An Approach for Creating Domain Specific Visualisations of CSP Models

Lukas Ladenberger, Ivaylo Dobrikov, and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
{ladenberger,dobrikov,leuschel}@cs.uni-duesseldorf.de

Abstract. A domain specific visualisation can greatly contribute to better understanding of formal models. In this work we propose an approach that supports the user in creating domain specific visualisations of CSP models. CSP (Communicating Sequential Processes) is a formal language that is mainly used for specifying concurrent and distributed systems. We have successfully created various visualisations of CSP models in order to demonstrate our approach. The visualisations of two case studies are presented in this paper: the bully algorithm and a level crossing gate. In addition, we discuss possible applications of our approach.

Keywords: Formal Methods, CSP, Domain Specific Visualisation, Validation, Method, Tool Support, Graphical Editor.

1 Introduction and Motivation

The feedback from a domain expert is crucial in the process of creating a formal model since certain types of errors can only be detected by a domain expert. Moreover, it is very important for the domain expert to make sure that his expectations are met in the formal model. However, the communication between the developer of a formal model and the domain expert can be challenging. One reason for this is the fact that discussing a formal model requires knowledge about the mathematical background of the respective formalism that the domain expert might not have. To overcome this challenge, it may be useful to create domain specific visualisations of formal models.

Inspired by the successful application of domain specific visualisations [1], [6] of Event-B models [3], we have started an attempt to develop an approach for creating domain specific visualisations for CSP (Communicating Sequential Processes). CSP is a notation used mainly for describing concurrent and distributed systems. There are two major CSP dialects: CSP-M [15] and CSP# [17]. The most popular tools that support model checking of CSP-M specifications are FDR [19] and ProB [10]. Support for animating processes of CSP-M specifications is provided by ProB and ProBE [5]. The more recent CSP# [17] is supported by the PAT system [18]. In this work, we concentrate on the creation of domain specific visualisations for CSP-M models.

** The work in this paper is partly funded by ADVANCE, an European Commission Information and Communication Technologies FP7 project.

Some of the tools provide features for visualising some aspects of the formal CSP model. For instance, ProB, PAT, and FDR can provide visualisations of counter examples that come in form of graphs. On the other hand, this work is concerned with creating domain specific visualisations. This means that if we were modelling, an interlocking system we could create a domain specific visualisation that shows a track layout with blocks and points as well as signals and trains. From now on, when we speak about a visualisation we mean a domain specific visualisation.

In this work we present an approach (method and tool) for visualising CSP-M models. We describe the method and present an implementation that comes as an extension for BMotion Studio [8]. BMotion Studio is a visual editor that supports the user in creating domain specific visualisations for Event-B, a formal language for state-based modelling and verification of systems.

The difference between our contribution and the original visualisation approach of BMotion Studio is imposed by the specifics of the CSP formal language. The basic idea of BMotion Studio is to visualise the information that is encoded in the states of an Event-B model (e.g. the values of variables), where each state of the model is mapped to a particular visualisation. In contrast to Event-B, in CSP the states of the modelled system are left uninterpreted and the behaviour is defined in terms of sequences of events (*traces*). Thus, the concepts of BMotion Studio are not longer applicable on event-based formalisms as CSP. The intention of our approach is to visualise the traces of the underlying CSP model.

In order to demonstrate our approach, we have created visualisations for various CSP-M models that we have found in the literature. In this paper, we focus on the presentation of the visualisations of the bully algorithm [13] and of a level crossing gate [14]. We also discuss how our approach can be of use in the process of analysing and validating CSP specifications.

The paper is organised as follows: Sections 2 and 3 describe the method and tool support, respectively. The presentation of the visualisation of both case studies is given in Section 4. The discussion of possible applications of our approach is outlined in Section 5. Finally, we present our conclusions and compare our work with related work.

Tool Website. The tool, various case studies, and a tutorial can be found at http://www.stups.hhu.de/bmotionstudio/index.php/CSP.

2 The Method

The mathematical semantics of CSP are mainly based on *traces*. A trace is a sequence of events performed by a process that can communicate and interact with other processes within the CSP model. The basic idea of our approach is to visualise the information encoded in the given sequence of events (*trace*). However, a process may perform many different traces and thus creating a visualisation manually for each possible trace is an almost impossible task.

Our method requires the user to set up only one visualisation that may be capable of representing any possible trace of a CSP process of a particular model. This is achieved by means of *observers* that are used to link the visualisation with the model. Formally, one can describe the method by means of Algorithm 1.

Algorithm 1: Visualising a CSP trace

```
1 procedure visualiseTrace(trace ⟨e_1, e_2, ..., e_n⟩, observers obs)
2     for i=1 to n do
3         foreach o ∈ obs do
4             if member(e_i, o.exp) then
5                 trigger(o.acts)
6             end if
7         end foreach
8     end for
9 end proc
```

For visualising a particular trace $tr = \langle e_1, e_2, \ldots, e_n \rangle$, we sequentially go through each event e_i of tr with $i \in \{1..n\}$ and execute all established observers obs for e_i. Note that by "visualisation of a trace" we mean the visualisation of the state reached after the sequential execution of the events of a trace.

Each observer o has a user-defined CSP expression $o.exp$ that constitutes a set of observed events. For instance, the CSP expression $\{e.x \mid x \leftarrow \{0..3\}\}$ will constitute the set of observed events $\{e.1, e.2, e.3\}$. In addition, an observer defines a list of actions $o.acts$ that determine the appearance and the behaviour of the visualisation. The actions are only triggered when the currently processed event e_i of the given trace is a member of the respective set of observed events defined by $o.exp$. More precisely, the actions are triggered (line 5) whenever the expression $member(e_i, o.exp)$ evaluates to *true* (line 4).

3 Tool Support

Fig. 1 shows an overview of the tools and components that are used in this work, as well as how our contribution fits into this overview (marked with dotted border).

We implemented the method presented in Section 2 as an extension for the new version[1] of BMotion Studio [8]. BMotion Studio is a visual editor for creating domain specific visualisations of formal models. It uses ProB [9] to interact with the model, to obtain trace information and to evaluate expressions. ProB is a validation tool for model checking and animating Event-B, Classical-B and CSP-M models [10], as well as other formalisms (e.g. [7] and [12]). The current version of BMotion Studio supports the user in creating visualisations for Event-B models [8]. This work extends BMotion Studio to support the creation of visualisations for CSP-M models.

In BMotion Studio, a visualisation is described by a *visualisation template* that contains *visual elements* and *observers*. Visual elements may be, for instance, shapes or images that represent some aspects of the model. For example, in case of modelling a communication protocol, we can use circles for representing the communicating entities of the protocol and arrows for the message exchanges between

[1] The new version of BMotion Studio is not officially released yet, but the source code is available from http://www.stups.hhu.de/bmotionstudio/index.php/Source.

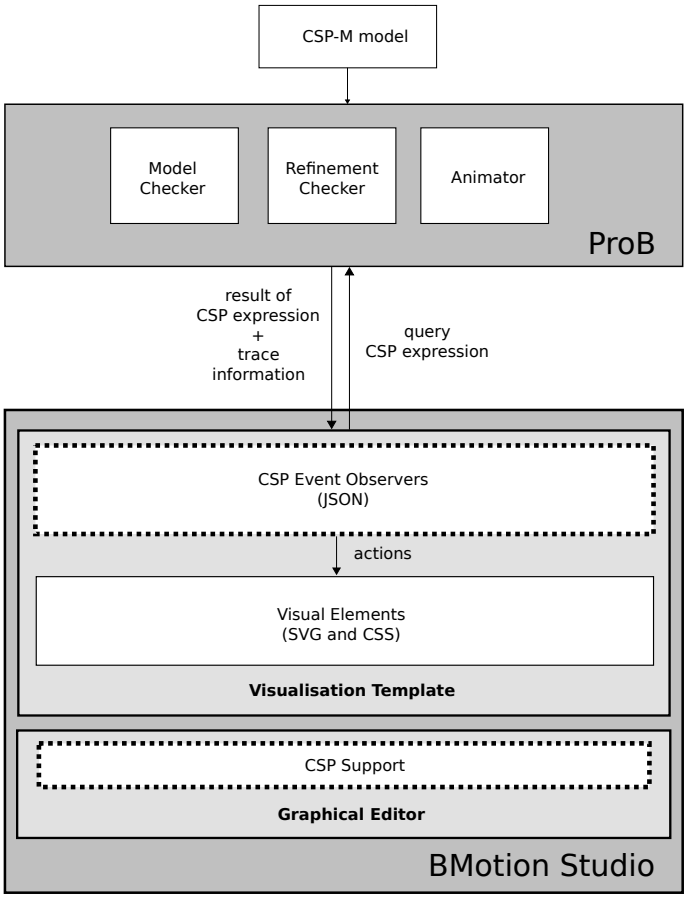

Fig. 1. Overview of the components that are used in this work

the entities. The new version of BMotion Studio uses web technologies like Scalable Vector Graphics (SVG) [21] and Cascading Style Sheets (CSS) [20] for this purpose. SVG is an XML-based markup language for describing two-dimensional vector graphics. It comes with a number of visual elements like shapes, images and paths. On the other hand, CSS is a language that can be used to describe the style of SVG visual elements (e.g. the colour or the dimension).

Observers are used to link visual elements with the model. An observer is notified whenever a model change its state, e.g. an event was executed. In response, the observer will query the model's state and triggers actions on the linked visual elements in respect to the new state. BMotion Studio comes with a number of default observers for creating visualisations for Event-B. For instance, BMotion Studio provides an observer that takes a user-defined predicate that is to be evaluated in every state. Depending on the result of the predicate (true or false), the observer will trigger an action to change the appearance of the linked visual elements (e.g. the colour of a shape).

We extended BMotion Studio with a new observer type called *CSP event observer* in order to support creating visualisations of CSP models. The observer has the following JSON structure (in BMotion Studio an observer is represented in JSON [2]):

```
{ "exp": "<user-defined CSP expression>",
  "actions": [
    { "selector":"<selector>", "attr":"<attribute>", "value":"<value>" },
    { ... }
  ] }
```

Each observer has a *user-defined CSP expression* and a list of *actions*. The user-defined expression constitutes a set of observed events, whereas the actions determine the changes made on visual elements.

An action defines a *selector* that matches a set of visual elements in the visualisation (SVG graphic). A selector follows the syntax provided by jQuery[2]. For instance, to match the visual element with the ID "elem1" (each element should have a unique ID in the visualisation) the user can define the selector "#elem1". The prefix "#" is used for matching a visual element by its ID in jQuery. An action also defines an *attribute* (e.g. "fill" for colouring the interior of a visual element like a circle shape) and a corresponding *value* that will be set as the new value of the attribute when the action is triggered. The actions of an observer o are triggered when the currently processed event is in the set of observed events of o.

The user can refer to the information given by the arguments of the currently processed event within the action fields (selector, attribute and value). This is achieved by means of the construct "{{aN}}" where aN refers to the N-th argument of the event. For instance, if the event has two arguments, then the first and the second one can be obtained with "{{a1}}" and "{{a2}}", respectively. To illustrate this, consider an event $evt.x$ with $x \leftarrow 0..4$. One may want to use the information given by the first argument x of evt within a selector in order to match visual elements that have an ID of the form "elem$x$". This can be done by defining the selector "#elem{{a1}}". The construct "{{a1}}" will be replaced by the value of the first argument of the currently processed event in the observer. For instance, if the currently processed event is $evt.2$, the selector "#elem{{a1}}" will become "#elem2".

Fig. 2 illustrates the function of the CSP event observer on a simple example. The visualisation consists of an SVG graphic with a text field element with the ID "txt" and one CSP event observer. The CSP event observer defines an expression that constitutes the set of observed events $evt = \{evt.2, evt.4, evt.6, ..\}$ and one action $act1$ that changes the value of the attribute "text" to "{{a1}}" of the visual element with the ID "txt" (the text field). According to our method (see Section 2), the observer is executed for each event of a given trace. This means that, whenever the currently processed event is in the set of observed events evt, the observer will trigger the defined action $act1$. For instance, the execution of the

[2] For more information about jQuery and selectors we refer the reader to the jQuery API documentation http://api.jquery.com/category/selectors/.

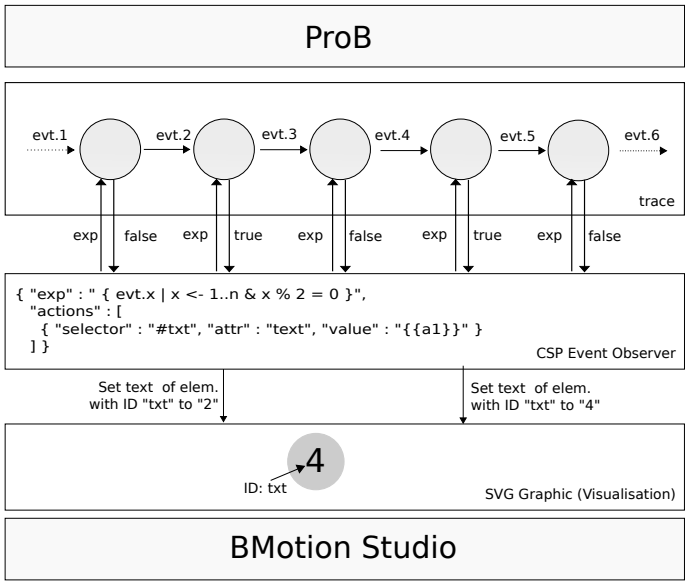

Fig. 2. The function of the CSP event observer

event *evt*.4 causes the observer to set the value of the text field element to "4" as demonstrated in Fig. 2.

Creating a Visualisation. BMotion Studio provides a graphical editor with different views and wizards that supports users in creating visualisations for formal models. Fig. 3 shows the bully algorithm visualisation template opened in the graphical editor (the bully algorithm visualisation will be introduced in Section 4). The editor consists of a set of tools (1) for creating SVG widgets (e.g. visual elements as shapes and images), a canvas (2) holding the actual visual elements, a view (3) for editing observers, and another view (4) for manipulating the attributes of the currently selected visual element in the canvas. The corresponding JSON file which contains the observers is created by the editor automatically. We extended the graphical editor of BMotion Studio in order to support the editing of CSP event observers.

Running a Visualisation. Once a visualisation template is created, it can be started with BMotion Studio as shown in Fig. 4. BMotion Studio uses the default web browser of the user's operating system to view the visualisation and the ProB tool to animate the corresponding CSP-M model.

The user can access the entire function range of ProB. For instance, Fig. 4 shows two views (Events and History) that come from ProB. The first one (Events) lists all possible events that are available in the current state of the animation. The second one (History) shows the executed events so far. The left side of Fig. 4 shows the visualisation of the trace that is displayed in the History view. If the user executes an event in the Events view, a new trace (the trace generated so far

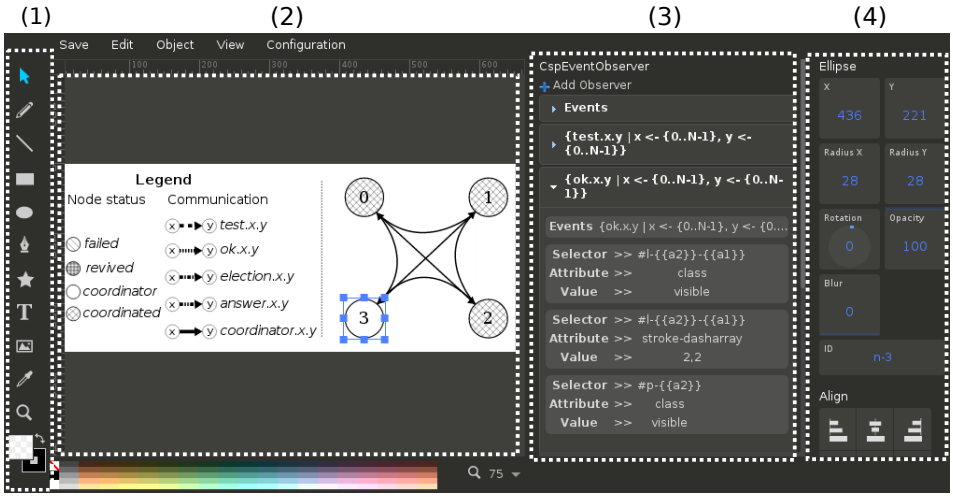

Fig. 3. CSP support within BMotion Studio graphical editor

plus the recently executed event) is provided which is visualised according to our approach.

4 Case Studies

In order to test our approach, we successfully created various visualisations for CSP specifications that we have found in the literature. In this work we present the visualisation of the bully algorithm specification from [13] and of the level crossing gate specification from [14]. The specifications are written in the machine readable dialect CSP-M and have not been modified for the visualisation we have created. Both visualisations were created by means of the built-in graphical editor of BMotion Studio. However, for presentation purposes the observers of the visualisations are described in the JSON notation in this section.

4.1 The Bully Algorithm

The algorithm represents a method of distributed computing for electing a node to be the coordinator amongst a group of nodes. Each node has a unique ID and the algorithm intends to select the node with the highest ID to be the coordinator. It is assumed that the nodes may fail and revive from time to time and the communication between the nodes is reliable. Three types of messages are defined within the design of the algorithm: *election* (announcing an election), *answer* (responding to an election message), and *coordinator* (announcing the identity of the coordinator).

The specification from [13] defines six additional types of events needed for the formalisation of the algorithm in CSP: the *fail* and *revive* events (for modelling

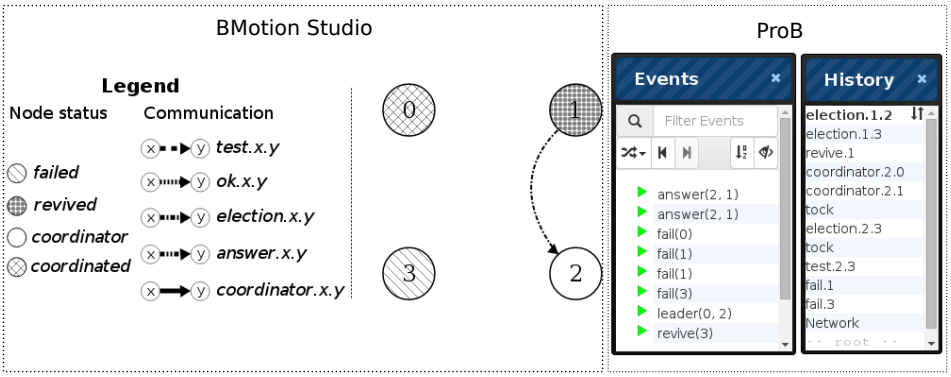

Fig. 4. The Bully Algorithm visualisation

failing and reviving of a node), the *test* and *ok* events (for simulating a test-response communication), the *leader* events (for indicating the coordinator of a living node), and the *tock* event (for modelling timeouts and time).

Visualising the Bully Algorithm. In general, we want to visualise the process of electing a leader in the network. More precisely, we aim to visualise the *Network* process of the CSP specification. As the bully algorithm specification in [13] is presented for a network with four nodes, we also intend to create a visualisation for four nodes (the nodes are enumerated from 0 to 3). Fig. 4 demonstrates the visualisation of a particular trace.

There are two major aspects of the specification that we want to visualise: the nodes and the communication between the nodes. Each node is visualised by means of a circle in which the respective ID is positioned, whereas the communication between the nodes is illustrated by directed arrows. Each directed arrow is made up of a line and a corresponding arrowhead.

To each visual element in the visualisation we assign a unique ID referring to the elements in the CSP specification. Thus, the node with ID x in the CSP specification is presented by the circle with ID "n-x" in the visualisation. Additionally, a message transfer from the node with ID x to the node with ID y is represented by the line with ID "l-x-y" and the arrowhead with ID "p-y" (i.e. the arrow connecting "n-x" and "n-y"). In this section, both symbols x and y stand for an integer ranging from 0 to 3.

We can classify all types of events in the specification into the following groups:

- **status:** Events that can change the status of a particular node x: $fail.x$, $revive.x$, $coordinator.x.y$, and $leader.x.y$.
- **message:** Events illustrating a message transfer from node x to node y: $test.x.y$, $ok.x.y$, $election.x.y$, $answer.x.y$, and $coordinator.x.y$.
- **hidden:** Events that are not considered in the visualisation: *tock*.

Thus, we can infer that there are two general types of observers to define: the *status* and the *message* observers. Note that each *coordinator* event ($coordinator.x.y$) has been included in the first two groups above. This is because in the specification

each of the *coordinator* events intends to identify the coordinator (x) and at the same time represents a message transfer (to node y).

The status of a node usually changes when one of the *status* events has been executed. Each node, except for the node with the lowest ID[3], can have the following status: `failed`, `revived`, `coordinator`, or `coordinated`. A unique fill pattern has been selected for distinguishing each possible status of a node (see legend in Fig. 4).

In order to associate a *status* event from the CSP specification with a node in the visualisation, we use the selector "#n-{{a1}}" in the definition of the respective observer. The construct "{{a1}}" is used in the selector for obtaining the value of the first argument of the respective *status* event. For example, the observer for changing a status of a node to `failed` can be defined as follows:

```
{ "exp": "{fail.x | x <- {0..N-1}}",
  "actions": [ {"selector":"#n-{{a1}}",
                "attr":"fill", "value":"url(#diagonalHatch)"} ] }
```

The observer will fill the respective node with a diagonal hatch pattern whenever a $fail$ event has been processed. For instance, the node with ID "n-3" will be filled with a diagonal hatch pattern when the event $fail.3$ has been processed. In a similar fashion we have defined the observers for the other node status changes.

For creating the *message* observers we need to consider both arguments of the *message* events. The types of the messages are distinguished by different stroke patterns (see Fig. 4). Thus, each *message* observer, except for the *coordinator* observer (this observer has three actions), has two actions: one action for appearing the arrow (the line and arrowhead constituting the respective arrow in the visualisation) and one action for changing the stroke pattern of the arrow. For instance, the observer for visualising the election message can be defined as follows:

```
{ "exp": "{election.x.y | x <- {0..N-1}, y <- {0..N-1}}",
  "actions": [ { "selector": "#l-{{a1}}-{{a2}}, #p-{{a2}}",
                 "attr": "class", "value": "visible" },
               { "selector": "#l-{{a1}}-{{a2}}",
                 "attr": "stroke-dasharray", "value": "5,2,2,2" } ] }
```

To provide a clear visualisation an additional observer has been added to hide all arrows after performing an arbitrary event. This observer is applied on the currently processed event before all other defined observers.

The initial state of the specification and the visualisation is the state in the network where all nodes are alive and the coordinator is the node with the ID 3 (the node with the greatest ID). Additionally, no message exchanges are performed.

4.2 Level Crossing Gate

The model of the first case study introduced in [14] specifies a level crossing gate of a single railway track along which trains move only in one direction. The track is divided into segments such that each of the segments is at least as long as any

[3] The node with ID 0 can never be a coordinator as there is no node with a lower ID.

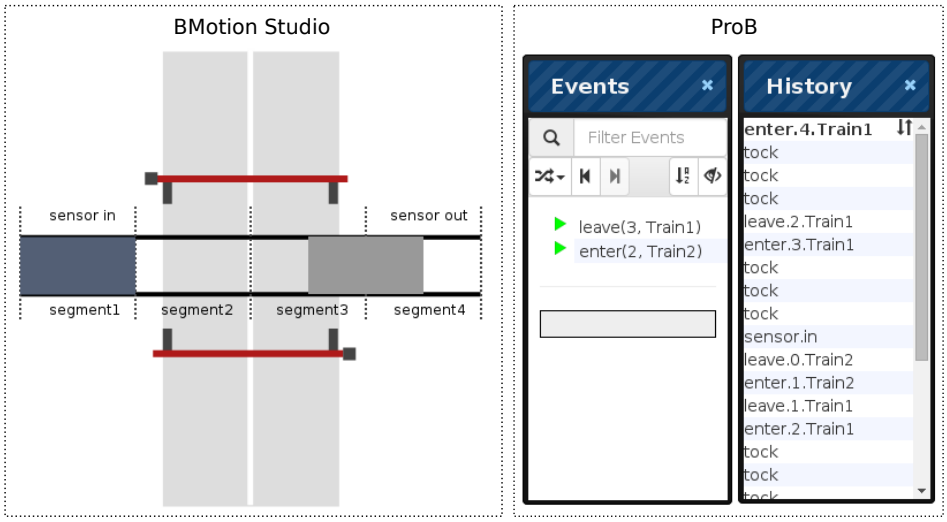

Fig. 5. The Level Crossing Gate visualisation

train. There are five track segments considered for the level crossing gate where one of the track segments represents the outside world.

The track segments are numbered. The input sensor is placed in segment 1 and the crossing and output sensors in segment 4. The outside world segment is identified by 0. A train enters segment $(i+1)$ before it leaves segment i. Entering and leaving of a segment are specified by the events *enter* and *leave*, respectively. The entering of train t into segment j is described by *enter.j.t*. Accordingly, the leaving of train t from segment j is designed by means of the event *leave.j.t*.

The sensors send control signals to the gate. The gate goes down after a train enters segment 1 and accordingly the gate goes up after the train leaves segment 3 and no train is moving along the segments 1 to 2. The control signals sent by the input and output sensors are specified by the events *sensor.in* and *sensor.out*, respectively. The communication between the controller and the gate processes is specified by the channel *gate* which defines four different events. The events *gate.go_down* and *gate.go_up* represent the commands from the controller to the gate for moving the barriers down or up. And the events *gate.down* and *gate.up* denote the confirmations from the gate sensors that the barriers are down or up, respectively.

In addition, timing constraint are set for the trains moving on the tracks. The speed of each train is determined by how many units of time a train can spend per track segment. This additional property is required since the goal of the system is to guarantee via timing that the gate is up and down at appropriate moments. In the CSP model the speed of a train per track segment has been set to three time units. A unit of time is denoted by the *tock* event in the level crossing gate specification.

Visualising the Level Crossing Gate. In our visualisation (see Fig. 5) we assume that the trains are moving from left to right. Track segments 1 to 4 are illustrated by rectangles separated by vertical, dotted lines. Segment 0, which represents the outside world, can be seen as the space left from track segment 1 and the space right from segment 4. A train leaves the outside world after entering track segment 1 and a train enters the outside world before leaving track segment 4. The length of each of the track segments 1 to 4 in the visualisation is considered to be 100 pixels.

Since the model from [14] handles two trains, we also intend to visualise only two trains (these are indicated as *Train1* and *Train2*). Both trains are represented by two boxes coloured in grey and slate grey, respectively. Moving of a train along the track is simulated by shifting the respective box from left to right. In order to simulate a movement along the track segments, we shift the respective box 50 pixels from left to right. In doing so, entering of a new segment is represented such that the box is laid half on the new segment and half on the previous. On the other hand, when the train leaves a track segment, the box is moved fully on the recently entered segment. Referring to Fig. 5, the grey box representing *Train1* is laid half on segment 4 and half on segment 3 after executing the event $enter.4.Train1$, whereas *Train2* (the slate grey box) is moved fully on segment 1 after performing consecutively the events $enter.1.Train2$ and $leave.0.Train2$. We have set each box representing a train to the length of 100 pixels.

For visualising the movement of the trains, we defined two observers that listen respectively to the events $enter.j.t$ and $leave.j.t$. Both observers contain an action that changes the *transform* attribute [21] of the matched visual element. For instance, the *leave* observer is defined such that by executing an event $leave.j.t$ the visual element with the ID "train-t" (t refers to the second argument of the *leave* events) will be moved 50 pixels to right by setting the $transform$ attribute to the value $translate(50, 0)$. Thus, the observer for leaving a track is defined as follows:

```
{ "exp": "{leave.j.t | j <- {0..3}, t <- {Train1,Train2}}",
  "actions": [ { "selector":"#train-{{a2}}",
                 "attr":"transform", "value":"translate(50,0)" } ] }
```

Note that the *leave* observer does not fire its actions when an event $leave.4.t$ is executed since in our visualisation the respective box "train-t" is intended to be moved on the left site of track segment 1 when the event $enter.0.t$ is executed. We decided to define the observers in this way because after entering the outside world (track segment 0) and leaving at last track segment 4, the same train can enter the crossing gate segments once again.

For the overall visualisation we defined four different observers. The other two observers are responsible for simulating the up and down movement of the barriers in the visualisation after proceeding of the events *gate.up* and *gate.down*, respectively. For this, we created for each of the barriers two visual elements that illustrate accordingly the two possible states of the appropriate barrier: barrier is up and barrier is down. This means that we have four visual elements illustrating the different positions of the barriers. When, for example, the event *gate.down* is processed, then the *go-down* observer executes two actions. The first is to hide all barrier elements and the second action is to display the visual elements representing that the barriers are down. The hiding and displaying of the barriers are

realised by setting the "opacity" attribute of the visual elements to 0 and 100, respectively. The *go-down* observer is given as follows:

```
{ "exp": "{gate.down}",
  "actions": [
    { "selector":"g[id^=gate]", "attr":"opacity", "value":"0" }
    { "selector":"g[id^=gate-go_down]", "attr":"opacity", "value":"100" }]}
```

Analogously, we defined the *go-up* observer. The initial state of the specification and its visualisation is the state in which both trains are in the "outside world" track segment and both barriers are up.

5 Application of the Approach

Using validation tools for performing various consistency checks automatically is a powerful technique for verifying the correctness of the analysed specification. A failure of a consistency check is mostly reported by producing of a counterexample (very often presented as a trace leading to an error state). However, trying to understand the failure behaviour of the model by simply examining the trace can sometimes be difficult as the error trace may, for example, be the result of the interaction of various components in the specified system. Thus, using a visualisation in order to facilitate the effort of understanding the error trace can be very useful.

In this Section we show how the bully algorithm visualisation introduced in Section 4 may, for example, contribute to the better understanding of an erroneous behaviour in the models.

For example, the trace of the *Network* process of the bully algorithm model

$$\langle fail.2, fail.3, test.1.3, tock, election.1.3, election.1.2, revive.2, revive.3,$$
$$coordinator.3.2, fail.3, test.0.3, tock, coordinator.1.0, leader.2.3\rangle$$

represents a sequence of events leading to a state in the network in which the elected leader is not the living node with the greatest ID. In general, the false behaviour that is explicitly discussed in [13] illustrates a problem occurring by a certain combination of node failures and mixing up various elections.

While examining the given error trace, it is hard for the user to reproduce and to see the actual problem. In contrast, Fig. 6 shows a stepwise graphical representation of the error trace. The user can see at a glance the erroneous behaviour that is shown in the last step of the trace (after performing leader.2.3) in the graphical representation.

6 Conclusion

In this paper, we presented an approach for creating domain specific visualisations of CSP-M models and an implementation based on BMotion Studio. In particular, we extended BMotion Studio and the built-in graphical editor with a new observer type (CSP event observer) that implements the algorithm presented in Section 2.

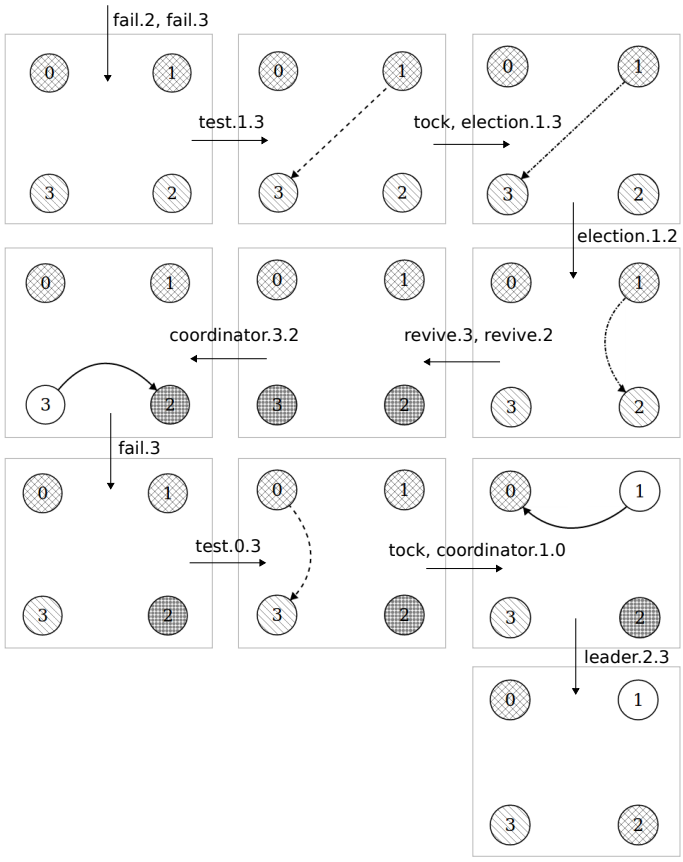

Fig. 6. A stepwise visualisation of a trace of the Bully Algorithm model

The difference between our contribution and the primary approach of BMotion Studio (the domain specific visualisation of Event-B models) is imposed by the question of what is to be visualised of a model. On the one hand, in CSP each trace is mapped to a particular visualisation. On the other hand, in Event-B the information to be visualised is given by the states (e.g. the values of variables) of an Event-B model, where each state is mapped to an individual visualisation.

We tested our approach by creating visualisations of various CSP-M models. A demonstration of our approach is given by visualising the bully algorithm specification from [13] and the level crossing gate specification from [14]. We also have shown how our approach could be of use in the process of analysing and validating CSP specifications.

Our tool comes with a graphical editor that can be used to create easily visualisations. The developer of a visualisation remains in the CSP domain. This means that only CSP expressions and jQuery selectors (see Section 3) are required for establishing the link between a visualisation and the CSP model. Moreover, a modification of the CSP model is not necessary to create a visualisation for it.

A domain specific visualisation of a CSP model can be useful in various ways. For example, the graphical representation of the behaviour of the CSP processes can be helpful for discussing the specification with non-formal method experts and for the further development of the specification.

We also believe that our approach may be of use to identify inconsistencies or unexpected behaviours within the specification. Indeed, in the process of examining the various case studies, the visualisation helped us to better understand some of the unexpected behaviours (error traces) discovered by validating the corresponding specification (see Section 5).

Finally, we believe that our approach may be useful for teaching formal methods, as the execution of a specification with a graphical representation may give a better idea and overview of the system being modelled. For instance, we used our approach successfully in our lectures as a way to present formal models to students and to motivate them to write their own formal models.

Related Work. BMotion Studio was initially developed for creating domain specific visualisations of Event-B models [8]. Our approach extends BMotion Studio to permit users to also create visualisations for CSP-M.

The tools presented in [4] and [16] support the creation of domain specific visualisations for Classical-B. In contrast to our approach, both tools require the user to set up scripts in order to link the visualisation to the model.

Our approach uses ProB [10] to execute a CSP-M specification. ProB and other CSP tools [18, 19] are capable of displaying graphs of processes and counterexamples. Whereas, the purpose of our work is to provide a tool that allows the user to create custom visualisations that are specific to a domain.

A central goal of our work is to gain a better understanding of CSP models by creating domain specific visualisations. A different approach has been taken by [11], which presents a tool for visualising CSP in UML.

References

1. ADVANCE Deliverable D4.2 (Issue 2). Methods and tools for simulation and testing I, Mar. 2013.
2. ECMA-404 The JSON Data Interchange Standard. Ecma International, Oct. 2013.
3. J. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, June 2010.
4. J. Bendisposto and M. Leuschel. A generic flash-based animation engine for prob. In J. Julliand and O. Kouchnarenko, editors, *Proceedings of B 2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 266–269. Springer, 2007.
5. Formal Systems (Europe) Ltd. *Process Behaviour Explorer (ProBE User Manual, version 1.30)*. Available at `http://www.fsel.com/probe_manual.html`.
6. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. Validation of the abz landing gear system using prob. In F. Boniol, V. Wiels, Y. Ait Ameur, and K.-D. Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 66–79. Springer International Publishing, 2014.
7. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.

8. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proceedings FMICS '09*, volume 5825 of *LNCS*, pages 202–204. Springer, 2009.
9. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.
10. M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, pages 278–297, 2008.
11. M. Y. Ng and M. Butler. Tool support for visualizing CSP in UML. In *Formal Methods and Software Engineering*, pages 287–298. Springer, 2002.
12. D. Plagge and M. Leuschel. Validating Z specifications using the ProB animator and model checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer-Verlag, 2007.
13. A. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.
14. A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
15. B. Scattergood and P. Armstrong. CSP-M: A Reference Manual. 2011.
16. T. Servat. Brama: A new graphic animation tool for B models. In *B 2007: Formal Specification and Development in B*, pages 274–276. Springer, 2006.
17. J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In W.-N. Chin and S. Qin, editors, *Proceedings TASE '09*, pages 127–135. IEEE Computer Society, 2009.
18. J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.
19. A. B. A. R. Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Model Checker for CSP. In E. brahm and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.
20. W3C CSS Working Group. Cascading Style Sheets (CSS) Snapshot 2010. http://www.w3.org/TR/css-2010/, May 2011.
21. W3C SVG Working Group. Scalable Vector Graphics (SVG) 1.1 (Second Edition). http://www.w3.org/TR/SVG11/, Aug. 2011.