Deductively Verified Program Models for Software Model Checking

Jesper Amilon(✉) and Dilian Gurov

KTH Royal Institute of Technology, Stockholm, Sweden
{jamilon,dilian}@kth.se

Abstract. Model checking temporal properties of software is algorithmically hard. To be practically feasible, it usually requires the creation of simpler, abstract models of the software, over which the properties are checked. However, creating suitable abstractions is another difficult problem. We argue that such abstract models can be obtained with little effort, when the state transformation properties of the software components have already been deductively verified. As a concrete, language-independent representation of such abstractions we propose the use of *flow graphs*, a formalism previously developed for the purposes of compositional model checking. In this paper, we describe how we envisage the work flow and tool chain to support the proposed verification approach in the context of embedded, safety-critical software written in C.

Keywords: Flow graphs · Deductive verification · Model checking

1 Introduction

In previous work [1], we introduced a workflow for software model checking of C programs, the key point of which being that we utilised deductive verification of contracts for creating abstract models of C programs. When creating the models for model checking, we first replaced certain components with a Hoare-style contract, thus viewing those components purely as state-transformers, ignoring their intermediate states. The abstraction is justified by deductively verifying that the components indeed satisfy their contracts.

In [1], we used as tool chain the WP plugin of Frama-C [3] for deductive verification, and TLC [15] as model checker. The translation from a program with deductively verified contracts to a TLA$^+$ specification was partially performed with the support of the C2TLA$^+$ [10] tool. However, much of the translation process was carried out manually, in an ad hoc fashion. In this paper, we propose an explicit model of the artefact corresponding to a given program with deductively verified contracts. This model, called *flow graph*, abstracts from the code, keeping just the information needed to verify temporal properties of the original program. The model is independent of the actual model checking technique,

This work has been funded by the FFI Programme of the Swedish Governmental Agency for Innovation Systems (VINNOVA) as the AVerT2 project 2021-02519.

T. Margaria and B. Steffen (Eds.): ISoLA 2024, LNCS 15221, pp. 8–25, 2025.
https://doi.org/10.1007/978-3-031-75380-0_2

and allows automated translations to the input language of the desired model checker, be it TLC or NuXmv. The model is adapted from previous work on modular model checking [13].

```
1  #define LO 0
2  #define HI 10
3
4  extern int glob_stee_primary_status;  //input
5  extern int glob_stee_sndary_status;   //output
6
7  /*@
8    ensures LO <= global_stee_primary_status <= HI &&       //c_havoc
9    assigns global_stee_primary_status;
10 */
11 void havoc_input() { ... }
12
13 /*@ ensures \result == glob_stee_status;                   //c_read
14     assigns \nothing; */
15 int rtdb_read_primary_stee_status() { ... } //β4
16
17 /*@ ensures \old(stee_info) != 0 ==> \result == 0 &&      //c_eval
18           \old(stee_info) == 0 ==> \result == 1;
19     assigns \nothing;  */
20 int evaluate_stee_status(int stee_info) { ... }
21
22 /*@ ensures glob_stee_primary_status == status;           //c_write
23     assigns glob_stee_primary_status;  */
24 void rtdb_write_sndary_stee_status(int status) { ... }
25
26 void steering() { //β3
27   //Read input data from the RTDB
28   // 0 == Flawless; Non-zero == Error
29   int primary_info = rtdb_read_primary_stee_status();     //s_read
30
31   //Evaluate the data
32   int sndary_info = evaluate_stee_status(primary_info);   //s_eval
33
34   //Write back the result of the evaluation to the RTDB
35   rtdb_write_sndary_stee_status(sndary_info);             //s_write
36 }
37
38 //scheduler
39 void main() { //β1
40   while(1) {  //β2
41     havoc_input(); //simulating environment
42     steering();
43   }
44 }
```

Fig. 1. STEE example

Running Example. We illustrate our approach on a simplified version of a software module from the automotive industry. The module is called Stee, and is intended to control the secondary steering of a heavy-vehicle, and, in case of malfunction of the primary steering module, activate the secondary steering functionality. The Stee function is called periodically by the scheduler of the embedded system's control software. This example has been used in earlier work and is described in more detail in [1]. Figure 1 shows a code skeleton of Stee and a main function which acts as the scheduler. The example is interesting in

this context, as it combines the temporal behaviour of the scheduler (the while loop) with the transformational behaviour of STEE. We use ACSL [2] to specify certain parts of the program with contracts.

Related Work. The closest work to ours is presented by Beyer et al. [5], which abstracts programs by automatically computing *summaries* of loop-free fragments of the code. The difference from our approach is that contracts may also specify code with loops and procedure calls. Also, the summaries are automatically extracted, whereas we assume contracts provided by, e.g., humans. A summary can, however, also be viewed as the strongest contract for the code block it is extracted from. Beckert et al. [4] have developed an approach for (bounded) model checking of Java programs with JML-contracts. The verification approach is similar to the one presented here, but model checking is carried out with a bounded model checker (JBMC), and they do not consider an intermediate (flow graph) representation. The KRATOS2 tool [7] allows for converting C-programs into the intermediary K2 language, which can in turn be converted into NUXMV models to be model checked. It does not, as far as we are aware, handle abstractions based on contracts or summaries.

The MOXI [11] language is another intermediate language for symbolic model checking. MOXI is more general than the flow graphs presented here and is, for example, well-adapted also for hardware modelling. Flow graphs are intended specifically for modelling procedural software that has been deductively verified.

Outline. The paper is structured as follows. In Sect. 2, we present our overall approach to deductive verification based program abstraction. In Sect. 3, we describe our notation for programs (annotated with contracts) and define the notion of flow graph, which we use to represent program abstractions, while in Sect. 4 we show how to extract such flow graphs from annotated programs. Once a flow graph is extracted, it can be model checked as described in Sect. 5. We discuss various aspects of our approach in Sect. 6, and conclude with Sect. 7.

2 Deductive Verification Based Program Abstraction

In previous work [1], we introduced a workflow and tool chain for software model checking of C programs, where we used contracts and deductive verification. Using contracts as a basis of abstraction, we could utilise the modularity of deductive verification in the otherwise monolithic model checking setting.

In that work, we relied on the TLA framework [8] to act as the *bridging agent* between the realms of deductive verification and model checking. Following this idea, we considered a translation of C programs, annotated with contracts, into TLA^+ [9] specifications. However, by depending heavily on TLA, we imposed certain limitations on the considered tool chain. For instance, the TLC model checker, which is part of the TLA framework, is an explicit-state model checker, whereas one might expect a symbolic model checker to be more suitable for models related to *symbolic transition systems*, such as the TLA^+ specifications.

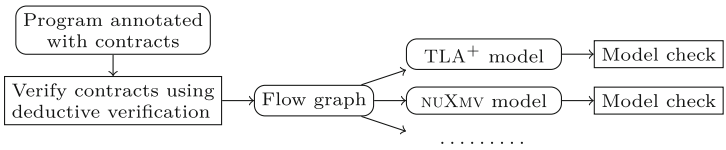


Fig. 2. Overview of the verification workflow using flow graphs

In this paper, we generalise our previous work by introducing an intermediate step, in which we first translate the program annotated with contracts, which we henceforth shall call *annotated program*, into a representation as a *flow graph*. The rationale for this is that this representation is independent of the source language, and can further be used to extract models in different modelling languages depending, for example, on which tool is most suitable for the verification task at hand. In this work, we consider extractions into TLA$^+$ and the nuXmv language, but we expect that also other modelling languages can be used. The verification approach, where flow graphs act as an intermediate step in the model extraction, is illustrated in Fig. 2. In the workflow, the role of the deductive verification step is to ensure certain *soundness* properties of the entire translation and verification chain.

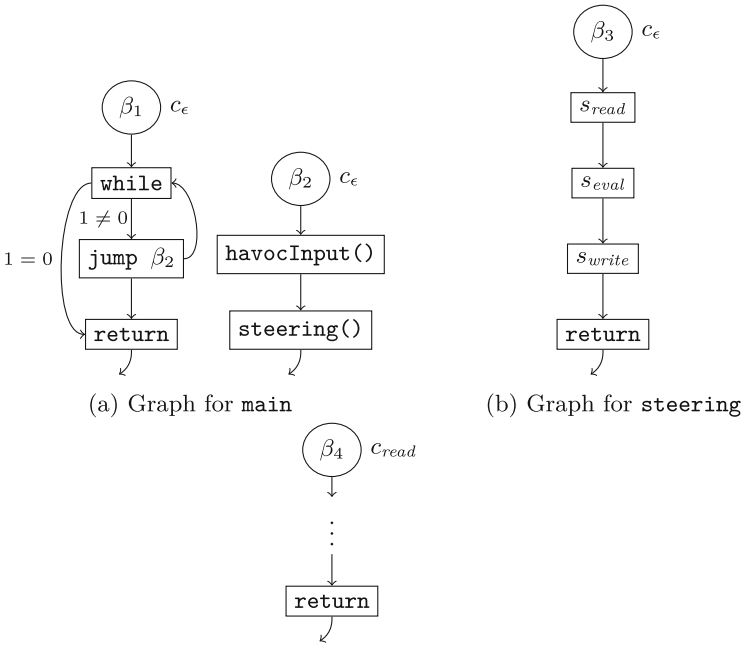
Flow graphs are a formalism previously developed for the purposes of compositional model checking [13]. We argue, for two main reasons, that they are also well-suited for the context presented here. First, there is a natural translation from a program annotated with contracts into a flow graph. Second, we believe that flow graphs have a natural translation into the modelling languages of several existing model checkers.

3 Annotated Programs and Flow Graphs

In this section, we first present our definition of annotated programs as directed graphs (akin to the standard notion of *control flow graphs*). We then define a similar object called a *flow graph*, which allows certain aspects of the original directed graph of the program to be abstracted away.

3.1 Annotated Programs

To simplify the presentation, we shall not consider here a specific programming language with its concrete syntax and semantics, but shall instead assume that programs are given to us as directed graphs. We consider programs as a set of procedures, which, in turn, consist of a set of (labelled) code blocks that map program locations to statements. With such a formalisation, contracts may specify either the entire procedure or smaller blocks within a procedure.



(a) Graph for `main` (b) Graph for `steering`

(c) Partial graph for `rtdb_read_primary_stee_status`

Fig. 3. Directed graph representation of parts of the program in Fig. 1.

The sets of statements *Stmt* and Boolean expressions *BExpr* are left unspecified, and similarly the set of contracts C.

Definition 1. (Annotated Code Block). *An annotated code block is a tuple* $\beta = (V_\beta, E_\beta, \pi_{V_\beta}, \pi_{E_\beta}, v_{e_\beta}, v_{r_\beta}, C_\beta)$*, where* V_β *is a set of control points,* $E_\beta \subseteq V_\beta \times V_\beta$ *a set of edges,* $\pi_{V_\beta} : V_\beta \to Stmt$ *a labelling of control points with statements, and* $\pi_{E_\beta} : E_\beta \to BExpr$ *a labelling of edges with path conditions. Finally,* $v_{e_\beta} \in V_\beta$ *is the entry point and* $v_{r_\beta} \in V_\beta$ *is the exit point of the block, and* C_β *is the contract for the block.*

In the set of contracts, we let c_ϵ denote the empty contract, indicating that the respective block does *not* been specified. The relation E_β describes the flow of control between statements, and the labelling π_{E_β} is induced from the Boolean guards of conditional or loop statements. In the set of Boolean expressions *BExpr*, we include an empty expression b_ϵ. When $\pi_{E_\beta}(v, v') = b$ and $b \neq b_\epsilon$, we say that v' is *guarded* by b (or that b is a *guard*).

In the example program from Fig. 1, we consider the contracts to annotate the entry block statement of the respective procedure body. A visual depiction of parts of the graph representation of the program is shown in Fig. 3.

Definition 2. (Annotated Procedure). *An* annotated procedure *with name p is a pair $P_p = (\mathcal{B}_p, \beta_p)$, where $\mathcal{B}_p$ is a collection of annotated block graphs, and β_p is the entry block.*

Definition 3. (Annotated Program). *An* annotated program *consists of a pair $(\mathcal{P}, main)$, where $\mathcal{P}$ is a collection of annotated procedures, and main the starting procedure.*

The definition of an annotated program puts no constraint on the correctness of the program relative to its annotations (that is, whether every annotated part implements its companion contract or not). However, as described in Sect. 2, verifying that the code fulfils the contracts is key to ensuring soundness of the abstraction approach considered in this paper. Thus, we shall say that a program is *correctly annotated* if every annotated block fulfils the contract annotation (which can be verified using, e.g., deductive verification tools).

3.2 Flow Graphs

We proceed with a formal definition of flow graphs, adapted, with significant modifications, from [13].

First, we define the set of program states as $State = State_L \times State_G$, where the local states $State_L$ and global states $State_G$ capture the values of the local and global variables, respectively. Given a state $s \in State$, we denote with s_l its component in $State_L$ and with s_g its component in $State_G$. Next, we define the notion of an *action*. We take the view of the TLA framework and evaluate actions on pairs of states, thus viewing them as relations between pre-states and post-states (similar to the well-known notion of a state transformer).

Definition 4. (Action). *An* action *$a \in A$ is a Boolean expression over* primed *variables x' and* non-primed *variables x.*

We do not provide the syntax in full detail here, but follow the idea of the TLA framework, where actions are logic formulas with variables being either *primed* or *non-primed*, with the intention that non-primed variables are evaluated in the pre-state and primed variables in the post-state. For example, $x' = x + 1$ is an action that holds over all state pairs (s, s') such that $s'(x) = s(x) + 1$. Semantically, an action is defined as a binary relation on states: $[\![a]\!] \subseteq State \times State$.

Definition 5. (Procedure Flow Graph). *A* procedure flow graph*, for a procedure with name p, is a finite transition system $F_p = (N_p, D, \rightarrow_p, A, \lambda_p, p_e, N_{p_r})$, where N_p is a set of nodes, D a set of procedure names, $\rightarrow_p \subseteq N_p \times (D \cup \{\epsilon\}) \times N_p$ a labelled transition relation, A a set of actions over State, $\lambda_p : N_p \rightarrow A$ a labelling of nodes with actions, $p_e \in N_p$ the entry node, and $N_{p_r} \subseteq N_p$ a set of return nodes.*

By labelling nodes with actions, each node can be seen as a state transformer, which in turn may correspond to, e.g., an assignment statement in the programming language, or to a contract. Edges are labelled either with a procedure name or with ϵ, the former case corresponding to procedure calls, and the latter to an intra-procedural transfer of control.

Definition 6. (Flow Graph). *A flow graph is a pair $(\mathcal{F}, main)$, where $\mathcal{F}$ is a collection of procedure flow graphs, and F_{main} is the main (or entry) procedure flow graph.*

An example of a flow graph is shown in Fig. 4, which is described in detail in Sect. 4.

We now proceed by defining the operational semantics of flow graphs by means of *pushdown systems*. We first introduce pushdown systems formally.

Definition 7. (Pushdown System). *A* pushdown system (PDS) *is a tuple $\mathcal{S} = (S, \Gamma, \triangle, I)$, where S is a set of control states, Γ a stack alphabet, and $\triangle \subseteq (S \times \Gamma) \times (S \times \Gamma^*)$ a set of rewrite rules. Moreover, we refer to $S \times \Gamma^*$ as the set of* configurations*, and $I \subseteq S \times \Gamma^*$ is the set of initial configurations.*

Let $\langle s, \gamma\rangle \rightarrow \langle s', \omega\rangle \in \triangle$ be a rewrite rule. Then, for each $\omega' \in \Gamma^*$, $\langle s', \omega \cdot \omega'\rangle$ is called an *immediate successor* of $\langle s, \gamma \cdot \omega'\rangle$.

Definition 8. (Run of a PDS). *A* run *of a pushdown system $\mathcal{S}$ is an infinite sequence of configurations $\langle s_1, \omega_1\rangle\langle s_2, \omega_2\rangle \dots$ where $\langle s_1, \omega_1\rangle \in I$, and for every $i \geq 1$, $\langle s_{i+1}, \omega_{i+1}\rangle$ is an immediate successor of $\langle s_i, \omega_i\rangle$. Moreover, a* state-run *of $\mathcal{S}$ is an infinite sequence of states $s_1 s_2 \dots$ such that there exists a run of $\mathcal{S}$ of the form $\langle s_1, \omega_1\rangle\langle s_2, \omega_2\rangle \dots$ for some $\omega_1, \omega_2, \dots \in \Gamma^*$.*

We are now ready to define how a flow graph *induces* a pushdown system, as inspired by Schwoon [12], so that runs of the flow graph can be defined in terms of runs of the induced pushdown system.

Definition 9. (Induced PDS). *Given a set of initial global states $G_{init} \subseteq State_G$, a flow graph $(\mathcal{F}, main)$* induces *a pushdown system $\mathcal{S}_{\mathcal{F}} = (S, \Gamma, \triangle, I)$, where:*

- *$S = State_G$, i.e., the global states of the program serve as control states;*
- *$\Gamma = (\bigcup_{F_p \in \mathcal{F}} N_p) \times State_L$, i.e., we store on the stack pairs of nodes and local states of procedures;*
- *Δ is induced as follows. Every procedure flow graph F_p induces the following sets of rewrite rules, where $init_l(p) \in State_L$ captures the initial values of the local variables in p:*
  (i) For each silent transition $(n_{p_1}, \epsilon, n_{p_2})$, the set of rewrite rules:

$$\{\langle s_g, (n_{p_1}, s_l)\rangle \rightarrow \langle s'_g, (n_{p_2}, s'_l)\rangle \mid (s, s') \in [\![\lambda_p(n_{p_1})]\!]\}$$

  (ii) For each call transition (n_{p_1}, p', n_{p_2}), the set of rewrite rules:

$$\{\langle s_g, (n_{p_1}, s_l)\rangle \rightarrow \langle s'_g, (p'_e, init_l(p')) \cdot (n_{p_2}, s'_l)\rangle \mid (s, s') \in [\![\lambda_p(n_{p_1})]\!]\}$$

(iii) If $p \neq main$, then for each return node $n_p \in N_{p_r}$, the set of rewrite rules:

$$\{\langle s_g, (n_p, l)\rangle \to \langle s'_g, \epsilon\rangle \mid (s, s') \in [\![\lambda_p(n_p)]\!]\}$$

The set of transition rules Δ is then the union of the above rules for each transition and return node.

- $I = \{\langle main_e, (init_l(main), g)\rangle \mid g \in G_{init}\}$.

The idea of the induced pushdown system is to model procedure calls using a stack. For a call transition (n_{p_1}, p', n_{p_2}) in procedure p, we push the next node in the current procedure graph (i.e., n_{p_2}) and the current local variable state to the stack. Then, for the case when the procedure p' returns, we simply add a rule that pops the local state and node from the stack (and discards the current local state), so that the execution of p can resume from n_{p_2} with correct values for the local variables. For silent transitions, we simply let the stack be untouched. The initial configurations are given by the possible initial global states, together with the entry node of the main procedure.

Definition 10. (Run of a Flow Graph). *A* run *of a flow graph $(\mathcal{F}, main)$ is a state-run of the pushdown system induced by $\mathcal{F}$.*

That is, when defining the run of the flow graph, we consider only the global component of the state. We view linear-time properties of a flow graph $\mathcal{F}$ as properties of its runs.

To illustrate, consider the flow graph in Fig. 4. The following sequence of configurations shows the first three steps of a run in the induced pushdown system.

$$\langle s_{1_g}, (n_{m_1}, s_{1_l})\rangle \langle s_{2_g}, (n_{m_2}, s_{2_l})\rangle \langle s_{2_g}, (n_{s_1}, init_l(stee)) \cdot (n_{m_3}, s_{2_l})\rangle \ldots$$

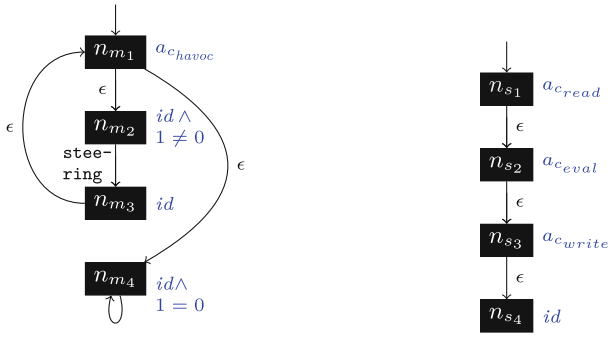
where $s_{1_l} = init_l(main)$, s_{1_g} is some initial values of the global variables, and $(s_1, s_2) \in [\![a_{c_{havoc}}]\!]$.

4 From Annotated Programs to Flow Graphs

In this section, we sketch a general translation from annotated programs into flow graphs, which was already hinted at in Sect. 3, where we showed in Fig. 4 the flow graph representation of Stee. We leave a fully formalised definition of the translation as future work.

For the translation sketch, we assume given a canonical, semantics-preserving representation of statements, contracts and Boolean expressions as actions, and use a_c, a_s and a_b to denote the actions corresponding to the contract c, statement s, and Boolean guard b, respectively. Then, an annotated program $(\mathcal{P}, main)$ is translated into a flow graph as follows.

For each annotated procedure $P_p = (\mathcal{B}_p, \beta_p)$, in the program, we construct a procedure flow graph $F_p = (N_p, D, \to_p, A, \lambda_p, p_e, \{p_r\})$. The nodes N_p are



(a) Flow graph for `main` (b) Flow graph for `steering`

Fig. 4. Flow graph for STEE

initially given by the control points $\bigcup_{\beta \in \mathcal{B}} V_\beta$. The node labelling λ_p is defined as follows:

$$\lambda_p(n_v) = \begin{cases} a_{c_\beta} & \text{if } \pi_V(v) = (\texttt{jump } \beta) \text{ and } c_\beta \neq c_\epsilon \\ a_{c_\beta} & \text{if } \pi_V(v) = s \text{ and } s \text{ is a procedure call to } p \text{ where the entry} \\ & \qquad \text{block } \beta_p \text{ is annotated with } c_{\beta_p} \neq c_\epsilon \\ a_s & \text{if } \pi_V(v) = (s) \text{ and } s \text{ is an assignment statement} \\ id & \text{otherwise} \end{cases}$$

That is, we replace annotated blocks, and calls to annotated procedures, with their respective contract. In addition to the labelling defined above, we also merge any remaining jumps, i.e., replace jumps to a non-specified block β with the body of β. Lastly, if s is guarded by b, then a_b is added as a conjunct to the label.

The sources and targets of the transition relation $\rightarrow_p$ for the remaining nodes are defined canonically from E_β for each block β. The labelling e of an edge (n_{v_i}, e, n_{v_j}) is defined as follows:

$$e = \begin{cases} p & \text{if } \pi_V(v_i) = (s) \text{ and } s \text{ is a call to procedure } p \text{ where the entry} \\ & \text{block } \beta_p \text{ of } p \text{ is annotated with } c_\epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

That is, we keep as edge labels all calls to procedures where the body is not specified. Lastly, we also add an ϵ-edge from the final node in the *main* procedure to itself, thus modelling terminating executions as ending by stuttering indefinitely in the final node of *main*.

In the example in Fig. 4, we see that the nodes n_{m_1}, n_{s_1}, n_{s_2}, and n_{s_3} are labelled with (the action of) the respective contract. We assume here for simplicity that the actions $a_{c_{read}}$ and $a_{c_{eval}}$ also describe the assignments to the local variables `primary_info` and `sndary_info`. The remaining nodes are labelled

with the identity action, and the guarded statements stemming from the while loop in the main procedure have a conjunct for the guards. The only non-specified procedure call, the one to `steering`, is maintained as the label of the transition from n_{m_2} to n_{m_3}.

5 Model Checking of Flow Graphs

Once a flow graph has been extracted from an annotated program, it can be model checked against temporal properties by translating it into the input modelling language of the specific model checking framework at hand. Here, we illustrate the idea using two well-known frameworks, TLA and NUXMV.

5.1 Model Checking with TLC

TLC [15] is an explicit state model checker that is part of the TLA [8] framework. TLC takes as input a TLA$^+$ [9] program, where TLA$^+$ is the language implementation of TLA. TLC can model check it either for deadlocks, or against temporal properties, which are also specified in TLA$^+$. A beautiful aspect of TLC is that both the models and the properties are specified in the same language. The TLA$^+$ language uses actions (see Sect. 3.2) as elementary propositions, and temporal properties are specified as in LTL, using, e.g., the temporal operators $\Diamond$ ("*eventually*") and $\Box$ ("*always*"). Programs in TLA$^+$ are defined as *symbolic transition systems*, comprised of a unary *initial-state* predicate *Init* and a binary *next-state* predicate M. Sometimes, a third component, a *fairness constraint* F is added explicitly as well. Given a triple $(Init, M, F)$, the corresponding TLA$^+$ program is the formula $Init \wedge \Box M \wedge F$.

To model check a flow graph with TLC, we translate the flow graph to a semantically equivalent TLA$^+$ specification. In particular, we define a variable n representing the current node in a run, and a variable st representing the stack in the pushdown system. We also define variables s_g and s_l corresponding to the state for the local and global variables.

Definition 11. (Induced TLA$^+$ Specification). *Given a flow graph $(\mathcal{F}, main)$, we define, for each procedure flow graph $F_p \in \mathcal{F}$, a set of actions as follows:*

(i) For each silent transition $(n_{p_1}, \epsilon, n_{p_2})$, an action:

$$n = n_{p_1} \wedge n' = n_{p_2} \wedge st' = st \wedge \lambda_p n_{p_1}$$

(ii) For each call transition (n_{p_1}, p', n_{p_2}), an action:

$$n = n_{p_1} \wedge n' = n_{p'} \wedge st' = push((p, s_l), st) \wedge \lambda_p n_{p_1}$$

(iii) For each return node $n_p \in N_{p_r}$, an action:

$$((n', l'), st') = pop(st) \wedge \lambda_p(n_p)$$

The full TLA$^+$ *specification of the program is then the specification:*

$$Init \wedge \Box M$$

where M is the disjunction of all actions induced as described above, and Init is the initial constraint on the global variables and sets n, and the local variables according to F_{main}.

```
\* Contracts
c_havoc == glob_stee_primary_status' \in LO..HI /\
           UNCHANGED(glob_stee_sndary_status)
c_read(res_var) == assign_loc(res_var, glob_stee_primary_status)
                   /\ fix_globs
c_eval(stee_info, res_var) ==
  (fetch_loc(stee_info) # 0 => assign_loc(res_var, 0))
  /\ (fetch_loc(stee_info) = 0 => assign_loc(res_var, 1))
  /\ fix_globs
c_write(status) == (glob_stee_sndary_status' = fetch_loc(status))
                   /\ fix_lvars /\ UNCHANGED(glob_stee_primary_status)

\*Main
main1 == n = "n_main1" /\ n' = "n_main2" /\ c_havoc /\ fix_stack /\ fix_lvars
main2 == n = "n_main2" /\ n' = "n_main4" /\ fix_stack /\ fix_lvars /\ 1 = 0
main3 == n = "n_main2" /\ n' = "n_stee1"  /\
         push("n_main3", lvars) /\ init_stee(lvars') /\ fix_globs /\ 1 # 0
main4 == n = "n_main3" /\ n' = "n_main1" /\ fix_stack /\ fix_lvars
         /\ fix_globs
main5 == n = "n_main4" /\ n' = "n_main4" /\ fix_stack /\ fix_lvars
         /\ fix_globs

\*Stee
stee1 == n = "n_stee1" /\ n' = "n_stee2" /\ fix_stack /\ c_read(primary_info)
stee2 == n = "n_stee2" /\ n' = "n_stee3" /\ fix_stack
         /\ c_eval(primary_info, sndary_info)
stee3 == n = "n_stee3" /\ n' = "n_stee4" /\ fix_stack /\ c_write(sndary_info)
stee4 == n = "n_stee4" /\ pop(n', lvars') /\ fix_globs

\*Program
Init == n = "n_main1" /\ init_main(lvars) /\ init_stack /\ init_globals
M == (main1 \/ main2 \/ main3 \/ main4 \/ main5 \/
      stee1 \/ stee2 \/ stee3 \/ stee4)

Spec == Init /\ [][M]_all_vars
```

Fig. 5. TLA$^+$ model for Stee

In Fig. 5, we show the key parts of the TLA$^+$ model induced by the flow graph in Fig. 4. The full code is available in Appendix A.

As TLC is an explicit-state model checker, model checking of the program is only possible for finite domains for the variables and the stack. In particular, we cannot model check flow graphs with infinite variable domains or unbounded recursion.

5.2 Model Checking with nuXmv

nuXmv [6] is a symbolic model checker for finite or infinite-state systems, which takes as input a model in the nuXmv language, and can verify the model against properties written in either LTL or CTL. Similar to TLA$^+$, a nuXmv model can also be given as a symbolic transition system, with an *initial constraint* and *transition constraints*, the latter corresponding to the next-state relation in TLA$^+$. Transition constraints are, semantically speaking, binary relations over the states of the transitions system, and are thus similar to TLA$^+$ actions. Syntactically, they differ from TLA$^+$ actions in that they use the `next` keyword instead of primed variables. For readability, nuXmv also allows *define* declarations, which function as macros.

To the best of our knowledge, the type system in nuXmv is less flexible than the one of TLA$^+$. In particular, there is no data type able to represent sequences of variable size, thus encoding the stack is less straightforward; for example, the maximum size of the stack must be both finite, and known beforehand. On the other hand, nuXmv supports model checking of infinite-state models, so that we may consider infinite domains for the program states.

The following definition shows how a flow graph induces a nuXmv model. Since both TLA$^+$ and nuXmv are symbolic-transition systems, the translation below closely resembles Definition 11.

Definition 12. (Induced nuXmv Model). *Given a flow graph $(\mathcal{F}, main)$, we define, for each procedure flow graph $F_p \in \mathcal{F}$, declarations as follows:*

(i) For each silent transition $(n_{p_1}, \epsilon, n_{p_2})$, a define declaration:

$$n = n_{p_1} \wedge next(n) = n_{p_2} \wedge next(st) = st \wedge \lambda_p n_{p_1}$$

(ii) For each call transition (n_{p_1}, p', n_{p_2}), a define declaration:

$$n = n_{p_1} \wedge next(n) = n_{p_2} \wedge next(st) = push((p, s_l), st) \wedge \lambda_p n_{p_1}$$

(iii) For each return node $n_p \in N_{p_r}$, a define declaration:

$$next(((n, l), st)) = pop(st) \wedge \lambda_p(n_p)$$

The nuXmv model is then given as a transition constraint corresponding to the disjunction of the define declarations in (i) - (iii), and an `INIT` constraint, which describes the initial constraints of the global variables, and sets the initial node, and local variables according to F_{main}.

The key parts of the nuXmv model for the flow graph in Fig. 4 are shown in Fig. 6. The full model is shown in Appendix A.

```
-- contracts
  c_havoc := next(glob_stee_primary_status) in 0..10
           & next(glob_stee_sndary_status) = glob_stee_sndary_status;
  c_read := next(lvars[primary_info]) = glob_stee_primary_status
          & fix_globs;
  c_eval := (lvars[primary_info] != 0 -> next(lvars[sndary_info]) = 0)
          & (lvars[primary_info] = 0 -> next(lvars[sndary_info]) = 1)
          & fix_globs;
  c_write := (next(glob_stee_sndary_status) = lvars[sndary_info])
           & fix_lvars
           & next(glob_stee_primary_status) = glob_stee_primary_status;


-- main
  Main1 := (n = n_main1 & next(n) = n_main2 & c_havoc & fix_lvars
           & fix_stack);
  Main2 := (n = n_main2 & next(n) = n_main4 & fix_lvars & fix_stack & 1 = 0);
  Main3 := n = n_main2 & next(n) = n_main3 & fix_stack & fix_lvars & push
         & init_stee_lvars & 1 = 0;
  Main4 := n = n_main3 & next(n) = n_stee1 & fix_stack & fix_lvars &
    fix_globs;
  Main5 := n = n_main4 & next(n) = n_main4 & fix_stack & fix_lvars &
    fix_globs;

-- stee
  Stee1 := n = n_stee1 & next(n) = n_stee2 & fix_stack & c_read;
  Stee2 := n = n_stee2 & next(n) = n_stee3 & fix_stack & c_eval;
  Stee3 := n = n_stee3 & next(n) = n_stee4 & fix_stack & c_write;
  Stee4 := n = n_stee4 & pop & fix_globs;

INIT
  lvars = init_main_lvars & vstack = init_vstack & curr_top = 0 &
  nstack = init_nstack & init_globals & n = n_main1;

TRANS
  Main1 | Main2 | Main3 | Main4 | Main5 | Stee1 | Stee2 | Stee3 | Stee4;
```

Fig. 6. nuXmv model of Stee

6 Discussion

We expect the approach presented in this paper to be particularly applicable to automotive embedded software. To see why, consider the simplified Stee example in Fig. 1. In a real example, the scheduler would periodically call numerous modules. Thus, if we annotate each module with a contract, we can perform model checking over the contracts of the modules, taking into account the actual code only of the scheduler. This also aligns, in our experience, with how requirements are specified in the automotive industry, where software modules are typically accompanied with a number of safety-critical requirements that can be deductively verified.

When considering the Stee example, as presented here, the use of a pushdown system in the formalisation may seem unnecessary. However, depending on the property to verify, one may want to abstract away, for example, only a subset of the procedures in the program model. Then, the pushdown system is necessary to capture the behaviour of procedure calls which one does not want to abstract away from.

In the translations considered in this work, the role of deductive verification is to justify the abstraction for the model checking task. That is, by showing that

the code satisfies the contracts, we may guarantee certain soundness properties of the abstraction. We are yet to work out the details of such guarantees, in particular the interplay between deductive verification and the temporal properties to be verified with model checking.

In this work, we assumed that the contracts have been provided by some (likely human) oracle. By taking this approach, we allow the *reuse* of earlier results in one domain (deductive verification), for the purpose of abstraction in another domain (model checking). The validity of this approach is corroborated by experience from an earlier case study on deductive verification of automotive embedded software [14], where we encountered requirements that we formalised and verified using deductive verification, and other requirements that were better suited for verification with model checking.

7 Conclusion

In this paper, we described an approach for deductive verification based program abstraction, where *flow graphs* act as an intermediate representation in extracting models from programs annotated with contracts. We have illustrated the approach on a code skeleton inspired by code from the heavy-vehicle industry, showing both how a program, annotated with contracts, can be translated into an abstract flow graph, and how the flow graph, in turn, induces TLA$^+$ and nuXmv models. We have also presented an operational semantics for flow graphs by means of an induced pushdown system, which allows semantic preservation properties of the translations to be established formally.

Future Work. Future work includes first and foremost experimental evaluation of model extraction. In particular, we are interested in exploring how the abstraction impacts verification times when model checking. We also plan to show semantic preservation properties of the defined translations, and identify fragments of LTL for which the verification (model checking) in our approach is sound. Future work also includes implementing the translation steps described. Lastly, we also consider combining the contracts-based method presented here with the large-block encoding presented in [5], to also consider abstraction based on *summaries*, which can be automatically extracted from the code.

Acknowledgement. Some of the concepts presented here were discussed at the Lorentz center workshop on Contract Languages, 4–8 March 2024.

A Full TLA$^+$ and nuXmv Model of Stee

A.1 TLA$^+$ Model

In the TLA$^+$ model, the stack consists of a tuple of stacks, one that maintains the node and one that maintains the variables. The model also defines a number of auxiliary actions, including popping from and pushing to the stack, and reading

from and writing to variables. The key parts of the model is the contract actions, which essentially captures the contracts as state transformers, and the program, which is based on applying Definition 11 to Fig. 4. Note that each contract action and program action must define also what variables are *not* written to, as required by TLC.

```
-----------------MODULE Stee_isola24------------------------------
EXTENDS Sequences, TLC, Naturals

VARIABLES
    nstack, vstack,
    \* Global vars
    glob_stee_primary_status,
    glob_stee_sndary_status,
    \* Current control state and local vars
    n, lvars

\*Constants
LO == 0
HI == 10
stack == <<nstack, vstack>>
primary_info == 1
sndary_info == 2

\*Inits
init_main(l) == l = <<>>
init_stee(l) == l = primary_info :> 0 @@ sndary_info :> 0

init_globals == glob_stee_primary_status \in {0,1} /\
                glob_stee_sndary_status \in {0,1}
init_stack == nstack = <<>> /\ vstack = <<>>

\*Auxiliary funcs
pop(_n, _v) == _n = Head(nstack) /\ _v = Head(vstack) /\ nstack' = Tail(
    nstack) /\ vstack' = Tail(vstack)
push(_n, _v) ==  nstack' = <<_n>> \o nstack /\ vstack' = <<_v>> \o vstack
assign_loc(x, v) == lvars' = [lvars EXCEPT ![x] = v] \*loc_env' = [loc_env
    EXCEPT !.lvars = [loc_env.lvars EXCEPT ![x] = v]]
fetch_loc(x) == lvars[x]
fix_globs == UNCHANGED(<<glob_stee_primary_status, glob_stee_sndary_status>>)
fix_stack == UNCHANGED(stack)
fix_lvars == UNCHANGED(lvars)
all_vars == <<glob_stee_primary_status, glob_stee_sndary_status, stack,lvars,
     n>>


\* Contracts
c_havoc == glob_stee_primary_status' \in LO..HI /\
           UNCHANGED(glob_stee_sndary_status)
c_read(res_var) == assign_loc(res_var, glob_stee_primary_status)
                   /\ fix_globs
c_eval(stee_info, res_var) ==
  (fetch_loc(stee_info) # 0 => assign_loc(res_var, 0))
  /\ (fetch_loc(stee_info) = 0 => assign_loc(res_var, 1))
  /\ fix_globs
c_write(status) == (glob_stee_sndary_status' = fetch_loc(status))
                   /\ fix_lvars /\ UNCHANGED(glob_stee_primary_status)

\*Main
main1 == n = "n_main1" /\ n' = "n_main2" /\ c_havoc /\ fix_stack /\ fix_lvars
main2 == n = "n_main2" /\ n' = "n_main4" /\ fix_stack /\ fix_lvars /\ 1 = 0
main3 == n = "n_main2" /\ n' = "n_stee1"  /\
         push("n_main3", lvars) /\ init_stee(lvars') /\ fix_globs /\ 1 # 0
main4 == n = "n_main3" /\ n' = "n_main1" /\ fix_stack /\ fix_lvars
         /\ fix_globs
main5 == n = "n_main4" /\ n' = "n_main4" /\ fix_stack /\ fix_lvars
```

```
58          /\ fix_globs
59
60  \*Stee
61  stee1 == n = "n_stee1" /\ n' = "n_stee2" /\ fix_stack /\ c_read(primary_info)
62  stee2 == n = "n_stee2" /\ n' = "n_stee3" /\ fix_stack
63          /\ c_eval(primary_info, sndary_info)
64  stee3 == n = "n_stee3" /\ n' = "n_stee4" /\ fix_stack /\ c_write(sndary_info)
65  stee4 == n = "n_stee4" /\ pop(n', lvars') /\ fix_globs
66
67  \*Program
68  Init == n = "n_main1" /\ init_main(lvars) /\ init_stack /\ init_globals
69  M == (main1 \/ main2 \/ main3 \/ main4 \/ main5 \/
70        stee1 \/ stee2 \/ stee3 \/ stee4)
71
72  Spec == Init /\ [][M]_all_vars
73  ============================================================================
```

A.2 nuXmv model

The nuXmv model is similar to the TLA$^+$ one, but differs slightly as nuXmv is, for example, more strictly types than TLA$^+$. Since array sizes must be fixed to a constant in nuXmv, we set the stack to be of size 10, as we know this to be enough in this model.

```
1   MODULE main
2
3   VAR
4       nstack : array 0..1 of {n_main1, n_main2, n_main3, n_main4, n_main5,
5                               n_stee1, n_stee2, n_stee3, n_stee4,dummy};
6       vstack : array 0..1 of array 0..1 of 0..10;
7       curr_top : 0..1;
8       --\* Global vars
9       glob_stee_primary_status : 0..10;
10      glob_stee_sndary_status : 0..10;
11      --\* Current control state and local vars
12      n : {n_main1, n_main2, n_main3, n_main4, n_main5, n_stee1, n_stee2,
        n_stee3, n_stee4};
13      lvars : array 0..1 of 0..10;
14
15
16  DEFINE
17    --loc vars index
18    primary_info := 0;
19    sndary_info := 1;
20
21  --\*Inits
22    init_main_lvars := [0,0];
23    init_stee_lvars := next(lvars[primary_info]) = 0 & next(lvars[sndary_info])
          = 0;
24
25    init_globals := glob_stee_primary_status in 0..1 &
26                    glob_stee_sndary_status in 0..1;
27    init_vstack := [[0,0],[0,0]];
28    init_nstack := [dummy,dummy];
29
30
31
32  -- Auxiliary funcs
33    pop := next(lvars[0]) = vstack[curr_top][0]
34           & next(lvars[1]) = vstack[curr_top][1] & next(n) = nstack[curr_top]
35           & next(curr_top) = curr_top - 1;
36    push := next(vstack[curr_top][0]) = lvars[0]
37            & next(vstack[curr_top][1]) = lvars[1]
38            & next(nstack[curr_top]) = n & next(curr_top) = curr_top + 1;
```

```
  fix_globs := next(glob_stee_sndary_status) = glob_stee_sndary_status
               & next(glob_stee_primary_status) = glob_stee_primary_status;
  fix_lvars := next(lvars[0]) = lvars[0] & next(lvars[1]) = lvars[1];
  fix_stack := next(vstack[0][0]) = vstack[0][0]
               & next(vstack[0][1]) = vstack[0][1]
               & next(vstack[1][0]) = vstack[1][0]
               & next(vstack[1][1]) = vstack[1][1];

-- contracts
  c_havoc := next(glob_stee_primary_status) in 0..10
             & next(glob_stee_sndary_status) = glob_stee_sndary_status;
  c_read := next(lvars[primary_info]) = glob_stee_primary_status
            & fix_globs;
  c_eval := (lvars[primary_info] != 0 -> next(lvars[sndary_info]) = 0)
            & (lvars[primary_info] = 0 -> next(lvars[sndary_info]) = 1)
            & fix_globs;
  c_write := (next(glob_stee_sndary_status) = lvars[sndary_info])
             & fix_lvars
             & next(glob_stee_primary_status) = glob_stee_primary_status;


-- main
  Main1 := (n = n_main1 & next(n) = n_main2 & c_havoc & fix_lvars & fix_stack
    );
  Main2 := (n = n_main2 & next(n) = n_main4 & fix_lvars & fix_stack & 1 = 0);
  Main3 := n = n_main2 & next(n) = n_main3 & fix_stack & fix_lvars & push
           & init_stee_lvars & 1 = 0;
  Main4 := n = n_main3 & next(n) = n_stee1 & fix_stack & fix_lvars &
    fix_globs;
  Main5 := n = n_main4 & next(n) = n_main4 & fix_stack & fix_lvars &
    fix_globs;

-- stee
  Stee1 := n = n_stee1 & next(n) = n_stee2 & fix_stack & c_read;
  Stee2 := n = n_stee2 & next(n) = n_stee3 & fix_stack & c_eval;
  Stee3 := n = n_stee3 & next(n) = n_stee4 & fix_stack & c_write;
  Stee4 := n = n_stee4 & pop & fix_globs;

INIT
  lvars = init_main_lvars & vstack = init_vstack & curr_top = 0 &
  nstack = init_nstack & init_globals & n = n_main1;

TRANS
  Main1 | Main2 | Main3 | Main4 | Main5 | Stee1 | Stee2 | Stee3 | Stee4;
```

References

1. Amilon, J., Lidström, C., Gurov, D.: Deductive verification based abstraction for software model checking. In: Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (ISoLA 2022). Lecture Notes in Computer Science, vol. 13701, pp. 7–28. Springer (2022). https://doi.org/10.1007/978-3-031-19849-6_2
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html
3. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual – Frama-C 23.1 (Vanadium). CEA LIST. https://frama-c.com/download/frama-c-wp-manual.pdf
4. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, pp. 60–80. Springer International Publishing, Cham (2020)

5. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., University, S.F., Sebastiani, R.: Software model checking via large-block encoding. In: 2009 Formal Methods in Computer-Aided Design, pp. 25–32 (2009). https://doi.org/10.1109/FMCAD.2009.5351147
6. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 334–342. Springer International Publishing, Cham (2014)
7. Griggio, A., Jonáš, M.: Kratos2: An SMT-based model checker for imperative programs. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 423–436. Springer Nature Switzerland, Cham (2023)
8. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
9. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002). http://research.microsoft.com/users/lamport/tla/book.html
10. Methni, A., Lemerre, M., Ben Hedia, B., Haddad, S., Barkaoui, K.: Specifying and verifying concurrent C programs with TLA+. In: Artho, C., Ölveczky, P.C. (eds.) Formal Techniques for Safety-Critical Systems, pp. 206–222. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-17581-2_14
11. Rozier, K.Y., et al.: MoXI: an intermediate language for symbolic model checking. In: Proceedings of the 30th International Symposium on Model Checking Software (SPIN), LNCS, Springer (April 2024). https://doi.org/10.1007/978-3-031-65627-9_10
12. Schwoon, S.: Model-Checking Pushdown Systems. Ph.D. thesis, Technical University of Munich (2002)
13. Soleimanifard, S., Gurov, D.: Algorithmic verification of procedural programs in the presence of code variability. Sci. Comput. Program. **127**, 76–102 (2016). https://doi.org/10.1016/J.SCICO.2015.08.010
14. Ung, G., Amilon, J., Gurov, D., Lidström, C., Nyberg, M., Palmskog, K.: Post-hoc formal verification of automotive software with informal requirements: an experience report. In: 2024 IEEE 32snd International Requirements Engineering Conference (RE) (2024). To appear
15. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods, pp. 54–66. Springer, Berlin Heidelberg, Berlin, Heidelberg (1999)