OCRA: Othello Contracts Refinement Analysis
Version 2.0 *

Abstract

Contract-based design enriches a component model with properties structured in pairs of assumptions and guarantees. These properties are expressed in term of the variables at the interface of the components, and specify how a component interacts with its environment: the assumption is a property that must be satisfied by the environment of the component, while the guarantee is a property that the component must satisfy in response. Contract-based design has been proposed in many methodologies for taming the complexity of embedded systems. In fact, contract-based design enables stepwise refinement, compositional verification, and reuse of components.

OCRA (Othello Contracts Refinement Analysis) is a tool that provides means for checking the refinement of contracts specified in a linear-time temporal logic. The specification language allows to express discrete as well as metric real-time constraints. The underlying reasoning engine allows checking if the contract refinement is correct. OCRA has been used in different projects and integrated in tools.

This document provides a manual on how to use OCRA, specifically its input language and its commands.

Contents

*For a bug report, a feature request or other suggestions, please send email to `ocra@list.fbk.eu`.

1 Introduction

OCRA is a tool that provides automated support for contract-based design with temporal logics. OCRA relies on the contract framework originally proposed in [CT12, CT], where assumptions and guarantees are specified as temporal formulas.

The main functionality of OCRA is the verification of the contract refinement. OCRA checks if a given contract refinement is correct, generating the corresponding set of proof obligations and checking their validity. OCRA takes as input a textual description of the components interfaces and their decomposition into sub-components, the components' contracts and their refinement with the contracts of the sub-components. The OCRA input file, also called OSS (OCRA System Specification), describes a tree of components (given by the decomposition into sub-components), which represents the system architecture.

The contracts are specified in Othello [CRST12], a human-readable language which can be mapped to temporal formulas in the HRELTL [CRT09], and thus represent sets of hybrid traces. The contract refinement is however independent from the nature of the traces and OCRA provides an option to interpret the contracts over discrete-time traces restricting the input to forbid continuous ports and allow LTL [Pnu77] contracts only.

In order to prove the validity of the proof obligations deriving from contract refinement, OCRA interacts with NuSMV3 [NuS], a framework that includes nuXmv [nuX], HyCOMP [HyC], and xSAP [XSA]. NUSMV3 provides the functionality to either prove that the formulas are valid, or to find counterexamples, which can be inspected by the user in order to find the bugs in the contract refinement. When the contracts are written in the standard discrete-time LTL, to prove or disprove the validity of the proof obligations the BDD-based engine is used. In the general case of HRELTL, reasoning relies on Satisfiability Modulo Theory (SMT). Since logical entailment for HRELTL is undecidable, bounded model checking techniques are used to find counterexamples to the contract refinement [CRT09, CT12].

OCRA has been developed within the European project SafeCer [Saf], focusing on the compositional certification of embedded systems. The tool is publicly available [OCR].

2 Input language[1]

2.1 Components

The main input of OCRA is a textual specification of a system architecture and the related contract refinement. The format of this input is the OCRA System Specification (OSS), which contains the specification of components, their ports and contracts, and their decomposition. Each component specification starts with the keyword **`COMPONENT`** and ends with the following **`COMPONENT`** or with the end of the input. For example, the OSS shown in Figure 1 contains three component specifications.

Each component specification contains a name, an interface part (introduced with the **`INTERFACE`**), and optionally a refinement part (introduced with the keyword **`REFINEMENT`**). A component without refinement is called a leaf component. In the interface part, the ports, the parameters, and the contracts are declared. In the refinement part, the subcomponents are declared and the connections and constraints among the ports of the component and subcomponents are specified. Finally, still in the refinement part, the refinement of the component contracts is also specified. In the following section, we detail the syntax of these elements.

One (and only one) of the component specifications is tagged with the keyword **`system`**. We refer to this component as the system component. The system architecture is a tree of component instances where the root is an instance of the system component and each node is either an instance of a leaf component and has no children or has as children the instances specified as subcomponents in the refinement. The system architecture of the example in Figure 1 has three nodes, if `simple_instance` is the name of the root, then `simple_instance.a` and then `simple_instance.b` are the leaves.

[1] The OCRA data types and operators are based on the NUXMV input language. See `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf` for a detailed description.

```
COMPONENT example1 system
 INTERFACE
  INPUT PORT in_data: boolean;
  OUTPUT PORT out_data: boolean;

  CONTRACT reaction
   assume: in the future in_data;
   guarantee: always (in_data implies in the future out_data);

 REFINEMENT
  SUB a: A;
  SUB b: B;

  CONNECTION a.in_data := in_data;
  CONNECTION b.in_data := a.out_data;
  CONNECTION out_data:= b.out_data;

  CONTRACT reaction REFINEDBY a.reaction, b.pass;


COMPONENT A
 INTERFACE
  INPUT PORT in_data: boolean;
  OUTPUT PORT out_data: boolean;

  CONTRACT reaction
   assume: in the future in_data;
   guarantee: always (in_data implies in the future out_data);


COMPONENT B
 INTERFACE
  INPUT PORT in_data: boolean;
  OUTPUT PORT out_data: boolean;

  CONTRACT pass
   assume: true;
   guarantee: always (in_data implies out_data);
```

Figure 1: Simple OSS example

2.2 Hybrid vs. Discrete Time

The semantics of components is defined in terms of traces (see Section B for a formal definition). Traces may be discrete where the model of time is the set of natural numbers or hybrid which combines discrete changes with continuous evolution of time (the time model in this case is the subset of $\mathbb{N} \times \mathbb{R}$ containing $\langle 0, 0 \rangle$ and such that for all $\langle i, t \rangle$, $\langle i', t' \rangle$, if $i < i'$ then $t \leq t'$). The default model is hybrid. In the discrete-time case, the language is syntactically restricted (see Section 2.3.1 and 2.5). The user can require to interpret the model with discrete time by writing at the beginning of the model:

```
@requires discrete-time
```

2.3 Interface

2.3.1 Ports

The component interface contains a (possibly empty) set of port declarations. Each port is declared with the keyword **`PORT`**, preceded by the direction (**`INPUT`**/**`OUTPUT`**) and followed by the name and the type. Allowed types are **`boolean`**, **`integer`**, **`real`**, **`event`**, **`continuous`**, **`signed word`**, **`unsigned word`**, range of integers, list of integers or symbolic constants. In the discrete-time case, **`continuous`** type cannot be used.

2.3.2 Parameters

A component interface may contain also parameters. These are introduced with the **PARAMETER**. The type of parameters can be **boolean**, **integer**, **real**, **signed word**, **unsigned word**, range of integers, list of integers or symbolic constants and uninterpreted functions. Parameters are like input port that do not change value along time. The parameters of subcomponents can be set by the connection (see the example in Figure 3). Therefore the same component can be instantiated with different parameters. Moreover, the value may depend on the parameters of the father component.

Parameters can be declared with the uninterpreted function type. The following code declares a function with two integer arguments and an integer return value:

```
PARAMETER f : integer * integer -> integer;
```

In contracts function will be referenced through calls, with the standard syntax:

```
guarantee : always f(0, 0) > 0;
```

2.3.3 Operations

A special kind of port is called operation. A component can provide an operation or require it from another one. Operations can be seen as a collection of ports: if an operation is declared in a component in the way shown below,

```
PROVIDED OPERATION PORT P (a : boolean, b : real) : boolean;
```

several ports are implicit declared and can be referenced:

```
INPUT P_call : event -- the event corresponding to the call of the operation
INPUT P_a_param : boolean -- the first parameter
INPUT P_b_param : real -- the second parameter
OUTPUT P_ret : event -- the event corresponding to the operation returning
OUTPUT P_ret_value : boolean -- the return value
```

Therefore, the form of an operation is much like a function in a traditional programming language. However, there is no semantic constraints on the implicit ports. A specific semantics should be enforce by means of constraints.

If no return value port is declared if the return type of the operation is void. As for the other ports, if a component is refined in subcomponents, it is possible to define the connections between the operations.

2.3.4 Defines

A define is a kind of macro. Every time a define is met in expressions, it is substituted by the expression associated with this define. Therefore, the type of a define is the type of the associated expression in the current context. Define expressions may contain **next** operators; normal rules apply: no nested **next** operators.

A define is not a port or a parameter, so it cannot be used in the left hand side of a **CONNECTION**.

2.3.5 Contracts

A component interface may contain a set of contracts. Each contract is introduced with the keyword **CONTRACT**, followed by the name of the contract and two constraints (see also Section 2.5) representing an assumption and a guarantee. The assumption is introduced with the keyword **assume** and represents a property that must be satisfied by the environment of the component. The guarantee is introduced with the keyword **guarantee** and represents a property that must be satisfied by the implementation of the component provided that the assumption holds.

2.3.6 Assertion

A component interface may contain a set of assertions. This specifications are like contract assumptions and guarantees. In order to create a specification we use the keywords **ASSERTION NAME** specification name := constraint.

2.4 Refinement

2.4.1 Refinement language

It is possible to declare variables local to the refinement of a component, not visible from its interface and consequently from the father components. They comes in handy when modelling **CONSTRAINT**s and refinement behaviour. These symbols are not ports and do not have a direction (input or output). Also they cannot have type **event**.

The syntax to declare them is **VAR** followed by an identifier and the type. For an example see Figure 4 below.

2.4.2 Subcomponents

The component refinement contains a set of subcomponent declarations. Each subcomponent is introduced with the keyword **SUB** followed by the name and the type. The type must be the name of another component. However, the type cannot be the refined component or any ancestor in the tree structure of the system architecture.

2.4.3 Connections

The component refinement contains a (possibly empty) set of connections. Each connection connects the output of the component or the input of the subcomponents to an expression over the inputs of the component and the outputs of the subcomponents. The connection is introduced with the keyword **CONNECTION**[2] followed by the connected port and the expression. The ports of subcomponents are referred to with the the dot notation (name of the subcomponent followed by "." followed by the name of the port). The type of the port and the expression must be the same. So, if the connected port is **real**, then also the expression must be a **real** expression (see the example in Figure 2).

Remark 1 *The presence of Boolean or arithmetic connectors in the connection may be seen as not a principled design choice, because it hides a component that implements such connector.* OCRA *supports it for flexibility but the user should consider the alternative model where additional components are introduced and all connections are atomic, just connecting one port to another port.*

Remark 2 *In a principled design, every output port of a refined component and every input port of its subcomponents should be connected by a connection. If not, such port is called dangling.* OCRA *warns the user about the presence of such ports but they are supported. The semantics of dangling ports is that their value is non-deterministically chosen. Note that the same behavior can be obtained by adding a subcomponent that generates such non-deterministic value.*

2.4.4 Constraints

The component refinement contains a (possibly empty) set of temporal constraints (see also Section 2.5) on the ports of the component and the ports of the subcomponents. Such constraints are introduced with the keyword **CONSTRAINT**. Constraints may be useful to model special kind of connections, such lossy channels or delays (see the example in Figure 3).

Remark 3 *Also in this case, the use of constraint may be avoided by introducing a component implementing the special connection (see also Remarks 1 and 2).*

2.4.5 Behaviour

OCRA provides a set of keywords to allow the modeller to specify a (in)finite state machine to control the refinement of a component. They respond to the same purpose of the **CONSTRAINTS**s, but are more expressive. Their argument is not an OTHELLO formula but a subset of them: an SMV expression. In particular SMV expressions lacks temporal operators (See `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf` for a detailed description). See figure 4 for an example.

- **INIT**: it determines the set of initial states. **next** is not allowed.

[2] In the earlier version of the language, connections were named **DEFINE**. This is no more supported.

```
COMPONENT example2 system
 INTERFACE
  INPUT PORT in_data: real;
  OUTPUT PORT positive: boolean;

  CONTRACT positive
   assume: always (in_data>=0);
   guarantee: always positive;

 REFINEMENT
  SUB a: A;
  SUB b: B;

  CONNECTION b.in_data := in_data + a.out_data;
  CONNECTION positive := b.out_data>0;

  CONTRACT positive REFINEDBY a.positive, b.past;


COMPONENT A
 INTERFACE
  OUTPUT PORT out_data: real;

  CONTRACT positive
   assume: true;
   guarantee: always (out_data>0);


COMPONENT B
 INTERFACE
  INPUT PORT in_data: real;
  OUTPUT PORT out_data: real;

  CONTRACT past
   assume: true;
   guarantee: always (out_data<=0 implies in the past in_data<=0);
```

Figure 2: Example with arithmetic expressions in the connections

- **TRANS**: it declares the transition relation.
- **INVAR**: specifies the set of invariant states. **`next`** is not allowed.
- **FAIRNESS**: a fairness constraint restricts the attention only to fair execution paths. When evaluating specifications, OCRA considers path quantifiers to apply only to fair paths. **`next`** is not allowed.

Remark 4 *The use of behavioural formulas may be avoided by introducing a component implementing the special connections (see also Remarks 1, 2 and 3).*

2.4.6 Assertion

The behavior is the same of the interface part, but in the refinement part the assertions can refer to elements that are contained in a subcomponent.

2.4.7 Validation Properties

The component refinement contains a (possibly empty) set of validation properties on a contract refinement. There are three types of validation properties that can be defined: *consistency*, *possibility*, and *entailment*. In their definitions, formula ids are used to make reference to assumption, guarantee, and norm guarantee [3] of the contracts of components. A formula id is made up of a component name, a contract name and one of the keyword: **`ASSUMPTION`**, **`GUARANTEE`**, or **`NORM_GUARANTEE`**. These three parts are separated using the dot notation (see below an example).

[3] Norm guarantee is defined as $A \rightarrow G$, where A is the assumption and G is the guarantee of a given contract.

```
COMPONENT example3 system
 INTERFACE
  INPUT PORT x: real;
  OUTPUT PORT alarm: event;
  PARAMETER min_period: real;
  PARAMETER threshold: real;
  CONTRACT alarm
   assume: always (change(x) implies then time_until(change(x))>min_period);
   guarantee: always ((x<=threshold) implies time_until(alarm)<=10);
 REFINEMENT
  SUB d: Device;
  SUB m: Monitor;
  CONNECTION d.x := x;
  CONNECTION alarm := m.out_alarm;
  CONNECTION m.period := (min_period<=2)?min_period:2;
  CONNECTION d.threshold := threshold;
  CONSTRAINT always (m.request implies time_until(d.request)<=3);
  CONSTRAINT always (d.alarm implies time_until(m.in_alarm)<=3);
  CONTRACT alarm REFINEDBY d.alarm, m.alarm, m.request;
  CONSISTENCY NAME c1 := example3.alarm.ASSUMPTION, d.alarm.ASSUMPTION, m.request.ASSUMPTION;
  POSSIBILITY NAME p1 := example3.alarm.GUARANTEE GIVEN d.alarm.ASSUMPTION, m.request.ASSUMPTION;
  ENTAILMENT  NAME e1 := m.request.GUARANTEE BY m.request.ASSUMPTION;

COMPONENT Device
 INTERFACE
  INPUT PORT x: real;
  INPUT PORT request: event;
  OUTPUT PORT alarm: event;
  PARAMETER threshold: real;
  CONTRACT alarm
   assume: true;
   guarantee: always ((request and x<=threshold) implies time_until(alarm)<=1);

COMPONENT Monitor
 INTERFACE
  INPUT PORT alarm: event;
  INPUT PORT in_alarm: event;
  OUTPUT PORT request: event;
  OUTPUT PORT out_alarm: event;
  PARAMETER period: real;
  CONTRACT request
   assume: true;
   guarantee: (time_until(request)<=period) and
              (always (request implies then time_until(request)<=period));
  CONTRACT alarm
   assume: true;
   guarantee: always (in_alarm implies time_until(out_alarm)<=1);
```

Figure 3: More complex OSS example

Let us now explain the three type of validation properties:

- *Consistency:* this type of validation property aims at formally verifying the absence of logical contradictions of a given subset of properties (assumption, guarantee, and norm guarantee of the contracts on a refined component and its subcomponents). It is possible to define a consistency property in the following form: **CONSISTENCY NAME** name := $prop_0, \ldots, prop_n$, where $prop_0, \ldots, prop_n$ is a list of formula ids.
- *Possibility:* this kind of validation property allows to verify whether a property is consistent with a set of other properties specified in the contracts. It is possible to define a possibility property in the following form: **POSSIBILITY NAME** name := $prop_0$ **GIVEN** $prop_1, \ldots, prop_n$, where $prop_0$ can be a formula or a formula id, and $prop_1, \ldots, prop_n$ is a list of formula ids.
- *Entailment:* this type of validation property aims at verifying whether an expected property is implied by a set of other properties specified in the contracts. It is possible to define a possibility property in the form: **ENTAILMENT NAME** name := $prop_0$ **BY** $prop_1, \ldots, prop_n$, where $prop_0$ can be a formula or a formula id, and $prop_1, \ldots, prop_n$ is a list of formula ids.

```
COMPONENT example4 system
 INTERFACE
  INPUT PORT x: real;
  OUTPUT PORT alarm: event;
  PARAMETER min_period: real;
  PARAMETER threshold: real;
  CONTRACT alarm
   assume: always (change(x) implies then time_until(change(x))>min_period);
   guarantee: always ((x<=threshold) implies time_until(alarm)<=10);

 REFINEMENT
  VAR local : boolean;
  SUB d: Device;
  SUB m: Monitor;
  CONNECTION d.x := x;
  CONNECTION alarm := m.out_alarm;
  CONNECTION m.period := min_period;
  CONNECTION d.threshold := threshold;

  TRANS m.request implies next(d.request)
  TRANS d.alarm implies next(d.in_alarm)
  INVAR min_period > 2 implies m.period = 2

  FAIRNESS not alarm;

  CONTRACT alarm REFINEDBY d.alarm, m.alarm, m.request;

COMPONENT Device
 INTERFACE
  INPUT PORT x: real;
  INPUT PORT request: event;
  OUTPUT PORT alarm: event;
  PARAMETER threshold: real;
  CONTRACT alarm
   assume: true;
   guarantee: always ((request and x<=threshold) implies time_until(alarm)<=1);

COMPONENT Monitor
 INTERFACE
  INPUT PORT in_alarm: event;
  OUTPUT PORT request: event;
  OUTPUT PORT out_alarm: event;
  PARAMETER period: real;
  CONTRACT request
   assume: true;
   guarantee: (time_until(request)<=period) and
              (always (request implies then time_until(request)<=period));
  CONTRACT alarm
   assume: true;
   guarantee: always (in_alarm implies time_until(out_alarm)<=1);
```

Figure 4: The example in figure 3 rewritten with behavioural statements

As we mentioned above, a formula id is made up of a component name, a contract name and one of the keyword: **ASSUMPTION**, **GUARANTEE**, or **NORM_GUARANTEE**. These three parts are separated using the dot notation. For example, the formula id corresponding to the `assumption` of the contract `alarm` on component `example3` in Figure 3 is denoted as: example3.alarm.ASSUMPTION. We remark that formula ids used in the definition of any validation properties can only make reference to contracts on the component where is defined the validation property and its subcomponents.

2.4.8 Asynchronous vs. Synchronous Refinement

The semantics of the composition of subcomponents is synchronous. This means that subcomponents progress simultaneously. At every discrete step or time evolution, every component behaves as specified by the contracts. It is possible to specify an asynchronous composition by using the keyword **ASYNC** in front of the keyword **REFINEMENT**. In this case, at every discrete step, each component either behaves as specified by the contracts or *stutters*. The relationship between the stuttering of one component and the other events must be specified with a **CONSTRAINT** using

the implicit event **stutter**. For example, a constraint may force a component to stutter when the other component it is performing an event. See the examples in the distribution of the tool. By default, no fairness is introduced to avoid endless stuttering of a component. By setting the option ocra_async_fairness the tool will introduce such constraint.

2.4.9 Contracts refinement

In the component refinement, each contract present in the component interface should be refined by some contracts of the subcomponents. This relationship is introduced with the keyword **REFINEDBY** preceded by the name of the refined contract followed by a list of contracts of the subcomponents. These are referred to with the the dot notation (name of the subcomponent followed by **"."** followed by the name of the contract).

2.5 Constraint language

OCRA input language uses OTHELLO [CRST12] to specify contracts. In the discrete-time case, it coincides with LTL and is interpreted over discrete traces.

The relevant syntax of OTHELLO has been summarized in Table **??** together with the corresponding mathematical formulation in HRELTL (see Section B for a formal definition of syntax and semantics).

In case of discrete time, **der**, **time_until**, and **time_since** are not allowed.

constraint	:=	*atom* |	ϕ	:=	a |
		not *constraint* |			$\neg\phi$ |
		constraint **and** *constraint* |			$\phi \wedge \phi$ |
		constraint **or** *constraint* |			$\phi \vee \phi$ |
		constraint **implies** *constraint* |			$\phi \rightarrow \phi$ |
		always *constraint* |			$G\phi$ |
		never *constraint* |			$G\neg\phi$ |
		in the future *constraint* |			$F\phi$ |
		constraint **until** *constraint*;			$\phi\, U\phi$;
		then *constraint* |			$X\phi$ |
		historically *constraint* |			$H\phi$ |
		in the past *constraint* |			$P\phi$ |
		constraint **since** *constraint*;			$\phi\, S\phi$;
		term **at next** *constraint*;			$t\ @F\phi$;
		term **at last** *constraint*;			$t\ @P\phi$;
atom	:=	**true** |	a	:=	$\top$ |
		false |			$\bot$ |
		term relation term |			$t \bowtie t$ |
		time_until(*term* **)** *relation term* |			$\rhd_{\bowtie t} t$ |
		time_since(*term* **)** *relation term* |			$\lhd_{\bowtie t} t$ |
		change(*term* **)** |			$v' \neq v$ |
		fall(*boolean_term*) |			$\phi \wedge \neg X\phi$ |
		rise(*boolean_term* **)** |			$\neg\phi \wedge X\phi$ |
		boolean_term ;			t;
term	:=	*port* |	t	:=	v |
		constant |			c |
		term function term |			$t \star t$ |
		der(*port* **)** |			$\dot{v}$ |
		next(*port* **)** ;			v';

Basic formulas are defined with linear arithmetic predicates over the variables or their derivatives. For examples, `x-e<limit` and **`der`**`(x)<0` are well-defined formulas. Predicates can be combined with Boolean and temporal operators. For example, `x-e<limit` **`and der`**`(x)<0` and **`always`** `x-e<limit` are well-defined formulas.

In temporal logic, a formula without temporal operators is interpreted in the initial state. Thus, `x=0` characterizes all traces that start with a state evaluating `x` to 0, and then `x` can evolve arbitrarily. Instead, to express that a predicates holds along the whole evolution, one may use the **`always`** operator as in **`always`** `x=0`.

Another classical example of properties is the response to a certain event. The formula **`always`** `(p` **`implies in the future`** `q)` defines the set of traces where every occurrence of `p` is followed by an occurrence of `q`. Note that `q` may happen with a certain delay (although there is no bound on such delay). The formula **`always`** `(p` **`implies`** `q)` instead forces `q` to happen at the instant of `p`.

The above formulas do not constrain the time model of the traces. Therefore, they can be interpreted either as discrete traces or as hybrid traces. However, the logic is suitable to characterize specific sets of hybrid traces, constraining when there should be discrete events and how the continuous variables should evolve along continuous evolutions.

The **`der`**`(.)` operator is used to specify constraints on the derivative of the continuous evolution of continuous variables. For example, the following OTHELLO constraint:

```
always (train.location<=target implies der(train.location)>=0)
```

characterizes the set of hybrid traces where in all states, if the train has not yet reached the target location, its speed (expressed as the derivative of the location) is greater than or equal to zero.

The **`next`**`(.)` operator is used to specify functional properties requiring discrete changes to variables. For example, we can express the property that the warning variable will change value after the train's speed passes the limit with the following constraint:

```
always (speed>limit implies
                    in the future next(warning)!=warning)
```

The expression **`change`**`(x)` can be used instead of **`next`**`(x)!=x`.

In order to constrain the delay between two events, we use the **`time_until`**`(.)` and **`time_since`**`(.)` operators, which denote respectively the time that will elapse until the next occurrence of an event and the time that elapsed since the last occurrence of an event. For example, the formula **`always`** `(p` **`implies time_until`**`(q)<max_delay)` defines the set of hybrid traces where `p` is always followed by `q` in less than `max_delay` time units.

The operator **`. at next .`** has a similar but more general purpose. It denotes the value of the left expression, known as the sample, at the next state (excluding the current one) in which the right expression, known as the trigger, will be true. For example, the formula **`always`** `(p` **`implies`** `(((time` **`at next`** `q)- time)< max_delay))` is the discrete time equivalent of the previous example, using an explicit user defined time variable. There is also an **`. at last .`** operator, which denotes the value of the sample at the last state (excluding the current one) in which the trigger was true (see [Ton17] for further details).

3 Verification of Contracts Refinement

The contract refinements specified in the OSS be seen as a set of proof claims that are resolved by the tool. The claim say 1) that if the subcomponents satisfy their contracts, then also the composite component satisfies its contracts, and 2) that if the subcomponents satisfy their contracts and the environment of the composite component satisfies the related assumptions, then also the assumptions of each subcomponents are satisfied.

The OCRA command to check the refinement of contracts is `ocra_check_refinement`. See Section 7 for a complete description of options. The command internally processes the input, generates the proof obligations corresponding to the proof claims, and check their validity. It reports the results to standard output, detailing for each proof claim if the claim is correct and if not, it produces a counterexample trace witnessing that the claim is wrong.

A typical usage of the tool is the following. First, the user writes the OSS in a textual file. The typical suffix used for these files is `.oss` (for example `example1.oss`). Emacs users can use the OCRA mode for syntax highlighting (see `ocra-mode.el` file provided with the tool distribution). Second, OCRA must be run in interactive mode with the command `ocra -int example1.oss`. Before checking the refinement, the user may check the syntax of the

input by calling `ocra_check_syntax`. OCRA provides also a simple validation check that verifies if each constraint in the contracts is satisfiable. This can be performed with the command `ocra_check_consistency`. Finally, after syntax checking and validation, the user may run the actual verification with the command `ocra_check_refinement`. The overall sequence therefore is:

```
$ ocra -int example1.oss
ocra > ocra_check_syntax
ocra > ocra_check_consistency
ocra > ocra_check_refinement
```

4 Integration with NUXMV and HYCOMP for Compositional Modeling and Verification

OCRA is tightly integrated with the NUXMV and HYCOMP model checkers to model and verify behavioral models that implement the OCRA components; NUXMV is used for discrete time models, while HYCOMP handles the hybrid time ones. The behavior of components can therefore described in SMV (the input language of NUSMV and NUXMV) or HyDI (the input language of HYCOMP, an extension of SMV). In SMV/HyDI the behavior is described by means of logical formulas that describe the initial states and the state transitions. An SMV/HyDI specification can be organized by modules, but the languages do not allow to have a component-based specification with a clear definition of the component interfaces. In fact, OCRA can be used even without contracts just to specify SMV/HyDI in a component-based fashion. Figure 5 shows in a nutshell three different scenario of modeling and verifying SMV/HyDI models: in the first, scenario only the SMV/HyDI language and the NUXMV/HYCOMP model checkers are used; in the second scenario, the system architecture is specified in OSS (even without contracts), the behavior of the leaf component is specified in SMV/HyDI, OCRA generates the SMV/HyDI of the system composing the leaves according to the architecture, and finally NUXMV/HYCOMP are used to verify the model as a whole; in the third scenario, contracts are also specified in the OSS model and OCRA is used for a composition verification.

Use case ID	Modeling	Verification & validation
1	Monolithic: Write the system behavior in SMV.	Monolithic: Verify and validate the SMV model with nuXmv.
2	Compositional: • Write the system architecture in OSS. • Write the behavior of the leaf components in SMV. • Build the SMV system behavior with OCRA.	Monolithic: Verify and validate the SMV model with nuXmv.
3	Compositional: • Write the system architecture in OSS. • Specify also the contracts. • Write the behavior of the leaf components in SMV.	Compositional: • Validate contracts with OCRA. • Verify contract refinement with OCRA. • Verify the contracts on the SMV models with OCRA. • Validate the SMV models with nuXmv.

Figure 5: Three scenarios of usage of nuXmv and OCRA in a nutshell

In order to support this usage, several commands of OCRA take as input or return as output SMV/HyDI models. These models represent an instantiation of the OCRA component in a free environment. They contain a module corresponding to the OCRA component and a module main instantiating the component.

Going to the detailed syntax, other than being valid SMV/HyDI models[4], to be used with OCRA, these models need to follow the syntax described below. When starting to model the implementations of an OCRA specification, it is possible to automatically generate the correct structure, using the command
`ocra_print_implementation_template`. Figure 5 shows a simple OCRA component and the corresponding SMV/HyDI template. Note that OCRA version 1.2.0 and earlier used a different (now deprecated) syntax. If needed, it can be enabled with
`set ocra_old_smv_format`. This works only for discrete time models.

General rules

- Types are slightly different in OCRA and SMV/HyDI so that the following correspondence must be followed:
  - OCRA input data port of type X corresponds to SMV/HyDI variable of type X
  - OCRA output data port of type X corresponds to SMV/HyDI variable of type X
  - OCRA input event port corresponds to SMV/HyDI input variable of type boolean
  - OCRA output event port corresponds to SMV/HyDI input variable of type boolean
  - OCRA parameter of type X corresponds to SMV/HyDI frozen variable of type X

  Note that the word ”input” has different meanings.
- Since in SMV/HyDI there is no distinction between input and output, we use the convention that inputs correspond to variables declared outside the component module and passed to module through MODULE parameters, while outputs correspond to some symbols declared inside the MODULE.

Main SMV module

- For each parameter, there must be a frozen variable (FROZENVAR). This has to be also an actual parameter to the module instance.
- For each input port, there must be a state (VAR) or input variable (IVAR, if the port is an event). This has to be an actual parameter to the module instance as well.
- For each output port, there must be a DEFINE. Its body must be a symbol with the same name in the module “component name”
- There must be a single module instance, of type “component name”.

Component SMV module

- For each input port and parameter, there must be a parameter
- For each output port, there must be a symbol

Main HyDI module

- There must be a single module instance, of type “component name”.

Component HyDI module

- For each parameter, there must be a frozen variable (FROZENVAR).
- For each input and output port, there must be a state (VAR) or input variable (IVAR, if the port is an event).

[4] see the NUXMV and HYCOMP websites: `https://es.fbk.eu/tools/nuxmv/`, `https://es.fbk.eu/tools/hycomp/`

```
COMPONENT Sensor system

    INTERFACE

        INPUT PORT ib : boolean;
        INPUT PORT ie : event;
        OUTPUT PORT ob : boolean;
        OUTPUT PORT oe : event;
        PARAMETER pb : boolean;
```

```
MODULE main
    VAR
        Sensor_inst : Sensor(pb, ie, ib);
    VAR
        ib : boolean;
    IVAR
        ie : boolean;
    FROZENVAR
        pb : boolean;
    DEFINE
        oe := Sensor_inst.oe;
        ob := Sensor_inst.ob;

MODULE Sensor(pb, ie, ib)
    IVAR
        oe : boolean;
    VAR
        ob : boolean;
```

Figure 6: A simple OCRA component and corresponding SMV template

4.1 Notes regarding the integration with HYCOMP

Even if the HyDI language provides a way to compose several process in a network, in the context of the integration with OCRA, each component is modelled as an HyDI model with a single process. The composition of the processes is rather modelled in the OSS model. Therefore, it is an error to use synchronization keywords such as EVENT and SYNC when modelling the implementation of an OCRA component.

5 Advanced features

5.1 Verification of Receptiveness

For compositionality of a component with its environment, it is necessary that the component does not block inputs from the environment. Since we are in the contract-based framework, it is sufficient to consider only environments satisfying the assumptions of the component contracts. This check is performed automatically by OCRA with the command `ocra_check_receptiveness`. However, the command is limited to consider only assumptions that predicate over the input ports.

5.2 Contract-based Safety Analysis

A specific safety analysis can be performed exploiting the contract-based design and identifying the component failures as the failure of its implementation in satisfying the contract. When the component is composite, its failure can be caused by the failure of one or more subcomponents and/or the failure of the environment in satisfying the assumption. This dependency can be automatically computed based on the contract refinement. The OCRA command `ocra_compute_fault_tree` produces a fault tree in which each intermediate event represents the failure of a component or its environment and is linked to a Boolean combination of other nodes; the top-level event is the failure of the system component, while the basic events are the failures of the leaf components and the failure of the system environment (see [BCMT14] for more details).

```
COMPONENT Sensor system

    INTERFACE

        INPUT PORT ib : continuous;
        INPUT PORT ie : event;
        OUTPUT PORT ob : continuous;
        OUTPUT PORT oe : event;
        PARAMETER pb : boolean;
```

```
MODULE main
    VAR
        Sensor_inst : Sensor();

MODULE Sensor()
    VAR
        ib : continuous;
    IVAR
        ie : boolean;
    FROZENVAR
        pb : boolean;
    IVAR
        oe : boolean;
    VAR
        ob : continuous;
```

Figure 7: A simple OCRA component and corresponding HyDI template

6 Parametrized Ocra

OCRA provides the possibility to specify also parametrized architectures, where the number of components and their interfaces are parametrized by some parameters. In particular, parameters can be used to specify the size of arrays (of subcomponents or ports) or to express conditions on the presence of a component or port. Parameters used in this way are called **`architectural parameters`**. OCRA models with this feature are called parametrized models.

A parametrized OCRA model represents a set of possible architectures, one for each evaluation of the architectural parameters. It is possible to instantiate a parametrized model using a specific assignment to its architectural parameteres using the command `ocra_instantiate_parametric_arch`. The assignments can be specified by the command or directly inside the model using connections. It is possible to perform the verification of the contract refinement of a parametrized model using the command `ocra_check_param_refinement`. Using this command it is possible to identify which configuration gives a proper contract refinement.

6.1 Architectural parameters

6.1.1 Definition

Architectural parameters are defined as **`PARAMETER`** and can be of type **`integer`**, numerical range or **`boolean`**. A Parameters is considered architectural in the following cases:

- it is used to set arrays size
- is used in a conditional declarations
- is used to define a parametrized connections
- is connected to a sub-component architectural parameter.

In the Figure 8 you can see for example that the architectural parameter `n_sub_comp` determinates the size of the sub-component array `block_1`.

```
@requires discrete-time

COMPONENT System1 system
  INTERFACE
    PARAMETER n_sub_comp : integer ;
  REFINEMENT
    SUB block_1 : array ( n_sub_comp ) of Block ;
    CONNECTION block_1[i].inputPort := block_1[i - 1].outputPort for 1 <= i < n_sub_comp ;
    CONNECTION block_1[0].inputPort := block_1[n_sub_comp -1].outputPort ;

COMPONENT Block
  INTERFACE
  INPUT inputPort : boolean ;
  OUTPUT outputPort : boolean ;
```

Figure 8: Simple parametrized OSS example

6.1.2 Instantiating architectural parameters

Instantiation using command:

Architectural parameters can be instantiated with the command `ocra_instantiate_parametric_arch` using the option `p` with a specific parameters assignment.

The instantiation process considers the specific component instance, not the component type. If the model contains multiple instances of the same component type, it is possible to provide different assignments to the component instances. For example consider the Figure 9: If that model is instantiated with the assignment `n_sub_comp=2` then the instantiated model will contain the component types `System1, Block, Block_1` as shown in Figure 10. That happens because each `Block` component instance has a different assignment to the parameter `n_ports`. See command `ocra_instantiate_parametric_arch` for further informations.

Sub-component architectural parameter assignment throught connections:

It is possible to connect architectural parameters to sub-components architectural parameters. To do that it is sufficient to add a `CONNECTION` between component architectural parameter to sub-component instance architectural parameter (e.g. `CONNECTION sub.size := size;`.

6.1.3 Architectural parameters dependencies

Since there could be dependencies between architectural parameters of various component instances it is possible to get the architectural parameters that should be instantiated first. See command `ocra_get_required_arch_params` for further information.

6.1.4 Parameter assumptions

It is possible to specify some constraints to the components architectural parameters. Using the keyword **`PARAMETER ASSUMPTIONS`** inside the **`INTERFACE`** block the user is able to specify a non-temporal boolean expression that contains the parameters. This expression is used in different ways internally:

- During the *instantiation* to ensure that the instantiated configuration respects the *constraints*
- During parametrized verification, where the proof obligation is generated considering the *parameter assumptions* of the various components.

An example of `PARAMETER ASSUMPTIONS`

```
PARAMETER size_0: integer;
PARAMETER size_1: integer;
PARAMETER boolean_param: boolean;
....
```

```
@requires discrete-time
COMPONENT System1 system
  INTERFACE
    PARAMETER n_sub_comp : integer ;
  REFINEMENT
    SUB block_1 : array ( n_sub_comp ) of Block ;
    CONNECTION block_1[i].n_ports := (i + 1) for 0 <= i < n_sub_comp;

COMPONENT Block
  INTERFACE
    INPUT flowport1 : array ( n_ports ) of integer ;
    PARAMETER n_ports : integer ;
```

Figure 9: Simple parametric OSS example

....
....
```
PARAMETER ASSUMPTIONS size_0 > size_1 & boolean_param;
```

6.2 Parametrized Language

Parametrized OCRA bring new features for the language, such as **`iterators`**, **`parametric connections`**, **`parametric subcomponents`**, **`parametric ports`** and **`parametric contracts`**.

6.2.1 Iterators

Iterators express ranges of values in expressions, they are mainly used in **`parametric`** structures in order to iterate through arrays.

They are defined with 3 different parts:

- lower bound
- index variable
- upper bound

The **`lower bound`** and the **`upper bound`** are positive **`integer`** values. They can be numbers (0,1,2,...), architectural parameters or expressions over numbers and/or architectural parameters (3 * 2 * n/5).

The **`index variable`** is the label of the iterator. In order to be valid, the index variable cannot contains dots or brackets. For example `elspilon[5]` and `index.var` are invalid **`index variables`**.
Examples of valid iterators:

`0 <= i < n`
where `n` is a parameter with type **`integer`**.
`m + 3 <= i < 10`
where `m` is a parameter with type **`integer`**.
`n < i <= m`
where `n` and `m` are parameters with type **`integer`**.
`n*2 + 1 <= i < n * 16`
where `n` is a parameter with type **`integer`**.

```
@requires discrete-time

COMPONENT System1 system
INTERFACE



REFINEMENT
SUB block_1[0]: Block;
SUB block_1[1]: Block_1;



COMPONENT Block
INTERFACE
INPUT PORT flowport1: array(1) of integer;



REFINEMENT



COMPONENT Block_1
INTERFACE
INPUT PORT flowport1: array(2) of integer;



REFINEMENT
```

Figure 10: Instantiated version of parametrized example from figure 9

6.2.2 Guard

A guard is defined as an **if** followed by a boolean expression, the purpose of the **guard** is to decide if the line will be instantiated through the boolean expression. The boolean expression can contains only architectural parameters, values, and index variables related to iterators in the same line.
Here there are various examples of **guards**:

```
if (alpha > 3)
```
where `alpha` is a parameter with type **integer**.
```
if (beta and not(gamma))
```
where `beta` and `gamma` are parameters with type **boolean**.
```
if (i > j)for 0 <= i < n, 0 <= j < m
```
where `n` and `m` are parameters with type **integer**.

6.2.3 Big or

big_or is an operator that allow the user to express the **or** operator over an iterator. Example:

```
INPUT PORT port_array : array(n_ports)of boolean;
...
CONNECTION sub.port := big_or(for 0 <= i < n_ports, port_array[i]);
```

When the bounds of the iterator are all defined the **big_or** get parsed into a group of **or**. Suppose for example that `n_ports` has been set to 3. The result will be the following:

```
CONNECTION sub.port := port_array[0] or port_array[1] or port_array[2];
```

6.2.4 Big and

big_and is an operator that allow the user to express the **and** operator over an iterator. Example:

```
INPUT PORT port_array : array(n_ports)of boolean;
...
CONNECTION sub.port := big_and(for (0 <= i < n_ports),port_array[i]);
```

When the bounds of the iterator are all defined the **big_and** get parsed into a group of **and**. Suppose for example that `n_ports` has been set to 3. The result will be the following:
`CONNECTION sub.port := port_array[0] and port_array[1] and port_array[2];`

6.2.5 Count iterator

We can use `count` with iteration bounds in order to operates with arrays. The syntax is the following: `count(for 0 <= i < n, ports[i])`.

When the bounds of the iterator are all defined the `count` get written as the classic `count` operator.

6.2.6 Parametric Ports

A port is considered parametric when:

- it contains an **if** guard
- it is a port array and his size depends on an architectural parameter

A parametric port example:

```
PARAMETER size : integer;
PARAMETER port_exists : boolean;
INPUT PORT ports: array(size)of boolean if (port_exists);
```

6.2.7 Parametric Subcomponent

A **SUB** is considered parametric if al least one of the following points is true:

- it contains a **guard**
- it is a subcomponent array and his size depends on an architectural parameter

Parametric subcomponents examples:

```
SUB sub : SubComp if (sub_exists);
SUB b_array : array(b_size * 5)of B;
SUB b_array_cond : array(b_size)of B if (b_exists);
```

Where `sub_exists`, `b_exists`, `b_size` are architectural parameters.

6.2.8 Parametric connection

Parametric connections are like normal connections with a non mandatory **if** guard and a **for** with a set of iterators separated by comma.

```
CONNECTION sub_array[i].ports[j] := ports[i] for 0 <= i <= (size - 1), 0 <= j < size;
CONNECTION sub.port := port if (port_instantiated);
CONNECTION sub_array[i].ports[j] := ports[i] if (array_inst);
```

Note that in these examples `size`, `port_instantiated` and `array_inst` are required to be architectural parameters.

After the instantiation, the parametric connection will be written as multiple connections with the iterators values. For example the last parametric connection with `size` assigned as 2 will became:

```
CONNECTION sub_array[0].iports[0] := ports[0];
CONNECTION sub_array[0].iports[1] := ports[0];
CONNECTION sub_array[1].iports[0] := ports[1];
CONNECTION sub_array[1].iports[1] := ports[1];
```

6.2.9 Parametric Contracts

A parametric contract is a contract that has an iterator, a guard or both. Here below are shown a couple of examples of parametric contracts:

```
CONTRACT c1 [i] (forall i, 0 <= i < n)if (exists)
assume: true;
guarantee: ports[i] > 3;
```

where `n`, `exists` are architectural parameters, `ports` is a parametric port array with size = `n`

```
CONTRACT c1 if (exists)
assume: true;
guarantee: ports[i] > 3;
```

where `exists` is an architectural parameter

6.2.10 Parametric Contract Refinement

Parametric refinedby is a a refinedby that has: an iterator, a guard or both.

Here there is an example of refinedby bodies:

```
CONTRACT Contract1 REFINEDBY
blocks[i].Contract1 for 0 <= i <= (n_blocks -1),
blocks[i].Contract2 if(ref_ctx2)for 0 <= i <= (n_blocks -1),
block_D.Contract if (have_d);
```

As for the other examples `n_blocks` should be an architectural parameter.

7 Interactive Commands of OCRA

ocra_check_refinement - *Checks the contract refinement of the OSS* Command

```
ocra_check_refinement [-C <string>] [-h] [-i <file>] [-k <int>] [-a <string>]
[-f text|xml ] [-o <file>]  [-s]
```

First, it checks the syntax of a model. In case of error, a message is given. Second, it checks that a given system architecture is correct with respect to the specified refinement of contracts. The verification guarantees that if the implementations of the leaf components are correct, then for every composite component C:

- every contract of C is satisfied by the implementations of the subcomponents of C
- every assumption of a subcomponent of C is satisfied by the implementations of the other subcomponents of C or by the environment of C.

If the check finds that the refinement is wrong, the system gives you a counterexample. In case that option `-s` is given, a summary of the checks is provided hiding the formula(s) and description of the trace(s).

Currently, past operators and option bmc_force_pltl_tableau are not supported are not supported with the bmc engine with discrete time interpretation.

Command Options:

`-C <string>`	Check just the specified contract, rather than all of them
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-k int`	Set the BMC length
`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-o <file>`	Set the output file [default=stdout]
`-s`	Prints summary of the results

ocra_check_implementation - *Verifies if an SMV/HyDI model satisfies the contracts defined in the OSS model* Command

```
ocra_check_implementation -I <file> [-C <string>] [-h] [-i <file>] [-c <name>]
[-k <int>] [-l <int>] [-a <string>] [-f (text|xml)] [-o <file>]
```

Given a finite state machine modeled in the SMV/HyDI language, and an Othello System Specification, verifies if the machine satisfies the contracts defined in the OSS.

Command Options:

`-I file`	Reads the SMV/HyDI specification
`-C <string>`	Check just the specified contract, rather than all of them

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-l <int>`	bound of K-Liveness [default=10]
`-c string`	Specify the component to be checked
`-k string`	Using bmc algorithm:set the bound on length of path [default=10]. Using ic3 or kzeno: set the bound of underlying IC3 [default=10]
`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-o <file>`	Set the output file [default=stdout]

ocra_check_syntax - *Parses an OSS specification and type checks it* Command

`ocra_check_syntax (-i <file> | -p <string>) [-h] [-Q]`

It checks the syntax of a model.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-p <string>`	Read the specification from a string as an assertion
`-Q`	Skip instantiation check

go_ocra - *Parses an OSS specification and type checks it* Command

`go_ocra [-h] [-i <file>]`

Parses the model and performs syntactic checks over it.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input

ocra_print_system_implementation - *Compute and prints a system implementation* Command

`ocra_print_system_implementation [-h] [-i <file>] [-L] [-m <file>] [-o <file>]`

Given an OSS specification, a set of SMV/HyDI files and a configuration file with a map between component names and the SMV/HyDI file representing their implementation, outputs a single SMV/HyDI file for the system implementation.

The configuration file looks like this:

Hydraulic Hydraulic.smv
Select_Switch Select_Switch.smv
subBSCU subBSCU.smv

or

Hydraulic Hydraulic.hydi
Select_Switch Select_Switch.hydi
subBSCU subBSCU.hydi

That is, a line for each space separeted association.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-L`	Disable printing of the contracts [default=enabled]
`-m file`	Reads the file with the map (component name, component smv implementation file)
`-o file`	Outputs the system implementation on file. If not specified, it prints to standard output

ocra_check_consistency - *Check if the contracts of the whole specification are satisfiable* Command

```
ocra_check_consistency [-h] [-i <file>] [-j]
```

By default, it checks every assertion (assumption and guarantee) for being consistent, that is, satisfiable. Alternatively, if the option -j is given, it performs richer checks:

- For each contract, it checks consistency of the conjunction of its assumption and guarantee. Notice that this check subsumes the one by default.
- For each composite component C, checks consistency among assumption of C and assumptions and guarantees of all subcomponents.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-j`	For each contract, checks consistency of the conjunction of its assumption and guarantee and for each composite component C, checks consistency among assumption of C and assumptions and guarantees of all subcomponents

```
ocra_check_validation_prop [-h] -i <file> [-a <string>] [-w ] [-P <string>]
                           [-T <string> [-c <string>] -r <string>
                           [-p] <string>] [-f <string>] [-o <file>] [-u] [-s]
```

By default, it check all validation properties defined in the input specification. Moreover, the validation properties defined in each refined component will be performed on it and its children in isolation. Alternatively, if the option `-w` is given, it checks all validation properties (consistency, possibility, and entailment) considering the entire architecture. In more details, the user can define some constraints on the architecture, for example on the input of the components like setting an input into a constant from the father refinement. So, enable this option `-w`, the check will consider all constrains defined on the architecture contrary to the default case. We remark that the generated additional checks for the different instances in the architecture using option `-w` are not stored in the data base of properties.

On the other hand, it is possible to check a validation property given by command line using options `-T`, `-c`, `-r`, and `-p`. In more details, option `-T` allows you to specify the type of validation (consistency, possibility, or entailment) to be checked in the given component `-c`. Option `-r` allows you to specify the subset of properties to be considered given as formula ids separated by comma. In case that an entailment or possibility is desired to be checked, option `-p` allows to specify it. In addition, it is possible to check consistency of all components given the options `-c` "ALL" and `-r` "ALL", or just simple checking the consistency of all properties and subcomponents properties for a specific component by `-c` "component_name" and `-r` "ALL".

Moreover, if the option `-u` is given, it computes unsat cores when an entailment property is ok and a possibility or consistency is not ok. Finally, if option `-s` is given, a summary of the checks is provided hiding the formula(s) and description of the trace(s).

Regarding the output of the checks, for consistency and possibility properties, if the validation check finds that is ok then the system gives you a trace as a witness. Otherwise, the system gives you a minimal subset of formulas still inconsistent (unsat cores) only if option `-u` is enabled. For an entailment property either the set of properties entails another formal property (representing the negation of the undesired behavior) and an unsat core is presented to the user (if option `-u` is given) or it violates the additional formal property where the system gives you a trace as a counterexample.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-w`	The whole architecture is considered for the checks
`-P "name"`	Checks only the validation property with the given `name`
`-T string`	Select type of validation property to be checked. Valid values are: consistency, possibility, and entailment

`-c string`	Component name. Valid value “ALL” to all components
`-r string`	Subset of properties given as formulas id separated by comma “,”. Valid value “ALL” to consider all contracts and subcomponent contracts
`-p string`	Possibility/Assertion given as an input formula
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-o <file>`	Set the output file [default=stdout]
`-u`	Compute unsat cores
`-s`	Prints summary of the results

ocra_check_receptiveness - *Verifies if an SMV model is compatible with every environment satisfying the assumption of the contracts* Command

```
ocra_check_receptiveness -I <file> [-h] [-i <file>] [-c <name>]
```

Given a finite state machine modeled in the SMV language, and an Othello System Specification, verifies if the machine is compatible with every environment satisfying the assumption of the contracts defined in the OSS. At the moment, the command can be used only with propositional models.

Command Options:

`-I file`	Reads the SMV specification
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-c string`	Specify the component to be checked

ocra_check_composite_impl - *Check the system implementation compositionally or monolithically* Command

```
ocra_check_composite_impl -m <file> [-C <string>] [-h] [-a <string>] [-f <string>]
[-i <file>] [-k <int>] [-l <int] [-L] [-M] [-o <file>] [-R] [-s <file>] [-S
<string>]
```

It checks the correctness of the system implementation with a *compositional* strategy by checking:

1. the contract refinement
2. the implementation of each leaf component

Equivalent to issue ocra_check_refinement and ocra_check_implementation on each leaf component.

Alternatively, if the option -M is given, it checks the correctness of the system implementation with a *monolithic* strategy. It computes the system implementation out of the leaf components and the system architecture and it checks its correctness. Equivalent to issue ocra_print_system_implementation and ocra_check_implementation.

The receptiveness check can be performed only on discrete-time models.

Command Options:

`-m file`	Reads the file with the map (component name, component SMV/HyDI implementation file)

`-C <string>`	Check just the specified contract, rather than all of them
`-h`	Shows a brief description of the available options.
`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-k int`	Set the BMC length
`-l <int>`	bound of K-Liveness [default=10]
`-L`	Disable printing of the contracts [default=enabled]
`-M`	Perform a monolithic verification by building the system implementation
`-o <file>`	Set the output file [default=stdout]
`-R`	Skip the refinement check
`-s <file>`	Set the output file for the system component [default=None]
`-S <file>`	Select which checks are performed on the leaves [default=implementation] Valid values are full, implementation, none, receptiveness

ocra_tighten_contract_refinement - *Tighten a contract refinement* Command

```
ocra_tighten_contract_refinement [-h] -i <file> -C <string> [-O <string>]
                          [-g <string>] [-f <string>] [-o <file>] [-s]
```

Tighten a contract refinement by either weakening the assumption of the parent contract and the guarantee of its subcontracts (top-down approach) or strengthening the guarantee of the parent contract and the assumption of its subcontracts (bottom-up approach). By default, it performs a top-down tightening. Alternatively, if the option `-g` is given, it allows to select either a top-down tightening ('w') or bottom-up tightening ('s').

Finally, if the option `-s` is given, it tighten the contract refinement considering those contracts refinement at the same level which have subcontracts in common.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-C string`	Contract name
`-O string`	Component name
`-g string`	Select approach to tighten the given contract refinement Valid values: 'w' (top-down) and 's' (bottom-up), [default=w]
`-f string`	Selects output format. Valid values are: text, xml [default=text]

`-o <file>`	Set the output file [default=stdout]
`-s`	Prints summary of the results

ocra_print_implementation_template - *Prints the SMV/HyDI templates of the leaf components* Command

`ocra_print_implementation_template [-h] [-c <name>] [-i <file>] [-L] [-m <file>]`

The purpose of this command is to aid the modelling of the implementations of the basic (leaf) components of a system.

It generates one o more SMV/HyDI files, each of them representing the implementation of a leaf component. The generated models are templates: they contain just the language of the component.

If `-m` is given, only the listed components are processed and the SMV/HyDI files are named according to the mapping. Otherwise, all the leaf components are processed and the SMV/HyDI files are named after the name of the components.

The configuration file looks like this:
Hydraulic Hydraulic.smv
Select_Switch Select_Switch.smv
subBSCU subBSCU.smv

or

Hydraulic Hydraulic.hydi
Select_Switch Select_Switch.hydi
subBSCU subBSCU.hydi

That is, a line for each space separated association.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-L`	Disable printing of the contracts [default=enabled]
`-c`	Consider the specified component, rather than the whole specification
`-m file`	Reads the file with the map (component name, component SMV/HyDI implementation file)

ocra_write_anonymized_models - *Prints the SMV templates of the leaf components* Command

`ocra_write_anonymized_models [-h] [-i <file>] [-m <file>] [-d <file>]`

The configuration file looks like this:
Hydraulic Hydraulic.smv
Select_Switch Select_Switch.smv
subBSCU subBSCU.smv

That is, a line for each space separated association.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-m file`	Reads the file with the map (component name, component smv implementation file)
`-d file`	Reads the file with the map (component name, component smv implementation file)

ocra_print_proof_obligations - *Prints on files the proof obligations composing the refinement check* Command

`ocra_print_proof_obligations[-h] [-i <file>] [-o <file>]`

It computes the proof obligations that compose the refinement check and prints them as SMV files, if discrete time interpretation is selected, as HRELTL files otherwise. Trivial proof obligations are not printed.

If -o option is given, its value is used as a prefix that, together with a counter, is used as filename. Otherwise, the filename will follow this format:
$< model > \_ < component > \_ < contract > \_(i|e)[\_ < subcontract >]. < extension >$
Where:

- model is the pathname of the input oss
- component is the name of the refined component
- contract is the name of the refined contract
- "i" or "e" stands for the type of the check, that is "implementation" or "environment"
- subcontract, present only for the environment check, is the name of a contract belonging to the refinement of the contract under checking

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-o file`	Set base output file name

ocra_discrete_time Environment Variable

Interpret the ocra specification over discrete time. Default is false, specification is interpreted over hybrid time.

ocra_make_warnings_errors Environment Variable

Treat every warning as an error. Default is false.

ocra_old_smv_format Environment Variable

Write and read SMV models with the format used before version 1.3.0. Default is false.

ocra_async_fairness Environment Variable

Add to the specification the constraint that each subcomponent will eventually not stutter. It affects only models with asynchronous refinement.

ocra_show_traces - *Shows the traces generated in an ocra session* Command

```
ocra_show_traces [-h] [-v] [-t] [-A] [-m | -o output-file]
[-a | trace_number[.from_state[:[to_state]]]
```

Shows the traces currently stored in system memory, if any. By default it shows the last generated trace, if any. Optional trace number can be followed by two indexes (from_state, to_state), denoting a trace "slice". Thus, it is possible to require printout only of an arbitrary fragment of the trace (this can be helpful when inspecting very big traces). Moreover, the trace descrition also shows information of the component where was generated the trace.

Command Options:

`-h`	Shows a brief description of the available options.
`-v`	Verbosely prints traces content (all state ports, input/output ports, and parameters, otherwise it prints out only the variables that have changed their value from previous state). This option only applies when the Basic Trace Explainer plugin is used to display the trace.
`-t`	Prints only the total number of currently stored traces.
`-a`	Prints all the currently stored traces.
`-m`	Pipes the output through the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
`-o output-file`	Writes the output generated by the command to `output-file`.
`trace_number`	The (ordinal) identifier number of the trace to be printed. Omitting the trace number causes the most recently generated trace to be printed.
`from_step`	The number of the first step of the trace to be printed. Negative numbers can be used to denote right-to-left indexes from the last step.
`to_step`	The number of the trace to be printed. Negative numbers can be used to denote right-to-left indexes from the last step. Omitting this parameter causes the entire suffix of the trace to be printed.
`-A`	Prints the trace(s) using a rewriting mapping for all symbols.

ocra_show_property - *Shows the currently stored properties in an ocra session.* Command

```
ocra_show_property [-h] [-v] [[ -c <name> | -a | -g | -q | -r | -s] |
                   [t | f] ] | [-n idx | -P <name>]
```

Shows the properties currently stored in the list of properties. This list is initialized with the properties present in the input file, if any; then all of the properties added by the user with the basic command `ocra_check_syntax` or with any command which requires as input an OSS specification. Generally, three types of properties are distinguished *contracts*, *validation props*, and *proof obligations*. In the last case, the proof obligations are only added to the list of properties through the command `ocra_check_refinement`.

For every property, the following informations are displayed:

- the identifier of the property (a progressive number);
- the property name;
- the property formula;
- the category (ASSUMPTION_PROP, GUARANTEE_PROP, CONSISTENCY_PROP, POSSIBILITY_PROP, ENTAILMENT_PROP, PROOF_OBLIG_PROP, and TRACE_PROP);
- the status of the property (OK, NOT OK, N/A);
- if the formula has been found to be false (true), the index number of the corresponding counterexample (witness) trace;
- the list of traces id that the property is satisfied on each one.
- the list of traces id that the property is not satisfied on each one.

Each property has a name associated except for property with category TRACE_PROP (see command `ocra_check_ltlspec_on_trace`). Validation properties are the only ones that contain a name by definition. On the other hand, for each contract the name for its assumption and guarantee is defined as follows:

$< component > . < contract > .[ASSUMPTION|GUARANTEE]$.

Finally, for each proof obligation, its name is defined as follows:

$< component > . < contract > \_(imp|env)[\_ < subcontract >]$
Where:

- component is the name of the refined component
- contract is the name of the refined contract
- "imp" or "env" stands for the type of the check, that is "implementation" or "environment"
- subcontract, present only for the environment check, is the name of a contract belonging to the refinement of the contract under checking

By default, all the properties currently stored in the list of properties are shown without printing the formula contents. The option `-v` enable the visualization of the formulas. Specifying the suitable options, properties with a certain category and/or of a certain status (OK or NOT OK), or with a given identifier, or with a given name it is possible to let the system show a restricted set of properties. Moreover, it is possible to show all properties of a given component name. It is allowed to insert several options per category.

Command Options:

`-h`	Shows a brief description of the available options.
`-v`	Verbosely prints formula contents.
`-c <name>`	Prints out the properties of component named "name".
`-a`	Prints only assumption properties.
`-g`	Prints only guarantee properties.
`-q`	Prints only validation properties.
`-r`	Prints only proof obligation properties.
`-t`	Prints only those properties found to be OK.

`-f`	Prints only those properties found to be NOT OK.
`-n idx`	Prints out the property numbered “idx”.
`-P <name>`	Prints out the property named “name”.
`-s`	Prints the number of stored properties.

ocra_check_ltlspec_on_trace - *Checks whether a property is satisfied on a trace* Command

```
ocra_check_ltlspec_on_trace [-h] [-i] [-n number | -P <name> |
                    [[-c <name>] -p "expr"] ] [-l loopback] trace_number
```

Checks whether a property is satisfied on a given trace. The problem generated can be checked using SAT/SMT backend.

Option `-i` forces the use of the engine for infinite domains. In case the user does not provide this option, a SAT solver is called by default for checking the problem generated if only the formula and trace does not contain infinite precision variables. Otherwise, a SMT solver will be called for solving the problem generated.

We take into account that each OCRA trace may correspond to an infinite number of traces due to the possible presence of more than one loopbacks. So, it is not possible (or at least straightforward) to check all of them. Therefore, we consider just one loopback and provide the user with the possibility to select it.

A `"expr"` to be checked can be specified at command line using option `-p`. This formula is added to the data base of property with the category TRACE_PROP. Each property in the data base is associated with component information, i.e., option `-c` allows to the user to associate `"expr"` with the component information provided. In case the user does not provide this information, the component associated to `"expr"` is the component **`system`** by default.

Alternatively, options `-n` and `-P` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` nor `-P` are used, then each contract (assumption and guarantee) in the OSS specification is checked on the given trace.

The loopback value can be specified at command line using option `-l`. This must a valid `loopback` value on the given trace. If it is not valid, then an error message is printed and also the available loopbacks are provided. In case that option `-l` is not used, then a warning is printed and the check is performed using the first loopback found on the given trace.

Finally, the last argument of the command is the trace number which has to correspond to a trace stored in the system memory. If the trace number is omitted, then an error message is printed. In case that the trace has not loopbacks, then an error message is printed informing the user that the selected trace is finite and cannot satisfy any LTL formula.

Command Options:

`-h`	Shows a brief description of the available options.
`-i`	Forces the use of the engine for infinite domains.
`-c <name>`	Component name to be assigned to the input formula
`-p "expr"`	Checks only the given formula “expr”
`-P <name>`	Checks only the property with the given name
`-n <number>`	Checks only the property with the given index `number` in the property database.

`-l <loopback>`	Checks the property on the trace using `loopback` value. This must a valid `loopback` value on the given trace.
`trace_number`	The (ordinal) identifier number of the trace to be used to check the property. This must be the last argument of the command.

ocra_debug_property - *Debugs a property* Command

```
ocra_debug_property [-h] [-n number | -P <name> ] [-s]
```

By default, it debugs all unsatisfied proof obligations and entailment properties in an oss specification. Generally speaking, the procedure debugs each of the subformulas of the property on the related trace that it was generated as a counterexample or as a witnesses during a check (e.g., `ocra_check_refinement`, `ocra_check_consistency`, or `ocra_check_validation_prop`). Thereby, only properties related to a trace can be debugged.

Alternatively, options `-n` and `-P` can be used for debugging a particular property. Finally, if option `-s` is given, only the not satisfied results are printed.

Command Options:

`-h`	Shows a brief description of the available options.
`-n <number>`	Debugs only the property with the given index `number` in the property database.
`-P "name"`	Debugs only the property with the given `name`.
`-s`	Prints summary of the results

ocra_compute_fault_tree - *Generates a hierarchical fault tree from a contract-based system decomposition* Command

```
ocra_compute_fault_tree [-C <string>] [-h] [-i <file>] [-k <int>] [-a <string>]
[-m <file>] [-x <file>] [-f <format>] [-o <file>]
```

Given a contract-based decomposition it generates a heirarchical fault tree according with the approach desribed in “M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal Safety Assessment via Contract-Based Design. In Proceedings of ATVA 2014. Sydney, Australia, November 3-7, 2014.”.

The optional parameter “-x” enables the additional fault tree computation on leaves implementations by providing a fault injection xml file description.

Command Options:

`-C <string>`	Check just the specified contract, rather than all of them
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-k <int>`	with bmc: bound on length of path [default=10] with ic3 or kzeno: bound of underlying IC3 [default=10]

`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-m <file>`	Set a file with the mapping OSS component -¿ SMV file. Its format is: (¡OSS component name¿ ¡SMV filename¿)+ Equivalent to deprecated -a
`-x <file>`	XML file describing the fault extension.
`-f <string>`	select output format [default=text] Valid values are: text, xml, xml_no_text
`-o <file>`	Set the output file [default=stdout]

ocra_instantiate_parametric_arch - *Instantiate an OSS specification and returns instantiated OSS file* Command

`ocra_instantiate_parametric_arch -i <file> (-I <file> | -p <string>) -o <file> [-h]`

Given an assignment list it instantiates parameters of a parametric OCRA model.

The assignment list is composed in this way:
system_parameter = value,
system_comp.subcomp.paremeter = value,
system_comp.subcomp.another_sub.parameter = value

In the assignment list is possible for the user to specify a range of components array to do multiple assignments in a single instruction. This can be done using this notation to specify every component in this range: component_array[lbound_value .. upper_bound_value]

Example of instantiation list:

system_param = 15,
system_comp.subcomp[0..10].param = 11,
system_comp.subcomp[0].other_sub[1..100].alpha = true

It is also possible to launch the command without any assignment list, in this case only parameters assigned (using connections) inside the OSS file will be instantiated.

The command will save the result specification in the file specified with `-o`, the returned specification can be also only partially instantiated, full instantiation is not a constraint.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-p <string>`	Read the parameter assignments from a string as a csv assignment list
`-I <file>`	Read the parameter assignments from a csv file

`-o <file>`	Write the instantiated OSS architecture in this file

ocra_instantiate_automatically_arch - *Instantiate a parametric architecture with all possible parameters values* Command

```
ocra_instantiate_automatically_arch -i <file> (-I <file> | -p <string>) [-h]
[-b <int>]
```

This command try to instantiate all possible specifications using parameter assumptions as constraints on parameters values. The command at the moment is not able to determinate if a parameter assumption provides infinite assignments. In that case the program might not terminate. This command will save the instantiated specifications in output files, also the assignments that generate the specifications are saved in csv output files.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-p <string>`	Read the parameter assignments from a string as a csv assignment list
`-I <file>`	Read the parameter assignments from a csv file
`-b <int>`	Set the bound of assignments from the `PARAMETER ASSUMPTIONS` of each component

ocra_get_required_arch_params - *Get the architectural parameters required in order to perform instantiation* Command

```
ocra_get_required_arch_params -i <file> [-h] [-o <file> ] [-X]
```

Given an OCRA model, get the required parameters in order to go further in the instantiation. This command does not always return all parameters required for instantiation, only the parameters that do not depend on other unsetted parameters.
For Example: Given a non instantiated parameter that belongs to a component C, if another component B have parametric conditions on an instance of a subcomponent of type C (existence, size of an array), it is not possible to say if this parameter should be initialized or not.

This command fill output file with comma separated value parameters, it is possible with option `-X` to show this parameters in a compact form that unify arrays in contiguos ranges.

Example:
Given required parameters a[0].b[0].a, a[0].b[1].a, a[1].b[0].a, a[1].b[1].a, compact form of these parameters is a[0 .. 1].b[0 .. 1].a.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input

`-o <file>`	Write the required parameters in this file
`-X`	Print compacted parameters

ocra_check_param_refinement - *Checks the contract refinement of the parametrized OSS* Command

```
ocra_check_param_refinement [-C <string>] [-h] [-i <file>] [-k <int>] [-b <int>]
[-a <string>] [-f text|xml ] [-o <file>]  [-s]
```

This command works as the command `ocra_check_refinement`. The difference is that this command supports parameterized models and it finds all the assignments to the architectural parameters such that the contract refinement is correct.

Currently, past operators are not supported. To prevent non-termination a bound on the number of configurations has been introduced. This bound can be specified with the option -b and it represents the maximum number of possible instantiations considered in each component.

Command Options:

`-C <string>`	Check just the specified contract, rather than all of them
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification. If not specified, the command reads from standard input
`-k int`	Set the BMC length
`-a string`	Force algorithm type. Valid values are: bmc, bdd, ic3, auto [default=auto]. Where auto in discrete time interpretation is bdd when the model is finite state, ic3 otherwise.
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-o <file>`	Set the output file [default=stdout]
`-b <int>`	Set the bound of assignments from the `PARAMETER ASSUMPTIONS` of each component

8 OCRA Command Line Options

OCRA accepts the following syntax:

`system_prompt>` **`ocra [command line options]`** *input-file* `<RET>`

input-file	Set the OSS input file
`-ocra-discrete-time`	Enables the interpretation of the model over discrete time instead of default hybrid time. The model must not contain: • continuous variables • der operator • time_until operator
`-ocra-old-smv-format`	Write and read SMV models with the format used before version 1.3.0
`-ocra-async-fairness`	Add to the specification the constraint that each subcomponent will eventually not stutter. It affects only models with asynchronous refinement.

References

[BCMT14] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal Safety Assessment via Contract-Based Design. In *ATVA*, 2014. To appear.

[CES] CESAR patterns. `http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP2_R2.2_M2_v1.000.pdf`.

[CRST12] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Validation of Requirements for Hybrid Systems: a Formal Approach. *ACM Trans. Softw. Eng. Methodol.*, 21(4):22, 2012.

[CRT09] A. Cimatti, M. Roveri, and S. Tonetta. Requirements Validation for Hybrid Systems. In *CAV*, pages 188–203, 2009.

[CT] A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems . *Sci. Comput. Program.* To appear.

[CT12] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In *SEAA*, 2012.

[FoR] FoReVer project. `https://es.fbk.eu/index.php?n=Projects.FoReVer`.

[HyC] HyCOMP website. `https://es.fbk.eu/tools/hycomp/`.

[NuS] NuSMV3 website. `https://es.fbk.eu/tools/nusmv3/`.

[nuX] nuXmv website. `https://es.fbk.eu/tools/nuXmv/`.

[OCR] OCRA website. `https://es.fbk.eu/tools/ocra/`.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.

[Saf] SafeCer project. `http://www.safecer.eu`.

[SPE] SPEED project. `http://www.speeds.eu.com/`.

[Ton17] Stefano Tonetta. Linear-time Temporal Logic with Event Freezing Functions. In *Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20-22 September 2017.*, pages 195–209, 2017.

[XSA] xSAP website. `https://es.fbk.eu/tools/xsap/`.

A Concrete syntax

A.1 Othello System Specification

An Othello System Specification (OSS) is written following the grammar below (in Extended Backus-Naur Form):

```
OSS               = requirements? system_comp component* ;
requirements      = "@requires" "discrete-time" | "@requires" "hybrid-time" ;
system_comp       = "COMPONENT" comptype? "system" interface refinement? ;
component         = "COMPONENT" comptype interface refinement? ;
interface         = "INTERFACE" var* contract* define* i_assertion* param_asms*;
refinement        = "ASYNC"? "REFINEMENT" subcomponent* connection* con_constraint* refinedby* valid_prop*
                    r_assertion*;
var               = (port | parameter | operation) ";" ;
port              = direction? "PORT" name ":" type ("if" param_logic_expr)?;
direction         = "INPUT" | "OUTPUT" ;
parameter         = "PARAMETER" name ":" type | euf ;
operation         = ("PROVIDED" | "REQUIRED") "OPERATION" "PORT"? name
                     "(" op_params ")" ":" (type | "void") ;
op_params         = port? | port ("," port)* ;
define            = "DEFINE" name ":=" i_constraint ";"
type              = "boolean" | "integer" | "real" | "continuous" | "event" |
                    "{" (number | name)+ "}" -- enumeration |
                    number ".." number -- range |
                    ("unsigned" | "signed")? "word" "[" number "]" |
                    "array" number ".." number "of" type |
                    "array" "(" param_expr ")" "of" type ;
euf               = euf_subtype("*" euf_subtype)* "->" euf_subtype ;
euf_subtype       = "boolean" | "real" | "integer" ;
contract          = "CONTRACT" name param_contract?
                    "assume" ":" i_constraint ";"
                    "guarantee" ":" i_constraint ";" ;
i_assertion       = "ASSERTION" "NAME" name ":=" i_constraint ";"
r_assertion       = "ASSERTION" "NAME" name ":=" r_constraint ";"
param_asms        = "PARAMETER" "ASSUMPTIONS" param_logic_expr ";"
subcomponent      = "SUB" name ":" subcomp_type ";"
subcomp_type      =  comptype ("if" param_logic_expr)? |
                    "array" "(" param_logic_expr ")" "of" comptype ("if" param_logic_expr)? ;
connection        = "CONNECTION" name ":=" right_connect ";"
right_connect     =  r_constraint |
                     r_constraint ("if" param_logic_expr)? iteration_bounds* ;
con_constraint    = "CONSTRAINT" r_constraint ";"
iteration_bounds  = "for" iteration_bound ("," iteration_bound)* ";" ;
iteration_bound   = param_logic_expr lt_e_op name lt_e_op param_logic_expr ;
refinedby         = "CONTRACT" name param_contract? "REFINEDBY" contr_id+ ";" ;
contr_id          = name "." name ("if" param_logic_expr)? ("for" iteration_bound)? ;
param_contract    = "[" name "]" "(" "forall" name "," iteration_bound ")" ("if" param_logic_expr)? ;
valid_prop        = "CONSISTENCY" "NAME" name ":=" formula_id+; |
                    "POSSIBILITY" "NAME" name ":=" (formula_id | constrain) "GIVEN" formula_id+ ";" |
                    "ENTAILMENT"  "NAME" name ":=" (formula_id | constrain) "BY" formula_id+ ";" ;
formula_id        = name "." name "." ("ASSUMPTION" | "GUARANTEE" | "NORM_GUARANTEE") ;
```

where:

- `name` is a string;
- there cannot be two components with the same name;
- for every component, there cannot be two subcomponents with the same name;
- `comptype` is a string that matches one of the components' name;
- the relationship that links a component to its subcomponents is not circular and form a tree rooted in the system component;
- `i_constraint` must be an Othello constraint as defined below where every variable must match a variable of the interface;
- `r_constraint` must be an Othello constraint as defined below where every variable must either match a variable of the interface or be in the form `sub.var` where `sub` matches a subcomponent's name of the refinement and `var` matches a variable of the interface of such subcomponent;

- `param_logic_expr` represents a logical or arithmetic expression restricted to referencing only parameters, next values of parameters, array sizes, constants and index variables.
- `lt_e_op` is $<$ or $<=$ operator.
- there cannot be two validation properties with the same name.

A.2 Othello Port Types

A.2.1 Boolean

The boolean type comprises symbolic values **`FALSE`** and **`TRUE`**. The **`event`** type has the same domain of the boolean type. The difference is that while a boolean port is evaluated in a state, an **`event`** is evaluated in a transition: we say that an **`event`** occurs during a transition if it is **`TRUE`** in that transition.

Accordingly, the values of event ports are listed in a separate "Transition" section in the OCRA counterexamples, between two "State" sections.

A.2.2 Enumeration Types

An enumeration type is a type specified by full enumerations of all the values that the type comprises. For example, the enumeration of values may be {`stopped, running, waiting, finished`}, {`2, 4, -2, 0`}, {`FAIL, 1, 3, 7, OK`}, etc. All elements of an enumeration have to be unique although the order of elements is not important. A range of whole numbers can be declared with `N..M`. For example the following declarations are equivalent:

```
2..5
{2, 3, 4, 5}
```

A.2.3 Word

The unsigned word[•] and signed word[•] types are used to model vector of bits (booleans) which allow bitwise logical and arithmetic operations (unsigned and signed, respectively). These types are distinguishable by their width. For example, type unsigned word[3] represents vector of three bits, which allows unsigned operations, and type signed word[7] represents vector of seven bits, which allows signed operations.

When values of unsigned word[N] are interpreted as integer numbers the bit representation used is the most popular one, i.e. each bit represents a successive power of 2 between 0 (bit number 0) and 2^{N-1} (bit number $N-1$). Thus unsigned word[N] is able to represent values from 0 to $2^N - 1$.

The bit representation of signed word[N] type is "two's complement", i.e., negative numbers are represented by 2^N minus their absolute value. Thus, the possible values for signed word[N] are from -2^{N-1} to $2^{N-1} - 1$.

A.2.4 Integer

The domain of the integer type is the set of integers. Note that the use of ports with integer type requires the usage of infinite-state model checking algorithms.

A.2.5 Real

The domain of the real type is the set of real numbers. Note that the use of ports with real type requires the usage of infinite-state model checking algorithms.

A.2.6 Continuous

The domain of the **`continuous`** type is the set of continuous and differentiable functions. They are used to represent a value that is a continuous function of time. Note that the use of ports with **`continuous`** type requires the usage of infinite-state model checking algorithms and is available only for the hybrid time mode of OCRA.

A.2.7 Array

Arrays are declared with a lower and upper bound for the index, and the type of the elements in the array. For example,

array `0..3 of` boolean
array `10..20 of {OK, y, z}`
array `1..8 of` array `-1..2 of` unsigned word[5]

The type array `1..8 of` array `-1..2 of` unsigned word[5] means an array of 8 elements (from 1 to 8), each of which is an array of 4 elements (from -1 to 2) that are 5-bit-long unsigned words.

Array subtype is the immediate subtype of an array type. For example, subtype of array `1..8 of` array `-1..2 of` unsigned word[5] is array `-1..2 of` unsigned word[5] which has its own subtype unsigned word[5].

Expression of array type can be constructed with variables of array type.

Internally, these arrays are treated as a set of variables.

A.2.8 Uninterpreted functions

In OCRA it is also possible to define uninterpreted functions. Only parameters can be declared with this type. These functions are rigid, i.e. their denotation does not change from two different time points. These functions can be seen as parameters: the denotation of the function is defined in the initial state and kept from that point on. Below is reported a simple example of declaring a function `funct1` that takes two reals as arguments, and returns an integer, and a function `funct2` that takes two reals and returns an unsigned word of size 32.

```
PARAMETER funct1 : real * real -> integer ;
PARAMETER funct2 : real * real -> unsigned word[32] ;
```

Note: Currently in OCRA we only support a limited number of data types both as return type and as type of each argument of a function. In particular, the supported types are: `boolean`, `real`, `integer`, and `word[N]`. Support for richer types (e.g. enumeratives, bouded integers and array) is ongoing.

A.3 Othello constraints[5]

An Othello constraint is written following the grammar below (in Extended Backus-Naur Form):

```
constraint    = atom |
                -- boolean operators
                "not" constraint |
                "big_or" "(" iteration_bound, constraint ")" |
                "big_and" "(" iteration_bound, constraint ")" |
                constraint "and" constraint |
                constraint "or" constraint |
                constraint "xor" constraint |
                constraint "implies" constraint |
                constraint "iff" constraint |
                -- temporal future operators
                "always" constraint |
                "never" constraint |
                "in the future" constraint |
                "then" constraint |
                constraint "until" constraint |
                constraint "releases" constraint |
                -- temporal past operators
                "historically" constraint |
                "in the past" constraint |
                "previously" constraint |
                constraint "since" constraint |
                constraint "triggered" constraint |
                term "at next" constraint |
                term "at last" constraint
```

[5]The OCRA data types and operators are based on the NUXMV input language. See `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf` for a detailed description.

```
                ;
atom          = "true" |
                "false" |
                term relational_op term |
                "time_until" "(" term ")" relational_op term |
                "time_since" "(" term ")" relational_op term |
                boolean_term |
                fall(boolean_term) |
                rise(boolean_term) |
                change(term);
term          = variable |
                "der" "(" variable ")" |
                "next" "(" variable ")" |
                constant |
                term "+" term |
                term "-" term |
                term "*" term |
                term "/" term |
                term "mod" term |
                abs "(" term ")" |
                "min" "(" term "," term ")" |
                "max" "(" term "," term ")" |
                term "[" index "]" | -- array indexing
                function "(" term(, term)* ")" | -- function call
                term "?" term ":" term | -- compact if then else
                "case" (term ":" term ";")+ "esac" | -- if then else
                count "(" term ("," term)* ")" | -- count of true boolean expressions
                -- word operators
                term ">>" term |  -- bit shift right
                term "<<" term |  -- bit shift left
                term "[" term ":" term "]" | -- word bits selection
                term "::" term | -- word concatenation
                sizeof "(" term ")" | -- word size as an integer
                extend "(" term "," term ")" | -- word width extension
                resize "(" term "," term ")" | -- word width resize
                signed word[N] "(" term ")" | -- integer to signed word conversion
                unsigned word[N] "(" term ")" | -- integer to unsigned word conversion
                -- set operators
                term "union" term |
                "{" term ("," term)"}" | -- set
                term "in" term | -- set membership
                -- cast operators
                word1 "(" term ")" | -- boolean to unsigned word[1] conversion
                bool "(" term ")" |
                toint "(" term ")" |
                swconst "(" term ")" | -- integer to signed word constant conversion
                uwconst "(" term ")" | -- integer to unsigned word constant conversion
                signed "(" term ")" | -- unsigned word to signed word conversion
                unsigned "(" term ")" | -- signed word to signed word conversion
                floor "(" term ")"
                ;

relational_op = ( "=" | "!=" | "<" | ">" | "<=" | ">=" ) ;
```

where:

- `constant` is a constant number;
- `variable` is a string.
- `boolean_term` is a `term` of type `boolean`.

B Abstract syntax and semantics

B.1 Abstract syntax and semantics of HRELTL constraints

B.1.1 Abstract syntax

The logic underlying Othello constraints is called HRELTL. HRELTL is built over a set of basic atoms, that are real arithmetic predicates over $V \cup \text{NEXT}(V)$, or over $V \cup \text{DER}(V)$, where V is a set of variables, $\text{NEXT}(V)$ the set of next variables and $\text{DER}(V)$ the set of derivatives.

The set $PRED$ is defined as the set of *linear arithmetic* predicates in one of the following forms:

- $a_0 + a_1 v_1 + a_2 v_2 + \cdots + a_n v_n \sim 0$ where $v_1, \ldots, v_n \in V$, $a_0, \ldots, a_n$ are constants, and $\sim \in \{<, >, =, \leq, \geq, \neq\}$[6].
- $a_0 + a_1 \dot{v} \bowtie 0$ where $v \in V_C$, a_0, a_1 are arithmetic predicates over variables in V_D, and $\sim \in \{<, >, =, \leq, \geq, \neq\}$.

The HRELTL is defined as the extension of LTL defined with the following rules: if $p \in PRED$, ϕ_1 and ϕ_2 are HRELTL formulas, e is an event, $\sim \in \{<, >, =, \leq, \geq, \neq\}$ and c an arithmetic term, then:

- p is a HRELTL formula;
- $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\mathbf{X}\ \phi_1$, $\phi_1\ \mathbf{U}\ \phi_2$ are HRELTL formulas;
- $\triangleright_{\sim c} e$ and $\triangleleft_{\sim c} e$ are HRELTL formulas.

We use standard abbreviations for $\vee$, $\rightarrow$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{R}$ (see, e.g., [CRT09]). $\triangleright$ corresponds to "time_until", $\triangleleft$ to "time_since". "never" is an abbreviation for $\mathbf{G}\ \neg$. $fall(\phi)$ and $rise(\phi)$ are abbreviations of $\phi \wedge X!\phi$ and $!\phi \wedge X\phi$ respectively. $change(u)$ is an abbreviation for $next(u) \neq u$. The other abstract connectives correspond the concrete ones used in the previous section in a straightforward way.

B.1.2 Semantics

HRELTL formulas are interpreted with hybrid traces, which are defined as follows. Let V be the finite disjoint union of the sets of variables V_D (with a discrete evolution) and V_C (with a continuous evolution). A state s is an assignment to the variables of V ($s : V \rightarrow \mathbb{R}$). We write Σ for the set of states. Let $f : \mathbb{R} \rightarrow \Sigma$ be a function describing a continuous evolution. We define the projection of f over a variable v, written f^v, as $f^v(t) \doteq f(t)(v)$. We say that a function $f : \mathbb{R} \rightarrow \mathbb{R}$ is piecewise analytic iff there exists a sequence of adjacent intervals $J_0, J_1, ... \subseteq \mathbb{R}$ and a sequence of analytic functions $h_0, h_1, ...$ such that $\cup_i J_i = \mathbb{R}$, and for all $i \in \mathbb{N}$, $f(t) = h_i(t)$ for all $t \in J_i$. Note that, if f is piecewise analytic, the left and right derivatives exist in all points. We denote with $\dot{f}$ the derivative of a real function f, with $\dot{f}(t)_-$ and $\dot{f}(t)_+$ the left and the right derivatives respectively of f in t. Let I be an interval of $\mathbb{R}$ or $\mathbb{N}$; we denote with $le(I)$ and $ue(I)$ the lower and upper endpoints of I, respectively. We denote with $\mathbb{R}^+$ the set of non-negative real numbers.

A hybrid trace over V is a sequence $\langle \overline{f}, \overline{I} \rangle \doteq \langle f_0, I_0 \rangle, \langle f_1, I_1 \rangle, \langle f_2, I_2 \rangle, \ldots$ such that, for all $i \in \mathbb{N}$,

- either I_i is an open interval ($I_i = (t, t')$ for some $t, t' \in \mathbb{R}^+$, $t < t'$) or is a singular interval ($I_i = [t, t]$ for some $t \in \mathbb{R}^+$);
- the intervals are adjacent, i.e. $ue(I_i) = le(I_{i+1})$;
- the intervals cover $\mathbb{R}^+$: $\bigcup_{i \in \mathbb{N}} I_i = \mathbb{R}^+$ (thus $I_0 = [0, 0]$);
- $f_i : \mathbb{R} \rightarrow \Sigma$ is a function such that, for all $v \in V_C$, f_i^v is continuous and piecewise analytic, and for all $v \in V_D$, f_i^v is constant;
- if $I_i = (t, t')$ then $f_i(t) = f_{i-1}(t)$, $f_i(t') = f_{i+1}(t')$.

We denote with $PRED$ the set of predicates, with p a generic predicate, with p_{curr} a predicate over V only, with p_{next} a predicate over V and $\text{NEXT}(V)$ containing at least an event variable or a next variable, and with p_{der} a predicate over V and $\text{DER}(V)$ containing at least a derivative variable. We denote with $\overline{p}$ the predicate obtained from p by replacing $<$ with $\geq$, $>$ with $\leq$, $=$ with $\neq$ and vice versa. We denote with $p_\sim$ the predicate obtained from p by substituting the top-level operator with $\sim$, for $\sim \in \{<, >, =, \leq, \geq, \neq\}$.

The HRELTL is interpreted over hybrid traces. The predicates p_{curr}, p_{next} and p_{der} are interpreted over hybrid traces as follows:

[6] As described in [CRT09], instead of constants, also discrete variables can be used, but the currently used SMT solver does not support non-linear constraints.

- $\langle \overline{f}, \overline{I} \rangle, i \models_p p_{curr}$ iff, for all $t \in I_i$, p_{curr} evaluates to true when v is equal to $f_i^v(t)$, denoted with $f_i(t) \models_p p$;
- $\langle \overline{f}, \overline{I} \rangle, i \models_p p_{next}$ iff there is a discrete step between i and $i+1$, i.e. $I_i = I_{i+1} = [t,t]$, and p_{next} evaluates to true when v is equal to $f_i^v(t)$ and $\text{NEXT}(v)$ to $f_{i+1}^v(t)$, denoted with $f_i(t), f_{i+1}(t) \models_p p_{next}$;
- $\langle \overline{f}, \overline{I} \rangle, i \models_p p_{der}$ iff, for all $t \in I_i$, p_{der} evaluates to true both when v is equal to $f_i^v(t)$ and $\text{DER}(v)$ to $\dot{f}_i^v(t)_+$, and when v is equal to $f_i^v(t)$ and $\text{DER}(v)$ to $\dot{f}_i^v(t)_-$, denoted with $f_i(t), \dot{f}_i(t)_+ \models_p p_{der}$ and $f_i(t), \dot{f}_i(t)_- \models_p p_{der}$ (when $\dot{f}_i(t)$ is defined this means that $f_i(t), \dot{f}_i(t) \models_p p_{der}$).

Finally, we can define the semantics of HRELTL:

- $\langle \overline{f}, \overline{I} \rangle, i \models p$ iff $\langle \overline{f}, \overline{I} \rangle, i \models_p p$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \neg\phi$ iff $\langle \overline{f}, \overline{I} \rangle, i \not\models \phi$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \phi \wedge \psi$ iff $\langle \overline{f}, \overline{I} \rangle, i \models \phi$ and $\langle \overline{f}, \overline{I} \rangle, i \models \psi$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \mathbf{X}\ \phi$ iff there is a discrete step at i ($I_i = I_{i+1}$), and $\langle \overline{f}, \overline{I} \rangle, i+1 \models \phi$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \phi\ \mathbf{U}\ \psi$ iff, for some $j \geq i$, $\langle \overline{f}, \overline{I} \rangle, j \models \psi$ and, for all $i \leq k < j$, $\langle \overline{f}, \overline{I} \rangle, k \models \phi$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \rhd_{\sim c}\phi$ iff, for some $j \geq i$, $I_j = I_{j+1} = [t,t]$ and $\langle \overline{f}, \overline{I} \rangle, j \models \phi$ and, for all $i \leq k < j$, $\langle \overline{f}, \overline{I} \rangle, k \not\models \phi$ and for all $t' \in I_i, t - t' \sim c$;
- $\langle \overline{f}, \overline{I} \rangle, i \models \lhd_{\sim c}\phi$ iff, for some $j \leq i$, $I_j = I_{j+1} = [t,t]$ and $\langle \overline{f}, \overline{I} \rangle, j \models \phi$ and, for all $j < k \leq i$, $\langle \overline{f}, \overline{I} \rangle, k \not\models \phi$ and for all $t' \in I_i, t' - t \sim c$.

B.1.3 Examples

The following formulas are examples of expressions with time-until: $G(p \rightarrow \rhd_{\leq 10} q)$: always the maximum delay between p and the next q is 10 time units; $\rhd_{\geq 5} p \wedge G(p \rightarrow \rhd_{=10} p)$: p happens exactly every 10 time units after an initial offset of 5 time units.

B.2 Abstract syntax and semantics of the system specification

A *component* S is described with a set V_S of ports, which are the variables representing the relevant information of the component that are visible from outside it. An *implementation* of a component S is defined as a subset of traces over V_S, i.e., a subset of Π_{V_S}. An *environment* of S is also defined as a subset of Π_{V_S} (since V_S are the the variables at the interface of S).

A *connection* γ over a set $\mathcal{S}_\gamma$ of components defines the possible interaction of the components. The semantics $[\![\gamma]\!]$ of a connection is defined as a subset of traces over $\bigcup_{S \in \mathcal{S}_\gamma} V_S$, i.e. $[\![\gamma]\!] \subseteq \Pi_{\bigcup_{S \in \mathcal{S}_\gamma} V_S}$.

A *decomposition* of a component S is is pair $\rho = \langle Sub_\rho, \gamma_\rho \rangle$ where:

- $Sub_\rho(S)$ is a set of subcomponents such that $Sub_\rho(S) \neq \emptyset$ and $S \notin Sub_\rho(S)$,
- $\gamma_\rho(S)$ is a connection over $\{S\} \cup Sub_\rho(S)$.

The implementation of a decomposed component S consists of the composition of the implementations of its sub-components $Sub_\rho(S)$. Namely, it is given by all the traces that are compatible with the subcomponents and their connection. Formally, if for every $S' \in Sub_\rho(S)$, $I_{S'}$ is the implementation of S', the implementation I_S of S induced by the decomposition ρ is defined as $\{\pi \in \Pi_{V_S} \mid \text{there exists } \pi_{S'} \in I_{S'} \text{ for every } S' \in Sub_\rho(S) \text{ such that } \pi \times \bigotimes_{S' \in Sub_\rho(S)} \pi_{S'} \in [\![\gamma_\rho(S)]\!]\}$. Similarly, the environment of a subcomponent $U \in Sub_\rho(S)$ is composed by the environment of S and by the sibling subcomponents of S. Formally, if E_S is the environment of S, the environment E_U of U induced by the decomposition ρ of S is defined as $\{\pi_U \in \Pi_{V_U} \mid \text{there exists } \pi_{S'} \in I_{S'} \text{ for every } S' \in Sub_\rho(S) \setminus \{U\} \text{ and } \pi \in E_U \text{ such that } \pi \times \bigotimes_{S' \in Sub_\rho(S)} \pi_{S'} \in [\![\gamma_\rho(S)]\!]\}$.

A *system* is defined by a tree of components. The root of the tree is called the system component. The leafs of the tree are called atomic (or basic or primitive) components. The tree structure is given by the decomposition of each non-atomic component. Thus, if we consider Sub^* as the transitive closure of the function Sub, then $S \notin Sub^*(S)$ for any S in the system architecture.

Note that we avoid to distinguish between component type and component instances to simplify the notation. Actually, the subcomponents of a component type may be instances of the same component type and the system is a component instance. We simply see two component instances as two distinct components with the same structure and renamed ports and subcomponents.

Given a component S, a contract for S is a pair $\langle A, G\rangle$ of assertions over V_S representing respectively an *assumption* and a *guarantee* for the component. Let $C = \langle A, G\rangle$ be a contract of S. Let I and E be respectively an implementation and an environment of S. We say that I is a implementation satisfying C iff $I \cap [\![A]\!] \subseteq [\![G]\!]$. We say that E is an environment satisfying C iff $E \subseteq [\![A]\!]$. We denote with $Imp(C)$ and with $Env(C)$, respectively, the implementations and the environments satisfying the contract C.

Two contracts C and C' are *equivalent* (denoted with $C \equiv C'$) iff they have the same implementations and environments, i.e., iff $Imp(C) = Imp(C')$ and $Env(C) = Env(C')$.

Contracts are used to specify the assumptions and guarantees of components in a system architecture. We denote with $\xi(S)$ the contracts of the component S.

Since the decomposition of a component S into subcomponents induces a composite implementation of S and composite environment for the subcomponents, it is necessary to prove that the decomposition is correct with regards to the contracts. In particular, it is necessary to prove that the composite implementation of S satisfies the guarantee of S's contracts and that the composite environment of each subcomponent U satisfies the assumptions of U's contracts. We perform this verification compositionally only reasoning with the contracts of the subcomponent independently from the specific implementation of the subcomponents or specific environment of the composite component.

Given a component S and a decomposition $\rho = \langle Sub, \gamma\rangle$, a set of contracts $\mathcal{C} \subseteq \bigcup_{S' \in Sub(S)} \xi(S')$ is a refinement of C, written $\mathcal{C} \leq_\rho C$, iff the following conditions hold:

1. for any implementation I of S induced by the implementations $\{I_{S'}\}_{S' \in Sub(S)}$ of the sub-components of S and the decomposition ρ, if, for all $S' \in Sub(S)$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in Imp(C')$, then $I \in Imp(C)$ (i.e., the correct implementations of the sub-contracts form a correct implementation of C);

2. for every subcomponent S'' of S, for every contract $C'' \in \xi(S'') \cap \mathcal{C}$, for any environment E'' of S'' induced by the decomposition ρ, the environment E of S and the implementations $\{I_{S'}\}_{S' \in Sub(S) \setminus \{S''\}}$ of the sub-components of S, if $E \in Env(C)$ and, for all $S' \in Sub(S) \setminus \{S''\}$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in Imp(C')$, then $E'' \in Env(C'')$ (i.e., for each sub-contract C'', the correct implementation of the other sub-contracts and a correct environment of C form a correct environment of C'').

C Patterns

The Othello language is an expressive and complex language. This can be a problem for a user non specifically trained with the formal languages. On one side, it is possible to write contracts that do not represent the wanted requirement. On the other side, it can happen to write a specification that represents a unnecessary hard satisfiability problem.

For this reason, we list here several constraint patterns, mostly inspired by the SPEED [SPE] and CESAR [CES] projects. The list is the result of an analysis of the use cases developed so far both internally and with our partners in Safecer [Saf] and Forever [FoR] projects. With a few exceptions all those realistic contracts can be expressed using only these patterns.

P01: Initial condition

The condition that must hold in the initial state of the system. It is a common mistake to write such constraint instead of **`always`** `condition`.

```
condition
```

P02: Invariant

A good condition will always hold (typically the negation of condition represents some bad state).

```
always condition
```

It is equivalent to the CESAR pattern F3.

P03: Bounded Delay

Each time a certain event has occurred, another event will occur within a minimum delay and a maximum delay. Very useful to express reactions to signals (e.g. a safe state is entered each time an error has occurred).

```
always (event1 implies (
(time_until(event2) >= interval_lower_bound) and
(time_until(event2) <= interval_upper_bound)
))
```

It is equivalent to the CESAR pattern R3: `Delay between event1 and event2 within interval`

P04: Assured Reaction

If a certain event has occurred, another event will eventually occur. Used in discrete time models.

```
always (event1 implies [not] in the future event2)
```

Analogue to the CESAR pattern F1: `whenever event1 occurs event2 [does not] occur[s]`

P05: Timed Reaction

Equivalent to **Bounded Delay** with 0 as lower bound, listed here because it is commonly needed.

```
always (event1 implies [not] time_until(event2) <= interval_upper_bound)
```

Analogue to CESAR pattern F1[7].

Combination of patterns

Several times there will be the need to use more than one pattern in the same assertion. For achieving this, just connect them together through conjuction or disjunction:

```
pattern1 (and | or)
pattern2 (and | or)
...
patternN
```

The following provides a reference sheet.

P01: Initial condition

```
condition
```

P02: Invariant

```
always condition
```

[7] As a technical note, notice that it not possible to express in Othello this CESAR pattern with a lower bound different from 0. The semantic of CESAR would be that event2 must occur during the interval and can occur before it. However, the current Othello language would force the event to not occur before the specified interval (as in **Bounded Delay**)

P03: Bounded Delay

```
always (event1 implies (
(time_until(event2) >= interval_lower_bound) and
(time_until(event2) <= interval_upper_bound)
))
```

P04: Assured Reaction

```
always (event1 implies [not] in the future event2)
```

P05: Timed Reaction

```
always (event1 implies [not] time_until(event2) <= interval_upper_bound)
```

Combination of patterns

```
pattern1 (and | or)
pattern2 (and | or)
...
patternN
```