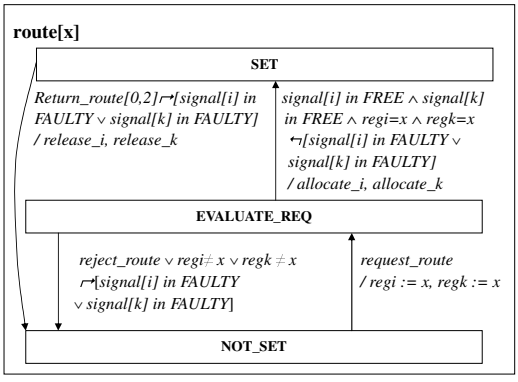

Fig. 7. Safechart for route[x] with mutual exclusion

Transition Urgency. As mentioned in Section 4.1, there are three types of transitions: *eager evaluation* (ϵ), *delayable evaluation* (δ), and *lazy evaluation* (λ). Transitions concerned with safety are given eager evaluation (ϵ) to ensure that when some malfunction or repair events happen, they can be executed first to reflect the correct status of a real-time system. In the railway signalling example, the model designer must give the transition with malfunctioning event ε an eager evaluation ϵ. As soon as the event ε occurs, the *signal*'s MODE is immediately changed to FAULTY. Thus *route* will not acquire the usage of *signal*, due to the safety-layer prohibiting guard condition $signal[i]$ in FAULTY $\vee$ $signal[k]$ in FAULTY.

To eliminate the safety-loopholes in Safecharts and avoid errors due to the loopholes, the above method must be used to extend Safecharts. We have implemented the proposed method in our Safecharts verification framework based on SGM.

4.5 Resource Access Mechanisms

Safecharts model both consumers and resources. However, when resources must be used in a mutually exclusive manner, a model designer may easily violate the mutual exclusion restriction by simultaneous checking and discovery of free resources, followed by their concurrent usages. A motivation example can be observed in the railway signalling system as illustrated in Fig. 4, Fig. 5, and Fig. 6, where signal[k] must be shared in a mutually exclusive way between route[x] and route[y]. However, each route checks if signal[k] is free and finds it free, then both route will be SET, assuming all signals are fault-free. This is clearly a modeling trap that violates mutually exclusive usages of resources. A serious tragedy could happen in this application example as two intersecting routes are set resulting in perhaps a future train collision.

From above we know that when consumers try to acquire resources that cannot be used concurrently, it is not safe to check only the status of resources. We need some kind of model-based mutual exclusion mechanism. A very simple policy would be like Fischer's mutual exclusion protocol [7]. For each mutually exclusive resource, a variable is used to record the id of the consumer currently using the resource. Before the consumer uses the resource, it has to check if the variable is set to its id. Fig. 7 is a corrected vari-

ant of the route Safechart from Fig. 5. When *route[id]* transits into EVALUATE_REQ, it sets variable *reg* to its id. When *route[x]* tries to transit into the SET mode to acquire the usage of *resource*, it needs to check if *reg* is still its id. If *reg* is still *x*, then *route[x]* acquires the usage of the resource. Other mechanisms such as atomic test-and-set performed on a single asynchronous transition can also achieve mutual exclusion.

5 Application Examples

The proposed model-based verification methodology for safety-critical systems was applied to several variants of the basic railway signaling system, which was illustrated in Fig. 4. The basic system was used to check the feasibility of the proposed methodology. The variants were used to check the scalability and efficiency of the methodology.

The basic system consists of two routes: route[x] and route[y], where route[x] requires signal[i] and signal[k], and route[y] requires signal[j] and signal[k]. The numbers and sizes of the Safecharts and the generated ETA are given in Table 1. As illustrated in Figs. 8 and 9, for each route Safechart, one ETA is obtained and for each signal Safechart, five ETA are generated. Thus, in the full system consisting of 5 Safecharts, 17 ETA are generated. It can be observed that the number of ETA modes, 40, is lesser than the number of Safecharts states, 56. The reason for this reduction is that hierarchical states do not exist in ETA. The synchronization and the mutual exclusion issues were both solved for this railway system as described in Sections 4.4 and 4.5, respectively.

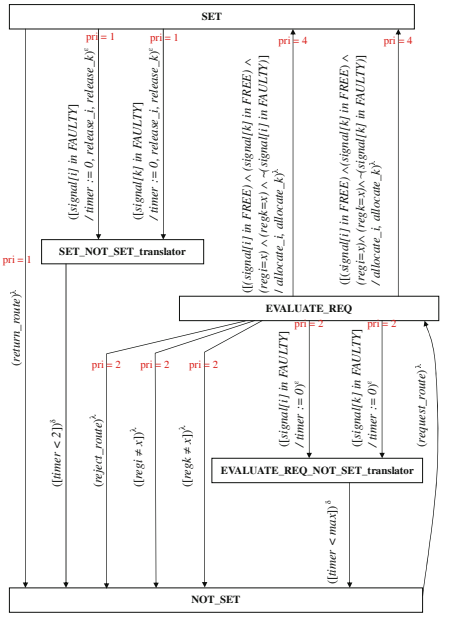

Fig. 8. ETA for route[x]

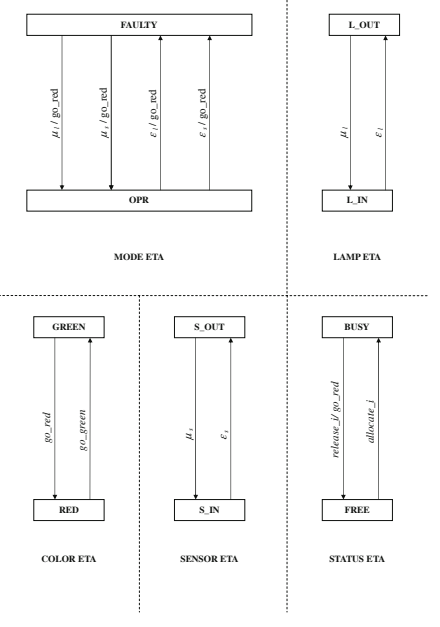

Fig. 9. ETA for signal[i]

Table 1. Results of the Railway Signaling System

Component		Safecharts			ETA		
Name	#	#	$\|\mathcal{S}\|$	$\|\mathcal{T}\|$	#	$\|M\|$	$\|T\|$
route	2	1	4	4	1	5	13
signal	3	1	16	10	5	10	12
full system	5	5	56	38	17	40	62

Table 2. Results of Application Examples

	System		Safecharts			ETA			$\|\phi\|$	Issues Solved		Time	Mem
	$\|R\|$	$\|S\|$	#	$\|\mathcal{S}\|$	$\|\mathcal{T}\|$	#	$\|M\|$	$\|T\|$		Sync	ME	(μs)	(MB)
A	2	3(1)	5	56	38	17	40	62	16	3	1	230	0.12
B	2	4(1)	6	72	48	22	50	78	19	4	1	292	0.12
C	2	4(2)	6	72	48	22	50	82	22	4	2	337	0.13
D	3	4(1)	7	76	52	23	55	87	24	4	1	326	0.14
E	3	5(2)	8	92	62	28	65	111	33	5	2	515	0.14
F	4	5(1)	9	96	66	29	70	112	32	5	1	634	0.14

$|R|$: total num of routes, $|S|$: total num of signals (num of shared signals), $|\phi|$: num of properties generated. Sync: Num of synchronization issues solved, ME: Num of mutual exclusion issues solved

A number of variants of the basic railway signaling system were used for validating the proposed method's scalability and efficiency. Varying the number of routes and the number of signals in each route increases the complexity and the concurrency of the system. However, we can observe from the verification results in Table 2 that the amount of time and memory expended for verification do not increase exponentially and are very well acceptable. The number of properties to be verified also increase and thus their automatic generation is also a crucial step for successful and easily accessible verification of safety critical systems. The number of issues solved imply how the proposed solutions in this work are significant for the successful verification of complex systems modeled by Safecharts.

6 Conclusions

Nowadays, safety-critical systems are becoming more and more pervasive in our daily lives. To reduce the probability of tragedy, we must have a formal and accurate methodology to verify if a safety-critical system is safe or not. We have proposed a formal method to verify safety-critical systems. Our methodology can be applied widely to safety-critical systems with a model-driven architecture. We hope our methodology can have some real contribution such as making the world a safer place along with the development of science and technology.

References

1. K. Altisen, G. Gössler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23:55–84, 2002.
2. E.M. Clarke and E.A. Emerson. Design and sythesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Logics of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer Verlag, 1981.
3. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
4. H. Dammag and N. Nissanke. Safecharts for specifying and designing safety critical systems. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 78–87, October 1999.
5. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for reat-time systems. In *Proceedings of the IEEE International Conference on Logics in Computer Science (LICS)*, pages 394–406, June 1992.
6. P.-A. Hsiung and F. Wang. A state-graph manipulator tool for real-time system specification and verification. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, October 1998.
7. K.G. Larsen, B. Steffen, and C. Weise. Fischer's protocol revisited: A simple proof using model constraints. In *Hybrid System III*, volume 1066 of *LNCS*, pages 604–615, 1996.
8. L. Lavazza, editor. *A methodology for formalising concepts underlying the DESS notation*. ITEA, December 2001.
9. N. Nissanke and H. Dammag. Risk bands - a novel feature of Safecharts. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE)*, pages 293–301, October 2000.
10. N. Nissanke and H. Dammag. Risk ordering of states in Safecharts. In *Proceedings of the 19th International Conference on Computer Safety, Reliability, and Security*, volume 1943 of *LNCS*, pages 395–405. Springer Verlag, October 2000.
11. N. Nissanke and H. Dammag. Design for safety in safecharts with risk ordering of states. *Safety Science*, 40(9):753–763, December 2002.
12. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer Verlag, 1982.
13. I. Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2001.
14. F. Wang and P.-A. Hsiung. Efficient and user-friendly verification. *IEEE Transactions on Computers*, 51(1):61–83, January 2002.