Mastering the Visualization of Larger State Spaces with Projection Diagrams (Technical Report)

Lukas Ladenberger and Michael Leuschel

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

Abstract. State space visualization is a popular technique for supporting the analysis and verification of formal models. It often allows users to get a global view of the system and to identify structural similarities, symmetries, and unanticipated properties (only to name a few). However, state spaces typically become very large, so human inspection of the visualization becomes difficult. To overcome this challenge, we present an approach which can considerably reduce the size of the state space by creating *projection diagrams*. Moreover, we present an approach to link a projection diagram with a domain specific visualization. The projection diagram construction can be initiated directly from user-selected graphical objects, without the user having to write formulas or having to know the variables or internal structure of the model. This makes the projection diagram inspection and construction accessible to non-formal method experts. These techniques have been implemented within the ProB toolset, and we demonstrate their benefits and usefulness on a case study.
Keywords: Formal Methods, B-Method, State Space, Visualization, Human Inspection, Domain Specific Visualization, Tool Support.

1 Introduction and Motivation

In state-based formal methods, such as the Classical-B method [2] and its successor Event-B [1], the system behaviour is modelled by *states* and *transitions*. A state is a particular configuration of variables, whereas transitions link two states and represent the evolution of the system. Transitions are triggered by the *execution* of an operation (or event in Event-B). Some states are marked as *initial* and the set of states and transitions reachable from the intial state is the *state space* of the model.

The state space can be constructed and validated automatically via model checking [6]. In this process, the validity of temporal properties will be checked, but the state space itself is "invisible" to the user. However, often it is important for the developer or a domain expert to inspect the state space (or parts of it) manually. This can be achieved interactively with animation [8] or by visualizing the state space [23]. The latter can be especially useful to identify structural similarities, symmetries, and unanticipated properties from the system (only to name a few) [23]. However, most state space visualization tools and techniques do not scale well. Indeed, the number of states and transitions typically become very

large, especially at more concrete, refinement levels, and human inspection of the visualization thus becomes a very difficult task. As an example, consider the state space visualization shown in Figure 1 (the reader is not expected to be able to read the diagram, just get a general impression of the problem statement). The visualization was generated with ProB [18], a validation toolset with support for Event-B [1] and Classical-B [2], as well as other formalisms (e.g. [19], [10] and [21]). The visualization shows the full state space (145 states and 673 transitions) of the first refinement level of the landing gear system Event-B model taken from [9]. Although the visualization shown in Figure 1 is produced at an abstract level (the whole model covers 6 refinement steps), it is already hard to grasp by humans.

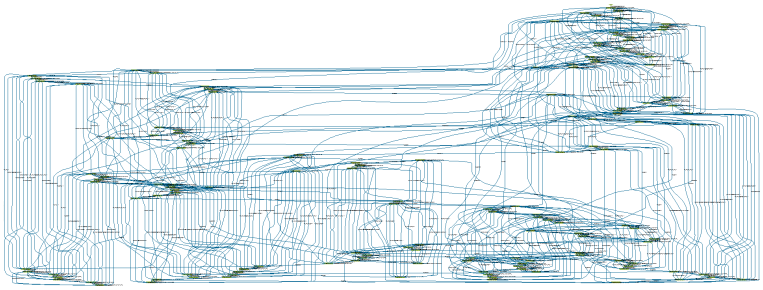

Fig. 1: Full state space visualization of the first refinement of the landing gear system

To overcome this challenge, we present an approach to considerably reduce the complexity of a state space visualization by creating *projection diagrams*. The main objective of the approach is to support human verification of the system by highlighting relevant aspects of the model (e.g. certain variables or a particular behaviour), while hiding information that is not relevant from the diagram. We describe the approach in detail and show that even for large or very large state spaces the diagram may provide valuable insights. The approach has been implemented into the ProB toolset with support for Event-B, Classical-B, TLA$^+$ and Z models. However it is generic so that it can also be integrated into another tool that is capable of producing a state space of a formal model.

In the second part of this paper, we present an approach to link a projection diagram to a domain specific visualization developed with BMotion Studio [15]. The resulted *domain specific diagram* consists of the basic projection diagram enhanced with graphical elements that come from the linked domain specific visualization. An important insight is the fact that the diagram can be generated from the graphical visualization directly, without the user having to know the variables of the model nor having to type expressions in a formal modelling language. We explain the underlying algorithm of the domain specific diagram and provide an implementation that comes as an extension of the ProB toolset. In order to demonstrate the approach, we provide a live visualization, that can be tested and evaluated online at [14]. Finally, we draw conclusions, discuss future improvements for the approach, and compare our work with related work.

Running Example Throughout the paper, we use the Event-B model and the domain specific visualization of the ABZ landing gear system case study[1] taken from [9] to demonstrate our approach. The full specification and a detailed description of the case study is available at [5].

The landing gear system is composed of three parts: a digital part including the control software, a pilot interface, and a mechanical part which contains the triplicated doors and gears. The system is in charge of controlling the retraction and extension sequence of the gears with respect to the doors and the pilot handle. The latter serves as the input to the system.

2 Basic Projection Diagram Algorithm

In this paper, we explain our approach based on the Event-B method [1] and the ProB tool [18]. However, the presented approach is generic and could also be integrated for other state-based formal methods and tools that are capable of producing a state space of a formal model. For instance, the approach has also been applied to Classical-B, TLA$^+$ and Z models.

The starting point of our approach is to explore the state space of a formal model. This can be achieved via model-checking [6] or interactively with animation [8]. Note that for our approach it is not mandatory to exhaustively explore the full state space of the formal model. The algorithm can also be applied on partial explored state spaces and provides feedback about which states have not yet been fully explored (see Section 2.1). As described in [20], the state space can be viewed as a non-deterministic labelled transition system (LTS):

Definition 1 (LTS). *An LTS is a 4-tuple (Q, Σ, q_0, δ) where Q is the set of states, Σ the alphabet for labelling the transitions, q_0 the initial state and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. By $q \xrightarrow{a} q'$ we denote that $(q, a, q') \in \delta$.*

Figure 2 shows a simple example of an LTS for an Event-B model with two variables x and y. Each node in the graph represents a state of the model, where each state is defined by a particular configuration of the two variables x and y. In the following, we use the notation $[v_1 = r_1, ..., v_n = r_n]$ to name the configuration of a state, where $v_1 = r_1, ..., v_n = r_n$ are the variables (v_x) and their values (r_x) in the respective state. For instance, the initial state q_0 (the node with the incoming serrated arrow) has the configuration $[x = 0, y = 0]$.

The edges in the graph represent the possible transitions of the LTS (δ). In Event-B, a transition is the execution of an *event*, which is specified as a generalised substitution allowing deterministic and non-deterministic assignments to be specified. Each transition is labelled with the corresponding event name, where $\Sigma = \{set\_x, set\_y, reset\}$ defines the names of the possible events. For instance, the event set_x can modify the value of the variable x from 0 to 1, which is denoted by the transition $[x = 0, y = 0] \xrightarrow{set\_x} [x = 1, y = 0]$ shown in Figure 2.

The next step in the construction of a projection diagram of an LTS consists of defining a *projection function*. All states with the same value for the projection

[1] The website http://stups.hhu.de/ProB/index.php5/ABZ14 contains the model and a live visualization of the ABZ landing gear system.

function are merged into an *equivalence class*. A transition leads from one equivalence class C to another C' if there is a transition from one state $s \in C$ to a state $s' \in C'$. Formally, one can define the projection of an LTS as follows:

Definition 2 (Projection). *Let $L = (Q, \Sigma, q_0, \delta)$ be an LTS and p a projection function with domain Q. The projection of the LTS using p, denoted by L^p, is defined to be the LTS $(Q^p, \Sigma, p(q_0), \delta^p)$, with $Q^p = \{p(s) \mid s \in Q\}$ and $\delta^p = \{p(s) \xrightarrow{ev} p(s') \mid s \xrightarrow{ev} s' \in \delta)\}$.*

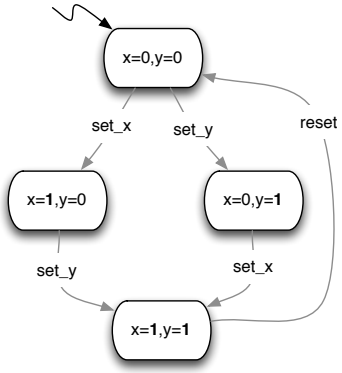

Fig. 2: Simple LTS

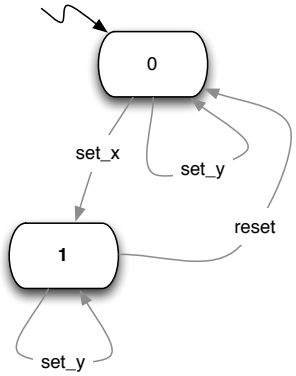

Fig. 3: Projection of the LTS onto the variable x

Each element in Q^p represents an equivalence class, where each equivalence class merges the states of Q (the states of the original LTS) that have the same value for the projection function p. To illustrate the idea of a projection, consider Figure 3. The diagram shows the projection of the simple LTS of Figure 2 onto the variable x using the projection function $p([x = vx, y = vy]) = vx$. The projection of an LTS may obviously not be equivalent to the original LTS, as the sequences of the events are not necessarily possible in the original LTS. However, all sequences of the original LTS are possible in any projection of it.[2] In order to reduce clutter in the projection diagram, one can also remove self loops, i.e., removing the event *set_y*. This is useful, if a user wants to focus on the transitions that can *change* the value of the projection function.

2.1 Categorizing Edges and Equivalence Classes

To provide a more refined visualization we categorize the equivalence classes and edges. We distinguish between *definite* and *non-definite*, as well as between *deterministic* and *non-deterministic* edges. In addition, we distinguish between two types of equivalence classes: the equivalence classes that contain only one single

[2] I.e., the original LTS is a trace refinement of the projection LTS.

state and the equivalence classes that have not yet been fully explored (e.g., if the state space has not been explored exhaustively).

In the following subsections we explain the different types of edges and equivalence classes and illustrate them with an example. To do this, let $L = (Q, \Sigma, q_0, \delta)$ be an LTS and $L^p = (Q^p, \Sigma, p(q_0), \delta^p, E)$ its projection. Given an edge $x \xrightarrow{ev} y \in \delta^p$, we denote x as the *source* and y as the *target* equivalence class. Moreover, we call an edge $x \xrightarrow{ev} y \in \delta^p$ *enabled* for a particular state s, with $s \in x$ if $\exists s' \cdot (s' \in y \wedge s \xrightarrow{ev} s' \in \delta)$.

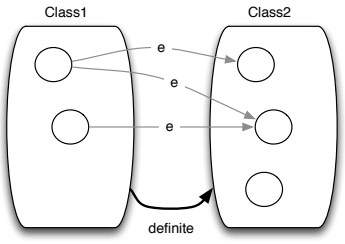

Fig. 4: Definite edge

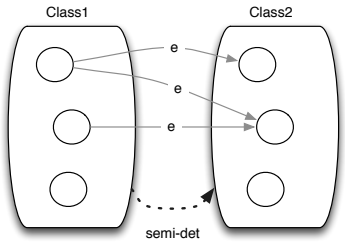

Fig. 5: Semi-deterministic edge

Definite Edges. An edge is definite, iff it is enabled in *all* states of the source equivalence class. Thus, the set of all definite edges of L^p can be defined as follows:

$$\mathit{Definite} = \{x \xrightarrow{ev} y \mid x \xrightarrow{ev} y \in \delta^p \wedge \forall s \cdot (s \in x \Rightarrow \exists s' \cdot (s' \in y \wedge s \xrightarrow{ev} s' \in \delta))\}.$$

Figure 4 illustrates the idea of a definite edge: there is a definite edge between the equivalence classes *Class1* and *Class2* whenever e is enabled in all states of the source equivalence class (*Class1*). An edge is non-definite iff it is not definite. In order to distinguish the different edge types in the projection diagram, definite edges are drawn as solid lines, while non-definite edges are drawn as dashed lines.[3]

Semi-Deterministic and Non-Deterministic Edges. As illustrated in Figure 5, an edge e is semi-deterministic iff the underlying event always leads to the same target equivalence class (*Class2*) from the source class (*Class1*). However, it does not have to be enabled in all states of the source equivalence class (*Class1*). Thus, the set of all semi-deterministic edges of L^p is defined as follows:

$$\mathit{SemiDet} = \{x \xrightarrow{ev} y \mid x \xrightarrow{ev} y \in \delta^p \wedge \neg(\exists z \cdot (z \neq y \wedge x \xrightarrow{ev} z \in \delta^p))\}.$$

Furthermore, we denote an edge as non-deterministic if it is *not* semi-deterministic. Thus, the set of all non-deterministic edges of L^p is composed of all edges (δ^p) expect of the semi-deterministic edges (*SemiDet*):

$$\mathit{NonDet} = \delta^p \setminus \mathit{SemiDet}.$$

[3] In ProB the user can customize the style and colour of the edge types.

Figure 6 shows an example of a non-deterministic edge. Given the three equivalence classes *Class1*, *Class2* and *Class3*, the edge e is non-deterministic if e is enabled and it leads to at least two distinct target equivalence classes (e.g. *Class2* and *Class3*).

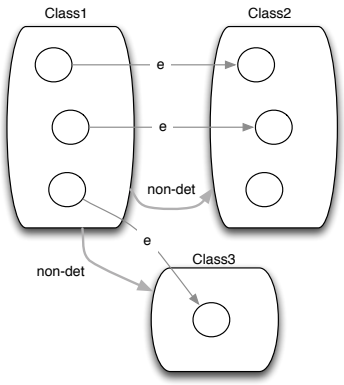

Fig. 6: Non-deterministic edge

Deterministic and Non-Deterministic Definite Edges. An edge is deterministic if it is both semi-deterministic and definite. As an example, the edge e shown in Figure 4 is definite and deterministic. This is because e is enabled in all states of the source equivalence class (*Class1*) and it leads to the same target equivalence class (*Class2*). The set of all edges of L^p that are deterministic and definite is defined as:

$$DetDef = SemiDet \cap Definite.$$

Moreover, edges that are non-deterministic and definite are defined as follows:

$$NonDetDef = NonDet \cap Definite.$$

For instance, the edge shown in Figure 6 is definite and non-deterministic as it is enabled in all states of the source equivalence class (*Class1*) and it leads to two distinct target equivalence classes (*Class2* and *Class3*).

Non-deterministic, semi-deterministic, deterministic and definite edges are distinguished by their colour (which can be set by the user).

Single State and Partial Equivalence Classes. An equivalence class is *single*, iff one state is merged into the equivalence class. Thus, the set of all single equivalence classes can be defined as follows:

$$Single = \{x \mid x \in Q^p \wedge card(\{s | s \in Q \wedge p(s) = x\}) = 1\}.$$

For instance, the equivalence class *Class3* in Figure 6 is *single*, as it contains only one state.

Furthermore, we highlight the equivalence classes that have non yet been fully explored. This may be the case, whenever the full state space has not been exhaustively explored. As in the categorization of edges, the different types of equivalence classes are distinguished by their (user defined) colour.

2.2 Application of the Projection Diagram

Note, from now on we will use projection functions of the form $p(s) = eval(E, s)$, where $s \in Q$, E an expression over the variables and constants of the model, and *eval* is the function that evaluates the expression E in state s. The projection function is thus defined by a "custom" expression E. With this scheme, we can project the state space of a model on a single variable v ($E = v$) but also on a set of variables $v_1, \ldots, v_k$ ($E = (v_1 \mapsto \ldots \mapsto v_k)$). We can also project on particular properties of a variable v, e.g., its cardinality ($E = card(v)$) or its range ($E = ran(v)$).

In this Section we present some example applications of the projection diagram and demonstrate how we have applied it in the process of validating the landing gear model introduced in Section 1.

The handle and gears of the landing gear system are closely related, since the extension and retraction of the gears can always be interrupted by a counter order of the handle. A projection on both aspects of the model (the gears and the pilot handle) helped us to inspect their behaviour in the process of modelling the landing gear system. To do this, consider the visualization of the projection diagram shown in Figure 7. The visualization was produced with ProB and demonstrates the projection of the fourth refinement level of an earlier version of the landing gear model (the full state space covers 6,283 states and 31,299 transitions) using the projection function $p(s) = eval(E, s)$, with $E = ran(gear) \mapsto handle$. Note that $ran(gear)$ is the set of states of the three gears, abstracting away which particular gear is in which state. E.g., $ran(gear)$ has the same value $\{retracted, gear\_moving\}$ for $gear = \{left \mapsto retracted, right \mapsto retracted, front \mapsto gear\_moving\}$ and $gear = \{left \mapsto gear\_moving, right \mapsto gear\_moving, front \mapsto retracted\}$.

Each rectangle represents an equivalence class (all states with the same value for the expression E) and is labelled with the associated expression value, as well as with the number of states that are merged into the equivalence class. A directed edge between two equivalence classes represents a transition which is labelled with the associated event name.

The diagram confirms that in every state the handle can be toggled (the corresponding transitions are definite) and that the only event which can modify the handle is *env_toggle_handle*. We can also see that the gears do not jump directly from *retracted* to *extended* or vice versa. The transitions for changing the gear state are not definite: this is to be expected, as the doors have to put into the correct position first. This again confirms our intuition about the modelled system. A quick look at the diagram also reveals an unexpected behaviour. The three gears are always in the same state ($card(ran(gear))$) is always 1), which is too restrictive: the gears and doors should move asynchronously. Figure 8 shows the

projection of the sixth refinement level, where the unexpected behaviour was fixed using the same projection function as before.[4] Now, the diagram clearly shows that the three gears can now move at different speeds.

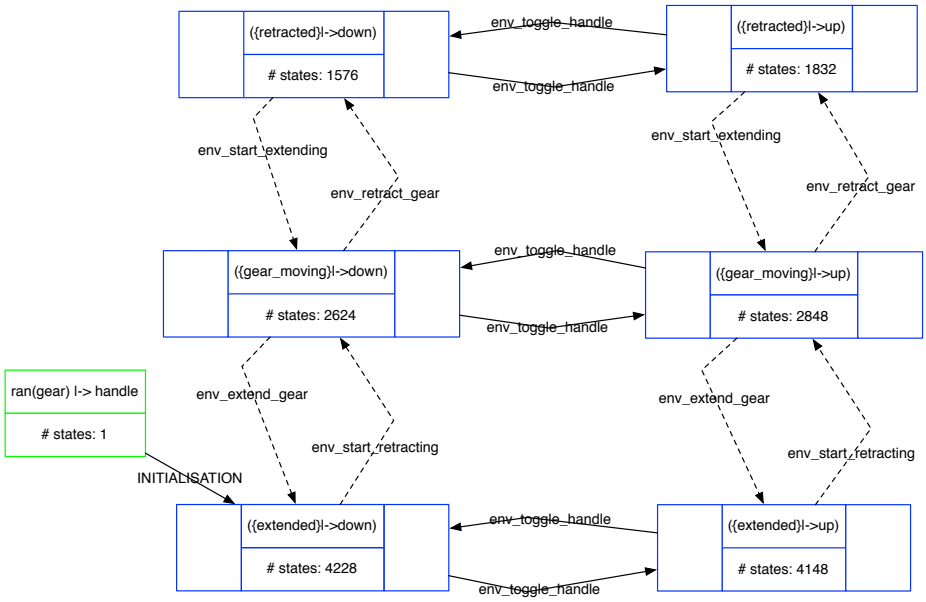

Fig. 7: State space projection of an earlier version of the fourth refinement level on variable handle and range of gears

Figure 9 illustrates how one combine various variables into a single expression. The Figure projects the state space of the "standard" scheduler benchmark example from [16] (also used, e.g., in [20]), which schedules processes and keeps disjoint sets of waiting, ready and active processes. In the Figure we abstract away from the process identities, by computing the cardinality of these sets. Furthermore, we add these sets together, to project on the total number of processes. One can clearly see that only two events change the total number of processes: *new* and *del*. Moreover, *new* is always enabled when less than 3 processes exist, while *del* is only possible when more than one process exists and is not always possible. This confirms our intution, as active processes cannot be deleted straightaway. Figure 9 shows how one can focus on very specific aspects of a model using the projection diagrams. We believe that one should probably generate a variety of projection diagrams for any particular model — each for very specific aspects — and that they can or should be incorporated into the documentation accompanying the model. In the next section we show how we can increase the value of these diagrams, by incorporating graphical elements and making them accessible to domain experts not versed in formal modelling.

[4] In the meantime, the variable *gear* has been renamed into *gears*.

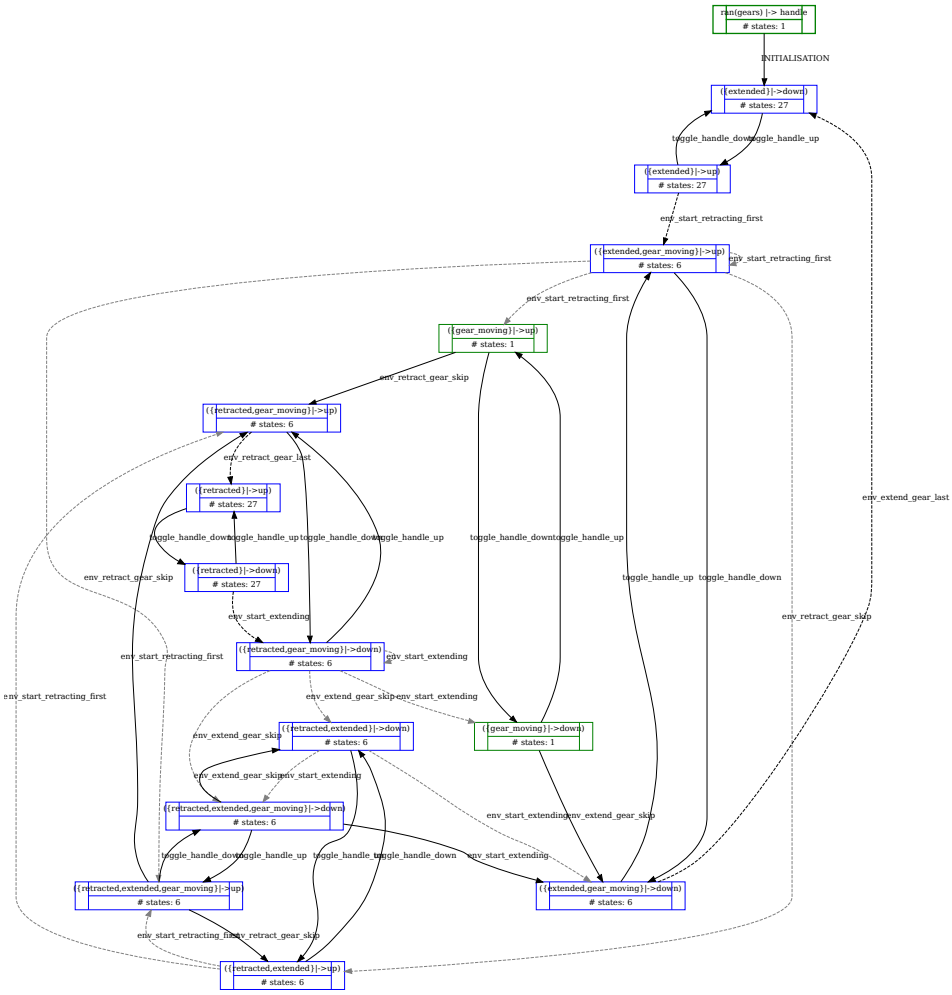

Fig. 8: Projection on expression $ran(gears) \mapsto handle$ of the third refinement

3 Linking with Domain Specific Visualization

BMotion Studio [15] is a tool for creating domain specific visualizations of formal models. The tool provides various graphical elements (shapes, images, buttons,...) that can be used to represent the different aspects of a model. As an example, consider the model of the landing gear system. A domain specific visualization of the model can show the physical environment (gears and doors) and the architecture of the hydraulic part (valves and cylinders). Moreover, *observers* are used to link the formal model with the visualization and allow the tool to compute a visualization for any given state. An observer binds an expression or a variable to a graphical element (e.g. a shape or an image) and changes its properties (e.g.

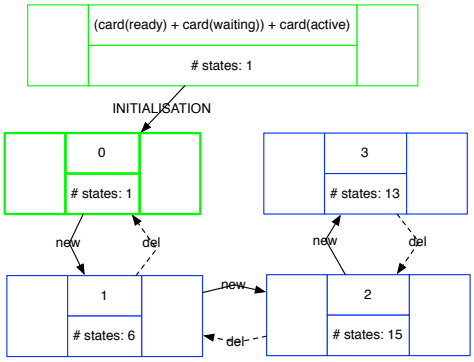

Fig. 9: Projection on expression $card(ready) + card(waiting) + card(active)$ of a process scheduler

the colour or position) according to the value of the expression or variable in the respective state.

In this Section we present another diagram type called *domain specific diagram*. The basic idea of the domain specific diagram is to create a projection of graphical elements of a domain specific visualization. To do this, we apply the following approach:

1. The user selects the graphical elements for the projection from the domain specific visualization.
2. We determine recursively the observers of the selected graphical elements[5] and derive the expressions f_i (which can be simple variables) that are required to draw the state of the selected graphical elements.
3. We construct the projection expression $E = f_1 \mapsto ... \mapsto f_n$ and compute the projection diagram using the projection function $p(s) = eval(E, s)$ as described in Section 2.
4. For each equivalence class of the projection diagram, we compute the representation of the selected graphical elements according to the value of the projection function of the respective equivalence class. Note that, if computed separately, all states in this equivalence class would yield the *same* visualization for the selected graphical elements.
5. We assign the adapted graphical elements to the corresponding equivalence classes and create the domain specific diagram.

For generating the diagram we use the *Cytoscape.js* framework[6], a JavaScript library for analysis and visualization of graphs. One main advantage of Cytoscape.js is, that the diagram becomes interactive, so that the user can rearrange the nodes

[5] In BMotion Studio graphical elements are arranged hierarchically. Thus, we also need to determine recursively the observers of the child graphical elements.

[6] http://js.cytoscape.org/.

an edges as desired. A good layout of the nodes and edges of the diagram can be crucial for its readability [11].

To illustrate the idea of the approach, consider the domain specific diagram in Figure 10 produced with ProB and the JavaScript observer in Figure 11. In line 1 we register a *formula observer* [13] on the graphical element that matches the selector "#handle", i.e. the image that represents the handle (the prefix "#" is used for matching a graphical element by its ID). Line 2 states that the observer should observe the variable *handle*, i.e. the variable that defines the state of the pilot handle of the landing gear system. In lines 3 to 6 we define the action which is applied on the image element whenever a state change occurred. In this case, the observer sets the path (*src*) of the image element (*origin*) to a new path that is constructed based on the value of the variable handle in the current state (*val[0]*)[7]. The diagram in Figure 10 demonstrates the projection on the pilot image element (the graphical element that matches the selector "#handle") using the projection function $p(s) = eval(E, s)$, where $E = handle$ is automatically derived from the observer shown in Figure 11. For each equivalence class, the representation of the image element is computed. To do this, we apply the observer in Figure 11 manually using the value of the projection function of the respective class. The adapted image element is then assigned to the equivalence class. For instance, the diagram in Figure 10 shows the two possible states of the handle (up and down) and its graphical representation. As with the basic projection diagram, every node in the graph represents an equivalence class and we categorize the equivalence classes and edges as described in Section 2.1.

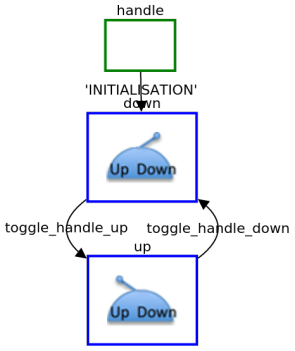

Fig. 10: Projection on handle

```
1 $("#handle").observe("formula", {
2  formulas: ["handle"],
3  trigger: function (origin, val) {
4   origin.attr("src",
5   "handle_" + val[0] + ".png");
6  }
7 });
```

Fig. 11: JavaScript observer for image element that represents the handle

Another example is shown in Figure 12, illustrating the projection on the graphical elements representing the handle and the front gear cylinder of the fifth refinement level of the landing gear system using the derived projection function $p(s) = eval(E, s)$, with $E = handle \mapsto gears(front)$. The visual feedback may help the user to localize specific equivalence classes for further inspection, even

[7] The domain specific visualization provides different images to represent the states of the handle (down and up).

if the diagram slightly becomes larger or the layout of the diagram is unflattering. For instance, one can identify at a glance the equivalence classes, where the cylinder is extended or where the handle is set to down.

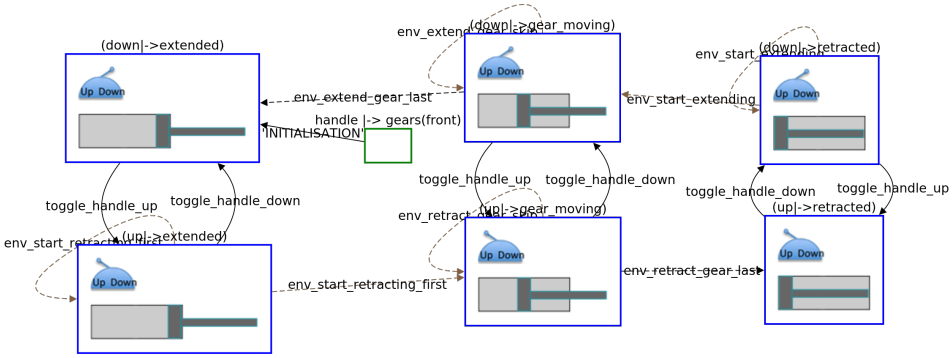

Fig. 12: Domain specific projection on handle and gear

Although developing a domain specific visualization requires extra effort, the benefits of combining it with a projection diagram can be considerable. For instance, it can be used to get a common understanding about the underlying model within a team or to discuss the model with non-formal methods experts. As an example, consider Figure 13 that shows the projection on the cabin of a simple lift model[8]. One can see at a glance (without knowledge about the underlying model or the used formalism), that the cabin is able to stop and to open the door at all three floors. The diagram also confirms that the cabin door must always be closed (indicated by a gray fill) before the lift can move. This is indicated by the solid edges labelled with the event *close_door*.

The domain specific diagram feature can even support the development of a domain specific visualization. For example, we have used it to check if a particular graphical element represents all relevant states properly and to eliminate undesirable behaviour in the domain specific visualization of the landing gear model. Consider the diagram shown in Figure 10 which is very small (although the full state space covers 25,217 states and 149,041 transitions) and easy to inspect by a human: one can see at a glance that the handle behaves as expected in the domain specific visualization (and in the formal model). Similar diagrams can also be created for other graphical elements (e.g. Figure 12).

4 Related Work

Several other approaches exist for state space visualization. In this work we are concerned with reducing the size and complexity of states space visualizations which have a large number of nodes and transitions. However, we are also concerned with supporting the verification of formal models, e.g. by producing state

[8] The formal model can be found at [14].

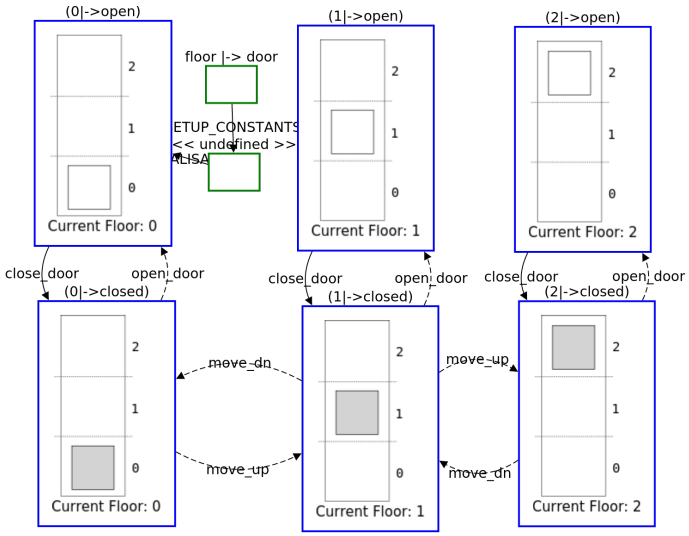

Fig. 13: Projection on cabin of simple lift model

space visualizations that are even accessible for non-formal method experts. Thus, we compare our work with other approaches that tend to improve the visualization of larger state spaces, as well as with other approaches that have the goal to make formal models accessible to non-formal method experts. Because our work has a strong focus on the B-method, we first compare our work with state space visualization approaches with support for the B-method. Afterwards, we compare our approach with other related work.

State Space Visualization for the B-Method. Our approach has been implemented into the ProB toolset [18]. ProB also provides further state space visualization features with the motivation to reduce the complexity of the produced graphs. Two of them are presented in [20]: the *signature merge approach* and the *DFA-abstraction algorithm*. The signature merge approach is very similar to our approach: while our approach merges all states based on a projection function, the signature merge approach merges all states with the same enabled events to a common *signature*. While the approach can be tuned by deselecting events from the signature, our approach can be tuned by adapting the projection function. For instance, our approach enables the user to focus on certain variables (or even just properties of those) of a formal model and to see only those events which can modify those variables. On the other hand, the basic idea of the DFA-abstraction algorithm is to abstract the labelling of the edges, i.e., to abstract away from event arguments and to apply the classical minimization algorithm for Deterministic Finite Automaton (DFA). The DFA-abstraction algorithm produces a visualization in which the transitions are equivalent to these in the original state space. However, this produces a larger graph which may still be difficult for humans to grasp [20].

In [12] the authors present two complementary approaches to increase the understanding of formal models by producing *behavioural views* from B models, rather than focusing on reducing the size and complexity of larger state spaces. In particular, the *under-approximation* approach also uses the ProB model-checker [18] to exhaustively explore the state space of a formal model as a first step. While our approach groups nodes based on a projection function, the under-approximation approach produces a graph by grouping concrete states satisfying a same abstract state predicate.

More State Space Visualization Approaches. The work done in [24, 25] addresses the problem of visualizing large state spaces and presents a tool called *DiaGraphica* with different features for the interactive visual analysis of state spaces. The tool supports the *fsm* input file format[9] for representing state spaces as plain text. It would be interesting to see if the work presented in this paper could be combined with the DiaGraphica tool, i.e. to export the state spaces produced by ProB (the full state space and the state space produced by the basic projection diagram algorithm) into the fsm format and to load it with the DiaGraphica tool.

The muCRL2 system also provides a 3-D state space visualization technique [7, 22], which tries to cater for showing a large number of nodes as opposed to our approach of projecting the state space onto a smaller refinement of it.

5 Conclusion

In this paper we have presented an approach for state space visualization with *projection diagrams*. The main objective of the approach is to considerably reduce the size of a state space and to support human visual verification of the system by highlighting relevant aspects of the model. In the second part of this paper, we have presented an extension of the approach to link a projection diagram to a domain specific visualization developed with BMotion Studio. The approach has been implemented into the ProB toolset with support for Event-B, Classical-B, TLA$^+$ and Z models.

Throughout the paper, we have demonstrated the benefits and usefulness of the approach by applying it on the Event-B model and domain specific visualization of the landing gear system taken from [9]. For this purpose, various example projection diagrams are presented in this paper. Moreover, we created a live visualization that can be tested online [14].

Although the produced projection diagram may not be equivalent to the original state space (as far as the sequences of the events are concerned), the projection may achieve a good result in reducing the size of the state space, while still preserving beneficial information. In particular, the categorization of the edges and equivalence classes proved to be very useful in supporting the inspection of the diagram and to infer useful properties of the respective formal model.

Our approach is also flexible, as the user may adjust the underlying projection function. The possibility to define an individual projection function enables the

[9] http://www.comp.leeds.ac.uk/scsajp/applications/data/fsm.html.

user to *query* the full state space and to obtain only the information in which the user is interested in (comparable with defining queries on a database).

Moreover, we believe that inspecting multiple small projection diagrams (with a manageable number of nodes and transitions) representing different aspects of the model can be more helpfully then inspecting only one big state space visualization. This was also confirmed by applying our approach on validating the landing gear model. One reason for this is that the user can concentrate on a specific aspect of the model (e.g. on certain variables) or to check a particular behaviour, while hiding nonrelevant information from the diagram.

We also believe that a projection diagram may help to verify properties of the model which are hard to express as invariants. For instance, in the landing gear model we used the diagram to verify that the extension and retraction sequence works as desired and that the controller responds correctly to toggling the handle during both sequences.

Finally, we believe that combining the projection diagram with a domain specific visualization affords further advantages. For example, the graphical representation of a specific aspect or behaviour of the model can be helpful for discussing the specification with non-formal method experts and for the further development of the specification. A non-formal method expert can even use this feature without any knowledge about the notation used in formal methods, since the projection is produced based on graphical elements and the underlying projection function is derived automatically from the attached observers.

Evaluation and Future Work. Table 1 shows some runtime statistics obtained after applying the basic projection algorithm (*runtime BP*) and the domain specific projection algorithm (*runtime DSP*) on the models presented in this paper, after the corresponding state space had been fully explored with ProB. We use the projection function $p(s) = eval(E, s)$, where $s \in Q$ and E is the *projection expression* (third column of Table 1). The measured time includes the actual runtime for both algorithms (implemented in ProB) without the time needed to exhaustively explore the full state space (i.e. the model checking time) and without the time needed for generating and layouting the actual diagram. In fact, the model checking and layouting time is not included, because it depends on the used model checking and layouting tool respectively. Moreover, the state space needs to be explored only once in order to generate multiple projection diagrams.

Table 1: Runtime of algorithm for various models and projection functions

Model	**States / Transitions***	**Projection Expression**	**Runtime BP**	**Runtime DSP**
Scheduler	36/121	$card(ready)$ $+card(waiting)$ $+card(active)$	0.01 s	-
Landing Gear (old), 4th Ref	6,283/31,299	$ran(gear) \mapsto handle$	0.02 s	-
Landing Gear, 5th Ref	25,217/149,041	$ran(gears) \mapsto handle$ $gears \mapsto handle$ $handle$ $handle \mapsto gears(front)$	0.98 s 1.06 s 0.77 s 0.88 s	- - 1.59 s 3.20 s
Simple Lift	186/838	$door$ $floor \mapsto door$	0.03 s 0.03 s	0.70 s 1.10 s

*The states and transitions of the full state space of the corresponding model.

In general, the runtime of the DSP takes longer than the BP. This is because the DSP uses the BP to generate the actual data (see Section 3) and needs some additional time to generate the graphical representation of the equivalence classes.

Table 1 also confirms that the runtime of both algorithms (DSP and BP) increases with the number of nodes and transitions of the state space.

In the future, we plan to apply the algorithms on more case studies to obtain additional statistics. Moreover, a good layout of the nodes and the edges of a projection diagram is crucial for its readability and accessibility. A next step would be also to adapt the underlying layout algorithm of the projection diagrams so that the nodes (the equivalence classes) are ordered based on the defined projection function. As an example, this was already done manually in the diagram shown in Figure 7. We ordered the nodes so that the left side of the diagram contains the equivalence classes where the handle is set to down, whereas the right side contains the equivalence classes where the handle is set to up.

We also plan to enhance the projection diagram with interactive features. For instance, it would be desirable to "jump" into an equivalence class and to inspect the individual states which have been merged into it. This could be in particular useful to take a closer look on equivalence classes that have unexpected outgoing edges, e.g. if the user expected a definite edge, but the equivalence class has a non-definite edge instead. One could jump into the affected class and inspect the states in which an event is not enabled.

Finally, we plan to symbolically construct a projection diagram statically using the built-in constraint solver of ProB [17] rather than first having to (exhaustively) explore the full state space using the model-checking feature of ProB. This is related to proof-based approaches such as [4] and [3].

References

1. J. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, June 2010.
2. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
3. J. Bendisposto and M. Leuschel. Automatic flow analysis for Event-B. In D. Giannakopoulou and F. Orejas, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*, volume 6603 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2011.
4. D. Bert, M.-L. Potet, and N. Stouls. Genesyst: A tool to reason about behavioral aspects of B event specifications. Application to security properties. In *ZB 2005*, pages 299–318, 2005.
5. F. Boniol and V. Wiels. The Landing Gear System Case Study. In *ABZ Case Study*, volume 433 of *Communications in Computer Information Science*. Springer, 2014.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
7. J. F. Groote and F. van Ham. Interactive visualization of large state spaces. *International Journal on Software Tools for Technology Transfer*, 8(1):77–91, 2006.
8. S. Hallerstede, M. Leuschel, and D. Plagge. Validation of formal models by refinement animation. *Science of Computer Programming*, 78(3):272–292, 2013.
9. 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.
10. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
11. I. Herman, G. Melancon, and M. Marshall. Graph visualization and navigation in information visualization: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 6(1):24–43, Jan 2000.
12. A. Idani and N. Stouls. When a formal model rhymes with a graphical notation. In C. Canal and A. Idani, editors, *Software Engineering and Formal Methods*, volume 8938 of *Lecture Notes in Computer Science*, pages 54–68. Springer International Publishing, 2015.
13. L. Ladenberger. BMotion Studio for ProB Project Website. http://stups.hhu.de/ProB/w/BMotion_Studio, May 2015.
14. L. Ladenberger. Projection Diagram Website. http://stups.hhu.de/ProB/w/ProjectionDiagram, May 2015.
15. 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.
16. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. Lindsay, editors, *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.
17. M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From animation to data validation: The prob constraint solver 10 years on. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems: Implementation of the B Method*, chapter Chapter 14, pages 427–446. Wiley ISTE, Hoboken, NJ, 2014.
18. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.
19. M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, pages 278–297, 2008.
20. M. Leuschel and E. Turner. Visualising larger state spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 6–23. Springer-Verlag, November 2005.
21. 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.
22. B. Ploeger and C. Tankink. Improving an interactive visualization of transition systems. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 115–124, New York, NY, USA, 2008. ACM.
23. A. Pretorius. *Visualization of State Transition Graphs*. 2008.
24. A. Pretorius and J. van Wijk. Multidimensional visualization of transition systems. In *Information Visualisation, 2005. Proceedings. Ninth International Conference on*, pages 323–328, July 2005.
25. A. Pretorius and J. van Wijk. Visual analysis of multivariate state transition graphs. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):685–692, Sept 2006.