Automatic Flow Analysis for Event-B[⋆]

Jens Bendisposto, Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{bendisposto,leuschel}@cs.uni-duesseldorf.de

Abstract. In Event-B a system is developed using refinement. The language is based on a relatively small core; in particular there is only a very small number of substitutions. This results in much simpler proof obligations, that can be handled by automatic tools. However, the downside is that, in case of software development, structural information is not explicitly available but hidden in the chain of refinements. This paper discusses a method to uncover these implicit algorithmic structures and use them in a model checker. Other applications are code generation, model comprehension, and test-case generation.
Keywords: Event-B, Model Checking, Theorem Proving, Tool Integration.

1 Introduction

Some specification formalisms only have limited ways to express ordering of events. In particular Event-B [1] lacks a notion of sequential composition, or other ways to explicitly describe the ordering of events. If we specify software or systems that include software in Event-B, we often have some implicit algorithmic structure.[1] Unfortunately this information is implicit only and therefore not directly usable by tools nor directly visible to users. This paper discusses a method to uncover this implicit algorithmic structure. This information can be useful for analyzing or comprehending models and for automatic code generation. In this paper we also show how to use this information to improve model checking.

2 Preliminaries

We follow the style of [1] of expressing variables and substitution in formulas. In particular, let $v = v_1, \ldots, v_n$ be a sequence of n distinct variables, $t = t_1, \ldots, t_n$ a sequence of n formulas and F a formula. Then $F[t/v]$ is obtained from F by replacing simultaneously all free occurrences of each v_i by t_i. We let $F(v)$ denote

[⋆] This research is being carried out as part of the DFG funded research project GEPAVAS.

[1] To order events in Event-B the usual method is to introduce abstract program counters.

a formula, whose free variables are among $v_1, \ldots, v_n$. Once the formula $F(v)$ has been introduced, we denote by $F(t)$ the formula $F[t/v]$ with v replaced by t.

In Event-B a state consists of a set of variables that are modified by events. The values of the variables are constrained by invariants $I(v)$. Each event is composed of a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are *parameters* of the event. We will only consider events of the form

```
$evt \mathrel{\widehat{=}}$ any $t$
        when $G(t, v)$
        then $v_{i_1}, \ldots, v_{i_k} := E_1(v, t), \ldots, E_k(v, t)$ end
```

for some $i_j \in i_1, \ldots, i_n$. Note that t can be empty and $G(t, v)$ can be *true*. Also note that k can be 0, in which case we write the action part as skip.

All assignments of an action $S(t, v)$ occur simultaneously. Variables $v_{j_1}, \ldots, v_{j_l}$ that do not appear on the left-hand side of an assignment of an action are not changed by the action. The effect of an assignment can be described by a before-after predicate:

$$S(v, t, v') \ \mathrel{\widehat{=}}\ v'_{i_1} = E_1(v, t) \wedge \ldots v'_{i_k} = E_k(v, t) \wedge v'_{j_1} = v_{j_1} \wedge \ldots v'_{j_l} = v_{j_l}$$

A before-after predicate describes the relationship between the state just before an assignment has occurred, x, and the state just after the assignment has occurred, x'.

Note that Event-B also allows non-deterministic actions of the form $x \ :\in E(t, v)$ or $x \ :\mid \ Q(t, v, x')$. Without loss of generality, we assume that those are rewritten to the above form using new parameters, one for every non-deterministic action which denotes the chosen element. For instance, we rewrite

any max when $max > 10$ then $x :\in 1..max$ end

into

any $max, choice$ when $max > 10 \wedge choice : 1..max$ then $x := choice$ end

3 Dependency Between Events

We are interested in how events influence each other. The motivations are multiple: either we may try to understand the dynamic behavior of our model, we may wish to generate code by determining the control flow or we may wish to improve the performance of model checking.

Suppose we have an event g with action $x, y := (x + 1), 0$. There are various ways it can influence another event:

1. it can disable another event. E.g., the event h with guard $y > 0$ will for sure be disabled after executing g.
2. it can enable another event. E.g., the event h' with guard $y = 0$ would for sure be enabled after executing g.

3. it can be independent of another event. For example, the enabling of the event h'' with guard $z > 0$ would not be modified by executing g, i.e., it will be enabled after g if and only if it was enabled before. (Note that, depending on the action part of h'', the effect of h'' could have been modified.)

In cases 1 and 2 the enabling or disabling may depend on the current state of the model. Take for example the event h''' with guard $y = 0 \wedge x > 1$. Then h''' would be enabled after g if $x > 0$ holds in the state before executing g, and disabled otherwise. The predicate $x > 0$ is what we call an enabling predicate, and which we define as follows:

Definition 1 (Enabling predicate). *The predicate P is called enabling predicate for an event h after an event g, denoted by $g \rightsquigarrow_{P(v,t,s)} h$, if and only if the following holds*

$$I(v) \wedge G(v,t) \wedge S(v,t,v') \Rightarrow (P(v,t,s) \Leftrightarrow H(v',s))$$

where $I(v)$ is the invariant of the machine, $G(v,t)$ is the guard of g with parameters t and $S(v,t,v')$ the before-after predicate of its action part, and where $H(v,s)$ is the guard of h with parameters s.

In the absence of non-deterministic actions, an equivalent definition can be obtained using the weakest precondition notation:

$$I(v) \wedge G(v,t) \Rightarrow (P(v,t,s) \Leftrightarrow [S(t,v)]H(v,s))$$

where $[S]P$ denotes the weakest precondition which ensures that after executing the action S the predicate P holds.

Note that it is important for us that the action part $S(t,v)$ of an event does not contain any non-determinism (i.e., that all non-determinism has been lifted to the parameters t; see Section 2). Indeed, in the absence of non-determinism, the negation of an enabling predicate is a disabling predicate, i.e., it guarantees that the event h is disabled after g if it holds (together with the invariant) before executing g. However, if we have non-determinism the situation is different. There may even exist no solution for $P(v,t,s)$ in Def. 1, as the following example shows.

Example 1. Take $x :\in \{1,2\}$ as the action part of an event g with no parameters and the guard $true$ and $x = 1$ as the guard of h. Then $[S(t,v)]H(v) \equiv false$ as there is no way to guarantee that h is enabled after g. Indeed, there is no predicate over x that is equivalent to $x' = 1$ in the context Def. 1 : the before after predicate $S(v,t,v')$ is $x' \in \{1,2\}$ and does not link x and x'. Similarly, there is no way to guarantee that h is disabled after g. In particular, $\neg[S(t,v)]H(v) \equiv true$ is not a disabling predicate.

Note that if $I(v) \wedge G(v) \wedge [S(t,v)]H(v,s)$ is inconsistent, then any predicate $P(v,t,s)$ is an enabling predicate, i.e., in particular $P(v,t,s) \equiv false$.

How can we compute enabling predicates? Obviously, $[S(t,v)]H(v)$ always satisfies the definition of an enabling predicate. What we can do, is simplify it in the context of $I(v) \wedge G(v)$.[2] We will explain later in Sect. 4 how we compute enabling predicates and discuss the requirements for a simplifier.

Example 2. Take for instance a model of a for loop that iterates over an array and increments each value by one. Assuming the array is modeled as a function $f : 0..n \rightarrow \mathbb{N}$ and we have a global counter $i : 0..(n+1)$, we can model the for loop (at a certain refinement level) using two events *terminate* and *loop*.

$$\mathit{terminate} \mathrel{\widehat{=}} \ \mathsf{when}\ i > n\ \mathsf{then\ skip\ end}$$

$$\mathit{loop} \mathrel{\widehat{=}} \ \mathsf{when}\ i \leq n\ \mathsf{then}\ f(i) := f(i) + 1 || i := i + 1\ \mathsf{end}$$

We can now try to find enabling predicates for each possible combination of events. Table 1 shows the proof obligations from Def. 1 and simplified predicates P which satisfy it.

Event Pairs (first $\leadsto_P$ second)	Enable Predicate Definition (wp notation)	Simplified P
terminate $\leadsto_P$ terminate	$i > n \implies (P \iff i > n)$	true
loop $\leadsto_P$ loop	$i \leq n \implies (P \iff (i+1) \leq n)$	$(i+1) \leq n$
loop $\leadsto_P$ terminate	$i \leq n \implies (P \iff (i+1) > n)$	$(i+1) > n$
terminate $\leadsto_P$ loop	$i > n \implies (P \iff i \leq n)$	false

Table 1. Enable Predicates for a simple model

The directed graph on the left in Figure 1 is a graphical representation of Table 1. Every event is represented by a node and there for every enabling predicate $\mathit{first} \leadsto_P \mathit{second}$ from Table 1 there is an edge between the corresponding nodes.

The right picture shows the same graph if we take independence of events into account, i.e., if an event g cannot change the guard of another event h, we do not insert an edge between g and h. In particular, as *terminate* does not modify any variables, it cannot modify the truth value of any guard. On first sight it seems as if we may have also lost some information, namely that after the execution of *terminate* the event *loop* is certainly disabled. We will return to this issue later and show that for the purpose of reducing model checking and other application, this is actually not relevant.

In Event-B models of software components independence between events occurs very often, e.g., if an abstract program counter is used to activate a specific subset of the events at a certain point in the computation. We can formally define independence as follows.

[2] This is similar to equivalence preserving rewriting steps within sequent calculus proofs, where $I(v), G(v)$ are the hypotheses and $[S(t, v)]H(v)$ is the goal of the sequent.

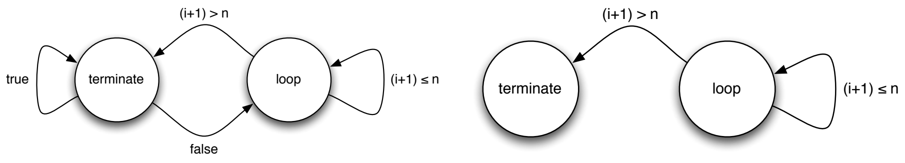

Fig. 1. Graph Representations of Dependence for a Simple Model

Definition 2 (Independence of events). *Let g and h be events. We say that h is independent from g — denoted by $g \not\rightsquigarrow h$ — if the guard of h is invariant under the substitution of g, i.e., iff the following holds:*

$$I(v) \wedge G(v,t) \wedge S(v,t,v') \implies (H(v,s) \iff H(v',s))$$

Our first observation is that an event g can only influence the enabledness of an event h (we do not require $g \neq h$) if g modifies some variables that are read in the guard of h. We denote the set of variables used in the guard of h by $read(h)$ and the set of variables modified by g by $write(g)$. If $write(g)$ and $read(h)$ are disjoint, then h is trivially independent from g:

Lemma 1. *For any two events h and g we have that $read(h) \cap write(g) = \varnothing \Rightarrow g \not\rightsquigarrow h$.*

This happens in our loop example, because $write(terminate) = \varnothing$, and hence all events (including *terminate* itself) are independent from *terminate*.

However, $read(h) \cap write(g) = \varnothing$ is sufficient for independence of events but not necessary. Take for instance the events from Figure 2. Event g clearly modifies variables that are read by h and therefore $read(h) \cap write(g) \neq \varnothing$ but g can not enable or disable h.

```
event g               event h
 begin                 when
   x := x + 1           x + y > 5
   y := y - 1          then
 end                   end
```

Fig. 2. Independent events

The trivial independence can be decided by simple static analysis, i.e., by checking if $read(h) \cap write(g) = \varnothing$. Non trivial independence is in general undecidable. In practice, it is a good idea to try to prove that two events are independent in the sense of Def. 2, as it will result in a graph representation with fewer edges. However, it is not crucial for our method that we detect all independent events.

As we have seen in the right side of Fig. 1, the information we gain about enabling and independence can be represented as a directed graph, now formally defined as follows.

Definition 3 (Enable Graph). *An Enable Graph for an Event-B model is a directed edge labeled graph $G = (V, E, L)$. The vertices V of the graph are the events of the model. Two events can be linked by an edge if they are not independent, i.e., $(g \mapsto h) \notin E \Rightarrow g \not\rightsquigarrow h$. Each existing edge $g \mapsto h$ is labeled with the enabling predicate, i.e., $g \rightsquigarrow_{L(g \mapsto h)} h$.*

Above we define a family of enable graphs, depending on how precise our information about independence is. Below, we often talk about the enable graph for a model, where we assume a fixed procedure for computing independence information.

Aside. There is another representation of the graph that is sometimes more convenient for human readers. We can represent the graph as a forest where each tree has one event as its root and only the successor nodes as leafs. The alternate representation is shown in Figure 3 for a small example.

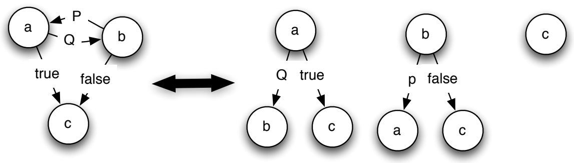

Fig. 3. Representations of the Enable Graph

4 Computing the Enabling Predicates

As mentioned before the weakest precondition $[S(t, v)]H(v, t, s)$ obviously satisfies the property of enabling predicates. We can use the syntax tree library of the Rodin tool to calculate the weakest precondition. However, these candidates for enabling predicates need to be simplified, otherwise they are as complicated as the original guard and we will gain no benefit from them. Therefore we simplify the candidate, in the context of the invariant $I(v)$ and the guard of the preceding event $G(v, t)$.

Consider the model shown in Figure 4. The weakest precondition for g preceding h is $[x := x + 2]x = 1$ which yields $x = -1$. This contradicts the invariant $x > 0$ and thus h can never be executed after g took place. In the context of the invariant, $x = -1$ is equivalent to false.

The simplification of the predicates is an important step in our method, deriving an enabling predicate $P(v, t, s)$ from the weakest precondition $[S(t, v)]H(v, s)$. Recall, that we simplify the predicate $[S(t, v)]H(v, s)$ in the context of the invariant $I(v)$ (and the guard $G(v, t)$)

$$I(v) \wedge G(v, t) \Rightarrow (P(v, t, s) \Leftrightarrow [S(t, v)]H(v, s))$$

```
invariant x > 0
event g               event h
 begin                 when x = 1
   x := x + 2         end
 end
```

Fig. 4. Simplification

A very important requirement in our setting is that the simplifier never increases the number of conjuncts. We have to keep the input for our enable and flow graph constructions small to prevent exponential blowup. Our simplifier shall find out if a conjunct is equivalent to true or false. In the first case the conjunct can be removed from the predicate in the second case the whole predicate is equivalent to false.

We have implemented a prototype simplifier in Prolog that uses a relatively simple approach. This prototype was used to carry out our case studies. The method does not rely on this implementation, we can replace it by more powerful simplification tools in the future.

5 Using the Enable Graph for Model Checking

The enable graph contains valuable information for a model checker. In this section we describe how it can be used within ProB. When checking the consistency of an Event-B model, ProB traverses the state space of the model starting from the initialization and checks the model's invariant for each state it encounters. The cost for checking a state is the sum of the cost of evaluating the invariant for the state and the calculation of the successors. Finding successor states requires to find solutions for the guards of each event. A solution means that the event is applicable and we can find some parameter values. ProB then applies the actions to the current state using the parameter values resulting in some successor states. In some cases the enable graph can be used to predict the outcome of the guard evaluation. The special case of an enabling predicate $P = false$ is very important. It means that no matter how we invoke g we can omit the evaluation of the guard of h because it will be false after observing g. In other words it is a proof that the property h *is disabled* holds in any state that is reachable using g.

When encountering a new state s via event e, we look up e in the enable graph. We can safely skip evaluation of the guards of all events f that have an edge (e,f) which is labeled with $false$ in the Enabled Graph. We can even go a step further if we have multiple ways to reach s. When considering an event to calculate successor states we can arbitrary choose one of the incoming events and use the information from the enable graph. For instance, if we have four events a, b, c and d and we know that a disables c and b disables d. Furthermore we encounter a state s via a but do not yet calculate the successors. Later we encounter s again, this time via b. When calculating the successors we can skip both, c and d.

The reason is that we have a proof for *c is disabled* because the state was reachable using event a and a proof that *d is disabled* because the state was reachable using event b. Thus the conjunction *c and d are disabled* is also true.

Because we use the invariant when simplifying the enabling predicate (see Section 4), the invariant must hold in the previous state in order to use the flow information. However we believe this is reasonable because most of the time we are hunting bugs and thus we stop at a state that violates the invariant. The implementation must take this into account and in case of an invariant violation it must not use the information gained by flow analysis. Also it needs to check not only the invariant but also the theorems if they are used in the simplifier.

6 Enable Graph Case Study

In this section, we will apply the concept to a model of the extended GCD algorithm taken from [7] using our prototype. The model consists of a refinement chain, where the last model consists of two loops. The first loop builds a stack of divisions. The second loop calculates the result from this stack. The last refinement level contains five events excluding the initialization. The events *up* and *dn* are the loop bodies, the events *upini*, *dnini* initialize the loops and *gcd* is the end of the computation. The event *init* is the *INITIALISATION* of the model.

event	read(event)	write(event)
init	$\varnothing$	$\{a, b, d, u, v, up, f, s, t, q, r, uk, vk, dn, dk\}$
upini	$\{up\}$	$\{up, f, s, t, q, r\}$
up	$\{up, r, f, dn\}$	$\{f, s, t, r, q\}$
gcd	$\{up, f, dn\}$	$\{d, u, v\}$
dnini	$\{up, dn, r, f\}$	$\{dn, dk, uk, vk\}$
dn	$\{dn, f\}$	$\{uk, vk, f\}$

Table 2. Read and write sets

The first step is to extract the read and write sets for each event; the result is shown in Table 2. Then we construct the enable graph. We calculate the weakest precondition for each pair of independent events and simplified them. Both steps were done manually but they were not very difficult. For instance, the most complicated weakest precondition was $[S_{up}]G_{dnini}$. In the presentation below we left out all parts of the guard and substitution that do not contain shared identifiers, e.g., the guard contains $up = TRUE$ but the substitution does not modify up. The next step is calculating the weakest precondition mechanically, finally we simplify the relational override using the rule $(r \mathbin{<\!\!+} a \mapsto b)(a) = b$.

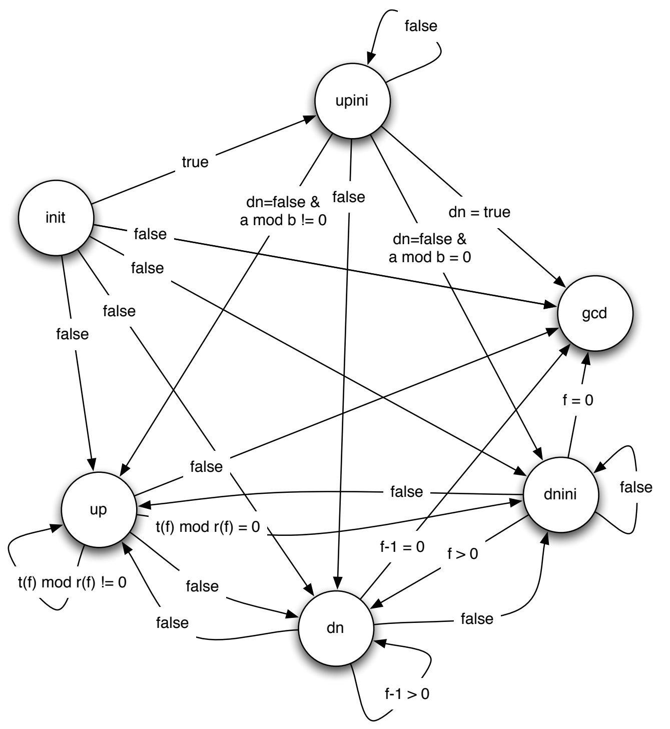

Fig. 5. Enable graph of the extended GCD example

$$
\begin{aligned}
[S_{up}]G_{dnini} &= [f := f+1, r \mathbin{\lhd\!\!-} \{f+1 \mapsto f(t) \bmod r(f)\}]\ (r(f) = 0) \\
&= (r \mathbin{\lhd\!\!-} \{f+1 \mapsto t(f) \bmod r(f)\})(f+1) = 0 \\
&= t(f) \bmod r(f) = 0
\end{aligned}
$$

The other simplification were much easier, for example, replacing $dn = \mathit{TRUE} \wedge dn = \mathit{FALSE}$ by $false$. The constructed graph is shown in Figure 5.

The enable graph can be used by the model checker to reduce the number of guard evaluations. Let us examine one particular run of the algorithm for fixed input numbers. The run will start with *init* and *upini* then contain a certain number of *up* events, say n. This will be followed by *dnini* and then exactly n *dn* events and will finish with one *gcd* event. In all, the calculation takes $2n + 4$ steps. After each step, the model checker needs to evaluate 5 event guards (one for each event, except for the guard of the initialization which does not need to be evaluated) yielding $10n+20$ guard evaluations in total. Using the information of the enable graph we only need a total of $4n+4$ guard evaluations. For example, after observing *up*, we only need to check the guards of *up* and *dnini*: they are the only outgoing edges of *up* in Fig. 5 which are not labelled by $false$.

7 Flow Construction

Beside the direct use in ProB, the enable graph can be used to construct a flow. A flow is an abstraction of the model's state space where an abstract state represents a set of concrete states. Each abstract state is characterized by a set of events, representing all those concrete states where those (and only those) events are enabled.

A flow describes the implicit algorithmic structure of an Event-B model. This information is valuable for a number of different applications, such as code generation, test-case generation, model comprehension and also model checking. In Section 8 we illustrate how we can gain and exploit knowledge about a model using the flow graph. We also briefly discuss how to generate code based on the flow.

The flow graph is a graph where the vertices are labeled with sets of events, i.e., the set of enabled events. The edges are labeled with an event and a predicate composed from the enable predicates for this event. The construction of the flow graph takes the enable graph as its input. Starting from the state where only the initialization event is enabled the algorithm unfolds the enable graph. We will describe the unfolding in a simple example, an algorithm is shown in Figure 7 and 8.

Figure 6 shows a simple flow graph construction. On the left side the enable graph for the events $init$, a and b are shown. The graph reveals that b always disables itself while it does not change the enabledness of a. The event a keeps itself enabled if and only if P holds and it enables b if and only if Q holds. The $init$ event enables a and disables b.

We start the unfolding in the state labeled with $\{init\}$. In this case we do not have a choice but to execute $init$. From the enable graph on the left hand side we know that after $init$ occurs a is the only enabled event. Therefore we have to execute a. We know that if P is true then a will be enabled afterwards and analogously if Q holds then b will be enabled. Combining all combination of P and Q and their negations, we get the new states $\{\}, \{b\}$ and $\{a, b\}$. If we continue, we finally get the graph shown on the right hand side. If more than one event is enabled, we add edges for each event separately. We can combine edges by disjunction of the predicates. In our case we did that for the transition from $\{a, b\}$ to $\{a\}$ which can be used by either executing b or a.

The algorithm in Figure 8 calculates for a given event e the successors in the flow graph by combining all possible configurations. The algorithm also uses a list of independent events that are enabled in the current state and therefore they are also enabled in any new state. The algorithm in Figure 7 produces the flow graph starting form the state $\{init\}$.

Generating the Flow Graph can be infeasible because the graph can blow up exponentially in the numbers of events. However, in cases where constructing the flow graph is feasible, we gain a lot of information about the algorithmic structure and we can generate code if the model is deterministic enough. We will discuss applicability and restrictions of the methods in section 9.

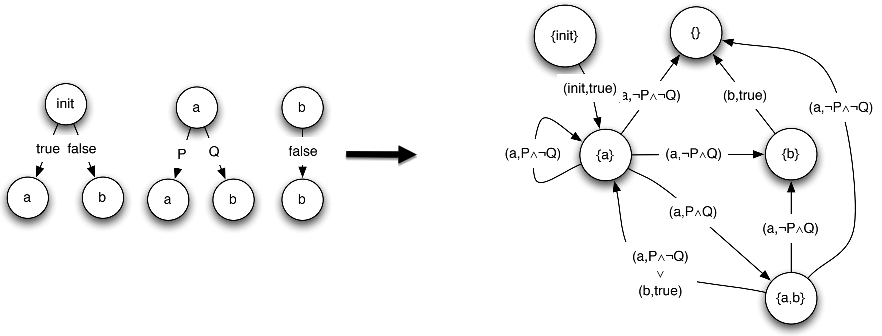

Fig. 6. Simple Flow Graph Construction

```
$todo := \{\{init\}\}$
$done := \varnothing$
$flow := \varnothing$
**while** $todo \neq emptyset$ **do**
  choose $node$ from $todo$
  **foreach** $e \in node$ **do**
    $keep := node \cap independent(e)$
    $atoms := expand(e, keep)$
    $todo := (todo \cup ran(atoms))$
    $flow := flow \cup \{node \mapsto atoms\}$
  **od**
  $done := done \cup \{node\}$
  $todo := todo - done$
**od**
```

Fig. 7. Algorithm for constructing a Flow Graph

8 Flow Graph Case Study

If we apply the flow construction to the example graph shown in figure 5 we get the flow graph shown in Figure 10. Compared to the structured model developed by Hallerstede in [7] shown in 9 we see a very similar shape.

However, the automatic flow analysis helped us to discover an interesting property. The flow graph contains a state that corresponds to concrete states where no event is enabled, i.e., states where the system deadlocks. Thus the model contains a potential deadlock. Inspection showed that the deadlock actually does not occur. The reason why the flow graph contains the deadlock state is a guard that is too strong. The guards of *dn* and *gcd* only cover $f \geq 0$. The invariant implicitly prevents the system from deadlocking by restricting the values of f.

In Figure 10 we can see that it is possible to automatically generate sequential code from a flow graph. The events *up* and *dn* can be translated into while loops

```
Given: enable graph as $EG : (Events \times Events) \nrightarrow Predicate$
def $expand(e, keep) =$
   $true\_pred := \{f \mapsto true | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = true\}$
   $maybe\_pred := \{f \mapsto p | (e \mapsto f) \in dom(EG) \wedge EG(e \mapsto f) = p \wedge p \neq false\}$
   $result := \varnothing$
     foreach $s \subseteq node$ do
       $targets := dom(true\_pred) \cup dom(s) \cup keep$
       $predicate := \bigwedge ran(s) \wedge \neg(\bigvee ran(s \lhd maybe\_pred)$
       $result := result \cup \{predicate \mapsto targets\}$
     od
  return $result$
end def
```

Fig. 8. Algorithm for expanding the Enable Graph (i.e., computing successor configurations)

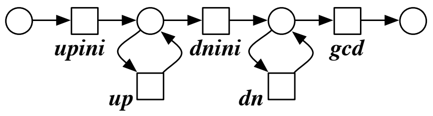

Fig. 9. Structural model from [7]

and *upini* and *dnini* are $if - then - else$ statements. In the particular case the termination of the computation was encoded into the *gcd* event.

9 Applicability and Restrictions

An important question is when to apply a method and maybe even more important when not to apply it. It is clear that flow analysis is probably not applicable if the model does not contain an algorithmic structure. In the worst case for flow construction, any combination of events can be enabled in some state, leading to $2^{card(Events)}$ states, where $card(Events)$ is the number of events. However in case of software developments it is very likely that eventually the model will contain events that are clustered, i.e, at each point during the computation a hopefully small set of events is enabled. We conjecture that the more concrete a model is, the better are results from simplification.

Constructing the enable graph is relatively efficient; it requires to calculate $O(card(Events)^2)$ enabling predicates. In case of software specifications generating the enable graph and using the information gained for guard reduction is probably worth trying. We can also influence the graph interactively. For instance, suppose the enable graph contains an edge labeled with $card(x) > 0$. Suppose we know that after the first event $x = \varnothing$ but the simplifier was too weak to figure it out, i.e., the empty set is written down in a difficult way, let's say $x := S \cap T$ where S and T are disjoint. By specifying (and proving) a theorem that helps the simplifier, e.g., $x = \varnothing$, we can interactively improve the graph.

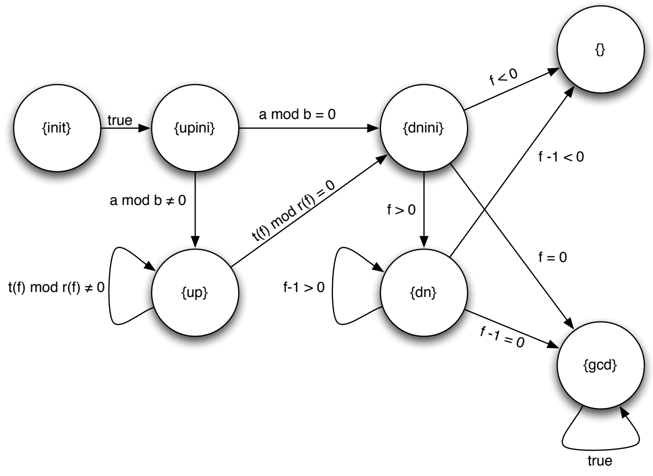

Fig. 10. Example for a relation between abstract and concrete states

We believe that expressing these theorems does not only improve the graph but also our understanding of a model because we explicitly formalize properties of the model that are not obvious (at least not for the automatic simplifier).

Constructing the flow graph is much more fragile; it can blow up very fast. It is crucial to inspect the enable graph and try to reduce the size of the predicates as much as possible. However our experience is that Event-B models of software at a sufficient low level of refinement typically have some notion of an abstract program counter that implicitly control the flow in a model. These abstract program counters are not very complicated and therefore it is likely that they are exploited by the simplifier.

10 Related and Future Work

Inferring Flow Information Model checking itself explores the state space of a model, and as such infers very fine-grained flow information. For Event-B, the ProB model checker [9, 10] can be used for that purpose. However, it is quite rare that the complete state space of a model can be explored. When it is possible, the state space can be very large and flow information difficult to extract. Still, the work in [11] provides various algorithms to visualize the state space in a condensed form. The signature merge algorithm from [11] merges all states with the same signature, and as such will produce a picture very similar to the flow graph. However, the arcs are not labelled by predicates and the construction requires prior traversal of the state space.

Specifying Flow Information There is quite a lot of related work, where flow information is provided explicitly by the modeler (rather than being deduced automatically, as in our paper). For example, several works use CSP to specify the sequencing of operations of B machines [14, 3, 5] or of Z specifications [6, 12, 13, 2].

In the context of Event-B, there are mainly three other approaches that are related to our flow analysis. Hallerstede introduced in [7] a new approach to support refinement in Event-B that contains information about the structure of a component. Also Butler showed in [4] how structural information can be kept during refinement of a component. Both approaches have the advantage to incorporate the information about structure into the method, resulting in better precision. However both methods require the developer to use the methods from the beginning while automatic flow analysis can be applied to existing projects. In particular automatic flow analysis can actually be used to discover properties of a model such as liveness and feasibility of events. Hallerstede's structural refinement approach does not fully replace our automatic flow analysis. Both methods overlap to some extent, but we think that they can be combined, such that the automatic flow analysis uses structural information to ease the generation of the flow graph. In return, our method can suggest candidates for the intermediate predicates used during structural refinement.

The third approach is yet unpublished but implemented as a plug-in for Rodin [8]. It allows the developer to express flow properties for a model and to verify them using proofs.

Future Work The next step is to fully integrate our method into the next release of ProB, and use it to improve the model checking procedure and help the user in analyzing or comprehending models. We also plan to use the technique to develop a new algorithm for test-case generation. In [15] we have introduced a first test-case generation algorithm for Event-B, tailored towards event coverage. One issue is that quite often it is very difficult to cover certain events. Here the flow analysis will hopefully help guide the model checker towards enabling those difficult events. We will also evaluate simplification tools that could be used within ProB to calculate good enabling predicates.

Conclusion In summary, we have developed techniques to infer algorithmic structure from a formal specification. From an Event-B model, we have derived the enable graph, which contains information about independence and dependence of events. This graph can be used for model comprehension and to improve model checking. We have described a more sophisticated flow analysis, which derives a flow graph from an Event-B model. It can again be used for model comprehension, model checking but also for code generation.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

2. D. A. Basin, E.-R. Olderog, and P. E. Sevinç. Specifying and analyzing security automata using csp-oz. In F. Bao and S. Miller, editors, *ASIACCS*, pages 70–81. ACM, 2007.
3. M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
4. M. Butler. Decomposition structures for event-b. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *Lecture Notes in Computer Science*. Springer, 2009.
5. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
6. C. Fischer. Combining object-z and csp. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *FBT*, volume 315 of *GMD-Studien*, pages 119–128. GMD-Forschungszentrum Informationstechnik GmbH, 1997.
7. S. Hallerstede. Structured Event-B Models and Proofs. In *ABZ 2010*, LNCS. Springer-Verlag, 2010.
8. A. Iliasov. Flows Plug-In for Rodin. `http://wiki.event-b.org/index.php/Flows#Flows_plugin`.
9. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
10. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
11. M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.
12. B. P. Mahony and J. S. Dong. Blending object-z and timed csp: An introduction to tcoz. In *ICSE*, pages 95–104, 1998.
13. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems-an integration of object-z and csp. *Formal Methods in System Design*, 18(3):249–284, 2001.
14. H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB'2000*, LNCS 1878, pages 188–208. Springer, 2000.
15. S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker. Applying Model Checking to Generate Model-based Integration Tests from Choreography Models. In *Proceedings TESTCOM/FATES 2009*, page to appear. Springer-Verlag, 2009.