Iris Groher, Thomas Vogel (Hrsg.): Software Engineering 2023 Workshops,
54 Digital Library, Gesellschaft für Informatik, 2023

Updates at Runtime for Cyber Physical Systems. A Game Theoretic Approach[1]

Janis Kröger[2], Martin Fränzle[3]

Abstract: Updates are becoming increasingly important in the field of cyber physical systems in order to increase the lifetime of these systems and to save resources and costs. One way to achieve this is to update the system at runtime. This paper describes our vision of a game-theoretic approach to determine when an update is possible at runtime, taking into account the system behavior. This ensures that the system behaves safely at all points in time. For this purpose we use so-called timed games and synthesize a strategy under which the update can be performed. We sketch the approach and illustrate its application on an automotive example of an autopilot.

Keywords: Game Theory; Software Updates; Update Process; Operating Modes

1 Introduction

Nowadays, it is increasingly important that systems can be used for as long as possible. This is due to a variety of reasons. One of the main reasons is the availability of resources: A long lifetime can save hardware resources as well as production and development costs. In order to make a long period of use possible, updates offer the possibility to keep the system up-to-date with respect to e.g. legal requirements, performance adaptations, and to allow the exchange of faulty components and extensions of functionality.

We encounter updates almost daily in a wide variety of areas of normal life. They are mainly used to increase the lifetime of the system, to fix bugs or to add new functionality. For example, they have already become the norm for mobile phones and consoles. As Guissouma et al discuss in [Gu18] (runtime) updates also play an important role in the automotive sector and increasingly include over-the-air updates (at runtime). These updates concern subsystems less critical systems for e.g. navigation or entertainment, but also safety-critical systems like an emergency braking assistant.

In order to keep the safety-critical systems up to date, they are often only updated during idle times or in the service centre due to their safety-critical nature. Updating the system at

[1] This work has been funded by the *Federal Ministry of Education and Research* (BMBF) as part of *AutoDevSafeOps* (01IS22087Q).

[2] Carl von Ossietzky Universität Oldenburg, Department für Informatik, Ammerländer Heerstraße 114-118, 26129 Oldenburg, Germany janis.kroeger@uni-oldenburg.de

[3] Carl von Ossietzky Universität Oldenburg, Department für Informatik, Ammerländer Heerstraße 114-118, 26129 Oldenburg, Germany martin.fraenzle@uni-oldenburg.de

doi:10.18420/se2023-ws-08

runtime, when the system operates in the field, saves the user money and time and increases safety, since then the vehicle can be updated directly in the field which reduces the time until the update.

When updating safety-critical systems at runtime, it is important that the system is always operational. In addition to correct functionality, the timing aspect plays a crucial role in safety-critical systems. The safety-critical systems, e.g. an emergency braking assistant, has to react fast enough to prevent collisions. For this reason, we focus on the timing properties of updates of safety-critical systems in this paper. In particular we are interested in answering “When is it possible to update a system at runtime, such that it remains fully functional during the update?”

An update installation process usually has to comply with time constraints because an update installation process blocks certain memory and computing resources in order to install the update. In order not to hinder the functioning of the system, the update installation process must follow a schedule that ensures that the subsystem being updated is not in use during the time allocated for the update. Otherwise, the subsystem could enter an unpredictable state. The schedule for update installation aims to update the subsystem while the (full) functionality of the overall system remains. We propose a game theoretic approach where the two players in a timed game are the (i) update installation process and (ii) the system in its environment. The structured behaviour of the system in different states and different operating modes, as well as the division of the update into different phases is used for this.

The paper is structured as follows. Section 2 presents existing related work that already deals with updates. Section 3 describes the basic concepts to understand the game theoretic approach presented in Section 4. In Section 5, the game-theoretic approach is demonstrated with an example use case in UPPAAL TIGA. Finally, a conclusion is given in Section 6.

2 Related Work

The following literature already deals with updating systems. Different approaches are taken to verify or perform updates. The present literature represents on the one hand the cornerstone of this work by showing different procedures of update processes, on the other hand also other formal methods are pointed out to the updating of system.

In [Be20], Bebawy et al. are dealing with the verification of updates. They first categorize updates into different update types and present a concept for incremental verification of these different update types for cyber physical systems. They consider the impact of the update on other components and use the contract based design methodology.

In [Na16] and [Na18], the authors deal with the problem of updating a controller during execution in reactive systems. They present a general approach for the specification of correctness criteria for the update as well as a technique to synthesize a controller. The controller handles the transition from the old to the new specification. The results are validated against several use cases. In contrast to our approach, no explicit timing aspects are considered here.

In [Ho16], the authors present a method for substituting lab-based verification with formal analysis techniques to determine whether an update can be performed safely. They consider different aspects such as timing and functional correctness. The authors use a contract-based method and illustrate this with an example from the automotive sector.

In [Ca05], the authors describe an on-the-fly algorithm to solve games in the sense of game theory based on timed automata with respect to reachability and safety properties. This algorithm has been implemented in the tool UPPAAL TIGA [Be06], so that it is possible to model so-called timed games and solve them with respect to reachability and safety properties. It is an extension of the model checking tool UPPAAL. With the help of UPPAAL TIGA, the present approach is demonstrated.

In [RK22], we present a framework in which they structure an update process into different phases which must be completed. In addition, they assign different responsibilities to different stakeholders, which must be complied with in the respective phases of the process. By ensuring that these responsibilities are fulfilled by the stakeholders and taking into account the system behaviour, an update can be made predictable so that an update can be guaranteed to be performed within the specified time span of the update process. For the specification of the responsibilities and for the system behaviour they use the contract based design methodology. This paper lays the grounds part of the approach in this paper.

3 Basic Concepts

Game theory is an established methodology to model and analyse situations with interactive decisions [MZS20]. Situations of decision taking are modelled as a game where the decision takers are called players. Moreover, game theory provides us with algorithms that determine winning strategies, i.e. the algorithms determine what a player has to do so that he will be guaranteed to achieve the specified winning condition. A timed game is a special type of game that can be used to model games over time. Timed automata are used for modelling [Ca05] by extending finite automata with finite set of real-valued clocks.

Due to the timing properties of a system considered in this paper, timed games are an important and appropriate method to model interaction of the system and the update installation process. The synthesis of winning strategies for the update player will achieve the winning condition of doing the update in time without interrupting the system functionality. A synthesis of these strategies with respect to timed games can be performed by the tool UPPAAL TIGA. In the following section, the basics of these methods are described, which are necessary for understanding the approach of this paper.

Timed Automata For the definition of a timed automaton [Al99] we follow the definition of [Ca05]. A timed automaton is a tuple $\mathcal{T} = (L, l_0, A, X, E, Inv)$ where L is a finite set of locations, l_0 is a set of initial locations, A is a set of actions, X is a finite set of real-valued variable clocks, $E \subseteq L \times A \times 2^X \times \mathcal{B}(X) \times L$ is a finite set of transitions and Inv is a set of clock constrains mapped to a location: $L \longrightarrow \mathcal{B}(X)$. $\mathcal{B}(X)$ defines a set of clock constrains $x \sim k$ where $x \in X$, $k \in \mathbb{Z}$ and $\sim \in \{<, \leq, =, >, \geq\}$. $S \subseteq L \times \mathbb{R}^X$ defines a set of states.

Timed Game Automata A timed game automaton $\mathcal{G}$ is a timed automaton where the actions A are separated into controllable (A_c) and uncontrollable (A_{uc}) actions, $A = A_c \cup A_{uc}$ [Ca05]. Figure 1 shows an example of a timed game automata in UPPAAL TIGA. The controllable actions are shown solid and the uncontrollable actions are shown dashed.

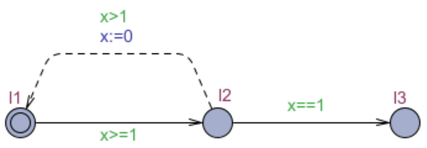


Fig. 1: Example of a timed game automaton in UPPAAL TIGA

Strategy A strategy $\mathcal{S}$ is a function that, for any location and clock evaluation, formalises how a player should act to reach a specific goal. For a synthesis of a strategy in, e.g., UPPAAL TIGA, as shown in Section 4.5, the goal to be achieved can specified in Computation Tree Logic (CTL). The goal of a player can be to win the game by reaching a certain location in the game graph (e.g. timed game automata in timed games). On the other hand, the goal can also be not to lose the game, i.e. not to reach a certain location. The strategy's behaviour can result in taking a controllable transition or in doing nothing, which is denoted by ϵ. A strategy is defined as follows: $\mathcal{S} : L \times \mathbb{R}^X_{\geq 0} \longrightarrow A_c \cup \epsilon$. A winning strategy is a strategy of a player with which the corresponding player always wins the game. In a two-player zero-sum game there can be several different winning strategies, although generally only one player has one or more winning strategies [MZS20].

Modes Modes are a good methodology to structure a system, to reduce complexity and to express the behaviour of a system in different situations or scenarios [Kr21]. For each mode of the system a different functionality or timing behaviour can be described. Modes can also be used as a safety mechanism, so that a basic functionality can be provided when failures are detected. A mode M can be considered as a set of states. For an easier understanding and for the modelling of a system, in the following a mode of the system is represented by a location of the timed automata.

4 Timed Game Approach

To answer the question of whether a system can be updated and how a possible update schedule might look like, we use a game theoretic approach. Since the goal is to install the update on the system, we model the update installation process as a timed game, wherein player one is the updater and the system including its environment is player two. We will synthesize a winning strategy for player one and consider player two as uncontrollable. The game is won, when the updater fully completes the update installation. Our approach is anchored in a model based design process, such that the devised game and the strategy synthesis can be done at design time, i.e. offline. There is no need to synthesize an update schedule at runtime, since the synthesized strategy is guaranteed to be applicable at runtime.

We first take a look at some limitations before we described the behaviour of both players and the strategy synthesis in more detail.

4.1 Limitations

As mentioned in the introduction, we stipulate that the system must not use a subsystem being updated during its update. Furthermore, we assume that the system and the update are non-cooperative. This assumption covers the worst case where the system cannot cooperate due to external events. We use the concept of *modes* [Kr21],[RK22] to structure a system. A mode of a system is the smallest part that can be updated. In our timed automata model a mode is represented by a location. Thus, the update aims to change a single location.

By a decomposition of the system a finer modelling can take place. The behavior of the system is completely known to the update. Based on this assumptions, we differentiate between two scenarios:

Scenario 1): In the first scenario, the system is updated without being actively transferred to a mode in which the update can be performed. The system decides itself when to change its mode. Since the updates know the system behaviour, it predicts when the update can be performed. The updater has no authority to command when the system changes modes.

In the following we concentrate an Scenario 1). We consider Scenario 2) as our next step in future work.

Scenario 2): In the second scenario, the update can activly transfer the system into a certain mode or keep it in a specific mode while the update is running.

4.2 Updater

To model the behavior of the updater, we structure an update installation process into different phases following [RK22]. The phases of the process are represented by the modes of our updater. Thus, the behavior of the updater represents valid sequences of update phases. When the updater reaches a final state the update installation process is been completed.

According to [RK22] each mode has a minimum and a maximum execution time. These time specifications become clock constraints in the automaton (invariants and guards). As we noted in [RK22], an update is not necessarily completed in one pass. It is quite possible that an update is performed step by step. This is captured via the timed automaton of the update.

For simplicity, we here consider only three modes *Off*, *Run*, *End*. The updater is initially in the phase *Off*. In this phase nothing happens at first. At any time, which can also be deterministically defined, the installation of the update can be started, i.e. the updater switches to the *Run* mode where the update is performed. When the installation is finished, it switches to the *End* mode. In this mode, the updater performs no action; the update has been installed. Initially, there is no provision for aborting or pausing the process. The required time for the installation of the update is defined via the clock constraints of *Run*.

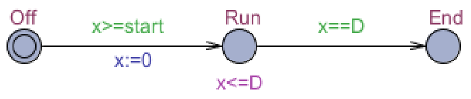


Fig. 2: Timed automaton of updater

Figure 2 shows an example of a timed automaton of the updater, where x is a clock. $start$ specifies a constant representing the time after the installation of the update can be started and D specifies the required time that is needed to perform the update. It is also possible to specify the exact time by the value *start* after which the update should start. By the exact determination of $start$ and D the maximum duration $D_{max} = start + D$ of the entire update process results. With the limitation of the time D_{max} it can be defined, after which time the update must be completed at the latest.

For the approach that is used in this paper, it is not relevant whether the update is installed by the target system itself or remotely by an external user. Furthermore, it is also not relevant whether the entire update process is performed completely on the target system or whether some phases are first processed externally before the update is installed.

4.3 System

The behaviour of the system can result e.g. from a formal specification, an informal specification like a simple system description, or be derived from a system implementation. When the system is modelled using a timed automaton, it should be divided into modes in such a way that updatable units, in the form of modes, are characterized. These modes are represented as locations in the automaton.

Let us consider the timed automaton of a system S with three modes $l1, l2, l3$which are represented by the respective locations in Figure 3 as an example. The system is initially in mode *l1* and remains here for at least 1 and at most 2 time units. If the guard is fulfilled and the invariant is not violated, it changes to mode *l2* and gives out an a. In mode *l2* the system stays exactly for a duration of 2 time units after which it changes to mode *l3* and gives out a b. From there it changes back to mode *l1* after a minimum time of 3 time units. It gives out a c. In every transition of the system, the clock x is reset.

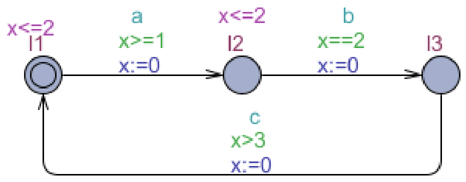


Fig. 3: Example of a timed automaton of a system

4.4 Strategy synthesis

As stated in Section 4.1 we will restrict our strategy synthesis to scenario 1 (see Section 4.1). A consideration of the second scenario will follow in future work. Since we are only concerned with strategy synthesis, which answers the question “if and when a certain update can be performed”, we are only interested in knowing a single winning strategy, from which we can derive a possible installation schedule.

As already mentioned, we synthesize a winning strategy for the updater. The game is won, when the updater is guaranteed to reach its final states. Reaching final states means that the update installation process will be fully completed.

In order to synthesize a possible winning strategy, it is first necessary to establish winning conditions which represents the goals of the player that must be fulfilled in order to win the game.

From the updater perspective, it wins the game if it has run through all modes and reached the *End* mode represented by the *End* location of the timed automaton. For a formal specification and for later use in UPPAAL TIGA, we specify the winning condition in CTL. We obtain the following CTL formula for the above condition:

$$A <> Updater.End.$$

The safety condition of running the update for one of the system modes is that the system must not reach that respective mode until the update is ended. In the following we will refer to the mode that is updated as M_{update}. That means the system may not change into the updating mode M_{update} as long as the updater is in mode *Run*. Equally, the updater may change only into the mode *Run*, if it is guaranteed that the system does not change into M_{update} in the maximum execution time of mode *Run* (see Figure 2). An occurrence of this combination results in losing the game. It is thus necessary to specify the condition, under which the updater does not lose, in addition to the condition for reaching the final state. As CTL formula this is specified as follows:

$$A[\,]\neg(Updater.Run\ and\ System.M_{update}).$$

4.5 UPPAAL TIGA

UPPAAL Tiga is a tool for modelling and solving timed games and is an extension of UPPAAL [Be06, Be07]. Using UPPAAL TIGA, we intend to illustrate our approach with a concrete example. We can first model the timed automata of the two players, establish corresponding winning conditions, check whether they are satisfied, and if so, synthesize a possible strategy. In UPPAAL TIGA it is not necessary to explicitly model the composition of the individual automata. It is sufficient to model the automata of the individual players and to specify the controllable and uncontrollable transitions. The specification of winning conditions in UPPAAL TIGA is in CTL. They are based on the regular expressions of UPPAAL. One of these expression is “Strict Reachability with Avoidance (Until)”. It is described by the expression

$$control : A[not\ lose\ U\ win].$$

This means that the set of states *win* must be reached and the set of states *lose* must be avoid. Through these winning condition, based on the presented approach, we can already express that the update must reach its final state to win the game and that certain states or state combinations must not occur. Through this specification, both necessary conditions described in Section 4.4 are expressed together.

5 Application Example

To demonstrate our approach, we consider the driving assistance system of an autopilot. To do this, we modelled the autopilot and the updater in UPPAAL TIGA synthesize a winning strategy for the updater.

The autopilot allows the driver to take his hands off the steering wheel for a short period of time and the vehicle takes over the driving. The autopilot specification can be summarized as follows, assuming D_1, D_2 are constant values of the autopilot. If the driver does not touch the steering wheel after at most D_1 seconds the autopilot gives a warning requesting the driver to touch the steering wheel. If the driver still does not touch the steering wheel after D_2 seconds the autopilot executes an emergency braking manoeuvre for safety reasons. The autopilot has the modes *Off*, *On*, *Warn* and *Brake*.

Figure 4a shows the timed automaton for the autopilot as modelled in UPPAAL TIGA. Its locations represent the corresponding modes. The (uncontrollable) transitions are shown as solid arcs. The times D_1 (until the warning is issued) and D_2 (until braking is initiated) have both been set to 15 time units, as some value. The event of "the driver touches the steering wheel before the warning" is represented as a self-loop at the location On. This transition can be taken non-deterministically at any time when the guard g_1 "at least one time unit has passed" is satisfied. Likewise the event of "the driver touches the steering wheel before the breaking" is represented as a transition from the location *Warn* to location *On*. This transition can be taken non-deterministically any time after 15 clock unit, i.e. guard g_2 "at least 15 time units have passed". Guards g_1 and g_2 rule out zeno behaviour, i.e. they rule out that an infinite number of transitions can be executed in zero time. The transitions are shown as uncontrollable (dashed connections).

The timed automaton for the updater sketched in Section 4.2 is shown in Figure 4b. Its transitions are controllable and hence the arcs are solid. The update installation may start the earliest after ten time units (modelled by the guard $g_3 := z > 10$). Once it is started, the installation must be completed after ten time units. Hence the updater changes to *End*. Note that, for the sake of a small example, we consider here a deterministic update installation process (e.g. the installation is not allowed to pause), although our approach straightforwardly covers also more general installation processes.

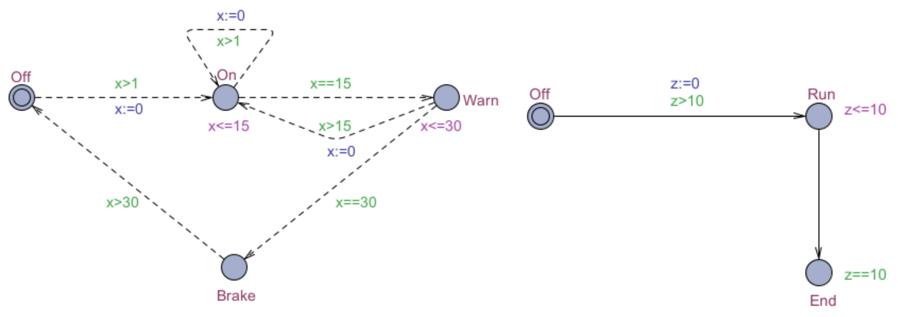
We chose the minimum of ten time units (cf. g_3) as an offset for the update installation in order to capture that the update is not available directly at the start of the system. In addition, this ensures that the autopilot can already be in *On* mode for a sufficiently long time. UPPAAL TIGA is thus forced to consider all state combinations in the strategy synthesis. Without the offset, the synthesized winning strategy will always install the update right at the start.

In both automata we have $x, z \in X$, where X is a set of real-valued clock variables

In our example we want to update the mode *Warn* of the autopilot. The updater must reach the *End* mode to win the game. Furthermore we require, as we described in Section 4.1, that the autopilot must not be in *Warn* mode or switch to *Warn* mode while the update installation is in progress, i.e. updater is in *Run* mode. To express these winning conditions formally in UPPAAL TIGA we use CTL as described in Section 4.5. This results in the following expression as the winning condition Ψ:

$$control : A[not(Updater.Run\ and\ Autopilot.Warn)\ U\ Updater.End]$$

UPPAAL TIGA now checks whether an winning strategy exists satisfying Ψ irrespective



(a) System Autopilot (b) Updater

Fig. 4: Timed Automata in UPPAAL TIGA

of any environmental disturbances. In our example, Ψ is satisfied, i.e. a winning strategy can be synthesized by the verifier that satisfies our winning condition. A possible winning strategy $\mathcal{S}_w$ as synthesized by UPPAAL TIGA is given in Figure 5. For reasons of brevity, only those actions are listed in the strategy in which transitions are taken. In all other state combinations and clock values, the ϵ action applies and the updater waits.

A significant point of the strategy $\mathcal{S}_w$ is pointed out in the following. Line 16 of Figure 5 refers to the state combination: autopilot is in state *On* and the updater is in state *Off*. As a reminder, the autopilot stays in *On* mode for at most 15 time units, i.e. the driver does not touch the steering wheel. According to $\mathcal{S}_w$ the updater may take the transition to *Run* in this state combination, if at least ten time units have passed (Update.z > 10) and the autopilot is less than five time units in the location *On* (Autopilot.x < 5) since the last driver action. This condition ensures that the updater has already left the *Run* location with a maximum execution time of 10 time units when the autopilot changes from *On* to *Warn* directly after a minimum execution time of 15 time units.

6 Conclusion

In this paper we describe a game-theoretic approach to determine an update schedule that can be applied at runtime. For this we modelled th update installation process of a system as timed game. In the end we demonstrated the approach using the example of an autopilot by synthesizing a suitable winning strategy with UPPAAL TIGA.

The approach is initially very simplified by fixed transition limits on the side of the update installation process as well as by various restrictions described in Section 4.1. In the future, the approach is to be extended to enable a synthesis that is as restriction-free as possible. Furthermore, in addition to the first scenario, scenario two (described in Section 4.1) will also be addressed. It would also be conceivable to consider incremental updates, as described in [Be20], or updates that have to be executed iteratively, as shown in [RK22].

```
Strategy to win:

State: ( Autopilot.Off Updater.Run )
When you are in (1<Autopilot.x && Updater.z==10),
take transition Updater.Run->Updater.End { z == 10, tau, 1 }

State: ( Autopilot.Off Updater.Off )
When you are in (1<Autopilot.x && 10<Updater.z),
take transition Updater.Off->Updater.Run { z > 10, tau, z := 0 }

State: ( Autopilot.Brake Updater.Off )
When you are in (30<=Autopilot.x && 10<Updater.z),
take transition Updater.Off->Updater.Run { z > 10, tau, z := 0 }

State: ( Autopilot.On Updater.Off )
When you are in (10<Updater.z && Autopilot.x<5),
take transition Updater.Off->Updater.Run { z > 10, tau, z := 0 }

State: ( Autopilot.Brake Updater.Run )
When you are in (30<Autopilot.x && Updater.z==10),
take transition Updater.Run->Updater.End { z == 10, tau, 1 }

State: ( Autopilot.On Updater.Run )
When you are in (Updater.z==10 && Autopilot.x<=15),
take transition Updater.Run->Updater.End { z == 10, tau, 1 }
```

Fig. 5: Winning strategy $\mathcal{S}_w$ of example

Bibliography

[Al99] Alur, Rajeev: Timed automata. In: International Conference on Computer Aided Verification. Springer, pp. 8–22, 1999.

[Be06] Behrmann, Gerd; Cougnard, Agnes; David, Alexandre; Fleury, Emmanuel; Larsen, Kim Guldstrand; Lime, Didier: UPPAAL-Tiga: Timed games for everyone. In: Nordic Workshop on Programming Theory (NWPT'06). 2006.

[Be07] Behrmann, Gerd; Cougnard, Agnes; David, Alexandre; Fleury, Emmanuel; Larsen, Kim G; Lime, Didier: Uppaal tiga user-manual. Aalborg University, 2007.

[Be20] Bebawy, Yosab; Guissouma, Houssem; Vander Maelen, Sebastian; Kröger, Janis; Hake, Georg; Stierand, Ingo; Fränzle, Martin; Sax, Eric; Hahn, Axel: Incremental contract-based verification of software updates for safety-critical cyber-physical systems. In: 2020 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE, pp. 1708–1714, 2020.

[Ca05] Cassez, Franck; David, Alexandre; Fleury, Emmanuel; Larsen, Kim G; Lime, Didier: Efficient on-the-fly algorithms for the analysis of timed games. In: International Conference on Concurrency Theory. Springer, pp. 66–80, 2005.

[Gu18] Guissouma, Houssem; Klare, Heiko; Sax, Eric; Burger, Erik: An empirical study on the current and future challenges of automotive software release and configuration management. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, pp. 298–305, 2018.

[Ho16] Holthusen, Sönke; Quinton, Sophie; Schaefer, Ina; Schlatow, Johannes; Wegner, Martin: Using multi-viewpoint contracts for negotiation of embedded software updates. arXiv preprint arXiv:1606.00504, 2016.

[Kr21] Kröger, Janis; Koopmann, Björn; Stierand, Ingo; Tabassam, Nadra; Fränzle, Martin: Handling of operating modes in contract-based timing specifications. In: International Conference on Verification and Evaluation of Computer and Communication Systems. Springer, pp. 59–74, 2021.

[MZS20] Maschler, Michael; Zamir, Shmuel; Solan, Eilon: Game theory. Cambridge University Press, 2020.

[Na16] Nahabedian, Leandro; Braberman, Victor; D'Ippolito, Nicolas; Honiden, Shinichi; Kramer, Jeff; Tei, Kenji; Uchitel, Sebastian: Assured and correct dynamic update of controllers. In: 2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). IEEE, pp. 96–107, 2016.

[Na18] Nahabedian, Leandro; Braberman, Victor; D'Ippolito, Nicolás; Honiden, Shinichi; Kramer, Jeff; Tei, Kenji; Uchitel, Sebastián: Dynamic update of discrete event controllers. IEEE Transactions on Software Engineering, 46(11):1220–1240, 2018.

[RK22] Rakow, Astrid; Kröger, Janis: Roles and Responsibilities for a Predictable Update Process–A Position Paper. In: International Conference on Verification and Evaluation of Computer and Communication Systems. Springer, pp. 17–26, 2022.