OCRA: Othello Contracts Refinement Analysis Version 1.1

Michele Dorigatti, Stefano Tonetta*

Contents

1 Introduction

OCRA is a package of NuSMV3 which supports the specification and verification of contracts refinement. OCRA takes in input an Othello System Specification (OSS) and returns a verification report.

Contracts are specified using the language Othello to specify assumptions and guarantees of components.

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

2 Input language

2.1 Othello constraints

2.1.1 Concrete syntax

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

```
constraint    = atom |
                "not" constraint |
                constraint "and" constraint |
                constraint "or" constraint |
                constraint "implies" constraint |
                "always" constraint |
                "never" constraint |
                "in the future" constraint |
                "then" constraint |
                constraint "until" constraint |
                constraint "releases" constraint ;
atom          = "TRUE" |
                "FALSE" |
                term relational_op term |
                "time_until" "(" term ")" relational_op term |
                term ;
term          = variable |
                constant |
                term "+" term |
                term "-" term |
                term "*" term |
                term "/" term |
                "der" "(" variable ")" |
                "next" "(" variable ")" ;

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

where:

- `constant` is a constant number;
- `variable` is a string.

2.1.2 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. Appendix A provides a reference sheet.

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[1].

[1]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**)

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
```

2.1.3 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 $\sim \in \{<, >, =, \leq, \geq, \neq\}$[2].
- $a_0 + a_1 \dot{v} \bowtie 0$ 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;
- $\triangleright_{\sim c} e$ is a HRELTL formula.

We use standard abbreviations for $\vee$, $\rightarrow$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{R}$ (see, e.g., [CRT09]). $\triangleright$ corresponds to "time_until". "never" is an abbreviation for $\mathbf{G}\ \neg$. The other abstract connectives correspond the the concrete ones used in the previous section in a straightforward way.

2.1.4 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

[2] 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.

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 $f : \mathbb{R} \to \mathbb{R}$ 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 $I_0 = [0, 0]$);
- $f_i : \mathbb{R} \to \Sigma$ 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:

- $\langle \overline{f}, \overline{I} \rangle, i \models_p p_{curr}$ 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 $t' \in I_i, t - t' \sim c$;

2.1.5 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.

2.2 Othello Component Specification

2.2.1 Concrete syntax

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

```
OSS          = system_comp component* ;
system_comp  = "COMPONENT" comptype? "system" interface refinement? ;
component    = "COMPONENT" comptype interface refinement? ;
interface    = "INTERFACE" var* contract* ;
refinement   = "ASYNC"? "REFINEMENT" subcomponent* connection* refinedby* ;
var          = port | parameter | operation ;
port         = ("IN" | "OUT")? "PORT" name ":" type ";" ;
parameter    = "PARAMETER" name ":" type ";" ;
operation    = ("PROVIDED" | "REQUIRED") "OPERATION" "PORT"? name
               "(" op_parameter* ")" ":" (type | "void") ";" ;
op_parameter = ("IN" | "OUT")? name ":" type ;
type         = "boolean" | "integer" | "real" | "continuous" | "event" |
               "{" (number | name)+ "}" | number ".." number;
contract     = "CONTRACT" name
               "assume" ":" constraint ";"
               "guarantee" ":" constraint ";" ;
subcomponent = "SUB" name ":" comptype ";" ;
connection   = "CONNECTION" name ":=" constraint ";"
formula      = "CONSTRAINT" constraint ";"
refinedby    = "CONTRACT" name "REFINEDBY" contr_id+ ";" ;
contr_id     = name "." name ;
```

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;

- in the list of components forming the OSS, there exists a component whose name is `system`;
- the relationship that links a component to its subcomponents is not circular and form a tree rooted in the system component;
- the `constraint` in the definition of a `contract` in an `interface` must be an Othello constraint as defined above where every variable must match a variable of the interface;
- the `constraint` in the definition of a `connection` in the `refinement` of a `component` must be an Othello constraint as defined above where every variable must either match a port 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.

2.2.2 Abstract syntax and semantics

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'').

2.2.3 Operations

A special kind of port is called operation. It is much like a function in a traditional programming language: it has typed parameters and a return type. A component can

provide an operation or require it from another one.

If an operation is declared in a component in the way shown below, several ports are implicit declared and can be referenced:

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

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

Notice that 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.

3 Interactive Commands of OCRA

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

```
ocra_check_refinement [-h] [-i <file>] -I <file> [-c <name>]
[-f (text|xml)] [-o <file>]
```

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.

Command Options:

`-a string`	Force algorithm type. Valid values are: bmc, bdd, auto [default=auto]
`-f string`	Selects output format. Valid values are: text, xml [default=text]

`-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
`-o <file>`	Set the output file [default=stdout]

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

```
ocra_check_implementation [-h] [-i <file>] -I <file> [-c
<name>] [-f (text|xml)] [-o <file>]
```

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

Command Options:

`-c string`	Specify the component to be checked
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification
`-I file`	Reads the SMV specification
`-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>) [-C]
```

It checks the syntax of a model.

Command Options:

`-C`	Consider the specification as an assertion [default=no] (rather than a whole OCRA system specification) Equivalent to deprecated -c
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification

`-p <string>`	Read the specification from a string

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

```
go_ocra [-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

ocra_print_system_implementation - *Compute and prints a system implementation* — Command

```
ocra_print_system_implementation [-h] -i <file> -a <file>
[-o <file>]
```

Given an OSS specification, a set of SMV files and a configuration file with a map between component names and the smv file representing their implementation, outputs a single SMV file for the system implementation. The configuration file looks like this:

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

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
`-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>
```

It checks every assertion (assumption and guarantee) for being consistent, that is, satisfiable.

Command Options:

`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification

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

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

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:

`-c string`	Specify the component to be checked
`-h`	Shows a brief description of the available options.
`-i file`	Reads the OSS specification
`-I file`	Reads the SMV specification

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

```
ocra_check_composite_impl [-a <string>] [-f <string>] [-h]
[-i <file>] [-k <int>] -m <file> [-M] [-o <file>] [-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.

Command Options:

`-a string`	Force algorithm type. Valid values are: bmc, bdd, auto [default=auto]
`-f string`	Selects output format. Valid values are: text, xml [default=text]
`-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
`-m file`	Reads the file with the map (component name, component smv implementation file)
`-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_make_warnings_errors	Environment Variable

Treat every warning as an error. Default is false.

4 OCRA Command Line Options

`-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

References

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

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

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

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

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

A Pattern Reference

P01: Initial condition

```
condition
```

P02: Invariant

```
always condition
```

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
```