Chapter 9

A Game-theoretic Approach to Behavioural Visualisation

The paper *A Game-theoretic Approach to Behavioural Visualisation* presented in his paper has been submitted to the 2^{nd} International Workshop on Formal Methods for Interactive Systems. The paper is a rewritten version of the workshop paper [C4]. The original workshop paper, [C4], focuses on the introduction of game coloured Petri nets, whereas the revised paper, [T5], focuses on a general formal framework for visualisations of formal models.

[T5] M. Westergaard. A Game-theoretic Approach to Behavioural Visualisation. Submitted, 2007.

[C4] M. Westergaard. Game Coloured Petri Nets. In *Proc. of 7th CPN Workshop*, volume 579 of *DAIMI-PB*, pages 281–300, 2006.

The version presented here is identical to the submitted paper except for typographical changes.

A Game-theoretic Approach to Behavioural Visualisation

Michael Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: mw@daimi.au.dk

Abstract

To bridge the gap between domain experts and formal methods experts, visualisations of the behaviour of formal models are used to let the domain expert understand and experiment with the formal model. In this paper we provide a definition of visualisations, founded in game-theory, which regards visualisations as transition systems synchronised with formal models. We show example visualisations, use them to show winning strategies of games, and demonstrate how an industrial application of formal models benefited from this approach.

9.1 Introduction

Formal models are being used for specification and verification of complex systems [T4, 10, 61, 64, 94, 103], provide valuable insight into the workings of the systems, and may detect errors early in the development process. One problem of constructing formal models of systems is that the domain experts, who have a lot of knowledge of the domain of the modelled system, typically have little or no knowledge of formal models. At the same time, experts in formal models typically have little knowledge of the system domain. One way to solve this is to let the domain expert describe the system to the formal methods expert, who then constructs a model for specification and validation. The drawback of this approach is that it is very difficult to know whether problems in the model represent errors in the model itself or in the modelled system. The formal methods expert typically does not know the domain well enough to make the judgement for subtle errors, and the domain expert does not understand the formal model or the error report well enough to make the judgement either. One way to facilitate the communication between the formal methods expert and the domain expert is to create a domain-specific visualisation, which the domain expert can inspect and stimulate. Examples of visualisations include cartoon-like representations of, e.g., computers on a network and how they communicate or a live updated UML sequence diagram [131] showing how messages are exchanged between people working in a bank. We also provide an example of how this can be used to visualise problems found in a model.

In order to facilitate communication of formal models, several tools [T3, 50, 66, 99, 117, 141, 149] have been conceived with the purpose of constructing domain-specific visualisations. These tools rely on the methodology depicted in Fig. 9.1. Here a domain expert writes a specification of the system. The specification is usually written in natural language and only uses semi-formal

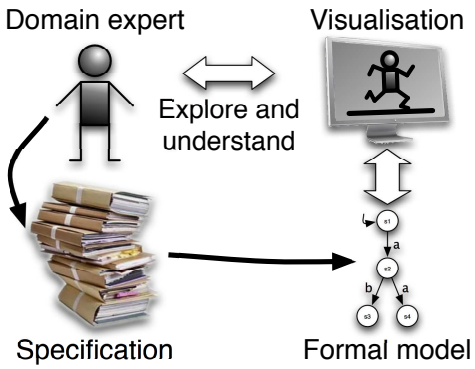

Figure 9.1: Methodology.

notation. Often the specification is vague and incomplete, maybe even self-contradictory. In order to make the specification clear, complete, and consistent, a formal executable model in constructed. This is usually done by a formal methods expert, who is not an expert on the domain. Ambiguities can be resolved during the construction of the model, which in itself makes the construction worthwhile. In order to ensure that the formal model actually reflects the specification, a visualisation is created. The behaviour of the visualisation is defined by the formal model, so the visualisation reflects the state of the model and changes in the model is reflected as updates to the visualisation. The domain expert is now able to see and understand what happens in the formal model, and can even interact with it. Inconsistencies between the model and the specification can be resolved as the domain expert identifies things that do not work as intended. We may then verify properties of the model required by the domain, e.g., that a network protocol cannot cause dead-locks, knowing that errors in the model probably reflect errors in the specification. One problem with tools facilitating the methodology in Fig. 9.1 is that they are built in an ad-hoc manner, as an afterthought, when the gap between domain experts and formal method experts becomes evident to researchers working with a specific formalism. Therefore the tools either mainly allows simple inspection of the state of the model during execution or require that the modeler spends a lot of time constructing a visualisation and integrating it with the model.

In this paper we propose a new, theoretically well-founded way to view visualisations, a declarative way for tying visualisations to a formal model, and a way to visualise error reports from formal verification so the domain expert is able to understand them. The idea is to view visualisations as transition systems, where the state of the system is what is visible to observers and labels on transitions are changes to what is visible. We tie visualisations to models by defining a synchronisation constraint [2], and require that the visualisation is able to simulate [124] the model, to make the behaviour of the synchronised product unconstrained by the visualisation and dictated by the model. This formulation only allows the domain expert to observe the behaviour of the formal model. In order to also allow the domain expert to provide input to the formal model, we regard it as describing a game between the modelled system and its surroundings; the domain expert then controls the environment and a computer tool controls the modelled system. The definition is formal and general, so it is possible to implement the method in computer tools supporting any formalism using transition systems as semantical foundation.

The paper is structured as follows: Sect. 9.2 describes related work, and in Sect. 9.3 some theoretical background material needed to understand the rest of the paper is provided. Sect. 9.4 introduces and exemplifies the idea of

regarding visualisations as transition systems synchronised with formal models, and in Sect. 9.5 two example uses of visualisations are described, namely a way to show winning strategies of games and an industrial application of the method in Fig. 9.1. In Sect. 9.6, we sum up our conclusions.

9.2 Related work

Several tools supporting the methodology in Fig. 9.1 exist. In this section we will describe some of them and discuss strengths and weaknesses of each.

ExSpect [50], a tool for modeling based on coloured Petri nets [91], allows the user to view the state of models by associating widgets with the state of the model, and allows users to asynchronously interact with the model using simple widgets. The disadvantage of this approach is, firstly, that it is specific to coloured Petri nets (as it relies on the special kind of state in a coloured Petri net) and, secondly, that input from the user is made by switching from one state of the system to another without formally executing a transition in the model.

The BRITNeY Suite [C2, T3] and Mimic/CPN [141] are libraries which facilitate visualisation of coloured Petri net models. They provide an API which can be used to define and update visualisations. By annotating a model, these functions are called during execution of the model. The disadvantage of this approach is that it is very inconvenient to have to change the model in order to add a visualisation and the changes unnecessarily clutter the model. Furthermore, these tools mainly focus on the state changes of the system, and everything shown to the user must be formulated as explicit updates, so it is not possible to easily monitor the value of, e.g., a counter like in ExSpect. Finally, these tools are unable to handle asynchronous input, which must be simulated by polling.

LTSA [116], a tool for modeling using timed transition systems, allows users to animate models using a library called SceneBeans [117, 149]. Animations are tied to models by associating animation activities with clocks. Resetting a clock corresponds to starting an animation sequence. The termination of an animation sequence, or a user with a mouse, sends events which correspond to progress of timers. The method is nice and declarative, but requires that we have clocks at our disposal, limiting the method to timed formalisms.

PNVis [99] is an add-on for the Petri Net Kernel [169], a modular tool for editing Petri nets [138]. PNVis associates 3D objects and locations in a 3D world with certain aspects of the state of the model and is hence suitable for modeling physical systems, but not aimed at systems that do not immediately have a physical counter-part.

The Play-Engine [66] allows a prototype of a program to be implemented by inputting scenarios (play-in) via an application-specific GUI. The resulting program can then be executed (play-out). Compared to the approach of the other described tools, this makes the model implicit as it is created indirectly via the input scenarios. Furthermore, the Play-Engine relies on heavy-weight techniques to perform visualisation as the model is given implicitly. In order to decide how to execute the model, a complete model-checking step is performed in each step, which is computationally expensive.

9.3 Theoretical background

Most of the work described in this paper has been developed in the context of coloured Petri nets (CP-nets or CPNs) [91] and game coloured Petri nets (game CPNs or game CP-nets) [C4], but applies to many other formalisms. In order to reflect that, we formulate our method using transition systems, which constitute the semantical foundation for several important modelling languages, e.g. CP-nets and the π-calculus [124]. In this section we recall definitions of transition systems, synchronised products, simulations, game transition systems (games), as well as winning strategies for games.

Definition 9.1 ((Labelled) Transition System) *A **transition system** (TS) is a tuple (S, T, δ, s_I), where S is a (finite or infinite) set of **states**, T is a (finite or infinite) set of **transitions**, $\delta \subseteq S \times T \times S$ is the **transition relation**, and $s_I \in S$ is the **initial state**.*

Four examples of transition systems can be seen in Fig. 9.2. Here we have represented each state by a circle and transitions as arcs leading from one circle to another. If there is an arc, labelled by a, leading from a circle labelled $s1$ to a circle labelled $s2$, it represents a transition $(s1, a, s2) \in \delta$. The initial state is marked by an incoming arc with no source. We will later explain why some arcs are dashed and some states are drawn using a double line. As an example, TS (a) can be represented as $\mathcal{TS} = (S, T, \delta, s_I)$ where $S = \{s1, s2, s3, s4\}$, $T = \{a, b\}$, $\delta = \{(s1, a, s2), (s2, b, s3), (s2, a, s4)\}$, and $s_I = s1$.

Let $\mathcal{TS} = (S, T, \delta, s_I)$ be a transition system, $s, s' \in S$ two states, and $t \in T$ a transition. If $(s, t, s') \in \delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. A *finite occurrence sequence*, σ, is an alternating sequence of states, s_i, and transitions, t_i, written $\sigma = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$ where $(s_i, t_i, s_{i+1}) \in \delta$ for $i = 1, \ldots, n-1$, and $s_1 = s_I$. An *infinite occurrence sequence*, σ', is an alternating sequence of states, s_i, and transitions, t_i, written $\sigma' = s_I = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} \cdots$ where $(s_i, t_i, s_{i+1}) \in \delta$ for $i \geq 1$, and $s_1 = s_I$. We denote by Σ^ω the set of all (finite and infinite) *occurrence sequences*.

We often wish to synchronise two or more transition systems, and a way to that is by forming a synchronised product by using a relation on transitions to define which must occur simultaneously, as formalised in Def. 9.2.

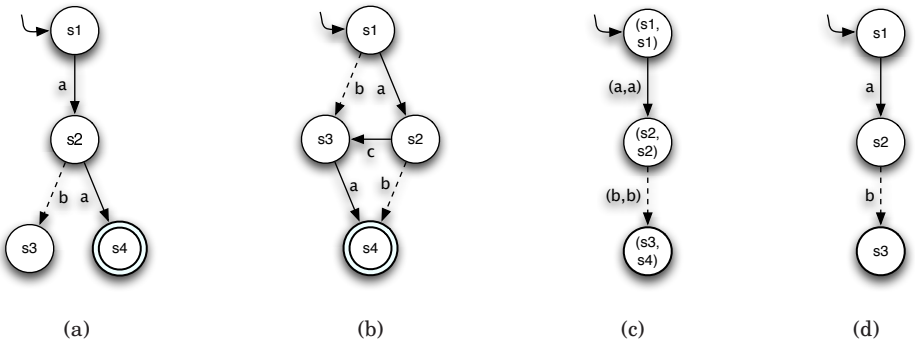

Figure 9.2: Four transition systems. (c) is a synchronisation of (a) and (b), (d) is equal to (c) except that its states and transitions have been renamed.

Definition 9.2 (Synchronised Product, [2]) *Let $\mathcal{TS}_i = (S_i, T_i, \delta_i, s_{Ii})$ for $i = 1, \ldots, n$ be transition systems. A **synchronisation constraint** is a relation $\mathcal{S} \subseteq T_1 \times T_2 \times \cdots \times T_n$. The **synchronised product** of $\mathcal{TS}_i$ w.r.t. $\mathcal{S}$ is $\mathcal{TS} = (S, T, \delta, s_I)$ with $S = S_1 \times S_2 \times \cdots \times S_n$, $T = \mathcal{S}$, $\delta = \{((s_1, \ldots, s_n), (t_1, \ldots, t_n), (s_1{}', \ldots, s_n{}')) \mid (t_1, \ldots, t_n) \in T, (s_i, t_i, s_i{}') \in \delta_i \text{ for } i = 1, \ldots, n\}$, and $s_I = (s_{I1}, s_{I2}, \ldots, s_{In})$.*

If we synchronise the TS (a) and TS (b) in Fig. 9.2 using the synchronisation constraint $\mathcal{S} = \{(a, a), (b, b)\}$, we obtain TS (c) (we have omitted states that are not reachable from the initial state). We notice that it is not possible for one of the TS to take a step autonomously using the above definition. We can simulate this by adding a distinguished transition Δ which leads from each state to itself. In the case of the TS in Fig. 9.2(a), we would add Δ to T and $\{(s1, \Delta, s1), (s2, \Delta, s2), (s3, \Delta, s3), (s4, \Delta, s4)\}$ to δ.

We often need to state that two transition systems behave in a similar way. We do this by defining a simulation, which states that one TS is able to exhibit the same behaviour as another (but not necessarily the other way around).

Definition 9.3 ((Strong) simulation [124]) *Let $\mathcal{TS}_i = (S_i, T, \delta_i, s_{Ii})$ for $i = 1, 2$ be transition systems sharing transitions. A relation $\preceq \subseteq S_1 \times S_2$ is a **simulation** iff whenever two states are in the relation, $s_1 \preceq s_2$, then for all transitions $\alpha \in T$, such that $s_1 \xrightarrow{\alpha} s_1{}'$, there exists a $s_2{}' \in S_2$ such that $s_1{}' \preceq s_2{}'$ and $s_2 \xrightarrow{\alpha} s_2{}'$. We say that $\mathcal{TS}_2$ **simulates** $\mathcal{TS}_1$ if there exists a simulation $\preceq \subseteq S_1 \times S_2$ such that $s_{I1} \preceq s_{I2}$.*

In Fig. 9.2 both (a) and (b) can simulate (d) using the simulations $\preceq_a = \{(s1, s1), (s2, s2), (s3, s3)\}$ respectively $\preceq_b = \{(s1, s1), (s2, s2), (s3, s4)\}$.

If we look at a game like tic-tac-toe, we see that it has two players, cross and naught. From the point of view of cross, it is only possible to add crosses to the board, naughts are added "automatically" according to the rules of the game. We want to reflect this in a transition system, so we split the transitions into two disjoints sets: the transitions controllable by the system we are modelling, and the transitions executed by the environment. We make the assumptions about the surroundings explicit in the model, yet provide a clear distinction between assumptions about the surroundings and the specification of the system. In the tic-tac-toe example, the action of adding a cross to the board is controllable by the modelled system and the action of adding a naught is not. Applying this to formal modelling, transitions of the modelled system are controllable, e.g., the actions of a network protocol, such as transmitting a packet or incrementing a counter, are controllable, whereas actions of the surroundings (e.g., a network), such as transmitting or altering a packet, are uncontrollable. Transitions of the environment formalise the assumptions about the surroundings (e.g. whether the network is allowed to alter packets). In normal games, like tic-tac-toe, we often also have some goal, e.g., ending up with three crosses in one row. This is also the case when modelling systems as games; in the case of a network protocol, a goal may be to successfully receive all packets in the correct order. A game is a TS where transitions are separated into disjoint sets: controllable and uncontrollable. Additionally we add a set of winning (goal) states. This is summarised in Def. 9.4.

Definition 9.4 (Game) *A **game** (or game transition system) is a tuple $(S, T^u, T^c, \delta, s_I, W)$, such that T^u is a set of **uncontrollable transitions** and T^c is a set of **controllable transitions** such that $T^u \cap T^c = \emptyset$, $W \subseteq S$ is a set of **winning states**, and $(S, T^u \cup T^c, \delta, s_I)$ is a transition system.*

We can turn any TS in Fig. 9.2 into a game by splitting the transitions into controllable and uncontrollable transitions and deciding which states are winning. For example, if we take $T^c = \{a\}$, $T^u = \{b\}$, and $W = \{s4\}$ we obtain a game for TS (a). In the figure we have shown uncontrollable transitions using dashed arcs. States in W are drawn using double lines.

A *strategy* is a function assigning to each state a controllable transition (if no controllable transitions are enabled in a given state, we can just map the state to any of the controllable transitions or add a distinguished transition Δ to T^c signifying "do nothing"). A winning strategy is a strategy, that ensures we always end up in a winning state, irregardless of what uncontrollable moves are chosen, i.e., a winning strategy is a "program" ensuring we and up in a good state. Formally:

Definition 9.5 (Winning Strategy) *Let* $(S, T^u, T^c, \delta, s_I, W)$ *be a game and* $\mathscr{S} : S \to T^c$ *a strategy. An occurrence sequence*

$\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n(\cdots) \in \Sigma^\omega, t_i \in T^u \cup T^c$ *is* ***consistent*** *with the strategy iff* $t_i \in T^c \implies t_i = \mathscr{S}(s_i)$ *for all* $i = 1, \ldots, n(, \ldots)$. *An occurrence sequence,* σ*, is* ***maximal*** *iff it is a) infinite, or b) finite,* $\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n$*, and if* $s_n \xrightarrow{t} s$ *for any* $s \in S$ *then* $t \in T^u$. *A strategy is a* ***winning strategy*** *iff all maximal occurrence sequences,* $\sigma = s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} s_n(\cdots) \in \Sigma^\omega$ *with* $s_1 = s_I$ *that are consistent with the strategy satisfy* $\exists k \geq 1$ *such that* $s_k \in W$.

If we take $T^c = \{a\}$, $T^u = \{b\}$, and $W = \{s4\}$ in Fig. 9.2(a), it is not possible to obtain a winning strategy (the only strategy is the mapping from all states to the transition a. The occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$ is consistent with that strategy, but does not lead to $s4$) whereas we can obtain a winning strategy in (b) using $T^c = \{a, c\}$, $T^u = \{b\}$, and $W = \{s4\}$ (the strategy mapping states $s1$, $s3$, and $s4$ to the transition a and $s2$ to c is winning).

In [18], an algorithm from [111] is instantiated to obtain an efficient (and optimal) algorithm to decide whether a given finite game (i.e. a game where $|S| + |T^u| + |T^c| < \infty$) has a winning strategy and to extract that strategy. The intuition of the algorithm is to calculate a minimal fix-point of all *good states*, where all states in W are good and all states where we can take a controllable step to a good state and all uncontrollable steps leading to a good state are good.

In Fig. 9.2(a), the only state which can be marked as good is $s4$ ($s3$ is not good as it has no successors, $s2$ not good as the b transition leads to $s3$, which is not good, and in $s1$ a leads to $s2$, which is not good). In (b), initially $s4$ is good. We can then mark $s3$ as good (as we can take an a transition to $s4$, which is good). After that, we can mark $s2$ as good (c, which is controllable, leads to $s3$ and b, all uncontrollable transitions enabled in $s2$, lead to $s4$). Finally we can mark $s1$ as good as both a and b lead to good states.

Using this algorithm, we can obtain a winning strategy for any game (if one exists). Often we are not satisfied knowing whether a winning strategy exists. If one does, we are interested in obtaining the winning strategy, as we can use as a guide to execute our model so we reach a winning state. We often require a counter example if no winning strategy exists so we can understand why. Until now, when concluding that a given game does not have a winning strategy, the best counter example we could provide was a list of all good states. This can be useful for small examples, but for systems with millions or more states this is not very useful. The purpose of the counter example is to convince a user that it is not possible to have a winning strategy. If a user needs conviction, it is probably because he thinks he knows a winning strategy. In this paper we will

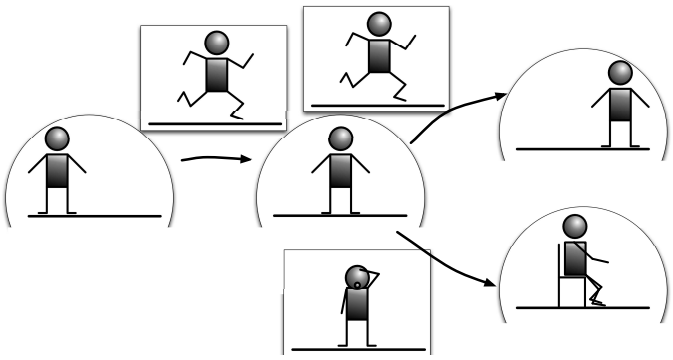

Figure 9.3: Visualisations as transitions systems.

propose a new way of providing counter examples to the existence of winning strategies. We let the user assume the role of the modelled system and let him try out his winning strategy against the computer, which knows how to counter all moves of the user. The user will try his winning strategy on a visualisation of the model. We will go into more detail about this in Sect. 9.5.2.

9.4 Visualisations as game transition systems

The idea of this work is to view visualisations as game transition systems, synchronised with formal models. The rationale behind the idea of considering visualisations as transition systems is that we can consider what is visible in the visualisation as a state and changes to what can be seen can be considered as transitions. As an example, consider Fig. 9.3. The semi-circles represent states of the visualisation and the rectangles are the labelled transitions leading from one state to another. In the left semi-circle we see one state, a person is standing at the left of a line. If we take the transition in the leftmost rectangle, the person runs to the right and we reach the state in the semi-circle in the middle of the figure, where the person is standing at the middle of the line. Now one of two things can happen: either the person keeps running (the transition to the upper right state), or the person gets tired and sits down (the transition to the lower right state). This visualisation is a renaming of the TS in Fig. 9.2(a), where "runs" corresponds to transition a, "gets tired and rests" corresponds to transition b, and $s1 \ldots s4$ corresponds to the various positions of the person. The states are graphical images and the transitions are transformations of one graphical image to another, e.g., an animation.

If we allow all synchronisation between a visualisation and a model, the behaviour of the synchronisation is not defined by the model, but by the model and the visualisation in unison, so if we, e.g., create a visualisation consisting of only one state and no transitions, the synchronisation is also without behaviour, which is not what we want to obtain. We want the behaviour of the synchronised system to be dictated by the model, and will only use the visualisation to show what happens in the model. In order to do this, we require that the visualisation is able to simulate the model. In that way, the behaviour of the synchronisation is dictated entirely by the model. A slight technicality is that the definition of a simulation (Def. 9.3) requires that the two systems share transitions. We remove this requirement and only require that, given a synchronisation constraint $\mathcal{S} \subseteq T_1 \times T_2$, whenever $s_1 \preceq s_2$, then for all $\alpha \in T_1$ if $s_1 \xrightarrow{\alpha} s_1'$ there exists a $s_2' \in S_2$ and a $\beta \in T_2$ such

that $s_1' \preceq s_2'$, $(\alpha, \beta) \in \mathcal{S}$, and $s_2 \xrightarrow{\beta} s_2'$. This allows us to say that (c) in Fig. 9.2 can be simulated by (a), (b), and (d), as we no longer care about the exact names of the transitions. For example, (c) can be simulated by (a) using the synchronisation constraint $\mathcal{S} = \{((a,a),a),((b,b),b)\}$ and the simulation $\preceq_b = \{((s1,s1),s1),((s2,s2),s2),((s3,s4),s3)\}$.

If we synchronise a model with a visualisation and require that the visualisation is able to simulate the model, the execution is defined by the model alone, which is fine if we only want to see the execution of the model. If we also want to manipulate the execution, we need to loosen the requirement that the visualisation must be able to simulate the model. Rather than allowing arbitrary synchronisations, which would make it difficult to distinguish between actions taken by the model itself and actions initiated by the user, we rely on games. The idea is that the visualisation plays one side of a game and the model plays the other side; controllable transitions of the visualisation corresponds to uncontrollable transitions of the model and vice versa. We require that the uncontrollable transitions of one side can simulate the controllable transitions of the other side. This is formulated in Def. 9.6.

Definition 9.6 (Visualisation) *Given a model as a game $\mathcal{TS}_M = (S_M, T_M^u, T_M^c, \delta_M, s_{IM}, W_M)$, a **visualisation** $\mathcal{TS}_V = (S_V, T_V^u, T_V^c, \delta_V, s_{IV}, W_V)$, and a synchronisation constraint $\mathcal{S} \subseteq (T_M^u \times T_V^c) \cup (T_M^c \times T_V^u)$, we say that $\mathcal{TS}_V$ can be used as a visualisation of $\mathcal{TS}_M$ with $\mathcal{S}$ iff there exists a relation $\sim \subseteq S_M \times S_V$ such that whenever $s_M \sim s_V$*

- *for all $\alpha \in T_M^c$ if $s_M \xrightarrow{\alpha} s_M'$ there exist $s_V' \in S_V, \beta \in T_V^u$ such that $s_M' \sim s_V'$, $(\alpha, \beta) \in \mathcal{S}$, and $s_V \xrightarrow{\beta} s_V'$, and*
- *for all $\beta \in T_V^c$ if $s_V \xrightarrow{\beta} s_V'$ there exist $s_M' \in S_M, \alpha \in T_M^u$ such that $s_M' \sim s_V'$, $(\alpha, \beta) \in \mathcal{S}$, and $s_M \xrightarrow{\beta} s_M'$.*

Furthermore we require that $s_{IM} \sim s_{IV}$.

The definition captures the intuition that whenever the model makes a move (a controllable transition in the model), the visualisation must be able to show that, and whenever the user provides some stimulation (a controllable transition in the visualisation), the model must be able to handle that and execute a corresponding uncontrollable transition.

One way to generate simple visualisations, is to use other formalisms as visualisation of our model. If we have created a model as a TS and need to communicate the model to an engineer who does not understand it, but who uses message sequence charts (MSC) on a daily basis (MSC can be seen as simplified UML sequence diagrams [131]), we can simply create an MSC and use it as visualisation of our model. In the following we present two visualisations that have proven themselves widely applicable and useful [T4, 94] for describing complex systems to domain experts, namely message sequence charts and cartoon-like visualisations created using a Java library called SceneBeans [149]. The MSC visualisation is an instance of the idea of using another formalism as visualisation of the model. The SceneBeans library is used by the LTSA tool as described in Sect. 9.2, but we use it in a way that makes it usable for a much wider range of formal models, as we do not require that the model is described as timed transition systems. The MSC visualisation exemplifies how to construct a visualisation that allows us to only see the behaviour of a model (not to manipulate it), whereas the SceneBeans visualisation allows us to see and manipulate the behaviour of the system.

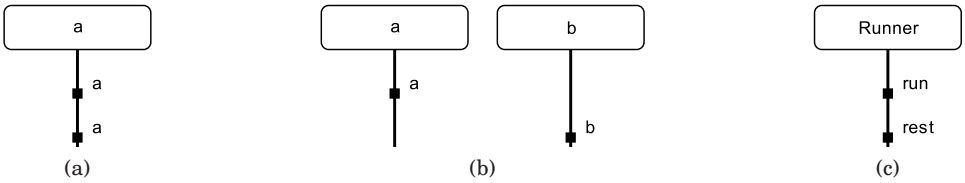

Figure 9.4: Simple MSC visualisations of the example from Fig. 9.2(a).

Example 1: Message sequence charts

Message sequence charts can be used either formally or informally to describe the behaviour of systems. A MSC consists of a set of processes, shown as vertical lines, which are able to exchange messages, represented as horizontal arrows from the source of the message to the destination, or which can execute internal events, represented as a dot on the process.

In its simplest form, this visualisation has a process for each transition and shows an internal event on the corresponding process whenever a transition is executed in the model. More formally, given a model $\mathcal{TS}_M = (S, \emptyset, T, \delta, s_I, W)$, we define a visualisation $\mathcal{TS}_V = (S_V, T_V, \emptyset, \delta_V, s_{IV}, \emptyset)$, where $T_V = T$. The set of states, S_V consists of all possible message sequence charts with T as processes. This is of course not manageable in reality (T may be infinite), so in practise we create processes as transitions are executed. The initial state is a MSC with processes T and no events, and transitions are enabled in $s \in S_V$, $s \xrightarrow{t} s'$, if $s' \in S_V$ is equal to s with an internal event added to the process t. The synchronisation used is equality. Using this visualisation directly on the model in Fig. 9.2(a), we can obtain the two leftmost visualisations in Fig. 9.4, the leftmost MSC, (a), corresponds to the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{a} s4$ and the middle MSC, (b), corresponds to the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$. The MSCs are updated as the model is executed, and the versions shown here are snapshots when no more transitions are enabled.

To make the visualisation more useful, we parametrise it with a function mapping transitions to process names and event labels so we can rename events and show "similar" events on a single process. Say the TS in Fig. 9.2(a) models a runner on a track (like the system in Fig. 9.3). The runner starts at the beginning of the track and runs towards the end. Optionally, the runner refuses to run any further halfway through the track, but sits down and rests. If we map the transition a to the process "Runner" and the event label "run" and b to "Runner" and "rest", we would obtain a visualisation as shown in Fig. 9.4(c) for the occurrence sequence $s1 \xrightarrow{a} s2 \xrightarrow{b} s3$. This visualisation makes it easier to see what was intended by the model than the ones in Figs. 9.4(a) and (b).

Synchronising visualisations with formal models using this technique is very useful and allows us to observe what happens in the model, but it does not allow us to interact with the model, e.g., to drive the model into states we find interesting. The next example makes full use of the separation of transitions into controllable and uncontrollable transitions, and allows the user to interact with the model using the visualisation.

Example 2: Visualisation using SceneBeans

The SceneBeans [149] library uses an XML specification for describing visualisations and allows programs using it to interact with the visualisation by

invoking commands in the visualisation and receiving events from the visualisation. By invoking a command on a SceneBeans visualisation, it is possible to change what is displayed on the screen, e.g. to move a graphical representation of a person, as in Fig. 9.3, and thereby provide feedback to the user. When a user, e.g., clicks on an object in a visualisation, the visualisation can raise an event, which can be handled by the application. We equate uncontrollable transitions of a SceneBeans visualisation with the provided commands, and controllable transitions with the events that can be raised by user interaction.

Using the SceneBeans library, it is possible to create a visualisation like the one sketched in Fig. 9.3. We want to control the runner, so we switch the transitions in Fig. 9.2(a), so the dashed arcs represent controllable transitions and solid arcs represent uncontrollable transitions. The start of the track corresponds to the left of the line and the end of the track is at the right. When we want the runner to progress along the track, we can click the figure representing the runner to raise a "run"-event (corresponding to a controllable a transition in the TS in Fig. 9.2(a)). If we do nothing when the runner is halfway through the track (the middle state in Fig. 9.3), it is possible that the uncontrollable b transition is executed, leading to executing of the command "rest", which makes the runner sit down and rest.

This kind of visualisation is formalism-independent, but the visualisation is heavily dependent on the model, as we need to support the required commands and events. Furthermore the user is required to specify how events and commands should be synchronised with the transitions of the model.

9.4.1 Tool support

Support for synchronising visualisations with formal models by regarding the visualisations as games has been added to the BRITNeY Suite [C2, T3], a tool for visualising formal models, typically created using coloured Petri nets.

The tool has been extended with an interface, written in Java, which gives developers the ability to write their own programs interfacing with formal models. The interface, which can be seen in Fig. 9.5, informs a visualisation, i.e. a class implementing the interface, of all enabled controllable transitions (line 2). The visualisation returns which controllable moves it would like to perform. The visualisation is informed whenever the computer makes a move (line 4) and when a user-specified move is executed (line 3). The names controllable/uncontrollable are from the point of view of the visualisation. The tool is able to switch controllable/uncontrollable transitions so the visualisation can control the controllable transitions of the model if we wish to experiment with the behaviour of the model, or control the uncontrollable transitions of the model, allowing us to see how the model reacts to the surroundings. Finally, the visualisation is informed when there are no more enabled transitions (the game is over, line 5). This can be used if the user should be alerted or cleanup is needed when the game is over. Classes implementing this interface act as both visualisation and synchronisation constraint. As uncontrollable is not allowed to raise exceptions, visualisations implementing this interface are able to execute a transition synchronised with any transition offered by the model, so the visualisation's uncontrollable transitions are able to simulate the model's controllable transitions. Additionally, if controllable (in line 2) returns a subset of the transitions provided as parameter, the model is able to simulate the controllable transitions of the visualisation. Thus, according to Def. 9.6, a class implementing the interface in Fig. 9.5 can be used as visualisation of any model.

```
public interface GameListener {
      List<Transition> controllable(List<Transition> ts);
      void controllable(Transition t);
      void uncontrollable(Transition t);
      void gameOver();
}
```

Figure 9.5: The GameListener interface.

Both of the visualisations presented in this section have been implemented using this interface, so despite the simplicity of the interface, it is versatile. In addition to the examples in this paper, the interface has also been used to implement a visualisation which automatically generates form-filling dialogues for CPN models (this visualisation is described in [C4]) as well as for ongoing work on implementing a work-flow system on top of game CP-nets.

Fairness

If the purpose of a visualisation is to get acquainted with the model or the modelled system, it is often reasonable to assume that a computer tool chooses controllable transitions at random. This can often be done very quickly, however, and this can make it difficult for the user to interact with the model. To overcome this, we may need to impose fairness during execution of the model.

A simple way to impose fairness is to make the game turn-based: the model makes one uncontrollable transition, followed by a user-selected controllable transition and so on until no transitions are enabled. If either of the players have no possible moves (i.e. no controllable resp. uncontrollable transitions are enabled) the turn is passed on to the other player. This approach is simple, easy to understand, and easy to implement. The disadvantage is that, depending on modelling detail, one player may gain an unfair advantage, and minor changes may make it difficult for one player to keep up with the moves of the other. This can be seen in the runner example in Fig. 9.2(a) (with $T^c = \{b\}$ and $T^u = \{a\}$), where we always end up in state $s3$ if we use this technique, as we need to first choose an a transition. The turn is then passed on to the computer, which chooses a b transition.

Another way to impose fairness is to give controllable transitions (from the visualisation's point of view) priority over uncontrollable transitions. This particularly useful for SceneBeans visualisations, where transitions of the model are expected to be executed while the user is observing, but we want interaction to happen immediately when the user requests it. The visualisation of the runner will do nothing until a controllable transition has been chosen in the runner example in Fig. 9.2(a) and Fig. 9.3. When the runner is in the state $s2$, at the middle of the track, a user can force the runner along the track by clicking on the graphical representation of the runner. If the user does nothing for a while, the computer will choose the uncontrollable transition b, and the runner will rest.

A third way to impose fairness is to make execution of transitions take time. A simple way to do this is to let the execution of every transition take, say, 0.1 second. The advantages and disadvantages of this approach is the same as those for turn-based execution. A slightly more involved way to use time to impose fairness is to use a timed formalism such as timed automata [1]. In this case, transitions may only be enabled for a certain amount of model-time

or only a certain amount of model-time after another transition. The idea is to let model-time correspond to real time. The advantage of this approach is that it is very general, and allows us to get a natural feeling of the behaviour of a timed model, but the disadvantage is that timed models may be more difficult to understand and this approach requires a timed formalism.

9.5 Use of visualisations

In this section we give two examples of use of visualisations. The first example in an industrial case study, where visualisation is used to improve a specification, using the methodology in Fig. 9.1, and the second example is to an application to verification of games, where we use visualisations to convince domain experts that no winning strategy exists (when it is believed that it should) as well as providing a means to find out if the error is in the specification or the formal model.

9.5.1 Industrial Case: Routing in Mobile Ad-hoc Networks

First, we look at an industrial application of visualisation, which uses an earlier version of the BRITNeY Suite without support for visualisations as games. The project is a collaboration between Ericsson Denmark A/S, Telebit and the CPN group at the University of Aarhus. For more details about the project, see [T4].

In Fig. 9.6, we see two visualisations created to visualise an interoperability protocol for mobile ad-hoc networks. The protocol ensures that mobile ad-hoc nodes (laptops) can communicate with a stationary host when on the move via the nearest gateway. Each gateway owns a specific sub-net of IP addresses. Based on the IP address of an ad-hoc node, it is possible to decide which gateway to use. The basic operation of the model is illustrated by the MSC in Fig. 9.6 (top). The protocol is modelled using coloured Petri nets in a model that contains modules, 54 places and 40 transitions. Altogether the model also contains 1000 lines of inscriptions, 200 of which are used to drive the visualisation. The exact details of the protocol are out of scope of this paper. The visualisation in Fig. 9.6 (bottom) makes it possible for the user to observe the behavior of the system as packets, visualised by colored dots, flow along the network and to provide stimuli to the protocol by dragging and dropping the laptops to indicate the node movement. These visualisations have been used in the project, both internally during protocol design, and externally, when presenting the protocol to management and protocol engineers unfamiliar with formal modeling.

The project uses the visualisations described in Sect. 9.4, namely a message sequence chart and a SceneBeans visualisation. The visualisations have been synchronised with the model using annotations of the model. One of the problems we encountered during the project, was this need to add annotations to the model. For example, in Fig. 9.7, we see the annotation “input...”, used to show packets flow. This is by far the largest annotation of the model, and clutters it unnecessarily. Furthermore annotations have to be added for each visualisation, making it difficult to turn off one or more visualisations, in order to focus on e.g. the MSCs. Using the approach described in this paper, we create our visualisations and for each specify how it should be synchronised with the model (in fact we would not need to specify synchronisation constraints as the implementation uses conventions, such as naming, to generate these automatically), and we can then turn off each visualisation independently and

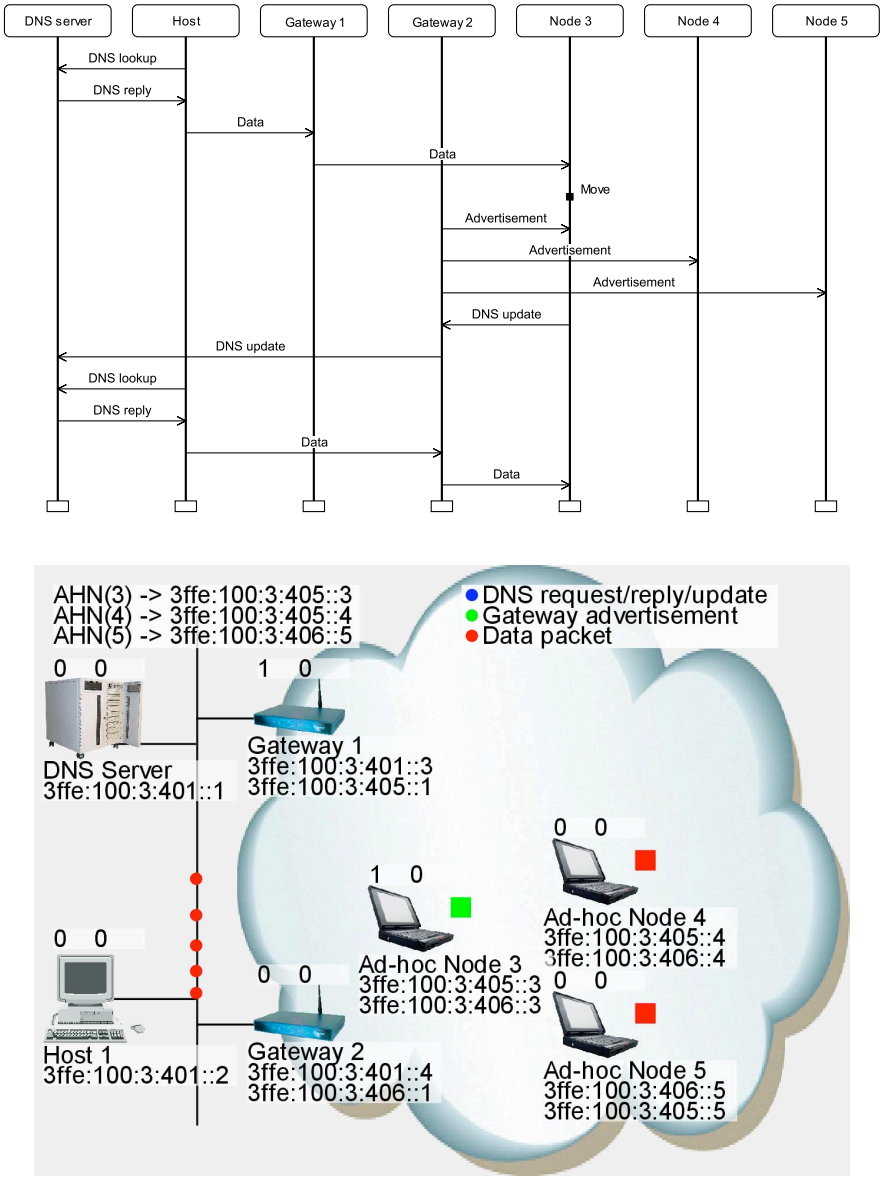

Figure 9.6: Visualisations used in an industrial project.

the model would be left uncluttered. Another major problem encountered in the project was that we wanted the model to perform actions when idle, e.g. send gateway advertisements, and react immediately when we moved a node or wanted to send packets. We only partially solved this by polling the visualisation for changes, which made the visualisation almost work, but was never satisfactory. Creating the visualisations as games, as proposed in this paper, making slight changes to the model in order to make it a game (make e.g. the movement of the ad-hoc nodes uncontrollable in the model), and using one of the fairness constraints discussed in Sect. 9.4.1, it is possible to make interaction with the model much more natural as the visualisation will we able to force actions in the formal model as desired.

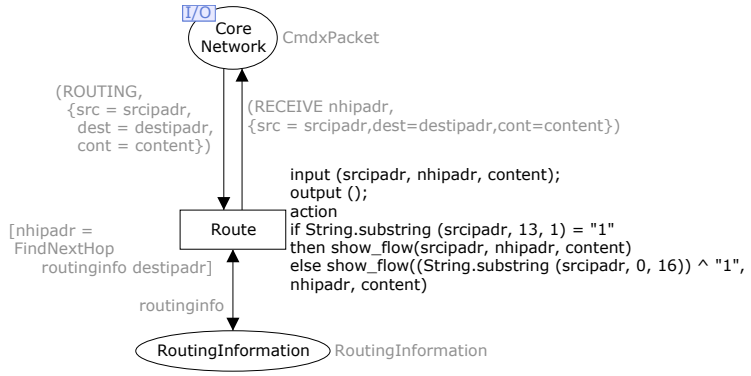

Figure 9.7: Part of the routing mechanism.

9.5.2 Visualising winning strategies

Hitherto, we have used visualisations primarily for validation that the formal model reflects the intended behaviour by letting a domain expert stimulate and observe the model using visualisations. Now, we will turn to using visualisation for communicating the result of formal verification, i.e., convincing users that no winning strategy exists, which is decided using an algorithm from [18] as outlined in Sect. 9.3. The purpose of a counter example is to convince users that it is impossible to have a winning strategy, so we let the domain expert assume the role of the modelled system and let him try out ideas for winning strategies. At the same time we let a computer tool take charge of the uncontrollable actions according to the counter example that has been calculated. The user is urged to reach a winning state while the tool executes uncontrollable transitions to prevent that (by ensuring that the user is not allowed the ability to execute a transition leading to a good state). We can do this using the formal model, but often the formal methods expert does not have enough domain knowledge to have understand why the system should have winning strategy, so the domain expert, who has little knowledge of the modelling language, has to find out whether the error is in the model or in the specification. Instead we let the domain expert control the controllable transitions of the model using a visualisation (the computer tool is able to let the visualisation assume control of either the controllable or uncontrollable transitions, as described in Sect. 9.4.1). We let the user stimulate the model in any way seen fit (according to the supposed winning strategy), and eventually the model will perform an unforeseen move (error in the specification) or the model will perform a disallowed move (error in the model).

In the example in Fig. 9.2(a), we may think we have a winning strategy: always pick transition a. This leads to the winning state $s4$, right? The computer tool knows that the only good state is $s4$, and will stay clear of it. If we let the user use the visualisation in Fig. 9.3, he would first click the runner to progress to $s2$. The tool then executes the b transition as it knows that $s4$ is good, but $s3$ is not. The game is over and the user is hopefully convinced that it is impossible to ensure we end up in a winning state.

9.6 Conclusion and future work

In this paper we have given a theoretical foundation for viewing visualisations as game transition systems synchronised with formal models, providing

a uniform and general framework for coupling formal models and behavioural visualisation. We have used game-theory to separate output from and input to the model and given two concrete examples of visualisations. We have demonstrated how an industrial case can benefit from using the method described in this paper. Furthermore, we have sketched how this can be used to create counter-examples to the existence of a winning strategy in games, so domain experts with no knowledge of the formalism used can understand them.

Future work includes using this technique in industrial settings. The visualisations described in this paper is already distributed as part of the BRITNeY Suite, and ongoing work on creating a detailed model of TCP/IP uses the MSC visualisation to communicate the model to protocol experts.