Using Language Engineering to Lift Languages and Analyses at the Domain Level

Daniel Ratiu[1], Markus Voelter[2], Bernd Kolb[3], and Bernhard Schaetz[1]

[1] fortiss, ratiu|schaetz@fortiss.org
[2] independent/itemis, voelter@acm.org
[3] itemis, kolb@itemis.de

Abstract. Developers who use C model checkers have to overcome three usability challenges: First, it is difficult to express application level properties as C-level verification conditions, due to the abstraction gap. Second, without advanced IDE support, it is difficult to interpret the counterexamples produced by the model checker and understand what went wrong in terms of application level properties. Third, most C model checkers support only a subset of C and it is easy for developers to inadvertently use C constructs outside this subset. In this paper we report on our preliminary experience with using the MPS language workbench to integrate the CBMC model checker with a set of domain-specific extensions of C for developing embedded software. Higher level language constructs such as components and decision tables makes it easier for end users to bridge the abstraction gap, to write verification conditions and to interpret the analysis results. Furthermore, the use of language workbenches allows the definition of analyzable language subsets, making the implementation of analyses simpler and their use more predictable.

1 Introduction

Current C model checkers have reached a level of scalability that makes them useful for real-world projects. However, their adoption in practice is much lower than it could be. There are three categories of challenges in using C model checkers [1,2]: First, it is difficult to formalize the to-be-verified application-level properties at the level of C, so model checkers are used only to verify implicit C-level properties (e.g., program does no crash, no overflow occurs). However, this is often not enough for end users. Second, once the result is obtained (at the abstraction level of C) it is difficult for a user to interpret it at the application level. Third, due to the complexity of C itself, many model checkers support only a subset of C and/or are simply buggy when certain C features are used. All these challenges are due to the gap between the abstractions relevant at the application level and how they are reflected in programs on the one hand, and the abstractions of the analysis tool on the other hand.

In this paper we propose a method to simplify the use of C model checkers that is based on the following three pillars: 1) we describe how various extensions of C encode higher level abstractions and their (explicit or implicit) properties;

2) we lift the analysis results to the application level, making them more understandable to the user; and 3) we define language restrictions that reflect limitations of C model checkers, making them evident to the user. We have implemented this method in mbeddr, an extensible version of C. As examples for C extensions we use components and decision tables. As analyses examples we show completeness and consistency of decision tables, and checking of interface contracts and protocols for components by using the CBMC model checker [3].

2 mbeddr: an Extensible C Language

mbeddr ([4] and `http://mbeddr.com`) is an extensible set of languages for embedded software development based on C, supporting the incremental, modular domain-specific extension of C. mbeddr also supports language restriction, in order to create subsets of existing languages. mbeddr is based on the JetBrains MPS language workbench (`http://jetbrains.com/mps`) and exploits its capabilities for language modularization and composition [5].

Out of the box, mbeddr comes with a set of extensions for interfaces and components, state machines, physical units and decision tables. Some of them lend themselves to formal analysis: currently we have integrated the Yices SMT solver (e.g. to verify decision table consistency) and the NuSMV model checker (for verifying state machines) [4,6,7]. In this paper we illustrate how language extension mechanisms allow a deep integration of the CBMC model checker.

```
exported cs interface SpeedComputer {
  void activate()
    protocol init(0) -> new Active(1)
  float computeSpeed(int16 distance, int16 time)
    pre(0) time > 0
    pre(1) distance > 0
    post(2) result > 0
    protocol Active -> Active
  void deactivate()
    protocol Active -> init(0)
}
```

```
instances comp {
  instance PlauzibilizedSpeedComputer sp
  adapt comp -> sp.speedComputer }

float emitCurrentSpeed() {
  int16 time = readTime();
  int16 dist = readDistance();
  if (dist >= 1 && time >= 0) {
    return comp.computeSpeed(dist, time); } if
  return 0; } emitCurrentSpeed (function)

exported int32 main(int32 argc, int8*[] argv) {
  initialize comp;
  emitCurrentSpeed();
  return 0; } main (function)
```

Fig. 1. Interface definition (left); Use of the interface in client code (right)

Interfaces, Components, Contracts. An interface defines a set of operations. In addition to the signature, each operation can define preconditions and postconditions. In addition, a protocol state machine defines the valid call sequences of the operations in an interface. The left part of Fig. 1 shows an example interface definition. `computeSpeed` has two preconditions, one postcondition and a protocol specification that specifies that `activate` must be called before calling `computeSpeed` (when `computeSpeed` is called, the interface must be in the `Active` state, which can be reached by calling `activate`). Fig. 2 shows a component that provides the `SpeedComputer` interface. The right part of Fig. 1 shows an example of client code of the component. Using model checking, we can verify whether a clients conforms to the preconditions and the protocol, and whether the implementation of the interface satisfies the postconditions.

```
[verifiable]
exported component PlauzibilizedSpeedComputer extends nothing {
  provides SpeedComputer speedComputer

  float lastSpeed = -1;
  boolean initialized = false;
  float maxPlausibleDelta = 10;

  float computeSpeed(int16 distance, int16 time) <= op speedComputer.computeSpeed {
    float currentSpeed = distance / time;
    float delta = (lastSpeed - currentSpeed < 0) ? currentSpeed - lastSpeed : lastSpeed - currentSpeed;
    return unsafe(float)<-1.0>
```

	delta < maxPlausibleDelta	delta > maxPlausibleDelta
initialized	lastSpeed = currentSpeed; yield currentSpeed;	lastSpeed
!initialized	initialized = false; yield currentSpeed;	initialized = true; yield currentSpeed;

```
[verifiable];
  } runnable computeSpeed

  void activate() <= op speedComputer.activate {
    initialized = true;
    return; } runnable activate
  void deactivate() <= op speedComputer.deactivate {
    initialized = false;
    return; } runnable deactivate
} component PlauzibilizedSpeedComputer
```

Fig. 2. Components implement each function of their provided interfaces. The users of a component must comply with the preconditions and use protocol defined in the interface. The implementation of each interface functions should comply with the defined post-conditions. In `computeSpeed` we show an example of decision tables.

Decision Tables. Decision tables [8] exploit JetBrains MPS' projectional editor in order to represent two-level nested if statements as a table (Fig. 2). The tabular notations makes it easier for developers to write and understand sets of input conditions. Decision tables suggest two verifications: completeness (check whether all possible input value combinations are covered), and determinism (checks that for any given set of input values only one option is valid).

3 Integrating CBMC into mbeddr

Fig. 3 shows the integration of CBMC: from programs written with higher-level constructs we generate C that includes a set of labels that represent higher-level verification properties (see next paragraph). The C code is then analyzed with CBMC and the analysis results are parsed and lifted back to the abstraction level of the higher-level constructs to make them easy to interpret.

Encoding verification conditions as reachability. We verify pre- and post-conditions, protocols and decision tables with the help of reachability analysis. As shown in Fig. 4, we generate labels (the things with the long numbers) to annotate locations in the code which represent violations of the high level properties. For example, operation implementations in components have `if` statements at the beginning that check the precoditions. The label is placed inside the body of the `if`. The body is only executed if the precondition fails. We maintain a

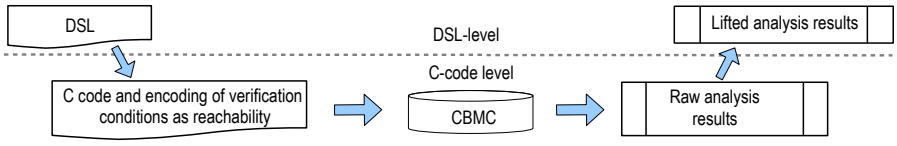

Fig. 3. Approach at a glance: generate C code, run CBMC and lift the raw results.

```
1  float computeSpeed(int16_t distance, int16_t time, void* ___inst) {
2    PlauzibilizedSpeedComputer* ___ci = ((PlauzibilizedSpeedComputer*)(___inst));
3    switch (___ci->___protocolState) {
4      case 2: { ___ci->___speedComputer_protocolState = 2; break; }
5      default: { protocolViolationForRunnable_2161187783549496741: break; }
6    }
7    if (!(time > 0)) { pre_2161187783549496724__2161187783549496741: ... }
8    float currentSpeed = distance / time;
9    float delta = ...
10   float ___result = decTabExp(delta, ___ci, delta, currentSpeed);
11   if (!(___result > 0)) { post_2161187783549496732__8053687140971342992: ...
12     return ___result;
13   }
14 static float decTabExp(float delta, struct PlauzibilizedSpeedComputer* ___inst,
15                         float delta, float currentSpeed) {
16   if (/* no case covered */) { label_dectab_completeness_8053687140971342993: ... }
17   if ((delta < ___inst->field_maxPlausibleDelta) && (___inst->field_initialized) &&
18       (delta < ___inst->field_maxPlausibleDelta) && !(___ci->field_initialized)) {
19     label_dectab_nondeterminism_0_8053687140971342993: ... } ... }
```

Fig. 4. Generated C code from the implementation of the interface

mapping between each label and the higher-level construct whose property the label represents. We then use CBMC to check whether the labels can be reached.

Lifting the Result. Running the reachability analysis with CBMC on the generated C provides a raw analysis result at the abstraction level of C. It specifies for each label whether it can be reached or not (if it can be reached the result includes a trace through the C code). This raw result needs to be interpreted with respect to the higher-level verification condition that is encoded by the label. In addition, the counterexample must be related to the program that includes the higher-level constructs. In Fig. 5 we illustrate examples for lifted results for checking contracts, protocol of components and the completeness of decision tables. Lifting the counterexample involves several abstraction steps:

1. *Eliminate the generation noise from the C code.* Part of the generated C code represents encodings of higher level concepts. For example, additional functions are generated that implement decision tables. In these cases, the corresponding sections of the counterexample are irrelevant in terms of the higher-level construct; they should not be visible in the lifted result.

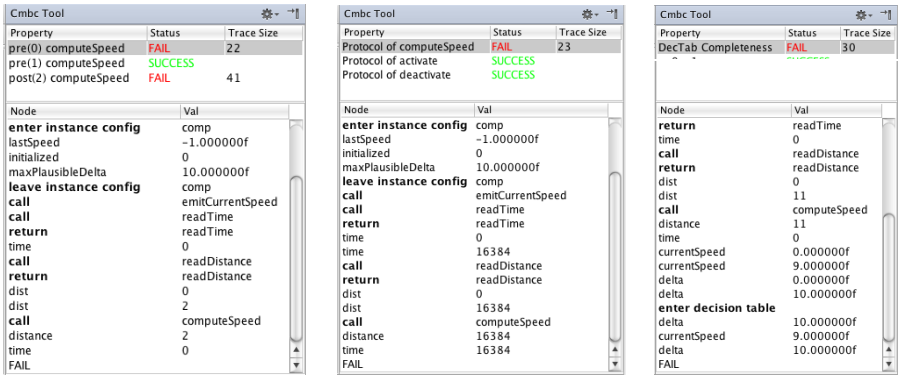

Fig. 5. Examples of lifted analyses results. In the case when an analysis fails, a lifted counterexample at the DSL-level is provided.

2. *Interpret the C-level counterexample.* Higher-level constructs are encoded in C through generation with the help of variables or function calls. These encodings need to be traced back. For example, the components are initialized in a function. If this function shows up in a C-level counterexample, it means that the components were initialized.
3. *Restore original names.* Since mbeddr supports namespaces, the names of the high-level program elements are mingled with module names in the C code. During lifting, we must recover the names of the higher-level abstractions.

Making users aware about the analyzability of their code. Due to the *model construction problem* [2], building robust verification tools for large and complex languages is challenging. In the case when the underlying verification tool does not support a language feature (intentionally, or because it has bugs), we inform the user about the non-analyzability of the code by showing a warning in the IDE. This way, unpleasant surprises are avoided and end user acceptance can be increased. For example, Fig. 6 (above the line) shows a program fragment that cannot be analyzed with CBMC 4.2 (there is a problem with function pointers that has since been fixed). The part below the line shows CBMC's error message if that code is used as input. Code like this is generated when the components in mbeddr are wired dynamically to support runtime polymorphism (via indirection through function pointers). mbeddr has a configuration option that forces static wiring of components (by using a language restriction), avoiding the use of function pointers in the generated C code. This makes the code analyzable, but it also limits the flexibility of the user. Making this tradeoff explicit allows users to make an informed decision regarding flexibility vs. analyzability.

```
1  struct PlauzibilizedSpeedComputer {     struct PlauzibilizedSpeedComputer anSC;
2    char (*activate)();                   void initializeComponents() {
3    int (*computeSpeed)(int, int);          anSC.activate = &activateImpl;
4  };                                        anSC.computeSpeed = &computeSpeedImpl;
5  char activateImpl() { return 0; }       }
6  int computeSpeedImpl(int d, int t) {    int main() {
7    return 0;                               initializeComponents();
8  }                                         int x = (*(snSC.computeSpeed))(2, 3);... }
9  ____________________________________________________________________________
10 Assertion failed (base_type_eq(assignment.lhs().type().assignment.rhs().type().ns)),
11   function return_assignment, file symex_function_call.cpp, line 483
```

Fig. 6. Example of a code fragment that is generated from mbeddr but is not supported by CBMC (top) and the error message provided by CBMC when this program is analyzed (bottom). We explicitly inform mbeddr users when they use a high-level construct that leads to a non-supported language fragment in the generated code. In this manner, we make the usage of analysis more predictible to the developers.

4 Related Work

In this paper we extend our previous work on using language workbenches to enable more user-friendly and high-level formal verification [4,6,7] by integrating a general purpose C-level model checker. There is significant related work on integrating C model checkers into development environments [9,10]. There is

also already work on generating verification properties from higher level models [11] and to trace the analyses results at a code level back at the model level [12].

Our work is different mainly in that instead of using models to generate verification properties, we use language extensions. This way we retain the benefits of generating verification conditions from higer-level abstractions: deriving the verification conditions is straight forward, and lifting the counterexample to the higher abstraction level eliminates a significant amount of noise and thereby improve usability. In addition, we avoid the semantic and tool integration issues that arise when (verifiable) parts of programs are expressed with different formalisms than the regular C code: the extensions have clearly defined semantics in terms of C, and the tool integration is seamless.

5 Conclusions and Future Work

We see domain-specific languages, language engineering and language workbenches as key enablers to increase the usability of formal verification. In the future, we will generate invariants from high level constructs and we will support to set different entry points in the analysis. A challenge that we foresee is that the semantics at DSL level ("big step") might miss many C-level errors ("small-step") and make the interpretation of the high-level counterexample unsound.

This work is developed in the LWES project, supported by the German BMBF, FKZ 01/S11014.

References

1. Loer, K., Harrison, M.: Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems. In: ASE. (2002)
2. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Păsăreanu, C., Zheng, H.: Bandera: extracting finite-state models from Java source code. In: ICSE'00. (2000)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS'04. (2004)
4. Voelter, M., Ratiu, D., Schätz, B., Kolb, B.: mbeddr: an extensible c-based programming language and ide for embedded systems. In: SPLASH'12. (2012)
5. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: 4th Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2011). LNCS. Springer (2011)
6. Ratiu, D., Voelter, M., Schaetz, B., Kolb, B.: Language Engineering as Enabler for Incrementally Defined Formal Analyses. In: FORMSERA'12. (2012)
7. Daniel Ratiu, Markus Voelter, Z.M., Schaetz, B.: Implementing modular domain specific languages and analyses. In: MoDEVVa'12. (2012)
8. Janicki, R., Parnas, D.L., Zucker, J.: Tabular representations in relational documents. In: Relational methods in computer science. (1997)
9. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: An eclipse plug-in for model checking. In: IWPC'04. (2004)
10. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In: IFM'04. (2004)
11. Zalila, F., Crégut, X., Pantel, M.: Leveraging formal verification tools for dsml users: a process modeling case study. In: ISoLA'12. (2012)
12. Combemale, B., Gonnord, L., Rusu, V.: A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In: ECMFA'11. (2011)