CoCoSpec: A Mode-aware Contract Language for Reactive Systems*

Adrien Champion[1], Arie Gurfinkel[2], Temesghen Kahsai[3], and Cesare Tinelli[1]

[1] The University of Iowa
[2] SEI / Carnegie Mellon University
[3] NASA Ames / Carnegie Mellon University

Abstract. Contract-based software development has long been a leading methodology for the construction of component-based reactive systems, embedded systems in particular. Contracts are an effective way to establish boundaries between components and can be used efficiently to verify global properties by using compositional reasoning techniques. A contract specifies the assumptions a component makes on its context and the guarantees it provides. Requirements in the specification of a component are often case-based, with each case describing what the component should do depending on a specific situation (or mode) the component is in. We introduce COCOSPEC, a mode-aware assume-guarantee-based contract language for embedded systems built as an extension of the Lustre language. COCOSPEC lets users specify mode behavior directly, instead of encoding it as conditional guarantees, thus preventing a loss of mode-specific information. Mode-aware model checkers supporting COCOSPEC can increase the effectiveness of the compositional analysis techniques found in assume-guarantee frameworks and improve scalability. Such tools can also produce much better feedback during the verification process, as well as valuable qualitative information on the contract itself. We presents the COCOSPEC language and illustrate the benefits of mode-aware model-checking on a case study involving a flight-critical avionics system. The evaluation uses KIND 2, a collaborative, parallel, SMT-based model checker extended to fully support COCOSPEC.

1 Introduction

The process of developing safety-critical embedded software (as used, for instance, in transportation, in aerospace and in medical devices) is becoming increasingly more challenging. The high number of functionalities now implemented at the software level, the inter-dependencies of software tasks, and the need to integrate different existing subsystems all lead to highly complex software-intensive cyber-physical systems. To manage this complexity embedded software is designed and implemented as the composition of several reactive components, each performing a specific, relatively simple functionality. A leading methodology to develop component-based software is *contract-based design*. In this paradigm, each component is associated with a contract specifying

* This material is based upon work funded and supported by NASA under Grant # NNX14AI09G, and by the Department of Defense under Contract # FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002921

its input-output behavior in terms of *guarantees* provided by the component when its environment satisfies certain given *assumptions*. When contracts are specified formally for individual components, they can facilitate a number of development activities such as compositional reasoning during static analysis, stepwise refinement, systematic component reuse, and component-level and integration-level test case generation.

Embedded system components often exhibit complex discrete internal behavior akin to state transitions in finite-state machines. At any one time, the component is in some of a number of different *modes* as a consequence of past events, and its response to the current inputs differs depending on the mode(s) it is in. For instance, in a flight guidance system, modes govern the choice of a specific control algorithm: an *approach* mode enables a controller that attempts to land the airplane, whereas a *climb* mode enables a controller that attempts to take the aircraft to a suitably safe altitude. The behavior of a multi-component system emerges from complex interactions between the modes of these components.

Despite the prevalence of modes in embedded system design, common contract formalisms for such systems are not *mode-aware*, as they only allow one to express general assumptions and guarantees. As a consequence, mode-based behavior, which is ubiquitous in specification documents, ends up being encoded in conditional guarantees of the form "*situation* ⇒ *behavior*". Correspondingly, assume-guarantee-based tools are *mode-agnostic*, they cannot easily distinguish between mode-specific requirements and general guarantees such as "the output shall always be positive" although the two kinds of requirement describe very different expectations. We see mode-awareness as a natural and important evolution of assume-guarantee contracts and compositional reasoning based on them. We argue that by distinguishing between modes and guarantees in contract-based design we avoid losing fine-grained information that can be used to further improve the scalability and the user feedback of automated analyses.

Contributions This paper focuses on a large class of embedded systems, (finite- and infinite-state) discrete synchronous reactive systems. For these systems, we introduce CoCoSpec, a mode-aware specification language for COntract-based COmpositional verification of *safety properties* that extends the assume-guarantee paradigm, and describe the sort of advantages that mode-aware tools can provide. We focus on features of the language that help with *i)* detecting shortcomings in the (modes of the) specification of a system independently of its implementation, *ii)* improving fault localization, *iii)* comparing the user's understanding of the contract / system pair with its actual behavior, and *iv)* improving the scalability of the verification process.

For concreteness, we have developed and implemented CoCoSpec as an extension of the synchronous dataflow language Lustre [12], and so we will describe it as such here. We stress, however, that its theory and applications are generic with respect to the whole class of specification languages for discrete synchronous reactive systems.

We briefly introduce the Lustre language and the assume-guarantee paradigm in Section 2. The syntax and semantics of CoCoSpec are described in Section 3, along with a running example extracted from a medium-size case study we did to showcase CoCoSpec's main features. We present our case-study in more details in Section 5 and report on the benefits of mode-awareness to write and debug contracts, raise the trust in their accuracy, and improve the scalability of automatic contract verification.

Related Work The notion of a *contract* has a long history in software engineering and traces its root to rely-guaranteed approaches introduced by Hoare, Dijkstra and others [14,10,17]. It is adopted in earnest in the *design by contract* methodology [25,16], which has been applied in different areas of software development and verification. Newer programming languages such as Dafny [22] incorporate formal contracts and compile-time contract checking as native features. Formal contracts have also been integrated into popular programming languages, via the addition of *ad hoc* specification languages, e.g., ACSL [20] for C/C++, JML [21] for Java, or SPARK [2] for Ada. ACSL in particular has a notion of *behavior* in function contracts which is similar to that of mode in COCOSPEC. One major difference is that predicates in an ASCL contract refer only to individual states (such as the pre- and the post-state of a function call), while in COCOSPEC, which is meant for reactive systems, they can use temporal operators.

A suitable notion of contract for reactive software, where components continuously process incoming data and produce output based on the input data and internal state information, is provided by the assume-guarantee paradigm for compositional verification [3]. A large number of contract formalisms have been proposed for reactive systems; for instance, Cimatti and Tonetta [7] develop a trace-based contract framework and adapt it to the properties specification language Othello [6]. Cofer *et al.* [8] follow a contract-based approach to perform compositional verification geared towards architectural models. Our approach differs from the techniques and languages above in the emphasis COCOSPEC puts on the mode-based behavior of the analyzed embedded system. In this sense, it is more in the spirit of Parnas tables [26], but for reactive systems.

2 Background

Lustre COCOSPEC was conceived as a contract extension to languages, such as Lustre [12], for modeling systems composed of *synchronous reactive* components. Such languages are based on the theory of synchronous time in which all components maintain a permanent interaction with their environment (e.g., a larger component, or the physical environment in case of top level components) and are triggered by an abstract universal clock. Lustre is a stream-based executable modeling language for finite- and infinite-state reactive systems. Every system in Lustre takes as input one or more infinite streams of values of the same type, and produces one or more infinite streams as output. Lustre systems are assumed to run on a universal base clock that represents the smallest time span the system is able to distinguish. Individual components can, however, be defined to run on coarser-grained clocks. For simplicity, we ignore this feature here and pretend that all components run on the same clock. In that case, each stream of type τ can be understood mathematically as a function from $\mathbb{N}$ to τ.

System components are expressed in Lustre as *nodes* with an externally visible set of inputs and outputs. Variables are used to represent input, output and locally defined streams. Basic value types include *real* numbers, *integer* numbers, and *Booleans*. Operationally, a node has a cyclic behavior: at each clock tick t it reads the value of each input stream at position or *time* t, and instantaneously computes and returns the value of each output stream at time t. Lustre nodes can be made stateful by having them refer to stream values from (a fixed number of) previous instants.

Typically, the body of a Lustre node consists in a set of stream equations of the form $x = s$, where x is a variable denoting an output or a locally defined stream and s is a stream algebra over input, output, and local variables. Most stream operators are point-wise liftings of the usual operators over stream values. For example, if x and y are two integer streams, the expression $x + y$ is the stream denoting the function $\lambda t.x(t) + y(t)$; an integer constant c, denotes the constant function $\lambda t.c$. Two important additional operators are a unary right-shift operator `pre`, used to specify state computations, and a binary initialization operator `->`, used to specify initial state values. At time $t = 0$, the value $(\texttt{pre}\ x)(t)$ is undefined; for each time $t > 0$, it is $x(t-1)$. In contrast, the value $(x\ \texttt{->}\ y)(t)$ equals $x(t)$ for $t = 0$ and $y(t)$ for $t > 0$. Syntactic restrictions guarantee that all streams in a node are inductively well defined.

Since a node is itself a mapping from input to (one or more) output streams, once defined, it can be used like any other stream operator in the right-hand side of equations in the body of other nodes, by applying it to streams of the proper type.

Example 1. As an example, here is how a stopwatch could be modeled in Lustre.

```
node previous ( x : int ) returns ( y : int )
let
  y = 0 -> pre x ;
tel

node stopwatch ( toggle, reset : bool ) returns ( count : int );
var running : bool;
let
  running = (false -> pre running) <> toggle ;
  count = if reset then 0
          else if running then previous(count) + 1
          else previous(count) ;
tel
```

Auxiliary node `previous` defines an initializing delay operator for integer streams that takes a stream with values $x_0, x_1, x_2, \ldots$ and returns the stream $0, x_0, x_1, x_2, \ldots$ Node `stopwatch` models a stopwatch with two buttons, modeled respectively by the Boolean input variables `toggle` and `reset`, one to start/stop the stopwatch and the other to reset its time to zero. The locally defined auxiliary stream `running` keeps track of when the clock is running. Its value is `true` initially iff `toggle` is not equal to (`<>`) `false` at that time; it is `true` later iff its previous value is different from the current value of `toggle`. Stream `count` counts the number of instants the clock has been running since the beginning or the last reset, if any. Initially, it is 0 unless `reset` is `false` and `toggle` is `true`, in which case it is 1. Afterwards, it is reset to 0 every time `reset` is `true`, is incremented by 1 while the clock is running, and is kept at its previous value when the clock is stopped. The definition of `count` contains two applications of node `previous`, to `count` itself. Note that despite the apparent circularity of this definition, `count` is well defined because of the delay in `previous`. □

Lustre has a formally specified semantics, which interprets nodes as a variant of extended-state Mealy machines [13] and node application as parallel composition. Discrete embedded systems developed in popular modeling languages such as Simulink or SCADE can be faithfully translated into Lustre (e.g., [9]). A large class of safety properties of Lustre models can be internalized as (Boolean) observer streams or observer nodes [11] and verified efficiently by SMT-based model checkers [18].

Assume-Guarantee paradigm Assume-guarantee contracts [24] in component-based reactive systems provide a mechanism for capturing the information needed to specify and reason about component-level properties. An *assume-guarantee contract* for a component K is a pair of *past linear temporal logic* (pLTL) [19] predicates $\langle A, G \rangle$ where the *assumption* A ranges over the inputs of K, and the *guarantee* G ranges over its inputs and outputs.

pLTL is a rich logic that uniformly supports the formulation of *bounded liveness* and *safety* properties, the kind of properties we focus on in this work. In terms of standard LTL, the semantics of an assume-guarantee contract $\langle A, G \rangle$ is the formula $\mathbf{G}\, A \Rightarrow \mathbf{G}\, G$ where $\mathbf{G}$ is the *globally* operator. From a verification point of view, however, proving that a component K satisfies that formula amounts to proving that the pLTL formula $\mathbf{H}\, A \Rightarrow G$ is invariant for K where $\mathbf{H}$ is the *historically* modality of pLTL [23].[4]

Compositional reasoning is achieved by proving that each component satisfies its own contract *as well as* the guarantees of any component it provides input to. More precisely, for the latter proof obligation, if a component K_1 is composed in parallel with a component K_2 and provides inputs to K_2, one must also prove that those inputs always satisfy the assumptions of K_2. The proof that K_1 satisfies its contract can then assume that any output provided by K_2 satisfies the guarantees in K_2's contract. In Lustre terms, one must prove that every application $n(s_1, \ldots, s_n)$ of a node n inside another node m is *safe* in the sense that the actual parameters $s_1, \ldots, s_n$ satisfy at all times the assumptions of n on its inputs. To prove that m satisfies its own contract one can assume that the result of the application $n(s_1, \ldots, s_n)$ satisfies the guarantees in n's contract.

3 The CoCoSpec Language

CoCoSpec extends Lustre by adding constructs to specify contracts for individual nodes, either as special Lustre comments added directly inside the node declaration, or as external, stand-alone contract declarations. The latter are similar in shape to nodes but are introduced with the `contract` instead of the `node` keyword. A node can *import* an external contract using a special Lustre comment of the form

```
(*@contract import <name>(<input params>) returns (<output params>); *)
```

For specification convenience, the body of a stand-alone contract can contain equalities defining local streams, using the `var` (`const`) keyword for (constant) streams. Besides local streams, a contract contains *assume* and *guarantee statements*, and *mode declarations*. Modes are named and consist of *require* and *ensure statements*. They have the form shown on Figure 1. Statements can be any well-typed Lustre expressions of type `bool`. In particular, expressions can contain applications to previously defined Lustre nodes. This is convenient, for instance but not exclusively, if one wants to use pLTL operators since those can be defined as Lustre nodes.

Example 2. A possible contract, and associated import, for the `stopwatch` component from Example 1 could be the following:

[4] Intuitively, $\mathbf{H}\, P$ states that P has been true in all states of an execution up to the current state.

```
contract stopwatchSpec ( tgl, rst : bool ) returns ( c : int ) ;
let
  var on: bool = tgl -> (pre on and not tgl) or (not pre on and tgl) ;
  assume not (rst and tgl) ; guarantee c >= 0 ;
  mode resetting ( require rst ; ensure c = 0 ; ) ;
  mode running ( require not rst ; require on ; ensure c = (1 -> pre c + 1) ; ) ;
  mode stopped ( require not rst ; require not on ; ensure c = (0 -> pre c) ; ) ;
tel

node stopwatch ( toggle, reset : bool ) returns ( time : int ) ;
(*@contract import stopwatchSpec(toggle, reset ) returns (time) ; *)
let ... tel
```

Note that `pre` binds more strongly than all other operators; `=>` is Boolean implication.

The contract has the same interface as the node. It uses an auxiliary Boolean variable `on` capturing the exact conditions under which the stopwatch should be on: initially when the start/stop button `tgl` is pressed (i.e., true); later when it was previously on and the start/stop button is not being pressed, or it was previously off and the start/stop button is being pressed. The contract contains a global assumption that the reset button `rst` and the start/stop button are never pressed at the same time, and a global guarantee that the time counter `c` is always non-negative. It also specifies three modes for the stopwatch. The component is in `resetting` mode if the reset button is pressed. When that button is not pressed, it is in `running` mode if the conditions captured by `on` hold, and is in `stopped` mode otherwise. The `ensure` statements of the three modes specify how `c`, the counter, should behave. It *i)* is reset to 0 in `resetting` mode, *ii)* is incremented by 1 in `running` mode, and *iii)* maintains its previous value in `stopped` mode. To import the contract, node `stopwatch` instantiates the contract's formal (input an output) parameters with any expression of the same type. □

In our experience, the ability of a node to import a stand-alone contract provides great flexibility. It makes writing specifications and implementations more independent, and facilitates the reuse of contracts between components. In general, a node can import more than one contract and have also local assumptions, guarantees and modes. The contract of a node is the union of all the local and imported assumptions, guarantees and modes.

Expressions in contracts can refer to a mode directly by using its name as if it were a Boolean variable. This is just a shorthand for the conjunction of all the `require` statements in the mode. CoCoSpec avoids potential dependency cycles between modes due to this feature by prohibiting forward and self references. Each stand-alone contract defines a namespace, with `::` as the namespace projection operator. As a consequence, modes can be referred to both inside and outside the contract they belong to. For example, in the `stopwatch` contract the `require` statement `not on` of mode `stopped` can be replaced, equivalently, by `not ::running`. In contrast, the `require` and `ensure` statements of `running` cannot contain a (forward) reference to mode `stopped`. The expression `::stopwatchSpec::running` can be used in the contract of `stopwatch` to refer to the `running` mode of the imported `stopwatchSpec` contract, as in

```
node stopwatch ( toggle, reset : bool) returns ( time : int ) ;
(*@contract import stopwatchSpec(toggle, reset) returns (time) ;
  guarantee true -> (
    (pre ::stopwatchSpec::running and tgl) => ::stopwatchSpec::stopped
  ) ; *)
```

Finally, neither `assume` nor `require` statements can contain references to current values of an output stream—although they may refer to previous values of those streams via the `pre` operator. This is a natural restriction because it does not make sense in practice to impose *pre*conditions on the current output values.

3.1 Formal Semantics and Methodology

A COCOSPEC *contract* for a Lustre node N is a triple $\langle A, G, M \rangle$ where A is a set of *assumptions*, G is a set of *guarantees*, and M is a set of *modes*. A *mode* is a pair $(R,\ E)$ where R is a set of *requires* and E is a set of *ensures*. Assumptions, guarantees, requires and ensures are all stream formulas, i.e., Boolean expressions over streams. A mode $(R,\ E)$ in the contract of N is *active* at time t in in an execution of N if $\bigwedge R$ is true at that time.

Formally, we define a COCOSPEC contract $C = \langle A, G, M = \{(R_i,\ E_i)\} \rangle$ for some node N as the assume-guarantee contract $C' = \langle A,\ G' \rangle$, with $G' = G \cup \{R_i \Rightarrow E_i\}$.[5] Node N *satisfies* C if its corresponding extended-state machine satisfies contract C' in the standard sense, that is, if it satisfies $\mathbf{G}\, A \Rightarrow \mathbf{G}\, G'$.

We require for a contract $C = \langle A, G, M = \{(R_i,\ E_i)\} \rangle$ to be such that the formula

$$\mathbf{G}\,(A \wedge G \wedge \{R_i \Rightarrow E_i\}) \;\Rightarrow\; \mathbf{G}\,(\bigvee \{R_i\}) \tag{1}$$

is logically valid in LTL. Note that this is a (meta)requirement on the contract itself, not on its associated node(s). Intuitively, it states that in the scenario where the contract's assumptions and guarantees both hold, at least one of the requires holds at all times. COCOSPEC modes are meant to formalize requirements coming from specification documents that describe a transient behavior. If property (1) holds, then any node satisfying contract C, and used in a context where C's assumptions are always met, has at all times at least one active mode. This ensures that the contract covers all possible cases whenever its assumptions hold.

In practice, the first step when verifying a COCOSPEC contract is to check the defensive property (1). If it does not hold, a situation unspecified by the contract is reachable, hence the contract is incomplete and must be fixed. If one desires, temporarily perhaps, to have an underspecified contract on purpose, one can add a mode with an empty set of ensures and a set of requires that captures the missing cases. The point is that the underspecification of mode behavior should be formalized explicitly and not be a consequence of a missing set of requirements.

If the defensive property of a contract C of a node N holds, the next step is to verify, using assume-guarantee reasoning, that N respects C. We abstract each application of another node inside N by that node's contract, replacing the contract's formal parameters with the actual parameters in the application. We then prove that N respects C whenever its subnodes respect their own contract. We also prove that N contains only safe applications of other nodes. Overall, the analysis of a system is successful if we can prove that *i)* none of the contracts used allow unspecified behavior, *ii)* all nodes respect their contract, and *iii)* all node applications are safe.

[5] We will identify sets of formulas, such as R_i and E_i, with the conjunction of their elements.

```
mode <id> (
  require <expr> ;
  ...
  require <expr> ;
  ensure <expr> ;
  ...
  ensure <expr> ;
) ;
```

Fig. 1: Mode syntax.

```
node ml (
  -- Control request flags.
  altRequest, fpaRequest : bool ;
  -- Deactivation flag.
  deactivate : bool ;
  -- Current and target altitude.
  altitude, targetAlt : real )
returns ( altEngaged, fpaEngaged: bool ) ;
```

Fig. 2: Activates one of the controllers.

```
node switch( on, off: bool )
returns ( out : bool ) ;
let
  out =
    not off and
    (on -> on or pre out) ;
tel
```

Fig. 3: `switch` helper node.

Note that a traditional assume-guarantee contracts $\langle A, G \rangle$ is expressible in CoCoSpec, as the contract $\langle A, G, \emptyset \rangle$. Property (1) is then trivially valid, and the analysis reduces to verifying $\langle A, G \rangle$. CoCoSpec is thus an extension of assume-guarantee contracts that natively supports, via the use of modes, requirements for transient behavior. We discuss the benefits that modes bring to mode-aware analyses in Section 5.

3.2 Using CoCoSpec: an Example

We now describe an example of system specification in CoCoSpec that allows us to illustrate concretely the main features of the language. The example is derived from an extensive case study where we took a realistic Lustre model of an avionics system developed by NASA [15,4], and wrote CoCoSpec contracts based on a natural language requirement specification. We discuss the study in detail in Section 5. For the purposes of this subsection, it is not crucial to explain the whole model and its expected overall functionality except to say that the system has a component `ml` that governs the engagement of two sub-controllers. Figure 2 shows the signature of the corresponding Lustre node[6]. This component decides whether two controllers, an *altitude controller* (`alt`) and a *flight path angle (FPA) controller* (`fpa`), should be engaged or not based on their respective request flags (`altRequest` and `fpaRequest`), a deactivation flag (`deactivate`), the current altitude (`altitude`), and the target altitude (`targetAlt`).[7]

Let $smallGap$ be a predicate that holds iff the distance between the current and the target altitude is smaller than a certain value, say 200ft. The requirements relevant to the `ml` component, namely Guide 170, 180, and 210 in [15], state that when $smallGap$ holds then the altitude controller has priority over the FPA controller: when requested to, the latter can engage provided that there is no request for the altitude controller to engage. When $smallGap$ is false the FPA controller has priority instead (Guide 170 and 180). The request protocol is the following. An engagement request for a controller becomes active as soon as the corresponding input flag becomes true, and remains active until the `deactivate` flag becomes true. A generic auxiliary node modeling this protocol for an arbitrary pair of activation and deactivation flags is shown in Figure 3.

[6] The node, called `MODE_LOGIC_AltAndFPAMode` in the original model, was slightly altered and its specification simplified for readability and simplicity.

[7] What the altitude and the FPA controllers actually do is not important at this point.

The specification for the `ml` component does not have any explicit assumptions. In the traditional assume-guarantee setting (e.g., in [1]) one would then be inclined to write a contract for `ml` with the following guarantee:

```
(    smallGap and altRequested                      => altEngaged) and
(    smallGap and fpaRequested and not altRequested => fpaEngaged) and
(not smallGap and fpaRequested                      => fpaEngaged) and
(not smallGap and altRequested and not fpaRequested => altEngaged)
```

where `altRequested = switch(altRequest, deactivate)` and `fpaRequested` is defined similarly. A contract with a single, complex guarantee, leads to loss of information in practice, for both human readers and static analysis tools such as model checkers. In contrast, CoCoSpec allows one to provide the same information but in a disaggregated form, explicitly accounting for the various cases through the use of modes. With a mode-based specification, assumptions only state general conditions on legal uses of the component—for instance, that the altitude values are always positive. Similarly, guarantees specify mode-independent behavior—in this case, that the altitude and FPA controllers never engage at the same time.

```
contract ml ( altRequest, fpaRequest, deactivate : bool ; altitude, targetAlt : real )
returns ( altEngaged, fpaEngaged : bool ) ;
let
  var altRequested = switch(altRequest, deactivate) ;
  var fpaRequested = switch(fpaRequest, deactivate) ;
  var smallGap = abs(altitude - targetAlt) < 200.0 ;
  assume altitude >= 0.0 ;
  guarantee targetAlt >= 0.0 ;
  guarantee not altEngaged or not fpaEngaged ;
  mode guide210Alt ( require smallGap ; require altRequested; ensure altEngaged ; ) ;
  mode guide210FPA ( require smallGap ; require fpaRequested ; require not altRequested;
                     ensure fpaEngaged; ) ;
  mode guide180 ( require not smallGap ; require fpaRequested; ensure fpaEngaged; ) ;
  mode guide170 ( require not smallGap ; require altRequested ; require not fpaRequested;
                  ensure altEngaged ; ) ;
tel
```

Debugging the specification early on We argue that, in addition to enabling compositional reasoning, CoCoSpec contracts also lead to more accurate analyses compared to traditional assume-guarantee by facilitating *blame assignment*. A mode-aware tool knows which modes are active at each step of a counterexample execution. Hence it can provide better feedback since modes are in effect user-provided abstractions of concrete states. Designers can reason about them to fix the system or its specification, instead of looking at concrete values, which may be less readable and informative.

In our running example, attempting to prove `ml` correct does not go very far: the defensive check fails right away and produces a counterexample triggering unspecified behavior. The problem is resolved by noting that the English specification means to say that `fpaRequested` and `altRequested` should be true *only* in the cases discussed above. Hence, this issue is easily addressed by adding the following two modes:

```
mode noAlt ( require not altRequested ; ensure not altEngaged ; ) ;
mode noFPA ( require not fpaRequested ; ensure not fpaEngaged ; ) ;
```

Now, because this example is quite simple, an experienced reader may have noticed the incompleteness in the specification already when we first introduced it. As we argue in Section 5, however, mode-based blame assignment is a very valuable feature on realistic systems with a large number of modes and complex require predicates.

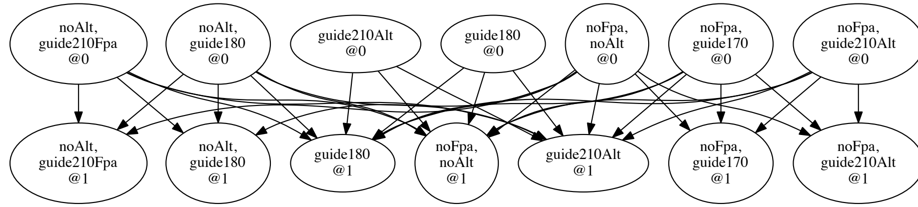

Fig. 4: Reachable combinations of modes in one transition from the initial state for `ml`.

Evaluating the specification We discuss next two approaches for checking that the semantics of a CoCoSpec contract corresponds to a user's understanding of it.

Unreachable properties over the specification as modes. Going back to the `ml` node, one could argue that mode `guide170` should not be reachable from mode `guide210FPA` in one step (i.e., from time t to time $t + 1$). That this is the case is not necessarily obvious because of the memorization capabilities provided by the `switch` component from Figure 3.[8] Since the property is expected to hold, verifying it would raise trust in the contract. Moreover, if the specification or the system later evolved to the point of not satisfying that property anymore, it would be useful for a new analysis to reveal that. This can be achieved by formulating the property explicitly as a CoCoSpec mode:

```
mode no170From210FPA ( require false -> pre ::guide210FPA ; ensure not ::guide170 ; ) ;
```

Exploration of reachable modes. When the defensive property (1) holds, modes provide effectively a small, user-defined abstraction of a component's reachable state set, with each abstract state represented by a set of active modes. One can then use explicit-state model checking techniques to analyze the possible executions of a component at the level of mode transitions. For instance, one can unroll the abstract transition relation to some depth to verify the presence of expected mode transition sequences or see if unexpected ones occur. Figure 4 shows (up to depth 1 only, for space constraints) the graph of reachable modes for the `ml` system, starting from each possible initial mode combination. Even by simple visual inspection, one can obtain a better high-level validation of one's understanding of the contract against the actual behavior of the model. For instance, is it expected that `guide170` is active only when `noFpa` is, or that the mode combination {`noFpa`, `noAlt`} can be reached from any initial mode combination?

4 Implementation

We added full support for CoCoSpec to Kind 2 [5], an open-source, multi-engine, SMT-based model checker for safety properties of Lustre programs, built as a successor of the PKind model checker [18].[9] Its basic version takes as input a Lustre file annotated with multiple properties to be proven invariant, and outputs for each property either a confirmation or a counterexample trace, a sequence of inputs that falsifies the property. Kind 2 is able to read Lustre models annotated with CoCoSpec contracts

[8] It is true in this instance because in the `switch` mode, `off` has priority over `on`.
[9] Kind 2 is available at http://kind.cs.uiowa.edu/ .

and verify them using compositional reasoning. We implemented all the features discussed in the previous section, including the exploration of the reachable modes of the input system to generate the corresponding graph.

Given a Lustre system S annotated with CoCoSpec contracts, Kind 2 can be run in *compositional mode* on S. In that case it will analyze the top node of S by abstracting its subnodes by their contracts, as discussed above. This is not enough to prove S correct though, since the correctness of the subsystems represented by the subnodes is not checked. Kind 2's *modular mode* addresses this shortcoming: in modular mode, Kind 2 will analyze each subsystem of the hierarchy, bottom-up, reusing previous results as it goes. When run in compositional and modular mode together, Kind 2 will analyze each subsystem compositionally, after proving the defensive check on its contract. If all systems of the hierarchy are proved correct, then the system as a whole is deemed safe.

Kind 2 also has a *refinement* mechanism. Say a node M contains an application of a node N, and the compositional analysis of M produces a counterexample. The counterexample might be spurious, as N was abstracted by its contract which might be too weak to verify M. In this case, if N was proved correct previously under some abstraction $\mathcal{A}$ of the node applications in its own body, then Kind 2 will launch a new analysis where the application of N in M is (in effect) replaced by the body of N under the abstraction $\mathcal{A}$. The failure of the compositional analysis signals that there is something wrong with the system and/or its specification. The refinement mechanism aims at giving more information about the problem. For instance, if M can be proved correct after refining the application of N as described above then probably the contract of N should be strengthened until the compositional analysis succeeds without having to use refinement.

5 Evaluation

As a case study to evaluate the usefulness and effectiveness of CoCoSpec, we chose a model derived from NASA Langley's Transport Class Model (TCM) [15], a control system for a mid-size (∼250K lb), twin-engine, commercial transport-class aircraft. While the TCM is not intended as a high-fidelity simulation of any particular transport aircraft, it was designed to be representative of the types of nonlinear behaviors of this class of aircraft. We specified in CoCoSpec some of the safety requirements for the TMC recently elicited by Brat *et al.* [4] from Federal Aviation Regulations and other documents. We will refer to those as *FAR requirements*. In this section, we discuss our specification of the FAR requirements and how CoCoSpec aided their automated compositional verification.[10]

The TCM includes submodels for the avionics (with transport delay), actuators, engines, landing gear, nonlinear aerodynamics, sensors (including noise), aircraft parameters, equations of motion, and gravity. It is primarily written in Simulink, consisting of approximately 5,700 Simulink blocks. The system also includes several thousand lines

[10] Full data on the case study, including models, contracts, reachability graphs, and instructions on how to reproduce our experimental results using the CoCoSpec version of Kind 2 are available at `https://github.com/kind2-mc/cocospec_tcm_experiments`.

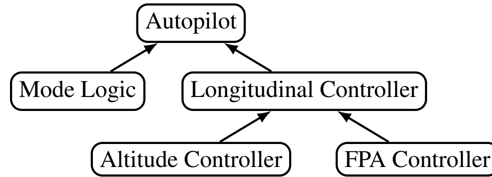

Fig. 5: Autopilot subsystem of the TCM.

of C/C++ code in libraries, primarily used for the simulation of the aircraft engines and the nonlinear aerodynamics models. Here, we focus on the guidance and control submodels and their properties within the context of the TCM. These models are written entirely in Simulink and so can be faithfully converted automatically to a (multi-node) Lustre model. We will call *Autopilot* the subsystem of the TCM that combines these subcomponents.

The subsystem is depicted in Figure 5. Each node in the graph corresponds to a component of the Autopilot system, and to a node in the Lustre model. The system actually has more nodes than this graph, which only shows the main components that are specified by a contract. While using the same name, we will distinguish between a component (for instance, the Autopilot node of the graph) and its corresponding subsystem, obtained as the composition of that component with all of its subcomponents. Of particular interest to us is the *Longitudinal Controller* subsystem, which combines two mutually exclusive subcontrollers, the *Altitude Controller* and the *FPA Controller*, to produce an elevation command for the aircraft. When engaged, the first subcontroller produces an elevation command aimed at reaching the target altitude it is given as input. The other subcontroller produces instead an elevation command aimed at reaching a target flight path angle (FPA). In practice, the two subcontrollers are not independent since the FPA Controller uses the output of the Altitude Controller to produce its output, regardless of which controller is engaged. The Longitudinal Controller and all its subcomponents are mostly numerical and include nonlinear arithmetic expressions.

Another subsystem of Autopilot, called *Mode Logic*, is in charge of deciding which of the subcontrollers, if any, should be active at any time. The decision is based on a number of parameters, including the aircraft's speed, altitude, and pitch, and the commands from the cockpit. A simplified version of the Mode Logic component is modeled by the `ml` Lustre node described in Section 3, and so we will not discuss it further here. Instead, we present the benefits of mode-awareness in the specification and verification of the Longitudinal Controller and of the Autopilot overall.

5.1 Benefits of CoCoSpec

Since we are not experts in flight control systems, we do not have a full understanding of the TCM or the details that the rather high-level FAR requirements leave unspecified. So, for our case study, we started from the FAR requirements and the TCM models (in Simulink and in their Lustre translation), and wrote naïve contracts for the Lustre components, which Kind 2 would then promptly disprove. In general, the concrete counterexample traces returned by Kind 2 were too detailed and specific for us to see

what was wrong. However, thanks to the mode information, KIND 2 could point us relatively precisely to relevant parts of a trace. Additionally, knowing what modes were active at any point in the trace provided a nice abstraction that would allow us to reframe the problem in more general terms and help us find ways to revise the contract.

Probably the most useful feature was the exploration of reachable modes yielding the reachability graphs introduced in Section 3. Even when KIND 2 proved the correctness of the modes we wrote, the mode reachability graph it generated would often reveal significant gaps in our understanding of the system's behavior. Sometimes only *idle* modes would be reachable, because of problems in our `require` statements; some other times the graph would contain mode transitions we expected not to be possible; or, even worse, it would contain deadlocked states. We cannot overstate the usefulness of this feature in the case study: it quickly became impossible for us to trust a contract without examining the mode reachability graph first.

We discussed in Section 3.2 how easy it is to express in COCOSPEC mode properties by using a mode's identifier to refer to the conjunction of its `require` statements. For instance, properties like "mode m_2 cannot immediately follow mode m_1" and "modes m_3 and m_4 cannot be active at the same time" can be encoded respectively as:

```
guarantee true -> pre ::m1 => not ::m2 ;
guarantee not (::m3 and ::m4) ;
```

Based on the reachability graphs and our understanding of the specification and the various subsystems, we ended up writing several properties like the above, to assess the quality of our contract. While reachability graphs provide a graphical mode-based exploration of the system and its specification up to some depth, expressing and checking properties over the specification itself considerably raises the trust in both the specification and the verification process. The fact that we do not duplicate mode requirements, but instead rely on a mode-aware tool to refer directly to mode identifiers, guarantees that these properties are synchronized with the current definition of each mode.

5.2 Verifying the Longitudinal Controller

KIND 2, like other model checkers for infinite-state systems, eventually relies on issuing queries to an SMT solver to reason about the system under analysis. Lustre models are converted internally into transition systems with an initial state condition and a two-state transition relation. These transition systems are then expressed as first-order formulas in one of the theories supported by the back-end SMT solver(s). Given the state of the art in SMT, without any compositional mechanism, an analysis of the Longitudinal Controller is currently impossible. The reason is that the system features nonlinear arithmetic constraints (with multiplications and divisions) which are very challenging for today's SMT solvers. On a system the size of this one, all solvers we tried give up as soon as we unroll the transition relation once, and return *unknown*.

The first step towards verifying any contract in the Autopilot system was thus for us to abstract the nonlinear expressions in it. To do so, we manually replaced nonlinear applications of the `*` and `/` Lustre operators with applications of Lustre nodes, written by us, meant to abstract those operators. For instance, an expression of the form $s * t$ with s and t of type `real` would be abstracted by `times(s,t)` where `time` is the node

```
node times( x, y : real ) returns ( z : real )
let
  z = x * y ;
tel
```

The abstracting nodes were provided with a contract specifying salient algebraic, and linear, properties of the abstracted operators, such as the existence of neutral and absorbing elements, sign, and proportionality. Because we isolated the nonlinear expressions, we were able to get the SMT solvers to prove these contracts. This allowed us to use just the contracts, which are on purpose weaker than the full implementation of the multiplication and division nodes, in the analysis of components using those nodes. As a nice side effect, by adding in the contract for the division node `divid` the assumption that the denominator argument is nonzero, we also got the analysis to check that no division by zero can happen in the system.

Armed with this sort of abstraction, we wrote contracts for the Longitudinal Controllers and its two subcontroller based on Guide 120 and 130 of the FAR requirements. A major challenge was that the output of the Altitude controller feeds also into the FPA controller, even when the former is disengaged. We thus had to write a contract for the Altitude Controller to specify its behavior even when it is not engaged. Now, the output in question is the result r of a nonlinear division of two values n and d, and is supposed to be within certain bounds. Our generic abstraction of division did not have a strong enough contract to guarantee that. However, the way the system is defined, when the Altitude Controller is disengaged both n and d are themselves bounded. So, we designed a custom abstraction for division, `divid_bounded_num`, which takes as input constant upper and lower bounds on the denominator and has a contract that extend our generic one for division with modes specifying that the result is within in an interval:[11]

```
contract divid_bounded_num( num, den: real ; const lbound, ubound: real )
returns ( res: real ) ;
let
  ...
  assume dem <> 0.0 and lbound <> 0.0 and u_bound <> 0.0 ;
  assume lbound <= den and den <= ubound ;
  ...
  mode num_pos_lbound_pos (
    require 0.0 <= num ; require 0.0 < lbound ;
    ensure  num/ubound <= res and res <= num/lbound ;
  ) ;
tel
```

There are six modes like `num_pos_lbound_pos` in the full contract, depending on the sign of the numerator and how the denominator compares to zero. Using this version of division we were able to prove that the output of the Altitude Controller is indeed within the expected bounds when the controller is disengaged.

Compositional analysis. Due to the nonlinearities discussed above, KIND 2 is unable to perform a monolithic analysis of the Longitudinal Controller subsystem, that is, one that looks at the subsystem as a whole, ignoring that it is the composition of several components. Hence, we evaluated the compositional approach by comparing a *linearized-monolithic* analysis, where only the nonlinear expressions are abstracted, with a compositional one, where the two Altitude and the FPA subcontrollers are abstracted.

[11] Full contracts for times, divid, and divid_bounded_num are available on the case study website.

```
contract logic_alt_fpa(...) returns (...) ;
let
  mode alt_170 (...) ; mode alt_210 (...) ;
  mode fpa_180 (...) ; mode fpa_210 (...) ;
tel
```

```
contract mode_logic (...) returns (...) ;
let
  import logic_alt_fpa (...) returns (...) ;
tel
```

```
contract logic_longitudinal (
  head_engage, alt_engage, fpa_engage: bool ; alt, alt_target, hdot,
  fpa, fpa_target, pitch, speed, gskts, cas, elev, ail: real ;
) returns (
  head_engaged, alt_engaged, fpa_engaged: bool ; out_alt, out_pitch, out_elev: real ;
) ;
let
  import mode_logic (
    head_engage, alt_engage, fpa_engage, elev <> 0.0 or ail <> 0.0, alt, alt_target
  ) returns ( head_engaged, alt_engaged, fpa_engaged ) ;
  import longitudinal (
    ::mode_logic::logic_alt_fpa::alt_170 or ::mode_logic::logic_alt_fpa::alt_210,
    ::mode_logic::logic_alt_fpa::fpa_180 or ::mode_logic::logic_alt_fpa::fpa_210,
    alt, alt_target, hdot, fpa, fpa_target, pitch, speed, gskts, cas, elev
  ) returns ( out_alt, out_pitch, out_elev ) ;
tel
```

Fig. 6: A sketch of the contract for the Autopilot node.

Both analysis were successful, but with no appreciable difference: they both terminate in a matter of seconds. This is not surprising because the implementation of those subcontrollers is not a lot more complex than their contract. In contrast, we did see a significant difference between the linearized monolithic analysis and the compositional one when we analyze the Autopilot system, as we explain next.

5.3 Verifying Autopilot

To verify the full Autopilot we wrote contracts also for its Mode Logic subsystem. The pertinent FAR requirements for that subsystem are Guide 170, 180, and 210, which specify how and when the altitude and the FPA controllers supposed to engage. We will not go over the contracts of Mode Logic here but describe instead our experience in verifying its composition with the Longitudinal Controller in the Autopilot system.

Before that, it is worth noting that during the verification of the Mode Logic component we found a bug in the Lustre model. The bug occurs when the input signals respectively enabling the Altitude and the FPA controller go from true to false at the same time.[12] In that case, the output flag for the controller that has been given priority by Mode Logic will become true, as expected, but then alternate between true and false at every step afterwards. Since fixing the model was beyond our level of expertise, we side-stepped the problem for this case study by adding a `require` clause stating that the two input signals never fall together.

Contracts for high-level components like Autopilot can be expressed in terms of the contracts for their subcomponents. Overall we found that lifting subcomponent contracts to their calling component is relatively straightforward thanks to the contract import feature discussed in Section 3. This feature is often flexible enough to let one write parametric contracts that can be adapted, by instantiation, to nodes with similar

[12] This is possible in principle if these signals come from distinct physical on/off buttons, as opposed to a switch, that are released at the same time.

behavior. In the case of the Autopilot node, its contract can be created by importing and suitably connecting the contracts of its Mode Logic and Longitudinal Controller subcomponents, as illustrated in Figure 6. Note that the two first parameters of the `longitudinal` import refer to the modes of the `mode_logic` contract to communicate whether the Altitude or the FPA controller is active. Reusing the contracts of the two subsystems through imports to write the contract for the Autopilot component reduces the duplication of specs across the overall system. This improves user-friendliness, maintainability and, hence, trust in the correctness of the specs.

There is still, however, room for errors in the contracts themselves. Mode information helps fix those errors that cause the contract to be falsifiable. Once a contract is proved, the exploration of reachable modes is again an invaluable tool to make sure all the modes can actually be activated, and that the system and the contract behave as expected, at least up to the explored depth of the reachability graph.

Compositional versus linearized-monolithic. The Autopilot system is rather complex. Recall that the Mode Logic component decides which controller is engaged based on information arbitrarily far in the past because of the request mechanism. Its outputs control the mutually-exclusive activation of the two subsystems of the Longitudinal Controller. Moreover, these subsystems are not independent as the FPA Controller takes as input the output of the Altitude Controller.

A monolithic analysis of this system in KIND 2 is again impossible because of the nonlinear expressions in the Longitudinal Controller subsystem, as discussed in Section 5.2. We therefore compared a linearized-monolithic analysis of Autopilot with a compositional one. The former could discharge some of the proof obligations generated for the Autopilot contract, but was overall inconclusive after running for one hour on an i7 (2014) CPU running Mac OS X. The compositional analysis, on the other hand, was able to prove the entire contract of the Autopilot node and all the proof obligations for the calls to its subcomponents in about 80 seconds.

We also had KIND 2 run a *full* analysis on Autopilot. As explained in Section 4, KIND 2 does that automatically by going through the hierarchy of nodes in a Lustre model bottom-up, and running a compositional analysis on each of them, where immediate subcomponents with contracts are abstracted by their contract. This guarantees that every node with a contract is correct, in the sense that it respects its contract as well as all the assumptions, if any, of the nodes it calls. The overhead of checking the correctness of *all* the subcomponents of Autopilot is minimal. The total runtime for this analysis, *including the nonlinear abstractions*, was under 100 seconds.

6 Conclusion

We described COCOSPEC, a mode aware assume-guarantee-based contract language for the specification of synchronous reactive systems. The starting point of COCOSPEC was the need to have a contract language able to accurately capture the behaviors of embedded systems. COCOSPEC is currently designed as an extension of the synchronous dataflow language Lustre. We have described COCOSPEC's main benefits, including *i)* bringing the specification language closer to the specification documents, *ii)* enabling

defensive semantics checking of the specification for oversights, *iii)* allowing more effective and more scalable compositional analyses, and *iv)* providing better feedback for fault localization. In addition to these direct benefits come features such as the exploration of reachable modes or the formulation of properties about the specification (by referring to mode requirements). This allows a mode-aware tool supporting CoCoSpec to provide several means to raise trust in the specification.

We added full support for CoCoSpec to the Lustre model-checker Kind 2. We demonstrated the usefulness of compositional reasoning in the context of CoCoSpec by applying it successfully to the TCM, a flight-critical system case study which, due to its realistic functionality, size, and complexity, is not amenable to monolithic analyses.

Future Work Kind 2 is also able to generate a concrete trace of inputs for each path in the tree of reachable modes. We conjecture that, by exploring the reachable modes of a contract, it is possible to generate specification-based test cases which are of better quality than those produced by syntactic test generation techniques. This is particularly relevant for outsourced components, which are often provided by subcontractors in executable form only. For such components, test cases are the only means to verify contract compliance. We plan to evaluate our conjecture experimentally in future work.

References

1. Backes, J., Cofer, D.D., Miller, S.P., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - 7th International Symposium, NFM 2015. Lecture Notes in Computer Science, vol. 9058. Springer (2015)
2. Barnes, J.G.P.: High Integrity Software - The SPARK Approach to Safety and Security. Addison-Wesley (2003)
3. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification, 20th International Conference, CAV 2008. Lecture Notes in Computer Science, vol. 5123. Springer (2008)
4. Brat, G., Bushnell, D.H., Davies, M., Giannakopoulou, D., Howar, F., Kahsai, T.: Verifying the safety of a flight-critical system. In: Bjørner, N., de Boer, F.S. (eds.) FM 2015: Formal Methods - 20th International Symposium, 2015. Lecture Notes in Computer Science, vol. 9109. Springer (2015)
5. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification, 28th International Conference, CAV 2016. Lecture Notes in Computer Science, Springer (2016), (To appear)
6. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. 21(4) (2012)
7. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: Cortellessa, V., Muccini, H., Demirörs, O. (eds.) 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012. IEEE Computer Society (2012)
8. Cofer, D.D., Gacek, A., Miller, S.P., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012. Lecture Notes in Computer Science, vol. 7226. Springer (2012)

9. Dieumegard, A., Garoche, P., Kahsai, T., Taillar, A., Thirioux, X.: Compilation of synchronous observers as code contracts. In: Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015. ACM (2015)
10. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
11. Halbwachs, N., Fernandez, J.C., Bouajjanni, A.: An executable temporal logic to express safety properties and its connection with the language lustre. In: Sixth International Symposium on Lucid and Intensional Programming, ISLIP 1993 (1993)
12. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. IEEE Trans. Software Eng. 18(9) (1992)
13. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) Algebraic Methodology and Software Technology (AMAST), Proceedings of the Third International Conference on Methodology and Software Technology, 1993. Workshops in Computing, Springer (1993)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
15. Hueschen, R.M.: Development of the Transport Class Model (TCM) aircraft simulation from a sub-scale Generic Transport Model (GTM) simulation. Tech. rep., NASA, Langley Research Center (2011)
16. Jézéquel, J., Meyer, B.: Design by contract: The lessons of Ariane. IEEE Computer 30(1) (1997)
17. Jones, C.: Development Methods for Computer Programs including a Notion of Interference. Ph.D. thesis, Oxford University (1981)
18. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Barnat, J., Heljanko, K. (eds.) Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011. EPTCS, vol. 72 (2011)
19. Kamp, J.: Tense Logic and the Theory of Order. Ph.D Thesis, UCLA (1968)
20. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3) (2015)
21. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, The Kluwer International Series in Engineering and Computer Science, vol. 523. Springer (1999)
22. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355. Springer (2010)
23. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. Springer (1995)
24. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME 1999. Lecture Notes in Computer Science, vol. 1703. Springer (1999)
25. Meyer, B.: Applying ”design by contract”. IEEE Computer 25(10) (1992)
26. Parnas, D.L.: Inspection of safety-critical software using program-function tables. In: Duncan, K.A., Krueger, K.H. (eds.) Linkage and Developing Countries, Information Processing, 1994, IFIP Transactions, vol. A-53. North-Holland (1994)