Concurrency Oracles for Free

Georgy Lukyanov and Andrey Mokhov

School of Engineering, Newcastle University, United Kingdom

Abstract. This paper presents an approach to deriving *concurrency oracles* for processor instructions whose behaviour is formally specified using a state transformer semantics. The presented approach does not require any modification of the existing semantics, nor does it rely on writing a parser for the language in which the semantics is described, thus justifying the "for free" part of the title.
The main tool in our arsenal is *ad-hoc polymorphism*: the presented approach is only applicable when the semantics of processor instructions is expressed using state transformation functions that can be reinterpreted in different contexts. As we show in the paper, such semantics can be interpreted not only for instruction simulation or verification, but also for extracting the information about instruction dependencies, thus allowing us to identify concurrency as well as various types of conflicts between instructions or blocks of instructions.

Keywords: concurrency oracle · instruction set architecture · functional programming · polymorphism

1 Introduction and motivation

Deciding whether two given events in a trace are *concurrent*, i.e. have no causal or data dependencies between them, is a major problem in the process discovery field [3]. Various methods for concurrency extraction, often referred to as *concurrency oracles*, have been introduced, including the classic α-algorithm [4], as well as a few less widely known methods, e.g. see [7], [15] and the review paper [5]. A good example of treating a concurrency oracle as a self-contained problem can be found in [8].

In this paper we present an approach for deriving concurrency oracles for events that correspond to processor instructions and blocks of instructions. The input to the proposed approach is the *microarchitectural semantics* of instructions, which gives a precise description of how an instruction execution changes the state of the processor. We show how the presented approach can be applied for program analysis and for synthesis of efficient hardware microcontrollers.

A popular method to describe microarchitectural semantics is to use a dedicated domain-specific language embedded in a high-level general-purpose host language, such as Haskell or Coq. Two pioneering works in this domain are [9], where the Arm v7 instruction set architecture is formalised in HOL4, and [10], where x86 architecture is formalised in Coq.

The authors of this paper have also used an embedded domain-specific language to describe the semantics of a space-grade microarchitecture [16]. In particular, it was demonstrated that the same semantics can be reused in different contexts: to simulate the processor and to perform formal verification of programs executed by the processor. In this paper we take this work further, by demonstrating that the very same semantics can be reused for deriving concurrency oracles that given two instructions, or blocks of instructions, can determine whether they are concurrent and, if not, report the data dependency conflicts.

We start by studying several common examples of processor instructions, noticing that different instructions require different features from the language used to describe them; see §2. We proceed by introducing the language for specifying semantics in more detail and then describe the semantics of a small instruction set (§3). The approach to deriving concurrency oracles is presented in §4. Precise analysis of data dependencies between program instructions allows us to synthesise efficient hardware controllers for executing predefined collections of programs, as demonstrated in §5, followed by a discussion.

2 Instruction set architecture semantics

In this section we introduce a metalanguage for describing the semantics of processor instructions by following a few examples. We will use the metalanguage to describe the semantics of a part of a simple generic instruction set architecture. The later sections will introduce a formal definition of the metalanguage, instruction semantics and present a method for extracting static data dependencies of instructions with certain properties from the semantic definitions leading to a construction of a concurrency oracle. Fig. 1 shows three example dependency graphs that can be automatically produced by the presented method.

Load Loading a data word from memory to a register is one of the simplest processor instructions. Here is how[1] we encode its semantics in our metalanguage:

```
load reg addr = \read write -> Just $
    write (Reg reg) (read (Addr addr))
```

In this definition we use two metalanguage terms `read` and `write` to specify the behaviour of the instruction, specifically to *read* the memory location at *address* `addr`, and *write* the result into the *register* `reg`. Crucially, `read` and `write` are *polymorphic over the computational context* `f` and have the following types:

```
read  :: forall f. Key -> f Value
write :: forall f. Key -> f Value -> f ()
```

Here `Key` is used to identify a particular component of the processor state, such as a memory location or a register, and `Value` is the type of data the processor operates on, e.g. `Value` for a 64-bit processor.

[1] We use Haskell throughout the paper, not only because it is one of the most popular functional programming languages, but also because it provides support for *higher-rank polymorphism* [17], which is essential for the presented approach.

The read term queries the microarchitectural state for the value of a key and returns it wrapped in a context f, which in this case captures the effect of reading from the state. The write term takes a key and a value, and modifies the microarchitectural state, returning no information (the *unit* ()), but capturing the effect of writing in the context f.

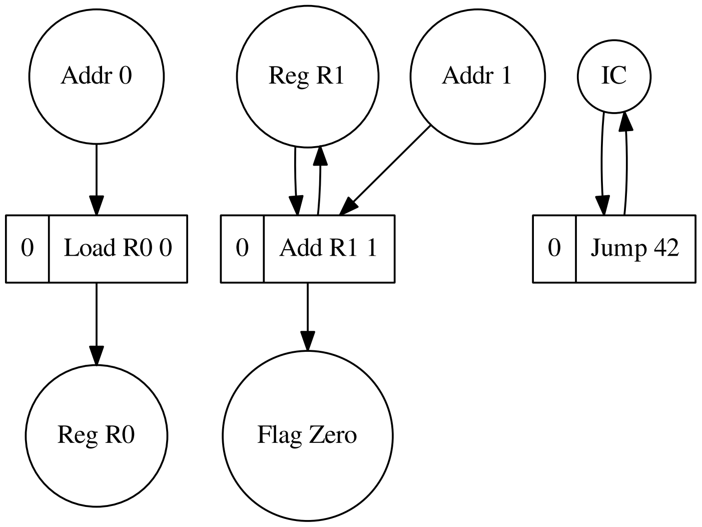

Fig. 1. Static data dependencies of `Load R0 0`, `Add R1 1` and `Jump 42` instructions[3]. The '0' inside the instruction boxes corresponds to the address of an instruction in the program and will be useful for visualising blocks of instructions.

Jump Another simple instruction is the unconditional control flow transfer:

```
jump offset = \read write -> Just $
    write IC ((+ offset) <$> (read IC))
```

This instruction adds an offset to the *instruction counter* IC to transfer the control to another instruction in the program. This definition has a crucial difference from load: it uses the function <$> of the Functor type class[4], thus restricting f of the read and write terms to be a Functor:

```
read  :: forall f. Functor f => Key -> f Value
write :: forall f. Functor f => Key -> f Value -> f ()
```

[3] We used the algebraic-graphs Haskell library [14] to export dependency graphs into the DOT format, and then applied GraphViz [18] for automated layout.

[4] Function <$> (pronounced "fmap") of type `Functor f => (a -> b) -> f a -> f b` transforms the values in a computational context f using a pure function.

In this definition the functorial constraint is required to apply the addition function `(+ offset) :: Value -> Value` to the instruction counter value enclosed in a computational context `f`. We therefore say that `jump` has *functorial semantics*, whereas `load` has *unconstrained semantics*, since the `read` and `write` terms had no constraint.

Later in the paper we will instantiate `f` with an appropriate context for data dependency tracking. As it turns out, instructions that have unconstrained or functorial semantics can have at most one read dependency: there is no way to combine multiple read results. To do that, we need *applicative semantics*, as demonstrated by the next example.

Add The `add` instruction performs the addition of the values of a register and a memory cell, writing the result back into the same register. If the result of the addition is equal to zero, the instruction sets the `Zero` flag of the processor to `True`, and to `False` otherwise.

The semantics definition is a bit more involved than the previous ones, because the `Functor` context is not expressive enough and a more powerful abstraction is needed. The following definition of `add` requires `f` to be at least an `Applicative`:

```
add reg addr = \read write -> Just $
    let result = (+)     <$> read (Reg reg) <*> read (Addr addr)
        isZero = (== 0) <$> result
    in  write (Reg reg) result *>
        write (Flag Zero) (boolToValue <$> isZero)
```

Let us elaborate on what is going on here. The definition may be broken down into three parts: reading data, processing it, and writing data back in the processor state.

The first `let`-binding uses `Applicative` notation to read the values from the register `reg` and memory address `addr` and add them up. Note that this notation is *declarative*, hence it rather states that the `result` is supposed to be a sum of values of two entities than performs actual computation. This intuition is very important for understanding the static dependency tracking of instructions: keys `Reg reg` and `Addr addr` are declared as static input dependencies of the `add` instruction. However, since the semantics may be executed in any `Applicative` context, this dependency-tracking interpretation does not prevent other possible interpretations of the very same definition of the semantics. For instance, in a simulation context, the `result` will be computed based on concrete data values read from the current processor state.

The second line of the `let`-binding is quite similar to the expression in the semantics of the `jump` instruction. The type of the `result` is `f Value`, hence the zero testing function `(== 0)` of type `Value -> Bool` must be mapped over the context `f` with the operator `<$>` to obtain the value of type `f Bool`.

The last two lines of the definition perform two `write` operations chained with the applicative operator `*>` of type `Applicative f => f a -> f b -> f b`.

This declares the keys Reg reg and Flag Zero to be output dependencies of the computation and that the writes must be both performed. The the read and write terms now have the following types:

```
read  :: forall f. Applicative f => Key -> f Value
write :: forall f. Applicative f => Key -> f Value -> f ()
```

An interesting feature of the Applicative notation is that it does not specify the exact order of performing actions. This is useful in embedded domain-specific languages with concurrency, for instance Facebook's Haxl [11]. This insight can also be used to extract concurrency from instruction descriptions.

Applicative functors are powerful enough to express the semantics of a large class of instructions. In this paper we exploit their features to not only specify the execution semantics but also automatically track static data dependencies of instructions. However not every instruction can be expressed with applicative semantics. If the behaviour depends on the actual data values, i.e. when *dynamic data dependencies* emerge, a more powerful *monadic semantics* is required.

Indirect load The indirect memory access instruction looks up a value in a memory cell and uses it as the effective address in the regular load instruction. Since the effective address can not be determined statically in the general case, this instruction has a dynamic data dependency. The polymorphic computational metalanguage requires the context f to be a Monad in order to be able to encode such behaviour. Consider the definition of the semantics of the loadMI instruction, which uses Haskell's monadic do-notation:

```
loadMI reg addr read write = Just $ do
    addr' <- read (Addr addr)
    write (Reg reg) (read (Addr addr'))
```

The first line extracts the effective address from the monadic context f and binds the identifier addr' to it. Here is the catch: expressions on left-hand-side and right-hand-side of the <- symbol have different types. The read (Addr addr) is of type Monad f => f Value and the identifier addr' has type Value. The main feature of Monad is the ability to extract a value from an effectful context and pass it in the further computation as if it was pure. This gives us a possibility to pass the addr' as an argument to the next read operation.

Monadic semantics is more powerful than unconstrained, functorial and applicative ones, but we are no longer able to extract all the dependencies of the computation if f is restricted to Monad, since some of them will not be static. Therefore, concurrency oracles can not be built for Monad-flavoured computations, or at least, they can no longer be exact and must be approximate (for example, one might conservatively say that loadMI has a read dependency on every possible memory address, but no register read dependencies).

We have given examples of four types of semantic computations: unrestricted, functorial, applicative and monadic. In every definition we used functions read and write with appropriate constraints on the context f. To recap, here are the four different types for the read function:

```
read :: forall f.                   Key -> f Value
read :: forall f.     Functor f => Key -> f Value
read :: forall f. Applicative f => Key -> f Value
read :: forall f.       Monad f => Key -> f Value
```

One can clearly see a pattern, and Haskell's type system is powerful enough to abstract over it. Generic read and write may be assigned the following types:

```
read  :: forall f. c f => Key -> f Value
write :: forall f. c f => Key -> f Value -> f ()
```

Here, the variable c must have the kind * -> Constraint. This allows to instantiate c with NoConstraint, Functor, Applicative, Monad or any other suitable constraint, thus making the metalanguage polymorphic in the computational context.

The next section will present a formal definition of the metalanguage, instruction and program semantics. The section 4 will describe the construction of concurrency oracles for programs comprising unrestricted, functorial, applicative, but not monadic instructions.

3 Polymorphic computational metalanguage

In the previous section we have described the semantics of several instructions of a generic computer architecture in terms of a polymorphic computational metalanguage. This section presents the formal definition of the metalanguage and provides a more formal description of the instruction and program semantics.

A remark on formal definitions Before we start, let us make a remark on what we consider a formal definition. We do not aim to formalise our tools in any kind of foundational mathematical system, such as ZF[5] or homotopy type theory. We are presenting an elegant way of solving a well-known problem and we use the Haskell programming language to implement the solution. Therefore, we consider a concept to be *formally* defined if it is expressed as a Haskell data type. This may sound hand-wavy, but since Haskell has a static type system (a variant of System F [19]) and operational semantics, we can be formal enough.

Definition (polymorphic computational metalanguage): A term of the metalanguage is a value of the following Haskell type:

```
type Semantics c a =
    forall f. c f => (Key -> f Value)
                  -> (Key -> f Value -> f ())
                  -> Maybe (f a)
```

[5] Zermelo-Fraenkel set theory.

A `Semantics` is essentially a rank-2 polymorphic[6] effectful computation depending on two functions, which we will usually refer to as `read` and `write`.

Let us now give some intuition for the components of the metalanguage. The `Semantics c a` type may be thought as a mutable dictionary. The `read` function has type `Key -> f Value` — it takes a key and gives back an effectful value looked up in the dictionary. The `write` function takes a key and an effectful value and alters the value of the key in the dictionary. The semantics can be partial, hence the the return type `f a` is wrapped in the `Maybe` type constructor. `Maybe`[7] is an idiomatic Haskell encoding of partial definitions. The semantics may become partial if we, for example, fix the constraint type variable `c` to `Applicative`, thus losing the possibility to encode the monadic components of the instruction set. `Maybe` allows us to treat such partially-defined semantics in a safe and formal way.

Definition (Instruction Set): An *instruction set* is an algebraic data type with as many data constructors as there are instructions. If an instruction has an argument, it is defined as an argument of the corresponding data constructor.

Consider an example definition of an instruction set consisting of instructions described in the previous sections and the related auxiliary types:

```
data Instruction = Load   Register MemoryAddress
                 | LoadMI Register MemoryAddress
                 | Add    Register MemoryAddress
                 | Jump   Value

data Register = R0 | R1 | R2 | R3

type MemoryAddress = Value
```

Definition (Instruction Set Semantics): The *semantics of an instruction set* is a Haskell function mapping data constructors of the instruction set to the terms of the polymorphic computational metalanguage.

The definition of an instruction set semantics is the point where the metalanguage has to be made monomorphic, i.e. the context constraint has to be instantiated with a concrete one. Below we present unrestricted, functorial, applicative and monadic semantics for the defined instruction set.

We start from the `Load` instruction which may be executed in any context[8]:

```
semanticsU :: Instruction -> Semantics Unrestricted ()
semanticsU (Load reg addr) = load reg addr
semanticsU _               = const (const Nothing)
```

[6] A rank-2 polymorphic function is one taking as a parameter another function, which is in turn (rank-1) polymorphic. This feature requires the `RankNTypes` language extension of the Glasgow Haskell Compiler.

[7] Defined in the Haskell's `base` library as `data Maybe a = Just a | Nothing`.

[8] The `Unrestricted` constraint is not exactly idiomatic Haskell and requires some tricks to be defined.

Note that the Haskell wildcard pattern '`_`' is used to match all instructions that require a more restrictive context. The `const (const Nothing)` expression is equivalent to `\read write -> Nothing` and constructs a stub for these more restricted semantics. The function `load` has been defined in §2.

The instantiation of `c` with a `Functor` allows us to implement the semantics of the instruction `Jump`:

```
semanticsF :: Instruction -> Semantics Functor ()
semanticsF (Jump simm) = jump simm
semanticsF i           = semanticU i
```

Here we use the definition of `jump` from §2 and fall back to unrestricted semantics definition `semanticU` for the `Load` instruction, hence avoiding code duplication.

The remaining definitions are analogous:

```
semanticsA :: Instruction -> Semantics Applicative ()
semanticsA (Add reg addr) = add reg addr
semanticsA i              = semanticsF i

semanticsM :: Instruction -> Semantics Monad ()
semanticsM (LoadMI reg addr) = loadMI reg addr
semanticsM i                 = semanticsA i
```

We can now define the semantics of a block of instructions by reducing a given list of instructions:

```
blockSemanticsA :: [Instruction] -> Semantics Applicative ()
blockSemanticsA xs = \r w ->
    foldr (\x acc -> (*>) <$> acc <*> semanticsA x r w) nop xs
    where nop = Just $ pure ()
```

The semantics of an empty block is `nop` (i.e. a *no-op* instruction). The semantics of a non-empty list is the semantics of its head chained with the semantics of the tail. We need to lift the applicative chaining operation since the `Maybe` type constructor also is an instance of `Applicative` and the behaviour of `*>` returns the contents of the last `Just`, which is would be wrong.

Now, with the instruction semantics defined in terms of the polymorphic computational metalanguage, we may proceed to evaluating the metalanguage in concrete contexts to get a practical interpretation of the instruction set.

The next section presents the interpretation of unrestricted, functorial and applicative instructions yielding concurrency oracles for programs.

Other interpretations of the metalanguage are also possible. In the technical report [16] we present a formal model of a processor developed for space missions, where we use a more restricted, monadic metalanguage, making emphasis on symbolic program execution and automated theorem proving: the framework allows to verify functional properties of programs and automatically check if two programs are semantically equivalent. The metalanguage presented in this paper also allows these interpretations, but the focus of this paper is different: automated derivation of concurrency oracles.

4 Concurrency oracles

In this section we present a method to derive concurrency oracles from the instruction semantics encoded in the metalanguage (Definition 3). More specifically, we aim to reason about instructions that have only *static dependencies*.

We start by introducing formal definitions and Haskell encodings of the concepts required for building concurrency oracles. First, we define the notions of input and output dependencies of a computation.

Definition (input dependency): Consider a term f of an applicative metalanguage `Semantics Applicative a`. A key k is an *input dependency* of the term f if the term f performs a `read` of the key k.

Definition (output dependency): Consider a term f of an applicative metalanguage `Semantics Applicative a`. A key k is an *output dependency* of the term f if the term f performs a `write` of the key k.

Definition (dependencies): Consider a term f of an applicative metalanguage `Semantics Applicative a`. The *dependencies* of the term f are sets I and O – the input and output dependencies of the term f, respectively.

In the Haskell implementation, we do not distinguish between input and output dependencies in the type level, thus the function determining the dependencies of a computation has the following type:

```
dependencies :: Semantics Applicative a -> Maybe ([Key], [Key])
```

The `Maybe` type constructor comes from the definition of the metalanguage: if the applicative semantics is partial (returns `Nothing`) it is impossible to extract its static dependencies. Successful static analysis yields a pair of lists representing the sets of input and output dependencies of a computation.

To extract the static data dependencies of an applicative computation we need to interpret its semantics in the special context of a *constant functor*.

4.1 The constant functor

The `Const a b` data type is defined as follows [12]:

```
newtype Const a b = Const { getConst :: a }
```

A value of the `Const a b` is just a value of any type `a` wrapped in a data constructor. However, it is important that the type constructor has a *phantom type* variable. This type variable allows us to declare useful instances of standard Haskell type classes such as `Functor` and `Applicative` for `Const a`. We would like to use this data type as a computational context for applicative semantics, hence we declare the corresponding instance[9]:

```
instance Monoid m => Applicative (Const m) where
    pure _              = Const mempty
    Const x <*> Const y = Const (x `mappend` y)
```

[9] `Const a` also has a `Functor` instance, where `fmap _ (Const x) = Const x`.

This instance exactly describes the desired behaviour of static dependency tracking computational context. `Const [Key]` is an applicative functor that ignores its enclosed value, but accumulates the declared dependencies using the `Monoid` instance for Haskell list data type[10].

4.2 Extracting dependencies

Using the seemingly vacuous datatype `Const` we define the function `dependencies`:

```
dependencies :: Semantics Applicative a -> Maybe ([Key], [Key])
dependencies s = partitionEithers . getConst <$> s read write
  where read  k    =        Const [Left  k]
        write k fv = fv *> Const [Right k]
```

We instantiate the polymorphic computation context with `f = Const [Key]` and supply custom tracking `read` and `write` functions. In fact, `read` does not perform any reading and just tracks the key as an input dependency, whereas `write` tracks the key as an output dependency and executes the effectful computation `fv`, to appropriately track its dependencies. The resulting list of keys gets unwrapped and unzipped by standard `partitionEithers . getConst`.

This whole paper is built around the above three-line implementation of the function `dependencies`: the rest is just functional programming background and folklore, which is required for understanding these three lines of code.

Fully armed with static dependency analysis, we can now define concurrency oracles for programs written in the presented metalanguage.

4.3 Concurrency oracle

A concurrency oracle is a function taking two computations and statically determining if they are data *concurrent*, i.e. do not share any data dependencies.

Definition (Concurrency Oracle Answer): Two terms of the metalanguage are *concurrent* if they do not share any data dependencies. They are in a *read* or *write conflict* if they share any input or output dependencies, respectively. If the share both input and output dependencies then they are considered to be in a *read-write conflict*. We use the following data type is used to encode concurrency oracle answers:

```
data OracleAnswer k = Concurrent
                    | ReadConflict [k]
                    | WriteConflict [k]
                    | ReadWriteConflict [k]
```

Definition (concurrency oracle): Consider two terms with applicative semantics `s1` and `s2` of the type `Semantics Applicative a`. A *concurrency oracle* is a function of the following type:

[10] The empty list `[]` is the monoid identity element and the list concatenation (++) is the associative binary operation.

```
concurrencyOracle :: Eq k => Semantics Applicative k v1 a
                          -> Semantics Applicative k v2 a
                          -> Maybe (OracleAnswer k)
```

The function is required to return `Nothing` when one of the given semantics is undefined, e.g. if it corresponds to an instruction with dynamic dependencies. Below is one possible implementation:

```
concurrencyOracle s1 s2 = do
    (r1, w1) <- dependencies s1
    (r2, w2) <- dependencies s2
    let readConflicts      = intersect r1 r2
        writeConflicts     = intersect w1 w2
        readWriteConflicts = intersect (r1 ++ w1) (r2 ++ w2)
    pure $ case (readConflicts, writeConflicts, readWriteConflicts) of
        ([], [], [] ) -> Concurrent
        (rs, [], rws) | rs == rws -> ReadConflict rs
        ([], ws, rws) | ws == rws -> WriteConflict ws
        (_ , _ , rws) -> ReadWriteConflict rws
```

The oracle determines static dependencies of two given terms and examines several possible cases of the intersections of their input and output dependencies.

4.4 Example oracles

In this subsection we show two examples to illustrate the usage of concurrency oracles. The examples are given in the form of interactive sessions, where '`ghci>`' denotes the command prompt.

Two `Load` instructions with different arguments are concurrent as confirmed by the oracle returning the result `Just Concurrent`:

```
ghci> concurrencyOracle (semanticsA (Load R0 0))
                        (semanticsA (Load R1 1))
Just Concurrent
```

Extending the first computation with an additional `Add` instruction causes a read conflict, as desired:

```
ghci> p1 = blockSemanticsA [Load R0 0, Add R0 1]
ghci> p2 = semanticsA (Load R1 1)
ghci> concurrencyOracle p1 p2
Just (ReadConflict [Addr 1])
```

We can overlay dependency graphs of programs `p1` and `p2` as follows (we place the programs at starting addresses 10 and 20, respectively):

```
ghci> Just g1 = programDataGraph (zip [10..] [Load R0 0, Add R0 1])
ghci> Just g2 = programDataGraph (zip [20..] [Load R1 1])
ghci> drawGraph (overlay g1 g2)
```

The resulting dependency graph is shown in Fig. 2. As one can see, the programs have a read conflict on key `Addr 1`, just like the oracle has determined.

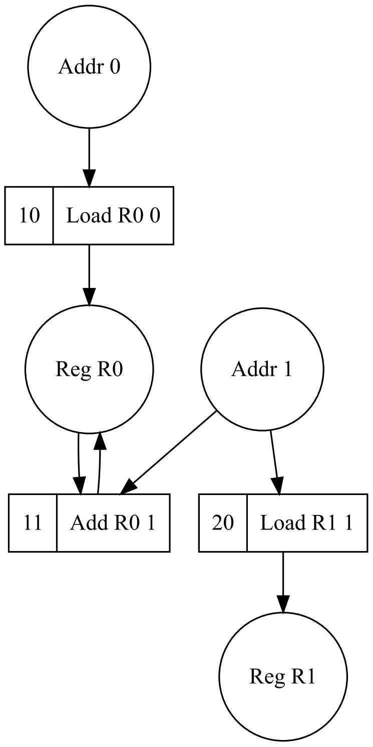

Fig. 2. An overlay of static dependency graphs of two blocks of instructions.

5 Synthesis of efficient hardware microcontrollers

In this section we present a method for extracting behavioural *scenarios* from system control programs and synthesising hardware microcontrollers that can execute these scenarios. We rely on Conditional Partial Order Graphs (CPOGs) [13] and associated tool support for synthesis; specifically, we use the CPOG plugin [1] of the Workcraft framework [2], which provides support for scenario specification and synthesis, and handles the translation of CPOGs to circuits to produce a physical implementation of the system microcontroller.

5.1 Extracting scenarios from programs

We define a *scenario* as a *partial order* (PO) $(\mathcal{I}, \prec)$, i.e. a binary precedence relation $\prec$ defined on a set of instructions $\mathcal{I}$ that satisfies two properties [6]:

- Irreflexivity: $\forall a \in \mathcal{I}, \neg(a \prec a)$
- Transitivity: $\forall a, b, c \in \mathcal{I}, (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$

To extract scenarios from programs, we reuse the dependency analysis that we developed for implementing concurrency oracles in the previous section. As one can see from our definition of scenarios, we only require to keep the information about the (partial) ordering of instructions. We can therefore discard the unnecessary details about which particular register, flag or memory location was the cause of the dependency, leading to the following procedure.

Extracting the behavioural scenario from a program:

1. Calculate all static data dependencies between the program instructions.
2. Construct the static dependency graph.
3. Contract data vertices, keeping the induced arcs between the instructions.
4. Perform the *transitive closure* of the resulting graph.

5.2 Scenario encoding and synthesis

In the context of this paper, we consider control systems which are programmed in low-level assembly-like languages. Using the procedure from the previous subsection, we can extract behavioural scenarios, represented as partial orders, from the system's control programs. These partial orders often have some shared functionality, which can be exploited by Conditional Partial Order Graphs and associated synthesis methods, leading to more efficient hardware implementations compared to those obtained by implementing each scenario separately.

Microcontroller synthesis:

1. Extract the scenarios from a given set of system control programs.
2. Synthesise a CPOG containing all the scenarios.
3. Apply the CPOG workflow for optimisation and compilation to hardware.

In the next subsection we will illustrate the described microcontroller synthesis approach on a simple example. We will consider a system with two control programs, extract the scenarios from these programs and synthesise them into a CPOG with some shared functionality.

5.3 Example

To illustrate the use of the scenario extraction procedure, let us sketch an example of a system controlling a hypothetical dual-motor autonomous vehicle. The vehicle has two motors directly driving its left and right front wheels. The system control unit has a simple application-specific instruction set, two general-purpose registers and a memory unit with four cells. The input to the unit is provided in memory cells 0 and 1. The unit controls the velocity of the left and right motors by outputting values to the cells 2 and 3, respectively. A motor's velocity may be adjusted with the instruction `Adjust` supplying the input value in a register and referring a target motor (2 or 3).

The first program (Fig. 3, left) implements the "drive straight" behaviour, driving both wheels with the same velocity. The graph on the right pictures the extracted partial order.

```
Load R0 0
Adjust R0 2
Adjust R0 3
```

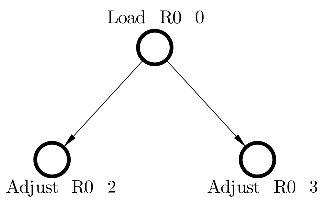

Fig. 3. Scenario “drive straight” and the corresponding partial order.

Another behaviour is “drive and turn”, which is implemented by adjusting the velocities with different values (Fig. 4). In this case, the resulting partial order contains two independent components.

```
Load R0 0
Load R1 1
Adjust R0 2
Adjust R1 3
```

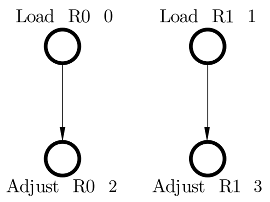

Fig. 4. Scenario “drive and turn” and the corresponding partial order.

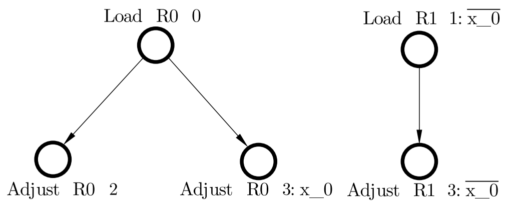

Fig. 5. Two operation scenarios and their composition.

Fig. 5 displays the Conditional Partial Order Graph which represents the composition of the two scenarios, where the additional control input x_0 is used to select the active scenario. The partial orders extracted from the programs are efficiently merged, thus rendering the CPOG composition to be more compact then a simple list of the extracted scenarios. The resulting hardware microcontroller, mapped to a 2-input gate library, is shown in Fig. 6.

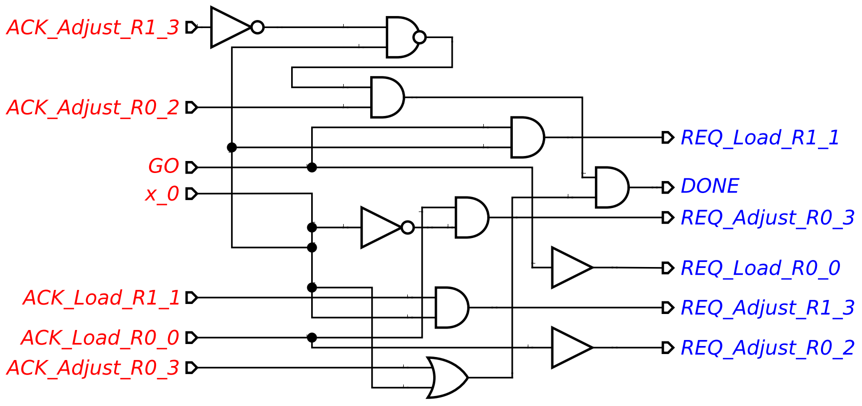

Fig. 6. Synthesised hardware microcontroller.

6 Conclusion

The paper presented a metalanguage for describing the semantics of instruction set architectures. Multiple interpretations of the metalanguage terms allow us to evaluate the semantics in different contexts without its modification. As the primary application, we present an approach to deriving concurrency oracles of the instructions with only static data dependencies.

To handle instructions whose dependencies are dynamic, it is possible to use conservative approximation of dependencies. For example, Fig. 7 shows the dependency graph for a program implementing the Euclidean algorithm for computing the greatest common divisor of two numbers that contains a conditional branch instruction `JumpZero`, which modifies the instruction counter `IC` only if the previous instruction set the `Zero` flag. In this example, we conservatively assume that `JumpZero` always depends on `IC`. Our future work includes the application of the presented methodology to extracting concurrency from real-world processor specifications, as described in [16].

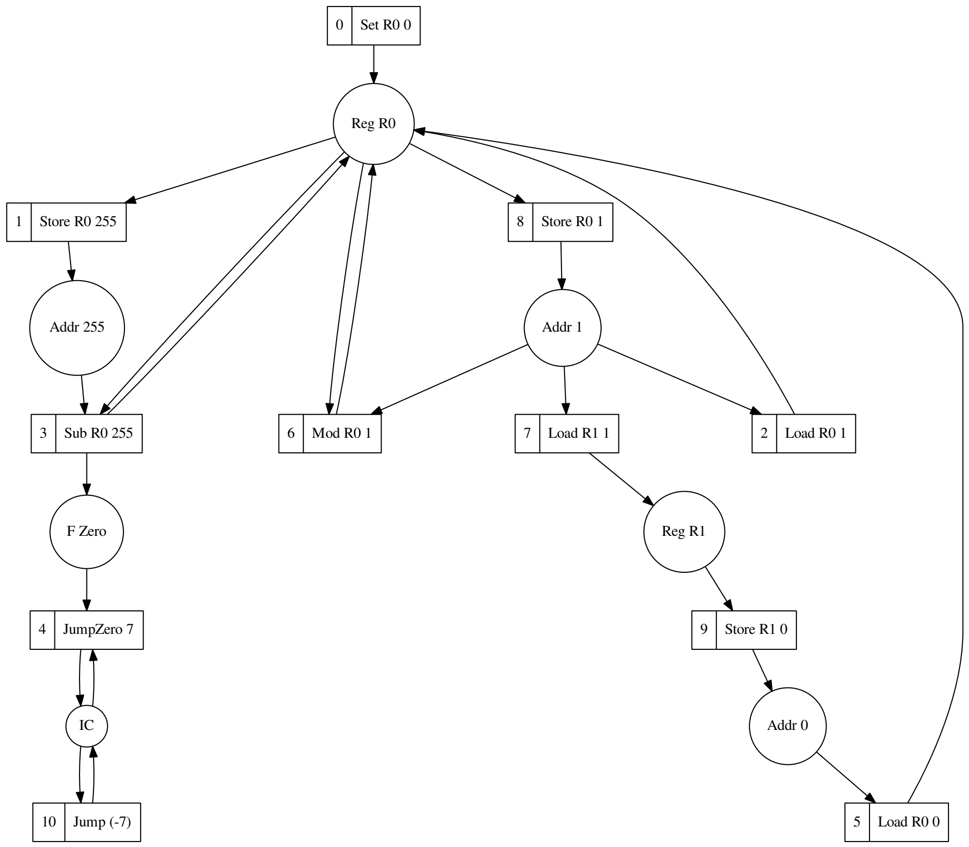

Fig. 7. Approximation of static dependencies of the Euclidean algorithm.

References

1. Scenco plugin repository, GitHub repository: github.com/tuura/scenco
2. Workcraft, GitHub repository: github.com/workcraft/workcraft, Workcraft website: www.workcraft.org
3. Van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
4. Van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering **16**(9), 1128–1142 (2004)
5. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. arXiv preprint arXiv:1705.02288 (2017)
6. Birkhoff, G.: Lattice Theory. No. v. 25, pt. 2 in American Mathematical Society colloquium publications, American Mathematical Society (1940), https://books.google.co.uk/books?id=0Y8d-MdtVwkC
7. Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: ACM SIGSOFT Software Engineering Notes. vol. 23, pp. 35–45. ACM (1998)
8. Dumas, M., García-Bañuelos, L.: Process mining reloaded: Event structures as a unified representation of process models and event logs. In: International Conference on Applications and Theory of Petri Nets and Concurrency. pp. 33–48. Springer (2015)
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: International Conference on Interactive Theorem Proving. pp. 243–258. Springer (2010)
10. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: the world's best macro assembler? In: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming. pp. 13–24. ACM (2013)
11. Marlow, S., Brandy, L., Coens, J., Purdy, J.: There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. SIGPLAN Not. **49**(9), 325–337 (Aug 2014)
12. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (Jan 2008)
13. Mokhov, A.: Conditional Partial Order Graphs. Ph.D. thesis, Newcastle University (2009)
14. Mokhov, A.: Algebraic Graphs with Class (Functional Pearl). In: Proceedings of the International Symposium on Haskell. ACM (2017)
15. Mokhov, A., Carmona, J., Beaumont, J.: Mining conditional partial order graphs from event logs. In: Transactions on Petri Nets and Other Models of Concurrency XI, pp. 114–136. Springer (2016)
16. Mokhov, A., Lukyanov, G., Lechner, J.: Formal verification of spacecraft control programs using a metalanguage for state transformers. arXiv preprint arXiv:1802.01738 (2018)
17. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of functional programming **17**(1), 1–82 (2007)
18. Research, A.L.: Graphviz – Graph Visualization Software (1991), https://www.graphviz.org/
19. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 53–66. TLDI '07, ACM (2007)