Using Effect Systems for Automating Model Extraction

R. L. Crole (R.Crole@mcs.le.ac.uk) & A. Furniss (mjf29@le.ac.uk)

Department of Computer Science,
University of Leicester,
University Road,
Leicester,
LE1 7RH, U.K.

Abstract. We present a programming language called Do which has an effect system. Do is motivated by Wadler's extension of the computational lambda calculus. Do is specified by augmenting (the language of) the extension with a set of predefined effects and with provision for programmers to create their own application-specific effects. Do is accompanied by Dome (Do model extractor), a tool to extract models of Do programs for model-checking. Models are created with regard to a particular computational effect or set of effects. A *key difference* between our approach and previous approaches is that vertices in program graphs represent the *effect behaviour* of statements, rather than the statement itself.

1 Introduction

Model checking [10] is a verification technique that determines if a specification in a temporal logic such as linear temporal logic or computation tree logic holds for a model of a system, represented as a transition system consisting of states connected by transitions. Model checkers exhaustively search the possible states of the model in order to find a counter-example - a state path that demonstrates that the specification does not hold for the model.

As current model checkers can only represent a limited number of states and the number of states in a typical program is potentially very high, the construction of a considerably simplified abstract model is often required. When creating a model one must ensure that it is a correct representation of the program, in that the model displays every behaviour of interest contained within the program and does not introduce additional behaviour that is not present in the program. Models are often created by hand or with a user-defined translation table to directly translate a program into the model checker definition language, but these methods introduce the potential for human error. Techniques for automatically deriving models from programs are therefore desirable, and we pursue this goal here.

Our aim here is to present a method for creating models of the interactions of a computer program (and constituent threads) with the computer system state.

We are currently implementing this method in our Dome (Do Model Extractor) tool to allow the automatic generation of models of Do programs for the Spin model checker.

We proceed as follows: We review the relevant background literature in the next section. In Section 3 we explain some foundational properties of states, effects and dependencies. In Section 4 we specify an operational semantics for the Do language. In Section 5 we specify a graph extraction algorithm. In Section 7 we explain the model extraction process. Finally in Section 8 we draw our conclusions.

2 Background

2.1 Effect systems

Effect systems make explicit the effect of executing a statement or expression by incorporating details of the effects carried out by the statement into its type. As a consequence one is able to determine not only what the return or result type of a statement is, but also the actions that the statement will have upon the state of the system. Effects are annotated with a region to indicate which resource or part of a resource they act upon. The terminology stems from research into region-based memory management in which the heap is partitioned into sections called regions, and memory effects are annotated with the region they act upon. The term tag is also used (for example in [12]) to indicate the same type of annotation.

Lucassen and Gifford [11] describe an effect system with types, effects and regions, and effects of reading, writing or allocating memory locations. They give an operational semantics for their type and effect inference rules and introduce the idea of effect masking, in which effects that are local to an expression appear only in contexts in which they are observable. This is accomplished through a 'PRIVATE' expression that creates an anonymous region which is inaccessible after the expression or statement has finished executing.

Marino and Millstein [12] provide a framework for a generic effect system, and demonstrate how memory effects and transactional memory can be implemented in terms of their generic framework. They use sets of privileges defining which effects a given section of a program may carry out, and show how the privileges may be enforced during type-checking. Although the effect system here (and other effect systems in the literature) are oriented more towards the description of effects carried out by code rather than restriction using the notion of privileges, we can consider the set of privileges required by a given section of code equivalent to the set of effects that the code performs.

2.2 Computational Monads and Monadic Effect Systems

Moggi [13] introduces the notion of computation types. Given a type τ one also has a type $T\tau$ which designates computations which may deliver a result of

type τ. For example the type might be `int` and $T\texttt{int} = \texttt{int}_\perp$, the type of all programs that may be non-terminating but otherwise return an integer. In [13], Moggi considers non-deterministic computation, side-effects and continuations. Computation types allow many different types of computations and effects to be treated in a uniform manner, and forms the basis for the effect system described in [15]. Crole has investigated the use of computation types in giving semantics to IO effects [5], and there are connections between the IO semantics and the Do semantics seen in the current paper.

Wadler and Thiemann [15] modify an existing effect system to produce one based upon computation types by labelling each (standard) computation type with a set of effects so that a computation of type τ but with an *explicit* effect σ actually has type $T^\sigma\tau$. The semantics of the Do language is based upon the systems specified in this paper. Like [11], Wadler and Thiemann give a type and effect inference algorithm, in which type and effect reconstruction is performed by creating and solving systems of constraints.

2.3 Program slicing

A program slice consists of a subprogram that computes or performs computation using a specific variable or set of variables at a particular point in the program. This variable or set of variables and program location is referred to as the slicing criterion. Slices are either forward slices, which include all elements of the program that could be affected by the slicing criterion, or backward slices that include all elements of the program that could affect the computation of the slicing criterion. Techniques in the literature generally either perform static slicing to compute a slice for all possible execution traces of the program, or dynamic slicing to create a slice of all parts of the program that affect a specific execution trace. *The slicing criterion in this paper differs from those discussed elsewhere as it consists of a set of effects to include in the generated model.* To extract a model the original program is reduced to one that solely includes all possible paths that include effects contained within the slicing criterion – essentially a series of static backwards slices.

The most common approaches to static backwards slicing in the literature have been graph-based. Hatcliff et al [8] consider program slicing in the context of model production for their Bandera tool using a control-flow graph. They show how to extract slicing criteria from a LTL formula, and prove that their slicing algorithm preserves the properties of the model that satisfy the criteria, allowing an optimised model to be constructed for each formula.

Horwitz, Reps and Binkley [9] discuss program slicing using dependence graphs. They start with a description of slicing programs with a single procedure or function using a program dependence graph. Such graphs have an entry vertex, variable declaration and variable final use vertices for each variable in the procedure, along with data and control dependence edges. They then extend the program representation to a system dependence graph to allow slicing of programs with multiple procedures or functions by introducing function call

site, formal in, formal out, actual in and actual out vertices, along with function call edges, parameter in and out edges. Their slicing algorithm operates in two stages. In the first stage the slice is computed by starting from the vertex representing the initial node of the slice, marking nodes that are either in the same function or are in functions that have called the function containing the initial node. In the second stage, the slicing algorithm traverses functions that are called by the function containing the starting point of the slice. The final slice is the union of the sets of vertices marked during the two stages.

Zhao [16] describes multi-threaded dependence graphs, an extension of the program dependence graphs used in methods of slicing sequential programs (most notably from [9]). Multi-threaded dependence graphs provide a thread dependence graph for each individual thread with edges to represent communication dependencies or synchronisation dependencies between threads. Communication dependencies occur when the result of a statement or expression in one thread is influenced by the result of a statement or expression in another thread. Synchronisation dependencies occur when the concurrency features of Java such as `notify`(), `wait`() and `join`() are used to synchronise threads.

2.4 A Note on Implementation Details

Appel and Palsberg [1] provide an overview of compiler design and related issues including type checking and symbol tables, which are an important aspect of both the model extractor and interpreter. They distinguish between imperative and functional symbol tables. Imperative symbol tables perform destructive updates on a single instance of the symbol table and have an 'undo' stack that restores the previous state upon leaving the scope. Functional symbol tables create a new instance of the symbol table when adding an entry, allowing the previous version to be restored upon leaving the scope. Strategies for efficiently implementing the different types of symbol table are also discussed. Muchnick [14] provides a further in-depth discussion of symbol tables, including their implementation using hash-tables for languages with different scoping rules.

The lexer and parser have been generated using Flex and Bison. During this process, the Flex documentation [7], Bison documentation [3] and C++ Flex/Bison example by Timo Bingmann [2] were useful.

3 States, Effects and Dependencies

Our aim is to present a method for creating models of the interactions of a computer program (and constituent threads) with the computer system state. We consider a system with concurrently and autonomously executing program threads, which carry out *computations* that may or may not interact with the state of the system. Here regions are labels that are used to identify system state such as memory locations or other system resources. An effect describes how a computation changes the state of the system or depends upon the state of the system. Each effect acts upon one or more regions. We further distinguish

between *atomic* effects, which are 'indivisible' primitive effects on the system, and *composite* effects which are composed of two or more primitive effects.

Effects describe transformations in the state of the system. We write $\sigma(s)$ to denote the result of updating the state of the system s with effect σ, and $\sigma_1 ; \sigma_2$ to denote composition of effects. For all effects σ_1 and σ_2, $\sigma_1 ; \sigma_2(s) = \sigma_2(\sigma_1(s))$. We define traces of effects in a conventional manner. A *trace* is a sequence of effects $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_{n-1} \rightarrow \sigma_n$ where n is 0 in the case of the *empty trace*. The *concatenation* of traces is defined as expected.

One of the key aspects of our approach concerns the dependence or independence of effects. An effect σ_1 is dependent upon another effect σ_2 if the result of carrying out σ_1 can alter depending upon how or if σ_2 has previously been carried out. For example, consider a system with memory locations, a *get* effect that describes accessing a memory location and a *set* effect that describes updating a location. The result of accessing a memory location is dependent upon the value that has previously been stored there, so the *get* effect is dependent upon the *set* effect. However, the converse does not hold, as the result of updating a memory location will be the same irrespective of how the location has previously been been accessed. We write the relationship of dependency as $\sigma_1 \dashv_{dep} \sigma_2$.

Once we understand the relationships between effects, we can use them to reason about the result on system state of performing computation; in particular we include effects and effect dependencies as "first class" citizens in the Do Language. To do this we first need to introduce effect and dependency sets. The effect set of a trace is the set of effects that the trace performs (in a similar manner to the concept of a sort in CCS). Note that we can consider a computation to be a trace of length 1, in which case the effect set is simply the effect of the computation. The *effect set* of a trace $t = \sigma_1 \rightarrow ... \rightarrow \sigma_n$ is $\mathit{Eff}(t) = \{\sigma_i \mid \sigma_i \in t\}$. Given the effect set we can now define the dependency set, which is essentially the set of effects which may influence the outcome of the trace. The dependency set is constructed from the set of all effects upon which any effect in the effect set of the trace is dependent upon. The *dependency set* of a trace t is $\mathit{Dep}(t) = \{\sigma' \mid \sigma \in \mathit{Eff}(t), \sigma \dashv^{+}_{dep} \sigma'\}$ where $^{+}$ denotes transitive closure.

To return to the previous example of memory locations, if we wanted to create a model of memory accesses it would not be enough to simply include all of the get effects in the model, as we cannot accurately represent the result of memory access without also including the effects that set the contents of the location. When we decide to include a type of effect in the model, we must also include all of the effects that it is dependent upon. Due to the nondeterministic nature of concurrent execution we cannot know what order the effects of the various threads will be performed, and it is difficult or impossible to know which sections of program code can be executed concurrently with each other. In the interest of simplicity, we do not attempt to determine which sections of the program may be run concurrently, and instead consider potential interactions between all instances across the entire program of the effects in question.

We include the notion of effect environments in the semantics of Do ranged over by β, a function from the set of identifiers to the set of effects. β_{id} denotes the effect environment β with the identifier id removed from its domain.

The notion of effect dependencies is important because it enables the system to determine which effects need to be included in extracted models when slicing. We determine the dependencies of an effect using an effect graph, with vertices representing effects and edges representing dependencies. We define the dependency set of a statement or expression to be the union of the dependency sets of each effect contained within the type of the expression or statement. The dependency set is determined using a breadth-first search to calculate the transitive closure of effect dependencies, following dependency edges and marking nodes to avoid infinite loops. Once the set cannot be expanded further, the effects represented by the marked nodes form the dependency set.

4 The Do Language

We have so far discussed effects as a mathematical abstraction, but we now present concrete effects in the context of the Do programming language. Do is a simple functional programming language with a conventional syntax and semantics, but with the addition of an *effect system*. Types τ range over integers, booleans, a unit type, and functions. Given a countable set Id of identifiers, the (raw/untyped) Do expressions and statements are specified by

$$
\begin{aligned}
V &::= \underline{b} \mid \underline{i} \mid () \mid id\ values \\
values &::= \epsilon \mid V\ values \\
E &::= \underline{b} \mid \underline{i} \mid id \mid uop\ E \mid E\ bop\ E \mid E\ E \\
uop &::= \sim \mid \neg \text{ and } bop ::= \ + \mid - \mid / \mid * \mid \ \&\& \ \mid \ || \\
S &::= \texttt{skip} \mid \texttt{new}\ \tau\ id \mid \texttt{get}\ id \mid \texttt{set}\ id\ := E \mid \texttt{let}\ id\ :=\ E\ \texttt{in}\ S \mid S\ \texttt{;}\ S \\
&\quad \mid \texttt{if}\ E\ \texttt{then}\ S \mid \texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{fun}\ id\ params\ = S \mid \texttt{return}\ E \\
params &::= \epsilon \mid id : \tau\ params
\end{aligned}
$$

Do has a 'small-step' operational semantics. This is an adaptation and extension of the Effect language presented in [15] (and also motivated by the Monad language loc. cit.). Do is call-by-value to simplify the order and sequence of evaluation (and hence simplify the extracted models of programs).[1] Other common constructs such as conditional statements are also introduced, along with the ability to declare new user effects and dependencies between them. To specify

[1] A simple form of concurrency has been introduced into the language with the addition of a `split` effect, which will cause a function to be used as the entry point for a new thread running concurrently with the existing one—the details are omitted from this paper.

the semantics we define a (countable) set of *locations* *Loc* to be a subset of the (countable) set of *Id* of identifiers. A *store* is a function from locations to the set *Val* of values. A semantic effect [15] f takes the form $new_{eff}(l)$, $get_{eff}(l)$ or $set_{eff}(l)$ where l is a location.

The operational semantics is specified by using evaluation contexts $\mathcal{C}$ (see for example [6]).

$$\begin{aligned}\mathcal{C} ::= {} & [\,] \mid uop\ \mathcal{C} \mid \mathcal{C}\ bop\ E \mid V\ bop\ \mathcal{C} \mid \mathcal{C}\ E \mid V\ \mathcal{C} \\ & \mid \texttt{if}\ \mathcal{C}\ \texttt{then}\ S \mid \texttt{if}\ \mathcal{C}\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\ & \mid \texttt{set}\ id\ :=\ \mathcal{C} \mid \texttt{let}\ id\ :=\ \mathcal{C}\ \texttt{in}\ S \\ & \mid \mathcal{C}\ ;\ E \mid V\ ;\ \mathcal{C} \mid \texttt{return}\ \mathcal{C}\end{aligned}$$

The operational semantics [4] transitions take the form $(s, \beta, \dashv_{dep}),\ X \xrightarrow{f} (s', \beta', \dashv_{dep}'),\ X'$ where X is an expression or a statement, generalising the transitions of [15]. In the case that no changes occur to s, β, or $\dashv_{dep}$ then we write simply $X \longrightarrow X'$. Much of the semantics is standard, but some of the non-standard transitions are given in Figure 1.

$$\frac{}{(s, \beta, \dashv_{dep}),\ \texttt{new}\ \tau\ l \rightarrow \xrightarrow{new_{eff}(l)} (s \cup \{l \rightarrow 0\}, \beta, \dashv_{dep}),\ ()}\ new\ [l \notin dom(s)]$$

$$\frac{}{(s_l \cup \{l \rightarrow V\},\ \beta, \dashv_{dep}),\ \texttt{get}\ l \xrightarrow{get_{eff}(l)} (s_l \cup \{l \rightarrow e\}, \beta, \dashv_{dep}),\ V}\ get$$

$$\frac{}{(s_l \cup \{l \rightarrow V\}, \beta, \dashv_{dep}),\ \texttt{set}\ l\ :=\ V' \xrightarrow{set_{eff}(l)} (s_l \cup \{l \rightarrow V'\}, \beta, \dashv_{dep}),\ ()}\ set$$

$$\frac{}{\texttt{let}\ id = V\ in\ S \rightarrow S[x := V]}\ let_2$$

$$\frac{}{id\ V_1\ \ldots\ V_a \rightarrow\ S[params := \boldsymbol{V}]}\ funcall\ [\texttt{fun}\ id\ params\ := S]$$

$$\frac{}{\texttt{return}\ V \rightarrow\ V}\ return$$

The rule for evaluation contexts is

$$\frac{X \xrightarrow{f} X'}{(s, \beta, \dashv_{dep}),\ \mathcal{C}[e] \xrightarrow{f} (s, \beta, \dashv_{dep}),\ \mathcal{C}[e']}\ evcxt$$

Fig. 1. Operational Semantics for Do

---

5 A Graph Extraction Algorithm and Implementation

The remainder of this paper focuses on an example program P given in Figure 2.

```
new int a;
new int b;
new int c;

fun fib2 :  int x :  int y :  int i :  int = {
(1)       if i = 0 then (2) return y;
(3)       if i = 1 then (4) return x + y;
(5)       set a := x + y;
(7)       set b := (6) get a + y;
(8)       set c := i - 2;
(14)       return (12) (13) fib2 (9) (get a) (10) (get b) (11) (get c)
};

fun fib :  int n :  int = {
(15)       let z := n - 1 in
(16)       return (17) (18) fib2 0 1 z
};

fun main :  unit d :  unit = {
(19)       return (20) (21) fib 42
};
```

Fig. 2. Progam P

One of the key aims during model creation is to limit the number of states and transitions in order to ensure the tractability of queries upon the model. To achieve this we create a control-flow graph representation of effects carried out by the program and then select a slice (or subgraph) that includes only behaviour matching a specified set of effects.

Our graph representation is based upon the system dependence graphs introduced in [9], although it has some important differences. While vertices in the system dependence graphs created by Horwitz et al represent statements themselves, we create graphs in which vertices represent the *behaviour* of the statement or expression in the form of its effect. Such graphs are therefore fundamentally a representation of the interaction of the program with system state rather than statements of the the program itself.

We introduce an additional type of edge, corresponding to the sequential order in which the effects represented by the vertices are performed. Unlike the system dependence graphs outlined in [9] we do not consider data-flow. As a result, the creation and slicing of the control-flow graphs described here is somewhat simpler than system dependence graphs, as we do not create vari-

able definition and last-use vertices, or formal and actual parameter in and out vertices.

However, the availability of data-flow information would enable a reduction in the number of transitions and states in the model. In the method outlined here we do not maintain a record of the state of regions, such as the contents of memory locations. When considering a statement such as `set` $a := b + 32$ we record only that an instance of the set effect has occurred; the value actually stored in the memory location represented by region a is not included in the model.

On one hand this reduces the amount of information stored for each state, and therefore the amount of memory required, and as a result improving the chance of creating a tractable model. However, as a consequence of this lack of state we cannot evaluate the condition of conditional statements during model-checking, so deterministic execution in the original program may become non-deterministic in the model representing the program. Given a statement such as `if get` $a := 55$ `then set` $b := 12$ `else set` $c := 4$ we cannot determine the contents of region a, and hence have to consider that the statements contained in either the true or false branches could be executed. It is likely that we will extend our dependence graph representation at some point in the future to include data-flow in a similar way to that described in [9].

We create a dependence graph for the program using the algorithm in Figure 3 for each statement s in the function. We use some notation to specify the algorithm: Let C be a stack to contain references to vertices with conventional $push_C(v)$, pop_C and top_C operations. Let t, t_{true}, t_{false} and t_{expr} be variables, each holding a reference to a vertex. We denote assignment to a variable with the := operator. t is always the last vertex to be created, t_{true}, t_{false} and t_{expr} are the last vertices of the true and false branches of a conditional statement, with t_expr the last vertex of the boolean expression that controls it. *CreateGraph*(F) means that we create a function entry vertex v_{entry} and function exit vertex v_{exit}, and perform $push_C(v_{entry})$. CD, TR, and V stand for control dependence, transition relation, and vertex.

The graph for program P is in Figure 4. Some simplifications have been made in the graph shown in this paper (for reasons of clarity). Arguments in a Do function call are applied one at a time, returning a function value after each application to which the next argument is applied until all of the arguments have been provided. A function call with n arguments in the program will therefore produce n pairs of function call nodes in the dependency graph; for example, each of the function calls in P will be represented by 3 pairs of vertices, each with function call and return edges. Since the complexity is the result of the language rather than the model extraction technique itself, we represent function calls in the graph with a single pair of nodes.

Vertices in the graph are classified into several vertex types, with each vertex having exactly one type. The different types consist of statement, expression, function entry, function exit, function call and function return vertices. The majority of statements in the source program are translated to vertices of the

`set` *id* := *E*	`get` *id*
Create a V v labelled {set id} Create a V v' labelled the effect of E. Add a CD edge from v to top_C Add a CD edge from v' to v Add a TR edge from t to v' Add a TR edge from v' to v $t := v$	Create a V v labelled {get id} Add a TR edge from t to v Add a CD edge from v to top_C $t := v$

`let` *id* := *E*	`return` *E*
Create a V v labelled the effect of E. Add a CD edge from v to top_C Add a TR edge from t to v $t := v$	Create a V v labelled the effect of E. Add a CD edge from v_{exit} to v. Add a CD edge from v to top_C Add a TR edge from t to v Add a TR edge from v to v_{exit} pop_t

`if` *E* `then` S_{true}	`if` *E* `then` S_{true} `else` S_{false}
Create a V v labelled the empty effect Add a CD edge from v to top_C Add a TR edge from t to v $push_C(v)$ $CreateGraph(E)$ $t_{expr} := t$ $CreateGraph(s_{true})$ $t_{true} := t$ If execution of S_{true} can reach the following statement Create a V v_{end} Create a CD edge from v_{end} to v Create TR edges from t_{true} to v_{end} and t_{expr} to v_{end} $t := v_{end}$ Otherwise $t := t_{expr}$ pop_C	Create a V v labelled the empty effect Add a CD edge from v to top_C Add a TR edge from t to v $push_C(v)$ $CreateGraph(E)$ $t_{expr} := t$ $CreateGraph(s_{true})$ $t_{true} := t$ $CreateGraph(s_{false})$ $t_{false} := t$ If execution of both s_{true} and s_{false} can reach the following statement Create a V v_{end} Create a CD edge from v_{end} to v Create TR edges from t_{true} to v_{end} and t_{false} to v_{end} $t := v_{end}$ Otherwise, if execution of only s_{true} can reach the following statement $t := s_{true}$ Otherwise $t := s_{false}$ $pop_C(v)$

id params
Omitted for space reasons

E_1 *bop* E_2	*uop* *E*	S_1 ; S_2
$CreateGraph(E_1)$ $CreateGraph(E_2)$	$CreateGraph(E)$	$CreateGraph(S_1)$ $CreateGraph(S_2)$

Fig. 3. Graph Extraction Algorithm

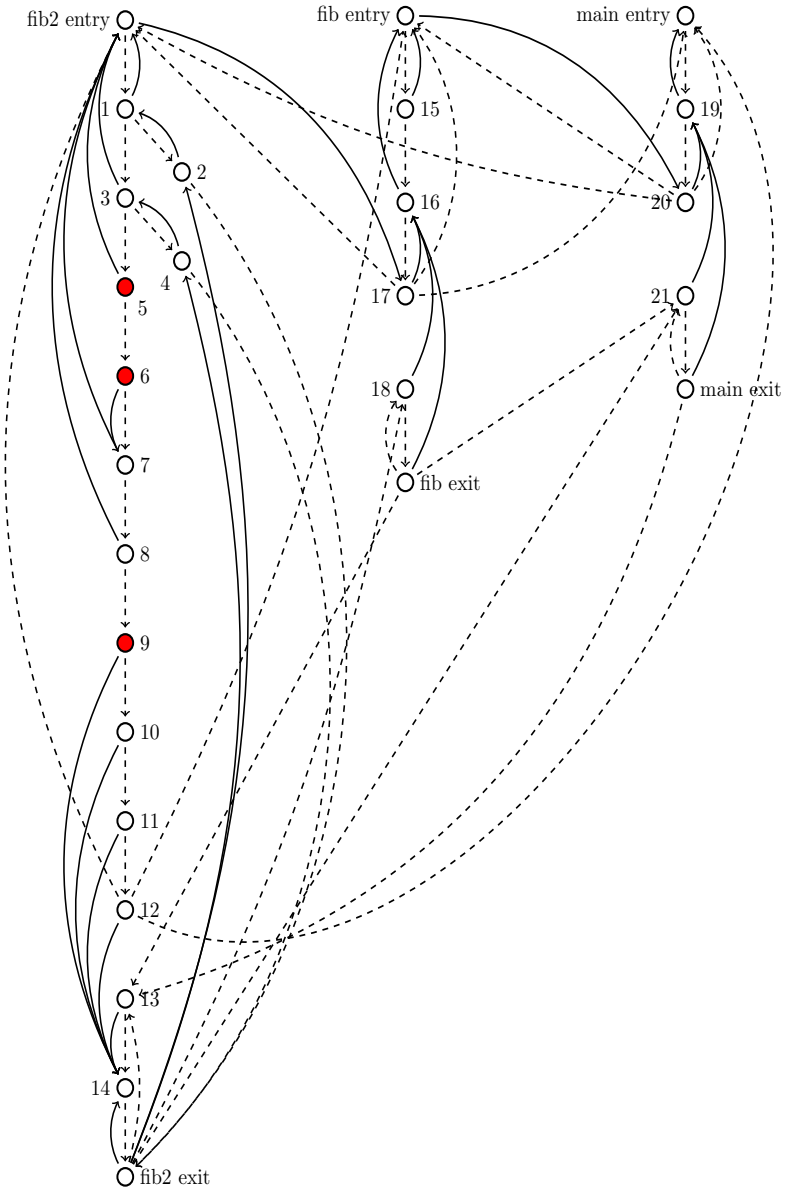

Fig. 4. Graph for Program P With Vertices Selected On First Stage of Slicing with Criterion $get < a >$

statement vertex type, which are labelled with the effect performed by the statement. The function entry and function exit vertices do not directly correspond to statements in the program: they are created as entry and exit points for the subset of the control flow graph corresponding to a particular function definition. Likewise, function call and function return vertices are created at function call sites to indicate the transfer of execution to and from another function.

Edges in the control-flow graph are classified in a similar manner to vertices, with 4 different edge types: control dependency edges, transition edges, function call and function call return edges.

We do not create nodes for region declaration, dependency declaration or thread creation statements, as regions, dependencies and threads exist throughout the entire life of the program and such statements cannot appear inside functions.

To enable the creation of control-dependence edges during the creation of the graph the last function entry or conditional statement vertex is tracked using a stack, and control dependence edges are generated during the creation of each new vertex by adding an edge to the vertex at the top of the stack. The transition relation is generated by keeping track of the last generated vertex to enable a transition edge to be added when a new vertex is created.

6 Slicing

Once the graph representation of program behaviour has been created we select a subset of the vertices in the graph called a *slice*. The slice is created with regard to a *slicing criterion* to obtain a subgraph containing only vertices with effects specified in the slicing criterion and vertices representing control flow statements.

Our slicing criterion and method differ from that described in [16] and [9] in some important respects. In existing approaches to program slicing, the slicing criterion consists of a variable x at a particular point in the program and produces a subprogram consisting of all statements that are involved in the computation of the value of x or can be affected by the x (depending upon whether forward or backward slicing is being performed). In contrast, we use a set of effects $\Sigma = \{\sigma_1, .. \sigma_n\}$ as the slicing criterion. Given a dependence graph g, we produce a subgraph g' containing only vertices labelled with instances of effects in $\Sigma \cup Dep(\sigma_1) \cup Dep(\sigma_2) \cup ... \cup Dep(\sigma_n)$ from the entire program. The sliced graph for program P is in Figure 5.

In doing so, we produce a slice that includes all instances of the specified behaviour from the entire program, and the effects of other statements (such as conditional statements and function calls) required for these effects and execution paths to be performed. The dependency relationships between effects are important here as they enable us to create a model that includes not only the behaviours specified in the slicing criterion but also behaviours that may influence the outcome of those in the slicing criterion and therefore must also be included. The dependency relationships bet-weens effects are important here, as

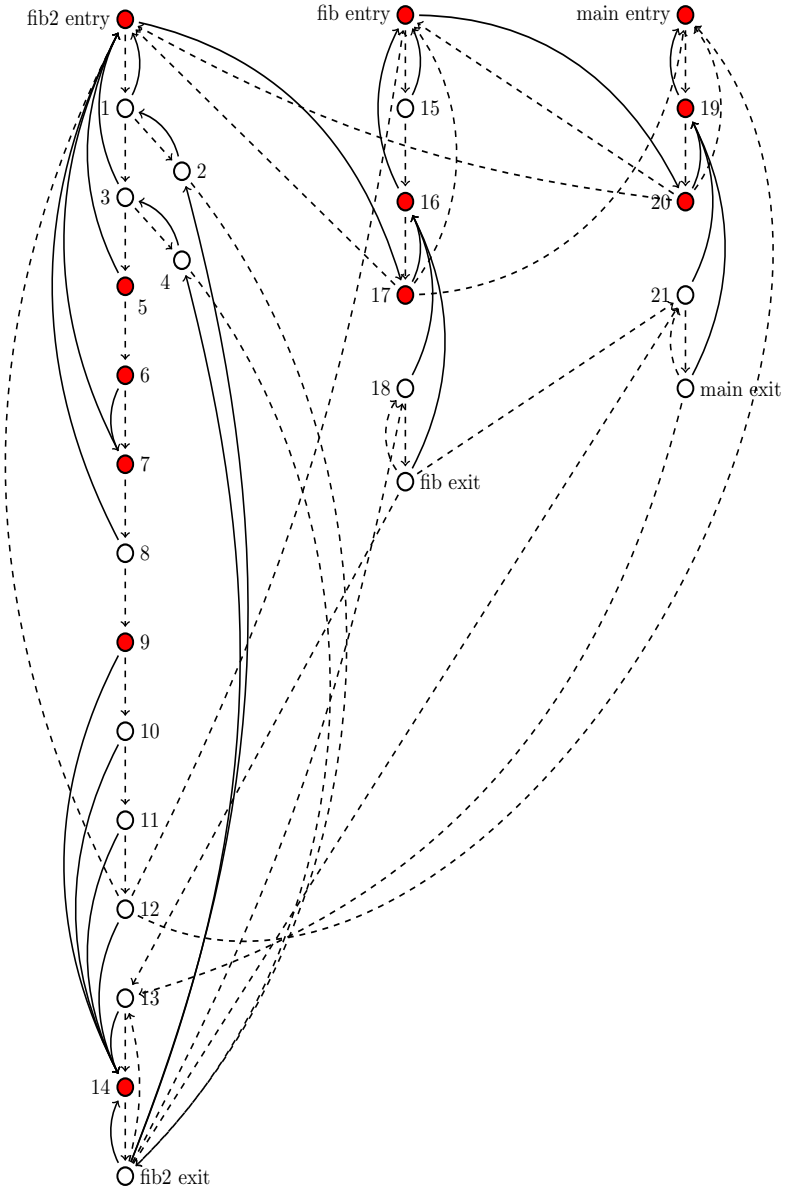

Fig. 5. Graph for P Sliced With Criterion *get* < *a* >

they enable us to create a model that includes not only the behaviours specified in the slicing criterion but also behaviours that may influence the outcome of those in the slicing criterion and therefore must also be included.

We utilise a breadth-first node-marking approach, in which vertices with effects that unify with some element in our slicing criterion or its dependencies are first marked and added to a queue. The nodes marked in this step are shown in Figure 4. The vertex v at the head of the queue is then removed, and all unmarked vertices immediately reachable from v via control dependency edges are also marked and added to the back of the queue. The process is repeated until the queue is empty, at which point the slice consists of the marked subgraph.

7 Model Extraction through a LTS

Once a subgraph of the graph from Section 5 has been created, we produce a labelled transition system in the form of a stack automaton from which the final PROMELA model is created. Each vertex v with label σ in the control flow graph becomes a transition t labelled with σ between states in the automaton.

The stack in the automaton provides the ability to return to the appropriate state after reaching the end of the states representing a function. Function call and function exit vertices in the graph become transitions states at which the stack must be manipulated or used to determine the next state in the model.

We denote transitions in the automaton with a quintuple $(s_1,\ s_2,\ \sigma,\ w,\ r)$ where s_1 and s_2 are the start and end states of the transition, σ is an effect, r is the identifier of a state to be read from the top of the stack and w the identifier of a state to push onto the stack (with λ indicating that nothing is to be read from the stack or popped onto the stack). We assume that values read from the stack are removed.

We indicate a transition edge between vertices v and v' in the graph as $(v, v')_T$, a control dependence edge between v and v' as $(v, v')_D$, a function call edge between v and v' as $(v, v')_F$ and a function call return edge between v and v' as $(v, v')_R$.

We create the automaton as follows: Let L be a stack to contain references to states with conventional $push_L(s)$, pop_L and top_L operations. Let E be a stack to contain references to states with conventional $push_E(s)$, pop_E and top_E operations. Let V be a list to contain visited vertices. Then proceed by applying these steps

- $M := \emptyset$
- Create a start state S_{start} and end state S_{end}
- Create start and end states for the main function $S_{mainstart}$ and $S_{mainend}$
- Add transitions $(S_{start}, S_{mainstart}, empty, S_{end}, \lambda)$
  and $(S_{mainend}, S_{end}, empty, \lambda, S_{end})$
- $push_L(S_{start})$
- If there is no v in the input graph such that $(v_{start}, v)_T$
  add a new transition $(S_{start}, S_{end}, empty, \lambda, \lambda)$ to the automaton and finish.

– Otherwise $process(S_{mainstart})$
– pop_L

together with $process(v)$ appearing in Figure 6. The automaton for P is in Figure 7.

It is relatively simple to create a representation of the program from the automaton in the PROMELA input language of the Spin model checker. We use a two-stage process. In the initial stage, the input program is read to build lists of the regions, processes and effects from the top-level declarations. A standard PROMELA program outline is used for elements common to all models of Do programs such as the declaration of the process stack and stack pointer, current state and last action variables. In the second phase, the automaton created from the sliced program is used to generate the states of the model. Each state in the automaton is numbered, and the behaviour of the model is produced by a PROMELA do statement indexed by the current state number. Transitions between states in the automaton become assignments to the `currentstate` variable in the model, and the `lasteffect` variable is used to record the effect label of the last transition made by the model checker to bring it into the current state. For reasons of space we omit the model for P from this paper.

8 Conclusions and Future Work

In this paper we have presented a method of creating simple models of program behaviour from programs in a language with an effect system.

It would be possible to extend our models to include system state such as the contents of memory regions rather than simply produce traces of effects that occur. Such an approach would allow execution in the source program that at present becomes non-deterministic in the generated model to be represented in a deterministic manner, and therefore remove traces in the model that do not correspond to possible execution paths in the program. Our graph representation and slicing algorithm would have to be extended to consider data-flow as well as control-flow in the original program, and we would need to consider additional types of effect dependencies. For instance, when an expression appearing in the condition of a conditional statement has an effect, we would need to include that effect and its dependencies in the model to properly evaluate the conditional statement. From the preliminary work undertaken here, we believe that such extensions will be a considerable undertaking.

The automata and models produced by the method detailed in this paper are certainly not minimal, so it would be possible to modify our algorithms to produce fewer states and empty transitions or add an additional minimisation stage before generating the final PROMELA model.

The Do language has a limited syntax and set of primitive effects. We intend to extend the language with further control statements and additional effects such as thread synchronisation and mutual exclusion, which would make the system significantly more useful for verifying concurrent programs.

$process(v) =$
Let $S_{funstart}$ and S_{funend} be the start and end states of the function containing v.
For each vertex v:

Function entry vertex:
- If v is marked and $v \notin V$
  - $push_L(S_{funstart})$
  - $V := V \cup \{v\}$
  - For every v' such that there exists a transition edge $(v, v')_T$ in the input graph $process(v')$
  - pop_L

Function exit vertex:
- Add a new transition $(top_L, S_{funend}, empty, \lambda, \lambda)$.
- $M := M \cup \{(v, top_L, S_{funend})\}$
- $V := V \cup \{v\}$

Function call vertex:
- If v is marked
  - Add a function call return state S_{return}
  - $V := V \cup \{v\}$
  - Add a transition $(top_L, s_{return}, empty, \lambda, \lambda)$
  - $push_L(s_{return})$
  - For every v' such that there exists a transition edge $(v, v')_T$ in the input graph process(v')
- For every marked vertex v'' such that there exists a function call edge $(v,\ v'')_F$ in the graph:
  - Create a transition $(top_L, v''_{funstart}, empty, \lambda, S_{return})$ where $v''_{funstart}$ is the start state of the function containing v''
  - $process(v'')$
  - Create a transition $(v''_{funend}, S_{return}, empty, S_{return}, \lambda)$ where v''_{funend} is the end state of the function containing v''
- pop_L

Function call return vertex:
- For every v' such that (v, v') is a transition edge in the graph:
  - If $(v', s, s') \notin M$ or $v' \neq top_L$ then $process(v')$
  - Otherwise add a transition $(top_L, s, empty, \lambda, \lambda)$ where $(v', s, s') \in M$

Other vertex types:
- If v is marked, $v \notin V$ and v has a non-empty effect:
  - Add a new state s
  - Add a new transition $(top_L,\ s,\ Eff(v),\ \lambda,\ \lambda)$
  - $V := V \cup \{v\}$
  - $M := M \cup (v, top_L, s)$
  - $push_L(s)$
  - $pushed := true$
- If $(v, s', s'') \notin M$ or $s'' \neq top_L$ then $process(v')$
- Otherwise add a transition $(top_L, s', empty, \lambda, \lambda)$
- If $pushed =$ true then pop_L

Fig. 6. Automaton Algorithm

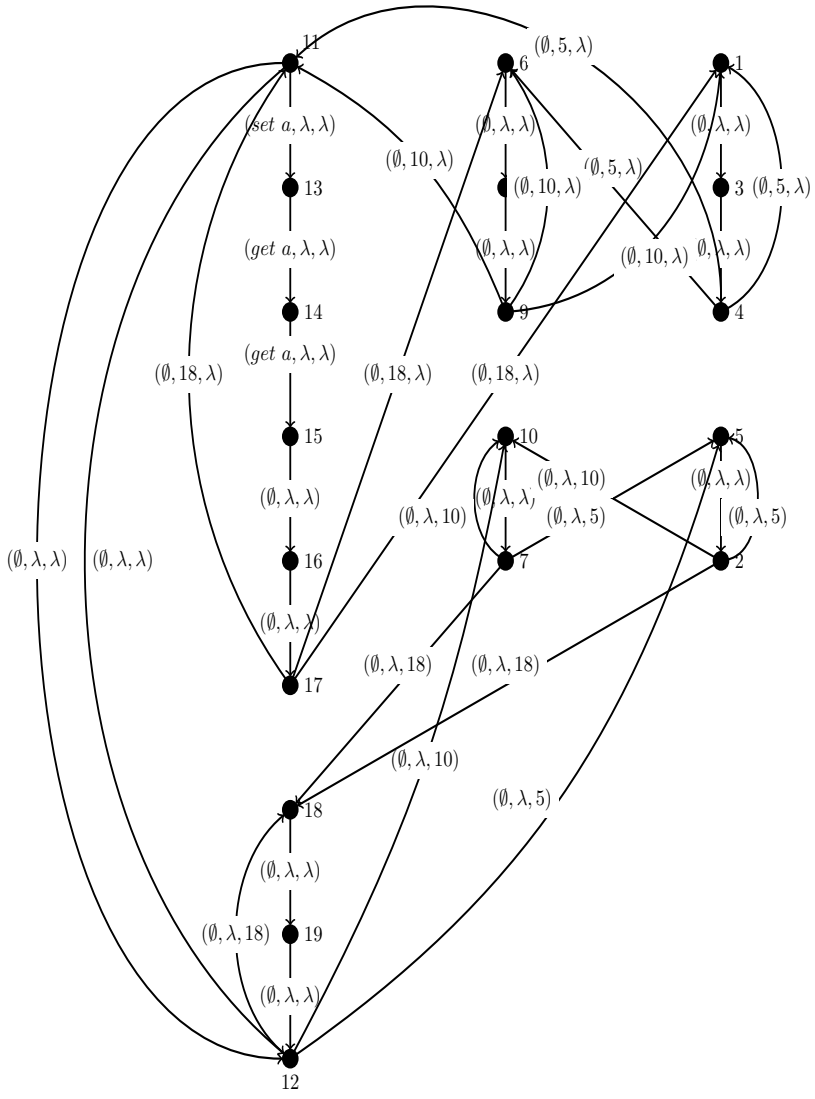

Fig. 7. The Automaton for P

Do has a formal operational semantics, but our work so far has not led to the formalization of the algorithms described and implemented here. Clearly a more complete formalization together with proofs of mutual correctness would be most desirable, and we plan to consider this in due course.

References

1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2003.
2. Timo Bingmann. Flex bison c++ template/example 0.1.4. http://idlebox.net/2007/flex-bison-cpp-example/.
3. Bison manual. http://www.gnu.org/software/bison/manual/.
4. R. L. Crole. Operational Semantics, Abstract Machines and Correctness, 2008. Lecture Notes for the Midlands Graduate School in the Foundations of Computer Science, LaTeX format 91 pages with subject and notation index, plus slides 1-up and 8-up.
5. R. L. Crole and A. D. Gordon. Relating Operational and Denotational Semantics for Input/Output Effects. *Mathematical Structures in Computer Science*, 9:125–158, 1999.
6. M. Felleisen and D. Friedman. *Control Operators, the SECD-machine, and the λ-calculus*, pages 193–217. North Holland, 1986.
7. Lexical analysis with flex. http://flex.sourceforge.net/manual/.
8. J. Hatcliff, M.B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13:315–353, 2000.
9. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transations on Programming Languages and Systems*, 12:26—60, 1990.
10. Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, Cambridge [U.K.], 2nd ed edition, 2004.
11. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, San Diego, California, United States, 1988. ACM.
12. Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50, Savannah, GA, USA, 2009. ACM.
13. Eugenio Moggi. Computational Lambda-Calculus and monads. *Information and Computation*, pages 14—23, 1988.
14. Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.
15. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.
16. J. Zhao. Multithreaded dependence graphs for concurrent java programs. In *Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on*, pages 13–23, 1999.