Research Projects, Ideas, and Open Problems

Maintained by Grigore Rosu (grigore.rosu@runtimeverification.com, grosu@illinois.edu)
(see additional contacts under each problem)

Here is a list of open problems and challenges that we are interested in solving, in no particular order.
The reason we decided to create and maintain this list is not only that it helps us understand these topics better by putting things at their place in the big picture of programming language semantics-driven reasoning and tooling, but also to tell you our story and hopefully motivate you to join our effort.
While we are doing our best to keep this list actual, it may well be the case that some of the problems have been solved in the meanwhile or that we have found a different way to approach them.
If you are interested in working on any of these problems, please contact us to make sure that the problem is still actual and nobody is already working on it.
There are more problems here than one person can finish in a lifetime.
If you choose to work on a problem and believe that we can help, please let us know and we may work together on it.
In same cases, we can also offer research grants to work on problems of immediate interest to our company.

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Table of Contents

- [Logic] 7. Notations in matching logic
- [Logic] 8. Matching logic in Coq
- [Logic] 9. Matching logic in Lean
- [PL, K] 10. Specifying medical guidelines in K
- [Logic, K] 11. Modeling hybrid systems in K
- [PL, K] 12. Intermediate verification languages (IVL) in K
- [PL] 13. ExeBench for RV-Match
- [Logic] 14. Separation logic in matching logic
- Solved

1. The K summarizer

[PL, K] Last update: 8/4/2022, 3:15:07 PM

The vision of the current K framework is that it is a unifying formal semantics framework for all programming languages. Given any programming language L, we can define its formal semantics in K, from which all the language tools (like a parser, an interpreter, a deductive verifier, a model checker, etc.) of L will be automatically generated. This way, we only need to implement language tools generically, once and for all, and then instantiate them by a formal programming language semantics.

The K Summarizer pushes the above idea even further. It allows us to extract from a given language semantics LL and a given program PP in that language, *a new semantics* L_PLP that is the K semantics of the program PP in language LL. In other words, L_PLP is how we would implement the same program PP directly in K if we didn't have LL. The key idea is to summarize statically all possible known (symbolic) behaviors of program PP and leave for runtime only what cannot be inferred statically. The result of that is a control-flow graph of the program PP, where all the basic blocks (i.e., straight-line executions/code) are pre-computed and decisions only need to be made at certain choice points such as conditional statements, while-loops, etc.

Compared to the original semantics of LL, the new semantics L_PLP is much simpler. Indeed, the entire C semantics can have thousands of semantic rules but the semantics of the following C program `sum.c` that computes the sum from 1 to the input `n`

```
while(n >= 0) {
  s += n;
  n--;
}
```

can be implemented using the following two rules:

```
    rule <k> while(n >= 0) { s += n; n-- } => .K </k>
         <state> s |-> S,
                 n |-> N
```

```
      </state>
  requires  notBool N >=Int 0
rule <k> while(n >= 0) { s += n; n-- } => while(n >= 0) { s += n; n-- } </k>
      <state> s |-> (S => S +Int N),
              n |-> (N => N -Int 1)
      </state>
  requires  N >=Int 0
```

If we only care about the `</state>` cell, we can simplify the configurations even further and have the entire semantics of `sum.c` implemented in one rule:

```
rule <n, sum> => <n -Int 1, sum +Int n> requires n >Int 0
```

At a high level, the new semantics L_PLP implements the control-flow graph of PP where each edge is an
all-path reachability rule that summarizes a basic block. When branching occurs,
we may split the state on the branch condition and continue execution from there.

This generalizes what compilers do, and also the classic "partial evaluation" approach, where a compiler is seen as a partial evaluation of an interpreter with the program (but not program input). The critical insight in our context is to do this at the level of a formal semantics, instead of a particular ad-hoc implementation of an interpreter.

The K summarize is currently under active development.
At this point it is an umbrella project for three different things (all important):

1. A web interface for the K prover and for K symbolic execution that allows visualizing K executions and manipulating them.
2. A new interactive approach to using the K prover, where we expose a command-line tool that brings us some light interactivity to the K prover.
3. The semantics-based compilation (SBC) project [link], which has the goal that we stated above to take a programming language LL, a program PP in that language, and produce a new programming language L_PLP for that one program.

For anyone who wants to learn more about the K summarize we recommend the 3-hour K summarizer workshop recording where we covered the history, the current status, the theoretical foundation, and the future plans about the project.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[video] The 3-hour K summarizer workshop recording

[related project] Semantic-based compilation

Contact

Everett Hildenbrandt [github]

Nishant Rodrigues (nishant2@illinois.edu)

Xiaohong Chen (xc3@illinois.edu)

2. Zero-knowledge proofs and the matching logic proof checker

[Logic] Last update: 8/2/2022, 3:08:01 PM

A zero-knowledge proof (abbreviated zk-proof) is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true while the prover avoids conveying any additional information apart from the fact that the statement is indeed true (cite: https://en.wikipedia.org/wiki/Zero-knowledge_proof).

Matching logic proof checker [link] is a small program written in Metamath that formally defines the syntax and proof system of matching logic. Formal proofs of matching logic can then be encoded as proof objects, which can be automatically checked by the checker. The following two papers

> **Towards a Trustworthy Semantics-Based Language Framework via Proof Generation**. Chen et al. CAV, 2021. [pdf]
>
> **Making Formal Verification Trustworthy via Proof Generation**. Lin et al. TechRep, 2022. [pdf]

have studied proof generation for K, where proof objects are generated for K as certificates that guarantee the correctness of the K-generated interpreters and deductive verifiers.

It has been realized with our experience with K that *every computation is a proof*. Program execution (concretely or symbolically), formal verification, model checking, and even parsing can and should be specified as rigorous and machine-checkable mathematical proofs. Language tools are then best-effort implementation of finding such proofs. It is then implied that there is no need to re-do a computation because the proof guarantees its correctness.

The idea of proof generation can be combined with that of zk-proofs to enable faithful and practical remote computation. The matching logic proof checker (abbreviated MLPC) is a program that takes as input a proof obligation (i.e., axioms imply a theorem) and a tentative proof of the obligation, and outputs

whether the given proof is correct, using a fully deterministic and simple program. For a third-party to verify the proof checking result, they can ask for the tentative proof given to MLPC and run the proof checking algorithm on it. In practice, the tentative proof can be huge (of millions or more lines of code) so re-doing the proof checking is inefficient.

Using zk-proofs techniques, the third-party can verify the proof checking result without re-doing the computation. Putting in zk-proofs terminology:

> One party (the ZK-prover, in our case it's MLPC) can prove to another party (the ZK-verifier, in our case the "third-party" above) that a given statement (P) is true while the ZL-prover avoids conveying any additional information (I) apart from the fact that the statement is indeed true.

where

- Statement (P): There exists a proof for the given obligation that passes the proof checking algorithm.
- Additional Information (I): The actual (millions-LOC) proof of the obligation.
  That is, the third-party ZK-verifier is convinced of the correctness of the proof obligation *without* needing to seeing to actual proof or re-running the proof checking algorithm.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[project] A Rust implementation of metamath [link]

[project] Cairo [link]

[paper] **Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications**. Chin et al. 2021. [link]

[project] Pepper [link]

Contact

Bolton Bailey (boltonb2@illinois.edu)

Xiaohong Chen (xc3@illinois.edu)

3. Semantics-based compilation

[PL, Logic, K] Last update: 8/3/2022, 8:21:37 PM

(*The following project description is based on an old article Open Problems and Challenges in 2017. See The K Summarizer for the latest development of the project.*)

An ideal language framework should allow us to generate compilers from language definitions. Specifically, we hope that soon the K framework will have the capability to take a language semantics and a program in that language, and generate an efficient binary for the program.
Conceptually, K already provides the ingredients needed for such a semantics-based compiler. One way to approach the problem is to use symbolic execution on all non-recursive non-iterative program fragments and calculate the mathematical summary of the fragment as a (potentially large) K rule; the left-hand side of that rule will contain the fragment. For example, consider the trivial IMP language and a while loop whose body contains no other while loops. Then we can conceptually replace the while loop body with a new statement, say `stmt17093`, whose semantics is given with a K rule that symbolically accumulates the semantics of the individual statements that composed `stmt17093`. We can do the same for all non-loop fragments of code, until we eventually obtain a program containing only while statements whose bodies are special statement constants like `stmt17093`, each with its own K semantics.

Once the above is achieved, there are two more important components left. One is to translate the while statements into jump statements in the backend language, say LLVM. In the long term, this should be inferred automatically from the K semantics of the while statement. In the short term, we can get the user involved by asking them to provide a translation for the while loop (yes, this is not nice, but hey, getting a compiler for your language is a big deal). The other component is to translate the K rules for statements like `stmt17093` into efficient target language code. We believe this is an orthogonal issue, which we want to have efficiently implemented in K anyway, as part of our effort on providing a fast execution engine.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[related project] K summarizer

Contact

Everett Hildenbrandt [github]

Nishant Rodrigues (nishant2@illinois.edu)

Xiaohong Chen (xc3@illinois.edu)

4. Completeness of Matching Logic

[Logic] Last update: 5/13/2023, 2:10:25 PM

The following paper

Matching mu-Logic. Chen and Rosu. LICS 2019. [pdf]

proposes a Hilbert-style proof system for matching logic (Fig. 1). The paper then shows two completeness results (Theorems 15 and 16) for the first-order (FO) fragment of matching logic that does not use set variables or fixpoint patterns, which we state as follows:

1. (Local Completeness) $\emptyset \vDash \phi$ implies $\emptyset \vdash \phi$ where $\emptyset$ is the empty theory (i.e., no axioms)
2. (Definedness Completeness) $\Gamma \vDash \phi\Gamma$ implies $\Gamma \vdash \phi\Gamma$ if $\Gamma\Gamma$ contains the axiom for the definedness symbol, from which equality and membership can be defined as derived constructs.

Note that the above completeness results are proved only for the FO fragment of matching logic, so no set variables or fixpoint patterns are allowed in $\Gamma\Gamma$ or in proofs.

On the other hand, it is known that matching logic in its full generality is incomplete. This is because matching logic has both quantifiers and fixpoint operators. Thus, we can define a theory $\Gamma_{Nat}\Gamma$Nat that captures precisely the standard model of natural numbers with addition and multiplication (This is proved in Proposition 23 in the LICS'2019 paper). By Godel's incompleteness theorem, such a theory $\Gamma_{Nat}\Gamma$Nat cannot have a complete deduction system.

However, it remains open whether the FO fragment of matching logic is complete or not. (Local Completeness) shows that it is complete for the empty theory. (Definedness Completeness) shows that it is complete for any theory that has one symbol $\lceil \_ \rceil \ \_$, called the definedness symbol, and the following one axiom:

(Definedness) $\forall x.\lceil x \rceil$ x. x

The remaining case is when $\Gamma\Gamma$ is not empty and doesn't have the definedness symbol/axiom.

The completeness problem can be asked for not only the FO fragment but other fragments, too. The following are three common fragments:

1. Modal fragment, which is the fragment that has only set variables, symbols, and propositional connectives.
2. FO fragment, which, as discussed above, has element variables, symbols, propositional connectives, and FO quantifiers ($\exists$ and $\forall$). Set variables or fixpoints are not allowed.
3. Fixpoint fragment, which has set variables, symbols, propositional connectives, and fixpoints (μ and ν). Element variables or FO quantifiers are not allowed.

For each fragment FF, we are interested in the local and global completeness properties:

1. (Local Completeness for Fragment FF) $\emptyset \vDash \phi$ implies $\emptyset \vdash \phi$ for all ϕ in the fragment.
2. (Global Completeness for Fragment FF) $\Gamma \vDash \phi\Gamma$ implies $\Gamma \vdash \phi\Gamma$ for all $\Gamma\Gamma$ and ϕ in the fragment.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[paper] **Matching mu-Logic**. Chen and Rosu. LICS 2019. [pdf]

[thesis] **[Deduction in matching logic]**. Adam Fiedler. 2022. [pdf]

[slides] **Deduction in matching logic - Master Thesis**. Adam Fiedler. 2022. [pdf]

Contact

Xiaohong Chen (xc3@illinois.edu)

Adam Fiedler (adam.fiedler@runtimeverification.com)

5. Henkin semantics of matching logic

[Logic] Last update: 8/2/2022, 3:09:31 PM

Matching logic in its full generality is not complete, which is not uncommon for a logic that can express both quantifiers and fixpoints. Without a full completeness theorem, we do not know whether the existing matching logic proof system is "good enough". Specifically, consider
$\Gamma\Gamma$ and ϕ such that $\Gamma \vDash \phi\Gamma$ but ***not*** $\Gamma \vdash \phi\Gamma$. Without a completeness theorem, there is no semantic argument as to why $\Gamma \vdash \phi\Gamma$ does not hold. In other words, we do not know whether it is because the proof system lacks some important proof rules, or there are some *intrinsic reasons* concerning the semantics of $\Gamma\Gamma$ and ϕ that will explain the lack of formal derivations.

In the literature, Henkin semantics (also called general semantics) are studied for second-order and higher-order logics which do not admit complete deduction with respect to their standard, "full" semantics. Usually, a Henkin semantics defines a new semantic entailment relation $\Gamma \vDash_{\text{Henkin}} \phi\Gamma$ Henkin that is *weaker* than the standard semantics $\Gamma \vDash \phi\Gamma$, by accepting more models of $\Gamma\Gamma$. For example, the second-order logic formula $\forall P.\Phi(P)$ P.Φ(P) holds in MM w.r.t. the standard semantics, if for all subsets PP of MM, $\Phi(P)\Phi$(P) holds. In

Henkin semantics, however, the formula holds if (intuitively) for all subsets PP *definable within the logic*, $\Phi(P)\Phi$(P) holds. A model that satisfies a formula w.r.t. the standard semantics is called a standard model. Henkin semantics allow non-standard models.

The challenge is to propose a Henkin semantics for matching logic and to prove that the matching logic proof system is complete w.r.t. the Henkin semantics:

(Henkin Completeness) $\Gamma \vDash_{\text{Henkin}} \phi\Gamma$ Henkin implies $\Gamma \vdash \phi\Gamma$ for all $\Gamma\Gamma$ and ϕ .

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[paper] **Matching mu-Logic**. Chen and Rosu. LICS 2019. [pdf]

Contact

Xiaohong Chen (xc3@illinois.edu)

6. Applicative matching logic and the conservative extension theorem

[Logic] Last update: 8/2/2022, 4:36:14 PM

Applicative matching logic is the simplest variant of matching logic that retains all of its expressive power. It is implemented by the matching logic proof checker [github].
Compared to the full matching logic proposed in the following paper

Matching mu-Logic. Chen and Rosu. LICS 2019. [pdf]

which has sorts and many-sorted symbols, applicative matching logic allows only one sort (which makes it an unsorted logic), one binary symbol called application `app(ph1, ph2)` or written in juxtaposition: `ph1 ph2`, and any user-provided constant symbols from the signature. Both multiary symbols and sorts are definable in applicative matching logic using axioms. For example, the binary function `plus` can be expressed by `((plus ph1) ph2)`, where `plus` is a constant that is first applied to `ph1` and then to `ph2`. Sorts can be defined using a special symbol `inh` called the inhabitant symbol. We use patterns such as `(inh Nat)`, `(inh Int)` to represent the inhabitant sets of the sorts `Nat`, `Int`, etc., where `Nat` and `Int` are constant symbols for the sort names.

Therefore, the full many-sorted matching logic can be defined axiomatically in applicative matching logic. For any many-sorted matching logic theory

ΓΓ and pattern φ , we have an encoding into applicative matching logic: AML(Γ)AML(Γ) and AML(φ)AML( ).

While many-sorted matching logic is useful to specify mathematical models and/or the formal semantics of programming languages, applicative matching logic is better for proof checking for its simplicity. An important open question is to prove that the encoding AMLAML is a conservative extension:

(Conservative Extension) $\Gamma \vdash \varphi\Gamma$ iff AML(Γ) $\vdash$ AML(φ)AML(Γ) AML( ).

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Contact

Xiaohong Chen (xc3@illinois.edu)

7. Notations in matching logic

[Logic] Last update: 8/2/2022, 3:09:59 PM

The syntax of matching logic defines only the following primitive constructs:

φ ::= $x \mid X \mid \sigma(\varphi_1, \ldots, \varphi_n) \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\varphi \mid \mu X.\varphi$ ::= x X (1,…, n) 1 → 2 x. X.

Many useful constructs are all defined as notations. For example, negation $\neg\varphi \equiv \varphi \to \bot$¬ → is defined using $\bot$ and →→. Disjunction $\varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2$ 1 2 ¬ 1→ 2 and conjunction $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$ 1 2 ¬(¬ 1 ¬ 2) can also be defined in the usual way as in propositional logic. Some notations involve binders. For example, $\forall x.\varphi \equiv \neg\exists x.\neg\varphi$ x. ¬ x.¬ creates a binding of xx in φ .

Substitution is also a notation. Indeed, $\varphi[\psi/x]$ [/x]---which means to substitute ψ for all the free occurrences xx in φ ---is a piece of syntax that exists *at the meta-level*. The greatest fixpoint pattern $\nu X.\varphi \equiv \neg\mu X.\neg\varphi[\neg X/X]$ X. ¬ X.¬ [¬X/X] is a notation definition that uses substitution.

At a high level, the K framework is a best-effort implementation of a framework for specifying matching logic theories and proving matching logic theorems. The front-end syntax used by K should therefore all be definable as notations, which came from our >20-year experience with defining the formal semantics of programming languages in K and are optimized for that purpose. The current

front-end tool `kompile` has built-in algorithms to handle these PL-specific notations, by de-sugaring them in the parsing phase into more primitive matching logic constructs.

The research question is: how to design a proof framework for matching logic that allows users to define *any notations* they like? Such a proof framework will allow us to define all the notations that we use while doing matching logic proofs on paper and eventually, the entire K, in a logically sound and rigorous way.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Contact

Xiaohong Chen (xc3@illinois.edu)

8. Matching logic in Coq

[Logic] Last update: 8/2/2022, 3:10:24 PM

The simplicity of matching logic makes it possible that it can be defined in other logical systems. For example,
Coq is an interactive theorem prover (a.k.a. a proof assistant) that is based on CIC---the calculus of inductive constructions. By defining matching logic in Coq, we can use Coq as a backend for carrying out interactive theorem proving in matching logic.

Dániel Horpácsi and his team from Eötvös Loránd University has started a project where the syntax, semantics (models), and proof system of matching logic are formalized as Coq definitions. The soundness of the matching logic proof system has been formalized and proved in Coq. The latest progress has been summarized in this preprint.

In the following we list a number of proposed work wrt the Coq formalization of matching logic.

A Coq proof mode for matching logic

Coq comes with a language for creating mathematical theories, as well as a dedicated proof tactic language for writing proofs for logical propositions. The tactic language can be used to describe proofs of propositions of form $H \vdash_{\text{Coq}} G$H CoqG by allowing the programmer to manipulate hypotheses HH and the goal GG in an intuitive way. Internally, the proofs are represented as terms in CIC.

When embedding matching logic in Coq's metalogic (i.e. CIC), we formalize/embed another type of consequence, namely $\vdash_{\mathrm{ML}}$ ML, which has its own set of axioms and proof rules. When reasoning about the embedded logic, we prove Coq formulas of form $H \vdash_{\mathrm{Coq}} (\Gamma \vdash_{\mathrm{ML}} \varphi)$H Coq(Γ ML). In examples like this Coq's tactic language is inconvenient to use: it is much more effective to use a separate tactic language for matching logic. A similar problem has been solved in the first-order logic proof mode and in Iris proof mode for separation logic. We plan to apply similar techniques to design a Coq proof mode for matching logic, e.g. by rendering the context of matching logic in the same way as the context of the metalogic of Coq (providing a better overview of the hypotheses) and providing dedicated matching logic tactics.

Supporting K

Matching logic is the logical foundation of K. It means that every K formal semantics will be translated into a matching logic theory, which is formalized in Coq by specifying the matching logic signature with the symbols of the target programming language and composing the set of axioms that describe the formal semantic rules. Currently, K is able to automatically generate these matching logic theories from its frontend tool `kompile`. It is therefore promising to start integrating the Coq formalization with K by using Coq to interactively reason about proof obligations that show up in formal verification in K.

Generating matching logic proof objects from Coq proofs

Proofs in Coq are CIC terms of the type associated with the specification formula. Once Coq's type checker accepts the term, it can be seen as a proof object proving the specification. However, Coq's trust base is rather extensive, consisting of more than ten thousand lines of code, and the matching logic proof system formalized in Coq is also a trusted component. The correctness of the matching logic proof depends on the correctness of this large trust base, so it would be desirable to reduce the trust base as much as possible.

The trust base can indeed be reduced by simply expressing the theory (ΓΓ), the formula (φ), and the proof (dd) in the existing matching logic proof checker. The trust base in this case becomes the code of the translation plus the trust base of the new checker. We can translate the proof terms CIC to matching logic proof objects and check them in this more trustworthy system, which brings substantially more confidence in the correctness of the proofs.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[project] AML Formalization [link]

[paper] **Mechanizing Matching Logic In Coq**. Bereczky et al. preprint 2022. [pdf]

Contact

Dániel Horpácsi (daniel-h@elte.hu)

Jan Tušil (jan.tusil@mail.muni.cz)

9. Matching logic in Lean

[Logic] Last update: 8/2/2022, 4:36:57 PM

Lean is a functional programming language that can also be used as an interactive theorem prover. The Lean project was launched by Leonardo de Moura and collaborators at Microsoft Research in 2013.

The goal of this project is to enhance the matching logic ecosystem by adding Lean support. The following objectives are proposed.

Implementation of matching logic in Lean

A formalization of matching logic in the Coq proof assistant was recently developed in the following paper

Mechanizing Matching Logic In Coq. Bereczky et al. Preprint 2022. [pdf]

Our first objective is to formalize matching logic in Lean.
This will be similar in principle to and will be inspired from the Coq formalization, but it will be optimized and specialized to Lean, taking advantage of its strengths. We shall

- formalize the syntax, semantics, and proof system of ML;
- verify the soundness of the formalized proof system;
- formalize proofs of ML theorems and derived deduction rules, the proof of the deduction theorem, as well as other proof-theoretical tools;
- give examples of theories with proofs about their properties.

Generating proof objects

A very small proof checker for matching logic was formalized in

Towards a trustworthy semantics-based language framework via proof generation. Chen et al. CAV 2021. [pdf]

Matching logic proof checker. [github]

using Metamath, a computer language for mathematical proofs that is simple, fast, and trustworthy.
We shall combine the Lean and Metamath formalizations of matching logic and translate matching logic proofs in Lean to proof objects in Metamath. In this way,

- we shall increase the confidence in the correctness of the ML proofs;
- proof checking will be much more efficient, as it will be done by a very small proof checker.

Ultimately, this will offer the K ecosystem a trusted interactive formal verification environment, where Lean, K, or any other complex system, are eliminated from the trust base.

This project is a research collaboration between RV and the Institute for Logic and Data Science
that has started at the middle of July 2022.

Contact

Laurențiu Leuştean (laurentiu.leustean@ilds.ro)

10. Specifying medical guidelines in K

[PL, K] Last update: 8/5/2022, 1:21:45 AM

MediK is a K-based domain-specific language (DSL)
for expressing medical guidelines as *concurrently executing finite state machines* (FSMs). Hospitals and medical associations often publish flowchart based Clinical best-Practice Guidelines to codify standard treatment for conditions such as Cardiac Arrest. In MediK, we systematically expresses these guidelines, along with other relevant data, such as the state of the patient's organ systems, and instructions to/from Healthcare Professionals, as FSMs that interact via passing events. Since MediK guidelines are executable, they can be used in a hospital setting as correct-by-construction guidance systems that assist Healthcare Professionals. For example, consider this simple model of a patient's cardiovascular system, and some treatment:

```
machine CardiovascularSystem {
  init state CardiovascularLoop {
    // Code to execute on entry
    entry(heartRateSensor) {
      // Read sensor data
      if (heartRateSensor.rate < 10) {
        // declare an emergency
        broadcast CardioEmergency;
      }
      sleep(10);
      goto CardiovascularLoop;
    }
}
machine Treatment receives CardioEmergency {
  ...
  on CardioEmergency do {
```

```
    // handle emergency
  }
}
interface HeartRateSensor {
  var rate;
}
```

In the example above, the `CardiovascularSystem` FSM starts in the state `CardiovascularLoop`, and uses a reference to a `HeartRateSensor` FSM to obtain the patient's heart rate. If the obtained heart rate is too low, the `CardiovascularSystem` FSM broadcasts a `CardioEmergency` event. Note that the heart rate sensor itself is not implemented in MediK. In medical guidance systems, it is common to interact with various external agents such as sensors and databases. We handle such actors by treating them as FSMs with transition systems external to MediK. In our example, the HeartRateSensor is declared as `interface` instead of a `machine`, signifying that the transition system is implemented elsewhere. This allows us to use the existing event passing mechanism to interact with external components.

While K suppports I/O through "stdin" and "stdout" attributes on cells, the I/O mechanism itself is synchronous, as reading from or writing to an I/O cell is blocking, i.e., no other rule can apply until the read or write is complete. Synchronous I/O can be a limitation for languages like MediK, where rules responsible for execution must continue to apply while waiting for I/O to be completed. This necessitates the need for a generic asynchronous I/O mechanism in K that allows rules for I/O to not block the application of other rules.

Resources

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

[project] MediK [link]

Contact

Manasvi Saxena [github] (msaxena2@illinois.edu)

11. Modeling hybrid systems in K

[Logic, K] Last update: 8/2/2022, 3:10:56 PM

Hybrid systems are dynamical systems that exhibit both discrete and continuous dynamic behavior. Such systems can both flow (described using differential equations) and jump (often described using discrete automata). Cyber-Physical Systems (CPS), or systems that combine cyber capabilities

(computation, communication and control) with physical capabilities are hybrid systems are examples of such systems. In recent years, CPS are increasingly being employed to tackle challenges in domains like medicine, transportation and energy. The use of CPS in safety-critical applications, however, makes their correctness and reliability extremely important.

Research Direction: Modeling Hybrid Automata in Matching Logic

The framework of Hybrid Automata (HA) is widely used to model CPS. The semantics of HA are described in this seminal paper by Henzinger.
The research question is: can we capture the semantics of Hybrid Automata in Matching Logic?
Such an embedding would be useful, as it would allow us to use the Matching Logic theorem proving infrastructure to reason about CPS.
This technical report describes both the problem and some preliminary work we've done towards addressing it.

Research Direction: Embedding Differential Dynamic Logic in Matching Logic

Differential Dynamic Logic is a variant of First Order Dynamic Logic over the domain of Reals that supports specifying and reasoning about hybrid systems. The research question is: can we define an embedding of Differential Dynamic Logic into Matching Logic? dL has specific proof rules for handling differential equations and invariants used to specify continuous components of hybrid systems. Is the proof system of matching logic sufficient to handle such proof rules?

Resources

[paper] **Formal semantics of hybrid automata** TechRep, 2020. [pdf]

Contact

Manasvi Saxena (msaxena2@illinois.edu)

12. Intermediate verification languages (IVL) in K

[PL, K] Last update: 8/18/2022, 1:28:52 AM

Many programming languages provide verification tools via translation to the Boogie [link] intermediate verification language (IVL).
Boogie provides powerful and efficient tools that allow the development of high-quality, easy-to-use annotation-based verifiers, and allows for modular verification of large software.

However, the translation from the target programming language to Boogie is essentially another compiler for the language, and,
besides the additional manpower needed for developing this tranlation, it also introduces scope for divergence between the actual language behaviour, severely reducing the confidence in their correctness.

We believe that a solution to this extra complexity introduced by the translation is to use
a semantics-based approach to language development. More specifically, we define the formal semantics of Boogie in K and let K
fulfil many of the use cases of Boogie within this semantic-based approach. To achieve that, we need to define
the following Boogie-powered features in K:

- loop invariants and function contracts specified in the programming language's concrete syntax,
- quantification over expressions,
- ghost variables,
- clear and easy-to-read error messages.

The goal of the project is a formal semantics of Boogie in K that supports all the above features.

Resources

[project] Boogie semantics in K [github]

Contact

Nishant Rodrigues (nishantjr@gmail.com)

13. ExeBench for RV-Match

[PL] Last update: 8/29/2022, 3:53:52 PM

Summary

ExeBench is the largest known database of compilable and executable C functions. By using a type-inferencer, the authors are able to "fill in" missing dependencies and make individual functions compile in isolation. Additionally, the dataset contains input-output pairs that specify how each program should behave (i.e. a set of unit tests).

Ideas

- Use the UB checker in RV-Match to try and quantify how frequent it is (by category etc.) in "real" code.

- Their idea is to use ExeBench as a training set for ML applications; cleaning out UB from that dataset (and vice-versa) would be an interesting task.
- Construction of a large correctness test suite for `kcc`.

Contacts

@Baltoli is in contact with Jordi Armengol.

Dataset

The dataset is hosted on HuggingFace.

Paper

> Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. 2022. ExeBench: an ML-scale dataset of executable C functions. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022). Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3520312.3534867

14. Separation logic in matching logic

[Logic] Last update: 3/5/2023, 2:24:38 AM

Separation logic (SL) is a logic specifically crafted for reasoning about heap structures. In the following paper

Matching Logic. Rosu. LMCS 2017. [pdf]

it is shown that SL is an instance of matching logic, if we fix the underlying model to be MapMap, the model of finite maps. All heap assertions in SL are patterns, and there is a nice correspondence between the semantics of two: a heap hh satisfies an SL assertion/formula φ if and only if hh matches φ as a pattern.

SL can be extended with recursive predicates. For example, $list(x) =_{\text{lfp}} emp \wedge x = 0 \vee \exists y.x \mapsto y * list(y)$ list(x)= emp x=0 y.x y∗list(y) defines a recursive predicate $list$list for singly-linked lists as a least fixpoint. SL with recursive predicates can also be defined in matching logic, as shown in the following paper

Towards A Unified Proof Framework for Automated Fixpoint Reasoning Using Matching Logic. Chen et al. OOPSLA 2020. [pdf]

In addition, the paper also proposes a set of high-level automated proof strategies for reasoning about matching logic fixpoints. When instantiated by the

model MapMap, these proof strategies can prove many challenging SL properties.

We can identify two important future research directions. One is inheriting automation from SL. And here we can think of two aspects:

1. reimplement the state-of-the-art automated reasoning techniques from SL in matching logic and K, as a decision procedure for theories that include the right signature and axioms (isomorphic to those of SL)
2. generalize the SL automation to a larger fragment of ML, so we can use it in more instances.

The other future direction is considering Hoare Logic + SL, which is the variant of Hoare Logic that uses SL as its base/underlying logic. We want to show that all the program proof rules of Hoare + SL are derivable in matching logic, just like how the vanilla/original Hoare Logic proof rules are derivable in matching logic via reachability logic; see Section IX of the following paper and the references there.

Matching mu-Logic. Chen and Rosu. LICS 2019. [pdf]