Viability Checking and Requirements Completion: How Constraint Programming Helps State Machines in Performance Requirements Engineering

Gefei Zhang

Hochschule für Technik und Wirtschaft Berlin, Germany
`gefei.zhang@htw-berlin.de`

Abstract

UML state machines are widely used for modeling software behavior. We present a proposal of constructing constraint systems out of state machines with performance requirements. Our approach takes into account hierarchical states and parallel regions, and facilitates viability checking. Also, missing requirements can be automatically deduced, which is particularly valuable in practice. Our approach thus provides a simple yet useful means for model driven engineering of requirements engineering.

1. Introduction

UML state machines are widely used for modeling software behavior. They are considered simple and intuitive, and even deemed to be "the most popular modeling language for reactive components" [2].

While it is straight-forward to incorporate notations for performance modeling, like such defined by MARTE [7], in UML state machines, their semantics may get rather involved. In particular, the most important feature of UML state machines is that they are *concurrent*. More than one states may be active simultaneously, and more than one actions may be executed simultaneously. Therefore, the dependencies of the states to each other and their impact on the system performance may get obscured. Moreover, it would be desirable to have some mechanism to check the viability of the performance requirements and detect conflicts, if any. Also, if for some states and actions their performance requirements are still missing, it would be helpful to deduce those automatically.

We propose a simple concept of constructing a *constraint system* out of a state machine, which allows us to employ a constraint solver to check the viability of the requirements, and deduce missing requirements if necessary. Our proposal takes into account the concurrent nature of state machines and the states' dependencies on each other.

The rest of the paper is organized as follows: in Sect. 2 we give a brief overview of the syntax and informal semantics of UML state machines. In Sect. 3 we introduce model elements for performance modeling in state machines. In Sect. 4 we show how to obtain a constraint system out of the state machine, as well as how to check the viability of the requirements and how to deduce missing requirements. Related work is discussed in Sect. 5 before conclusions are drawn and some future work is sketched in Sect. 6.

2. UML State Machines

A UML state machine provides a model for the behavior of an object or component. Figure 1 shows a state machine modeling the behavior of a system that automatically manages the repair of a car in case of an accident:[1], the system is Idle. Upon a car-repair request (request), it enters the state Running, where it first charges the credit card of the driver, orders a garage for the repair, and then, in the state OrderTruckAndCar, it orders in parallel, i.e., simultaneously, a tow truck and a rental car. For illustration reasons we suppose it is necessary first to reserve a tow truck and a rental car before you can order them. When the orders are sent, the state machine gets idle again.

2.1. Syntax and Informal Semantics

We briefly review the syntax and semantics of UML state machines according to the UML specification [6] by means of Fig. 1. A UML state machine con-

[1]This example was adapted from [9].

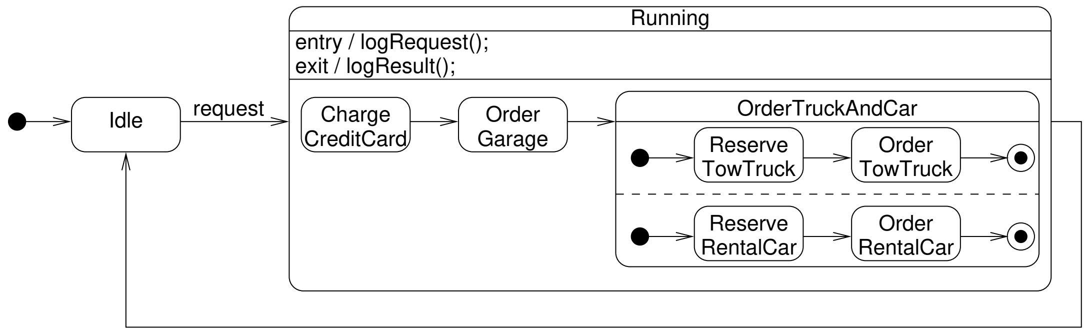

Figure 1. Example: UML state machine

sists of *regions* which contain *vertices* and *transitions* between vertices. A vertex is either a *state*, which may show hierarchically contained regions; or a *pseudo state* regulating how transitions are compound in execution. Transitions are triggered by *events* and describe, by leaving and entering states, the possible state changes of the state machine. The events are drawn from an *event pool* associated with the state machine, which receives events from its own or from different state machines.

A state is *simple* if it contains no regions (two examples are Idle and ChargeCreditCard in Fig. 1); a state is *composite* if it contains at least one region; a composite state is said to be *orthogonal* if it contains more than one region, visually separated by dashed lines (such as Running and OrderTruckAndCar). A region may contain state and other vertices. A state, if not on the top-level itself, must be contained in exactly one region. A composite state and all the states directly or recursively contained in it thus build a tree.

A state may also contain an entry and an exit action (and a do action, which is not relevant for this paper). The entry action is executed every time the state is entered and gets active, the exit action is executed every time the state is left and gets inactive. For example, in Fig. 1, every time Running is activated, the action logRequest is executed, and every time Running is deactivated, logResult is executed.

Transitions are triggered by events (e.g. request) may show guards and may have effects to be executed when a transition is fired. For simplicity, our running example does not include guards or effects. Completion transitions do not have an explicit trigger, instead they are triggered by an implicit *completion event* emitted when a state completes all its internal activities. In Fig. 1, all transitions leaving the states other than Idle are triggered by completion events.

An *initial* pseudo state, depicted as a filled circle (●), represents the starting point for the execution of a region. A *final* state, depicted as a circle with a filled circle inside (◉), represents the completion of its containing region; if all top-level regions of a state machine are completed then the state machine terminates. In the very simple example of Fig. 1, the only pseudo states are the three *init* vertices . For simplicity, we do not consider the other kinds of *pseudo states* (entry and exit points, shallow and deep history, junction, choice, and terminate). Final states are not pseudo states in the UML, but states.

At run time, states get activated and deactivated as a consequence of transitions being fired. The active states at a stable step in the execution of the state machine form the active *state configuration*. Active state configurations are hierarchical: when a composite state is active, then exactly one state in each of its regions is also active; when a substate of a composite state is active, so is the containing state too. The execution of the state machine can be viewed as different active state configurations getting active or inactive upon the state machine receiving events. Note that for any given region, at most one direct substate of the region can be active at any given time, because a state configuration can contain at most one direct substate of the region.

For example, a workflow of the above state machine handling an accident, given in terms of active state configurations, might be as follows: (Idle), (Running, ChargeCreditCard), (Running, OrderGarage), (Running, OrderTruckAndCar, ReserveTowTruck, ReserveRentalCar), (Running, OrderTruckAndCar, OrderTowTruck, ReserveRentalCar), (Running, OrderTruckAndCar, OrderTowTruck, OrderRentalCar), (Running, OrderTruckAndCar, OrderTowTruck, ◉), (Running, OrderTruckAndCar, ◉, ◉), (Idle).

Usually, it is assumed that a state remains active for some time and a transition is finished as soon as its effect is finished. That is, if a transition does not have an effect, then it does not need any time to get finished.

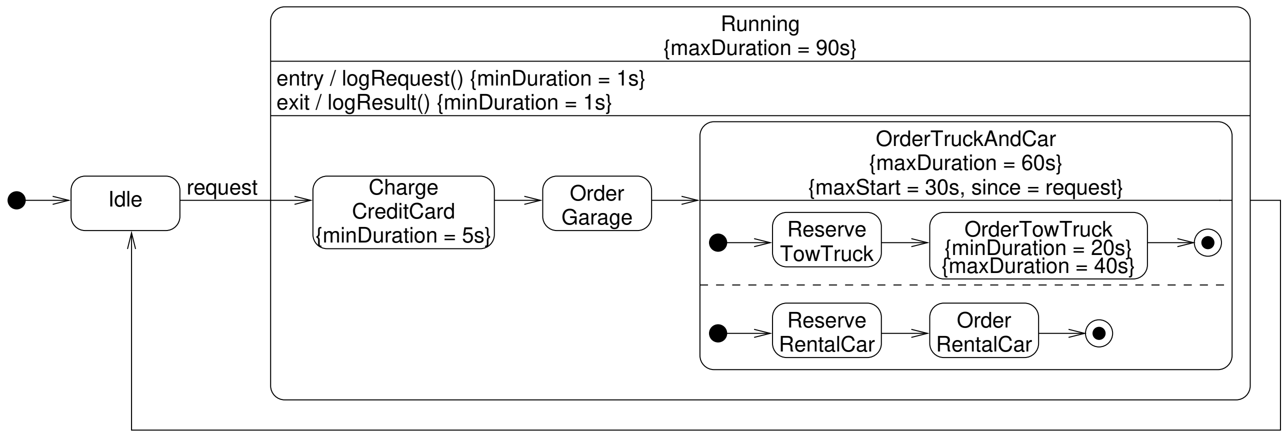

Figure 2. Example: UML state machine with performance modeling

Otherwise it takes as much time as its effect does.

In Fig. 1, since the transitions do not have any effect, they are supposed to get completed immediately. On the other hand, the states may stay active for some time. For example, the duration of OrderGarage being active is the same as the time the for the system to send a request to the garage and receives a confirmation.

3. Modeling of Performance Requirements

We use a standard extension mechanism, *tagged values*, of the UML to define a simple extension of state machines to incorporate performance modeling. We introduce the following tagged values:

- maxDuration to model the maximal time a state can remain active or an action can take, and
- minDuration to model the minimal time a state can remain active or an action can take.
- maxStart to model the maximum time allowed to elapse for a state to get activated since the occurrence of some event, (to be specified by tagged value since, see below).
- since to specify the event upon whose occurrence the start point for the timing of maxStart.

Of these tagged values, maxDuration and maxStart (together with since) model requirements for the system, and minDuration provides a constraint of the system.

Figure 2 shows an example of using these tagged values to model performance requirements of our vending machine. We require that the state Running remain active for a maximum of 90 seconds, that is, the complete service, starting from charging the credit card through ordering a tow truck and a rental car, must be completed within this time. We also require that OrderTruckAndCar remain active for 60 seconds at most and therefore the two parallel regions for reserving and order a tow truck respectively a rental car not take more than this time. We know that OrderTowTruck will take at least 20 seconds (minDuration), and require that it remain active for at most 40 seconds.

We also gave OrderTruckAndCar the tagged values maxStart = 30s and since = request. This way we model the requirement that the state OrderTruckAndCar get activated no later than 30 seconds after the event occurred.

For ChargeCreditCard, we define a minDuration of 5 seconds, modeling the time needed to charge the credit card. Finally, we define a minDuration = 1s for the entry and exit actions of Running, modeling the time the logging actions take.

Note that the requirements are not complete yet: many states do not carry a tagged value which models its active time. For example, it is not intuitively clear how much time is allowed for OrderGarage to be active or how much time the activity of ordering a garage is allowed to take.

4. Viability Checking and Completion of Requirements by Constraint Programming

Our state machine, enhanced with performance modeling, is an excellent starting point to check if the requirements are viable or if they conflict each other. Also, it allows us to deduce requirements that are missing or unknown at first. For example, in Fig. 2, is it satisfiable at all that OrderTruckAndCar remain active for no more than 60 seconds? How much time can the state ReserveTowTruck take at most? We show how to use constraint programming [1] to answer these questions.

4.1. Constraint Programming

Constraint programming is a declarative programming paradigm. A constraint program defines a set of variables, each having a certain domain, and a set of constraints imposed on the variables. The constraint solver can then automatically determine if there exists a valuation of the variables to satisfy the constraints or the system is unsatisfiable.

For example[2], the following code (written in the language *MiniZinc*[3]) defines three variables $1 \leq x \leq 20$, $9 \leq y \leq 11$, and $150 \leq z \leq 161$, as well as a constraint $x \cdot y = z$.

```
var 1..20:    x;
var 9..11:    y;
var 150..161: z;

constraint x*y = z;
```

If we ask MiniZinc to solve this constraint system by `solve satisfy;` we will get a solution $x = 17, y = 9, z = 153$.

Moreover, many constraint solvers can also find optimised solutions. For example, we can ask for a solution of the above constraint system in which y is maximized by `solve maximize y;` and get the solution $x = 14, y = 11, z = 154$, or ask for a z-maximized solution by `solve maximize z;` and get $x = 16, y = 10, z = 160$.

4.2. Variables

In order to obtain a constraint system out of a state machine like the one in Fig. 2, we first define a constant m to model the default maximal duration of a state being active. In this example, we assume $m = 90s$, that is, a state that does not have an explicit maxDuration can remain active for 90 seconds at most. We then introduce a variable for each state and each entry or exit action. The variable should represent the duration of the state being active or the time needed by the action. Its name is the name of the state or the action. Its domain is the range $0..m$.

For example, we define thirteen variables for the state machine in Fig. 2: eleven for the eleven states (including the two final states), and two for the two actions logRequest and logResult. Recall that final states are also states and therefore cannot be ignored when we are defining variables for states.

[2]Adapted from [1].

[3]`https://www.minizinc.org/`, accessed on 2025-05-31.

4.3. Constraints

The constraints are defined as follows:

- For each state s with minDuration $= d$, define a constraint $s \geq d$; for each state s with maxDuration $= d$, define a constraint $s \leq d$.

  For example, for the state ChargeCreditCard in Fig. 2, we define a constraint $\mathit{ChargeCreditCard} \geq 5$, and for OrderTowTruck we have two constraints: $\mathit{OrderTowTruck} \geq 20$ and $\mathit{OrderTowTruck} \leq 40$.

- For each entry or exit action a with minDuration $= d$, define a constraint $a \geq d$; for each entry or exit action a with maxDuration $= d$, define a constraint $a \leq d$.

  For example, for the entry action logRequest in Fig. 2 we define $\mathit{logRequest} \geq 1$ and for logResult we define $\mathit{logResult} \geq 1$.

- For each sequential path of states contained in a composite state, define a constraint that the sum of the values of the states on the path is less than or equals to the maxDuration of the composite state minus its entry and exit actions, if any.

  For example, for the upper region of the state OrderTruckAndCar, we define $\mathit{ReserveTowTruck} + \mathit{OrderTowTruck} + \mathit{Final}_1 \leq \mathit{OrderTruckAndCar}$, and for the lower region we define $\mathit{ReserveRentalCar} + \mathit{OrderRentalCar} + \mathit{Final}_2 \leq \mathit{OrderTruckAndCar}$. Note that since in Fig. 2 the final states are not named, we refer to them as Final_1 and Final_2.

  Moreover, for the state Running, which contains one region, we define $\mathit{ChargeCreditCard} + \mathit{OrderGarage} + \mathit{OrderTruckAndCar} \leq \mathit{Running} - \mathit{logRequest} - \mathit{logResult}$.

- All regions of the same composite state have the same time of execution. That is, the sum of the time of the states in every region being active is the same.

  For example, the two regions of OrderTruckAndCar should have the same duration, hence $\mathit{ReserveTowTruck} + \mathit{OrderTowTruck} + \mathit{Final}_1 = \mathit{ReserveRentalCar} + \mathit{OrderRentalCar} + \mathit{Final}_2$.

- For each state s with maxStart $= d$, define a constraint that the sum of the path from the target state of its since transition to the stats is less than or equals to d.

For example, for OrderTruckAndCar we define a constraint $\mathit{ChargeCreditCard} + \mathit{OrderGarage} \leq 30$.

The complete constraint system for the state machine of Fig. 2, using the syntax of MiniZinc, is given in `https://people.f4.htw-berlin.de/~zhangg/car.mzn`.

4.4. Validation

Equipped with the constraint system, we are now in a position to validate the performance requirements.

Satisfiability. It is important to know if the requirements are satisfiable or conflicting. To this end, we simply ask the constraint to solve our constraint system.

For example, we can ask MiniZinc to do this by `solve satisfy;` Then MiniZinc will respond that the requirements given in Fig. 2 are satisfiable and provide a set of possible values of the variables, i.e., the duration of the states and the actions.

Deduction of Requirements. Often the requirements are not complete, i.e., some states or actions do not have an explicit maxDuration. It may be important to deduce the performance requirements for these states. To this end, we simply ask the constraint solver to maximize the value for the state of action we are interested in.

For example, deducing the maximal time the task of reserving a tow truck can take, i.e., how long the state ReserveTowTruck is allowed to remain active, can be achieved by the goal `solve maximize ReserveTowTruck;`. MiniZinc will give an answer `ReserveTowTruck = 40`, which is actually the maximal time for this state to be active, since otherwise it would not be possible to hold $\mathit{Running} \leq 60$. Another example would be to calculate the maximal active time of OrderGarage by `solve maximize OrderGarage;`, which reveals that in this design, ordering a garage is allowed to take 25 seconds at most (since otherwise OrderTruckAndCar would not start in time).

5. Related Work

Model-driven performance engineering has been investigate by quite a few researcher. Process Algebra and Markov Chains are used in PEPA [8]. Performance of component-based software systems is considered by [3], just to name two examples. An overview is give in [5].

Concentrating on the UML, modeling time requirements is standardized in MARTE [7], which contains a lot more elements than our modeling elements (see Sect. 3). In fact, our modeling elements can be seen as a small subset of those defined in MARTE. We plan to adopt MARTE's notation and extend the translation to constraint system in order to analyse more complicated requirements.

In the UML [6], part of the semantics of state machines is left unspecified intentionally as *variation points*. The runtime model of this paper is based on that defined in Hugo/RT [4], which provides means for model checking UML state machines with time information.

Deriving a constraint system out of UML state machines to check the viability of its performance requirements and deducting new requirements is to the author's best knowledge a novel contribution of this paper. The latter may be particularly valuable in practical software development.

6. Conclusions and Future Work

We have presented a concept of constructing a constraint system out of a UML state machine with performance requirements. The constraint system can be used to check the requirements' viability, and also to deduce requirements that are not known at first. These requirements may be of important when the state machine is implemented by code.

As future work, we plan to extend this approach to handle more involved state machines, in particular those including cycles. Tool support to automate this approach is also planned.

References

[1] Krzysztof Apt. *Principles of Contraint Programming*. Cambridge University Press, 2009.

[2] Doron Drusinsky. *Modeling and Verification Using UML Statecharts*. Elsevier, 2006.

[3] Lucia Happe, Barbora Buhnova, and Ralf H. Reussner. Stateful component-based performance models. *Softw. Syst. Model.*, 13(4):1319–1343, 2014.

[4] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.

[5] Manoj Nambiar, Ajay Kattepur, Gopal Bhaskaran, Rekha Singhal, and Subhasri Duttagupta. Model Driven Software Performance Engineering: Current Challenges and Way Ahead. *SIGMETRICS Perform. Evaluation Rev.*, 43(4):53–62, 2016.

[6] OMG. Unified Modeling Language, Version 2.5.1. Specification, Object Management Group, 2017. `https://www.omg.org/spec/UML/2.5.1/PDF`, Accessed on 2025-05-31.

[7] OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Specification, Object Management Group, 2024. `https://www.omg.org/spec/MARTE/1.3/PDF`, Accessed on 2025-05-31.

[8] Mirco Tribastone, Adam Duguid, and Stephen Gilmore. The PEPA eclipse plugin. *SIGMETRICS Perform. Eval. Rev.*, 36(4):28–33, 2009.

[9] Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias M. Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-Based Development of Service-Oriented Systems. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *Proc.* 26th *IFIP WG 6.1 Int. Conf. FORTE'06*, volume 4229 of *Lect. Notes Comp. Sci.*, pages 24–45. Springer, 2006.