Validating Timed Component Contracts

Thibaut Le Guilly*, Shaoying Liu†, Petur Olsen*, Anders P. Ravn*, Arne Skou*
*Department of Computer Science
Aalborg University, Denmark
{thibaut,petur,apr,ask}@cs.aau.dk
†Department of Computer Science
Hosei University, Japan
sliu@hosei.ac.jp

***Abstract*—This paper presents a technique for testing software components with contracts that specify functional behavior, synchronization, as well as timing behavior. The approach combines elements from unit testing with model-based testing techniques for timed automata. The technique is implemented in an online testing tool, and we demonstrate its use on a concrete use case.**

***Keywords*—Component; Contract; Formal Specifications; Timed Automata; Test Case Generation; Online Testing**

I. Introduction

Component contracts need, in order to be completely specified, four levels of constraints [1]: syntactic, behavioral, synchronization and quality of service. Syntactical and behavioral specifications are well captured by pre-/post-condition languages such as VDM, SOFL [2] or rCOS [3]. Based on these, unit tests can be generated to validate that the specifications are satisfied, as shown in [4], [5], [6]. For specifications in the form of predicates, the *Functional Scenario Form* (FSF) [4], [5] constitute a practical way for generating test inputs. We recall these notions in Section II.

The synchronization level specifies the protocol of the component; the way in which its environment should interact with it. It is specified using regular expressions or equivalently Finite State Machines (FSM), or Timed Automata (TA) when timing information is important. Testing that a developed component satisfies its protocol and real time specifications is done during integration testing. We recall these notions in Section III.

A potential problem with separate unit and integration testing is that integration testing often does not test that the functional specifications of the component are still satisfied. Moreover, the behavior of some components is influenced by their environment, making it impossible to test it before deployment. An example is a home automation systems (considered as a component in a smart grid) or control systems in general that behave differently when deployed in different environments. Also, the interface to such systems is often provided through a middleware [7] which may introduce errors. In order to ensure an integrated test of this kind of component, Section IV presents an approach that combines the behavioral, synchronization and timing part of the component specifications into a timed automata model suited for model based testing. It enables different timing constraints for different executions of a single method. This model is in Section V used to build a testing tool that combines unit tests and integration tests. The tool is evaluated in Section VI on a use case taken from a European research project. We compare our approach to related work in Section VII, discuss limitations and potential enhancements in Section VIII, and conclude in Section IX.

II. Unit Testing

Unit testing of a component validates that its methods return correct output when provided with specific inputs, based on its contract [8]. Formal syntax and semantics similar to the ones defined in rCOS [9] and SOFL [2] provide good support for such specifications. Here we borrow the syntax used in rCOS. An interface I has a set $I.fields$ of typed variables of the form $x : T$, called fields, and a set of method signatures $I.methods$ of the form $m(x : T; y : V)$, where $x : T$ and $y : V$ are the input and output parameters of the method with their types. Pre-/post-conditions define the expected output based on the state of the component (the value of its fields) and the allowed inputs. The interface also has a set of *invariants* representing constraints on the component fields that should hold at entry and exit of methods. This notation can be used to generate inputs that cover the different possible executions of a method, and validating its outputs. To do so, the FSF transforms the pre-/post-conditions of the component interface to a part used for test case generation and one used as the oracle of the test case. Note that deriving the FSF of a method is based only on its specifications, and not its implementation. It is therefore likely that different scenarios cover different program paths, but it is also possible for a single scenario to be implemented by several program paths. Therefore, testing a method using its FSF is still considered black box testing. We recall here this notion.

Let's assume a method m, and its guarded design $pre_m(x) \vdash post_m(x, x')$ with x and x' the values of the variables before and after executing the method, and $pre_m(x)$ and $post_m(x, x')$ the pre- and post-conditions of the method.

Definition 1. *Let* $post_m \equiv \bigvee_{i=1}^{n} C_i \wedge D_i$*, where each* $C_i (i \in 1, \cdots, n)$ *is a predicate called a* "guard condition" *that contains no output variable, and where the* C_i *are mutually disjoint. Also,* D_i *is a* "defining condition" *that contains at least one output variable. Then, a* (functional) scenario F_i *of* m *is a conjunction* $pre_m(x) \wedge C_i \wedge D_i$*, and the expression* $\bigvee_{i=1}^{n} F_i$ *is called a* Functional Scenario Form *(FSF) of* m*.*

Intuitively, a scenario of a method represents one of its possible normal executions. For example, let's take a method `authenticate` of a component `AccountManager`,

TABLE I. AN EXAMPLE OF A COMPONENT INTERFACE IN THE PRE/POST FORMAT

Component	`AccountManager`
Fields	`int acc_num, int psw, int balance`
Methods	`authenticate(int p, int num; boolean auth){`
	`pre : p > 0 ∧ num > 0`
	`post : p = psw ∧ auth′ ∨ p ≠ psw ∧ ¬auth′`
	`}`
Invariant	`balance ≥ 0`

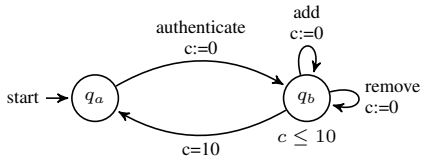

Fig. 1. A TA representing the interaction protocol of a component. This component requires authentication before requesting any of its functionalities. After 10 time units of inactivity, authentication should be performed again.

defined in a pre-/post-condition form shown in Table I. From this definition we can identify two scenarios. One that is triggered when the password `psw` is correct, another that is triggered when it is incorrect. We can thus rewrite the method in FSF with the two following scenarios:

Scenario 1 $p > 0 \wedge num > 0 \quad \wedge p = psw \quad \wedge auth'$
Scenario 2 $p > 0 \wedge num > 0 \quad \wedge p \neq psw \quad \wedge \neg auth'$

The first part represents the pre-condition, the second part the guard condition and the third part the defining condition. The first two thus define the condition on the state of the component fields for the scenario to execute, while the defining condition acts as an oracle predicting the result of the execution. Testing a functional scenario is done by executing a method in a state where the precondition of the scenario holds. In a more extensive robustness test, the precondition could be violated in an additional test case.

III. PROTOCOL AND REAL TIME TESTING

A synchronization specification of the component contract gives its interaction protocol; in other words the order in which its methods may be invoked. It thus encodes the state of a component, as well as the possible transitions from its current state, when a method of its interface is invoked. When time is of importance in the component contracts, TA can be used to capture the timing aspects. The objective is then to model the component states, the transitions between them and the timing constraints associated with the transitions. The transitions are triggered either by calls on a method of the component, or by time passing. We call transitions that are not triggered by a method call, *time only transitions*. These transitions make it possible to specify timeout constraints as well as delays. An example of a specification is shown in Figure 1. Such specifications can be used for conformance and robustness testing. In the example of Figure 1, one would for example be interested in testing that the methods `add` and `remove` can be called after authentication, but also that they cannot be called before.

With this simple model however we cannot test extra-functional requirements, as for example minimum and maximum response time for a method. To do so we model each method call with an input synchronization (denoted by a trailing "!") while the return from a method is modeled by an

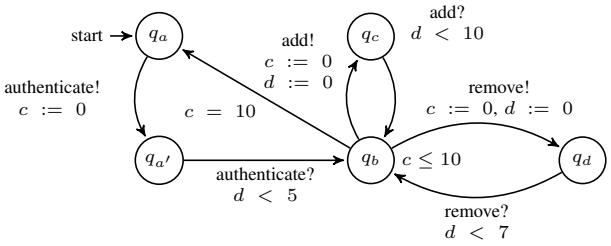

Fig. 2. A TA representing the interaction protocol of a component specifying the maximum execution time for each method.

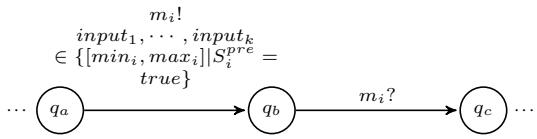

Fig. 3. The environment is modeled by a TA corresponding to the component protocol, where function calls are replaced as shown in this figure.

output action (denoted by a trailing "?"). Note that we assume that each of the methods provided by the component have an observable start and an observable end. An example of such an automaton is shown in Figure 2.

We are now capable of expressing extra-functional timing requirements for each methods. However, we cannot express different timing requirements for different scenario execution of each method. To do so we investigate a different approach.

IV. COMBINED SPECIFICATIONS

To combine the functional and timing specifications, we divide the component model into three parts. The first part represents the component environment, triggering the method calls. The second part represents the component methods, reacting on method calls, updating variables during execution and (possibly) returning a value. The third part of the model is a *scenario observer* that validates the correct execution of method scenarios. To include the functional properties of the component in the model, we augment it with a set of discrete global variables that consist of:

- the inputs and outputs for each method;
- the observable component fields used in at least one of the specification predicates.

A. Environment Model

The environment model is a modified version of the component protocol model where transitions triggered by method executions are split between output (the method call) and input (the method return) as in Figure 2. In addition, the method calls are augmented with assignments of values to discrete variables representing the input values of the method. A generic example is shown in Figure 3. The set of inputs is restricted to the ones that satisfy the method pre-condition.

B. Component Model

The component model is the parallel composition of its method models. An example of a method model is shown in

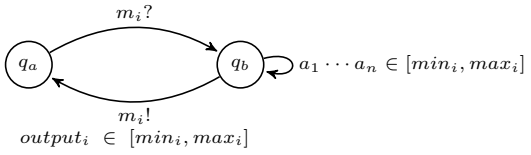

Fig. 4. Model of a method.

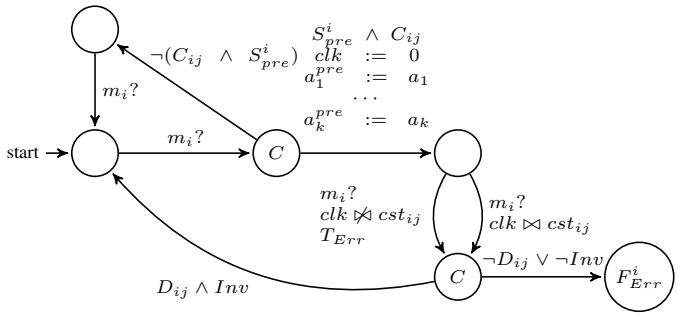

Fig. 5. Model of a scenario observer.

Figure 4. The first transition is triggered by a method call from the environment. During the execution of the method, the component fields are updated. Finally, method call termination is signaled to the environment and a return value ($output$) can be provided. Note that compared to other model based testing approaches, here the model can violate its specification. Instead, the conformance with respect to the specification is checked by an observer of a method scenario.

C. Scenario Observer

For each scenario of each method, we define a *scenario observer*. A scenario observer validates the conformance of the component execution with respect to its functional and timing specifications. Figure 5 shows a generic scenario observer. A scenario observer is triggered when the method it is associated with is called in a state that satisfy the guard conditions of the specific scenario it should observes. It then checks that the scenario is executed correctly according to both the functional and timing specification of the scenario. A scenario observer can detects two types of errors. Functional faults are detected when the F_{Err} location is reached by the observer, indicating that the defining condition of the scenario was violated. Timing faults are detected when taking the T_{Err} transition indicating that the timing relation $clk \bowtie cst$ (where clk is a clock and cst a constant) was not satisfied.

We have obtained a component model that encodes both the functional specification of the component as well as its protocol and timing requirements. We will now interpret this information for testing.

V. Testing Tool

To interpret the models, we built an online testing tool in Java, based on previous work in interpreting TA [10]. The tool is implemented as a Java library providing an interface for loading the component model as well as an instance of the component in the form of a Java object. The tool is implemented in Java and as such only targets testing of soft real time systems. However, the theory could be used and implemented differently to test hard real-time as well. The testing process has two phases:

Initialization	where the information contained in the model is parsed to generate required data structures;
Execution	where the TA is interpreted to generate test inputs and validate the outputs.

A. Testing Objectives

The objective of the testing tool is to detect functional and timing errors. In addition, it provides statistics on execution time of each scenario enabling a profiling of each of them. To obtain such statistics, each scenario is executed a defined number of times. As we cannot guarantee that all scenarios are tested, we terminate the test when all executable scenarios given the state of the model have been tested enough times.

B. Initialization Phase

At initialization, the model is parsed to load information about methods and scenarios. Method names are extracted from the channel names, and scenario constraints are extracted from observers. Each scenario constraint is passed to the Choco solver [11], which computes the domain of each variable given the constraint. This will be used to generate method inputs and check if the component fields satisfy a given scenario constraint. Note that time only transitions are also extracted and considered as a special type of scenario without associated method. This ensures that they are tested as many times as any method scenario.

C. Execution Phase

The execution of a test case is divided into two parts. First, the scenario to be executed is chosen, based on the current state of the component and the number of times each executable scenario has been tested. In case of ambiguity a scenario is chosen at random. Once selected, the method is executed with adequate inputs that satisfy the scenario constraints. In practice the input selection is implemented using a select statement that generates a single transition for each possible assignment of the inputs. This may lead to an explosion of the number of possible assignments which we discuss in Section VIII. After selecting the inputs and invoking the method, the transition representing the method call is taken in the model. At the end of execution, the return value and possibly modified fields are updated in the model (by taking corresponding transitions in the method model), and the transition representing the method termination is taken. Timing information is recorded, and eventually timing error, and in case of functional error the test is stopped and the error highlighted. The testing process thus consists of choosing a scenario and executing it, using the TA model to validate the timing and functional requirements and the specified protocol. Doing so we collect information that is used to generate a test result.

D. Test Result

As already mentioned, if a functional error is detected, it is immediately reported. If none are found, a summary of the scenario execution is presented, indicating if timing

errors have been observed and providing for each scenario: the number of test performed, the average, maximum and minimum execution time, and the success rate with respects to the timing constraints.

VI. Use Case

As illustration of the testing technique, we use a component that is part of a demonstration for the projects Arrowhead[1] and TotalFlex[2]. The objective in these projects is to improve the balance between energy consumption and production using the flexibility offered by some appliances. The flexibility of devices is encoded using the concept of flexoffer [12]. Flexoffers are then sent by appliances to an entity called an aggregator that combines the flexibility of multiple devices to make it easier to handle. More information on the concepts and system architecture can be found in [13]. In the demonstration system, two types of flex offers are created: from heat pumps, providing energy flexibility, and from washing machines, providing time flexibility. In order to preserve a maximum of flexibility, the aggregator is expected to aggregate these two sets of flex offers separately into at least two aggregated flex offers. Finally, the timing requirements are specified so that the demonstration can be conducted without too much delay when showing example of aggregation. Note that in a production context the aggregation would also be required to be time bounded as decisions about energy planning need to be taken fast.

The component interface of an aggregator has four methods and six fields, shown in Table II. Its protocol is shown in Figure 6. The timing constraints of each scenario are added to their FSF to facilitate the presentation. The fields represent in order, the total number of flex offers that were added, the number of flex offers of type heat pump, the number of flex offers of type washing machine, the number of aggregated flex offers, the number of scheduled aggregated flex offers and the number of scheduled flex offers. The first method provided by the component is `addFlexOffer`, used to add new flexoffers of a certain type. The two scenarios of this method reflect the possibility of adding two different type of flex offers, and the different timing constraints the fact that it takes longer to generate a flex offer for a heat pump than for a washing machine. The second method is the `aggregate` method, that performs the aggregation. Here three scenarios are defined, that depicts the fact that when only one type of flexoffer is available then only one aggregated flexoffer should be obtained. The last two methods perform the scheduling (assigning a consumption period and value) and disaggregation operations (the reverse of aggregation).

For testing, the component interface is wrapped into an adapter object that provides the adequate bindings between the variables of the interface and their implementation. For example, the number of flexoffers is obtained by getting the list of available flexoffers and return its length. Adapters for component interfaces are commonly used for online testing, and enable the testing of different component implementations. To be able to add large numbers of flexoffers but avoid issues with the select statement, we use a scaling factor for the

[1]http://www.arrowhead.eu/

[2]http://totalflex.dk/

TABLE II. The interface of the aggregator component.

Component	`Aggregator`
Fields	`int nb_fo, int nb_hp_fo, int nb_wm_fo, int nb_agg_fo, int nb_agg_sch, int nb_fo_sch`
Methods	`addFlexOffers(int num, int type)`
	$\texttt{scenario}_1 : \texttt{num} > 0 \wedge \texttt{num} \leq 10 \wedge$
	$\texttt{type} = \texttt{HP} \wedge \texttt{nb\_hp\_fo}' = \texttt{nb\_hp\_fo} + \texttt{num} \wedge \texttt{clk} \leq 5$
	$\texttt{scenario}_2 : \texttt{num} > 0 \wedge \texttt{num} \leq 10 \wedge$
	$\texttt{type} = \texttt{WM} \wedge \texttt{nb\_wm\_fo}' = \texttt{nb\_wm\_fo} + \texttt{num} \wedge \texttt{clk} \leq 2$
	`aggregate()`
	$\texttt{scenario}_1 : \texttt{nb\_wm\_fo} > 0 \wedge \texttt{nb\_hp\_fo} > 0 \wedge \texttt{nb\_agg\_fo}' > 1$
	$\wedge \texttt{clk} \leq 2 * \texttt{nb\_fo}$
	$\texttt{scenario}_2 : \texttt{nb\_wm\_fo} = 0 \wedge \texttt{nb\_hp\_fo} > 0 \wedge \texttt{nb\_agg\_fo}' \geq 1$
	$\wedge \texttt{clk} \leq 2 * \texttt{nb\_fo}$
	$\texttt{scenario}_3 : \texttt{nb\_hp\_fo} = 0 \wedge \texttt{nb\_wm\_fo} > 0 \wedge \texttt{nb\_agg\_fo}' \geq 1$
	$\wedge \texttt{clk} \leq 2 * \texttt{nb\_fo}$
	`schedule(){`
	$\texttt{scenario}_1 : \texttt{nb\_agg\_fo} > 0 \wedge \texttt{nb\_agg\_sch}' = \texttt{nb\_agg\_fo} \wedge \texttt{clk} \leq \texttt{nb\_fo}$
	`disaggregate()`
	$\texttt{scenario}_1 : \texttt{nb\_agg\_sch} > 0 \wedge \texttt{nb\_fo\_sch}' = \texttt{nb\_fo} \wedge \texttt{clk} \leq \texttt{nb\_fo}$
Invariant	$\texttt{nb\_fo} = \texttt{nb\_hp\_fo} + \texttt{nb\_wm\_fo}$

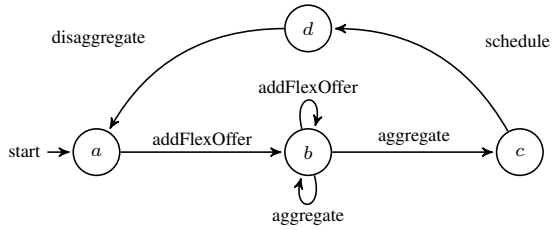

Fig. 6. The protocol of the aggregator component.

`addFlexOffer` method, that we varied between 10^1 to 10^5. Note that this does not impact the testing results but only their interpretation, and that fixing the select statement issue would make it possible to test without this scaling factor. For each test, the maximum number of execution per scenario was set to 30. We can draw two types of results from these experiments. Regarding the component, we did not detect any violation of the functional requirements, which increases our trust in it. For timing requirements, the generation of flexoffers for heatpump violated the constraints when trying to generate more than 10^5 flexoffers. All other methods satisfied their constraints. Regarding the testing tool, we note that test case generation did not noticeably impact the execution of the test, and that the fact that two methods had only one scenario influenced the coverage of other method scenarios. Since in location c in Figure 6 only `schedule` is active, the testing process stops, even though the scenarios of the methods `addFlexOffer` and `aggregate` were tested only half this number of times.

VII. Related Work

To solve the select statement issue, a symbolic representation of variable could be used, as in [14]. The authors present an extension of TA and Input Output Transition Systems that we will consider for future work. However, we note that the objectives mentioned in that paper differ from ours in that they use offline test case generation.

A natural choice of tool to interpret the presented models could have been Uppaal TRON [15], an online conformance testing tool for TA. However, the random exploration of the model would make it less likely to test different scenarios of a method. In order to do so, a constraint solver seems to be better. We also experimented with generating test cases using

reachability queries as proposed in [16]. However, the use of discrete variables and select statements leads to state space explosion rapidly when using multiple variables. Even when a test case can be generated, the computation time is higher than that of the constraint solver.

VIII. Discussion and Further Work

A limitation of the tool comes from the implementation of the select statement in UPPAAL. A select statement assigns a value within an interval to a temporary variable, which can then be used to update a local or global variable. In UPPAAL, select statements are replaced with a set of transitions that correspond to each possible assignment of a variable. The number of transitions per select statement is thus equal to the length of the select interval. This implies that if several select statements are defined on a single edge, the number of transitions generated from them will be equal to the product of their interval length. When this number grows large, it becomes impossible to select the correct transition within an acceptable time. This limits the scalability of the tool with respect to the number of inputs and their domain. However, even though this implementation of the select statement can be useful for model checking and exploring the entire set of possible values for each variable, for testing its interest is limited. Instead, we plan a different implementation where one can choose which value to assign to a variable within a single transition. The Timed Input Output Symbolic Transition System defined in [14] is a possible solution.

We also note that the construction of the models could be automated. In fact, the model presented in Section IV can be derived from a TA model of the component protocol, the specification of the component interface in FSF and the timing requirements for each scenario. These timing requirements are easily integrated into the FSF by adding a conjunction with a clock constraint to each scenario constraint as we showed in the use case. This would further simplify the testing process and make it easier to integrate into development methodologies such as rCOS or SOFL.

Finally, the experiments showed that depending on the topology of the protocol and the number of scenarios per method, it can be difficult to obtain a good coverage. Proposing different criteria for stopping the tests could help at improving coverage. An example could be for the user to specify a sequence of method calls to be repeated a number of times.

IX. Conclusion

In this paper we proposed a technique that allows the modeling of functional, synchronization and timing requirements of component contracts. The technique combines aspects of unit testing and integration testing. This allows the validation of functional requirements while testing the protocol of the application. We then showed how we used these models to develop an online testing tool that checks the conformance of a component implementation under test with respect to the models. The tool also generates statistical information about the method execution timing constraints. We demonstrated the use of this tool on a concrete use case taken from ongoing research project in the domain of the energy efficiency.

Acknowledgment

This work is partially supported by the Danish project TotalFlex, the European project Arrowhead and by JSPS KAKENHI Grant Number 26240008.

References

[1] A. Beugnard, J. Jezequel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, Jul 1999.

[2] S. Liu, *Formal Engineering for Industrial Software Development*. SpringerVerlag, 2004.

[3] H. Jifeng, X. Li, and Z. Liu, "rCOS: A refinement calculus of object systems," *Theoretical Computer Science*, vol. 365, no. 12, pp. 109 – 142, 2006, formal Methods for Components and Objects Formal Methods for Components and Objects.

[4] S. Liu, "Integrating specification-based review and testing for detecting errors in programs," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, M. Butler, M. Hinchey, and M. Larrondo-Petrie, Eds. Springer Berlin Heidelberg, 2007, vol. 4789, pp. 136–150.

[5] S. Liu and S. Nakajima, "A decompositional approach to automatic test case generation based on formal specifications," in *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, June 2010, pp. 147–155.

[6] P. Olsen, J. Foederer, and J. Tretmans, "Model-based testing of industrial transformational systems," in *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*, 2011, pp. 131–145.

[7] T. Le Guilly, P. Olsen, A. Ravn, J. Rosenkilde, and A. Skou, "Homeport: Middleware for heterogeneous home automation networks," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, 2013, pp. 627–633.

[8] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992.

[9] Z. Liu, C. Morisset, and V. Stolz, "rCOS: Theory and tool for component-based model driven development," in *Fundamentals of Software Engineering*, ser. Lecture Notes in Computer Science, F. Arbab and M. Sirjani, Eds. Springer Berlin Heidelberg, 2010, vol. 5961, pp. 62–80.

[10] P. Dalsgaard, T. Le Guilly, D. Middelhede, P. Olsen, T. Pedersen, A. Ravn, and A. Skou, "A toolchain for home automation controller development," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, 2013, pp. 122–129.

[11] X. L. Charles Prud'homme, Jean-Guillaume Fages, *Choco3 Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[12] L. Siksnys, M. Khalefa, and T. Pedersen, "Aggregating and disaggregating flexibility objects," in *Scientific and Statistical Database Management*, ser. Lecture Notes in Computer Science, A. Ailamaki and S. Bowers, Eds. Springer Berlin Heidelberg, 2012, vol. 7338, pp. 379–396.

[13] L. Ferreira, L. Siksnys, P. Pedersen, P. Stluka, C. Chrysoulas, T. le Guilly, M. Albano, A. Skou, C. Teixeira, and T. Pedersen, "Arrowhead compliant virtual market of energy," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–8.

[14] W. Andrade, P. Machado, T. Jeron, and H. Marchand, "Abstracting time and data for conformance testing of real-time systems," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 9–17.

[15] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using Uppaal-TRON: An industrial case study," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 299–306.

[16] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds. Springer Berlin Heidelberg, 2008, vol. 4949, pp. 77–117.