COMPUTATION TOOLS 2014

The Fifth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking

ISBN: 978-1-61208-344-5

May 25 - 29, 2014

Venice, Italy

COMPUTATION TOOLS 2014 Editors

Wolf Zimmermann, Martin-Luther University Halle-Wittenberg, Germany

Petre Dini, Concordia University, Canada | China Space Agency Center, China

COMPUTATION TOOLS 2014

Foreword

The Fifth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2014), held between May 25-29, 2014 in Venice, Italy, continued an event under the umbrella of ComputationWorld 2014 dealing with logics, algebras, advanced computation techniques, specialized programming languages, and tools for distributed computation. Mainly, the event targeted those aspects supporting context-oriented systems, adaptive systems, service computing, patterns and content-oriented features, temporal and ubiquitous aspects, and many facets of computational benchmarking.

The advent of advanced computing embracing various forms of computational intelligence, large-scale strategies, and technology-oriented approaches relays on fundamental achievements in systems and feature specification, domain-oriented programming and deployment platforms and benchmarking.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2014 Technical Program Committee, as well as all of the reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and efforts to contribute to COMPUTATION TOOLS 2014. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations, and sponsors. We are grateful to the members of the COMPUTATION TOOLS 2014 organizing committee for their help in handling the logistics and for their work to make this professional meeting a success.

We hope that COMPUTATION TOOLS 2014 was a successful international forum for the exchange of ideas and results between academia and industry and for the promotion of progress in the areas of computational logics, algebras, programming, tools, and benchmarking.

We are convinced that the participants found the event useful and communications very open. We hope that Venice, Italy, provided a pleasant environment during the conference and everyone saved some time to enjoy the charm of the city.

COMPUTATION TOOLS 2014 Chairs:

Kenneth Scerri, University of Malta, Malta
Alexander Gegov, University of Portsmouth, UK
Ahmed Khedr, University of Sharjah, UAE
Torsten Ullrich, Fraunhofer Austria Research GmbH - Graz, Austria
Zhiming Liu, UNU-IIST, Macao
Lev Naiman, University of Toronto, Canada
Ingram Bondin, University of Malta, Malta
Tomáš Bublík, Czech Technical University in Prague, Czech Republic

COMPUTATION TOOLS 2014

Committee

COMPUTATION TOOLS Advisory Chairs

Kenneth Scerri, University of Malta, Malta
Alexander Gegov, University of Portsmouth, UK
Ahmed Khedr, University of Sharjah, UAE

COMPUTATIONAL TOOLS Industry/Research Chairs

Torsten Ullrich, Fraunhofer Austria Research GmbH - Graz, Austria
Zhiming Liu, UNU-IIST, Macao

COMPUTATION TOOLS Publicity Chair

Lev Naiman, University of Toronto, Canada
Ingram Bondin, University of Malta, Malta
Tomáš Bublík, Czech Technical University in Prague, Czech Republic

COMPUTATION TOOLS 2014 Technical Program Committee

François Anton, Technical University of Denmark, Denmark
Henri Basson, University of Lille North of France (Littoral), France
Steffen Bernhard, TU-Dortmund, Germany
Ateet Bhalla, Oriental Institute of Science & Technology - Bhopal, India
Paul-Antoine Bisgambiglia, Université de Corse, France
Narhimene Boustia, Saad Dahlab University - Blida, Algeria
Luca Cassano, University of Pisa, Italy
Emanuele Covino, Università di Bari, Italy
Hepu Deng, RMIT University - Melbourne, Australia
Eugene Feinberg, Stony Brook University, USA
Tommaso Flaminio, University of Insubria, Italy
Janos Fodor, Obuda University, Hungary
Alexander Gegov, University of Portsmouth, UK
Luis Gomes, Universidade Nova de Lisboa, Portugal
Rajiv Gupta, University of California - Riverside, USA
Fikret Gurgen, Bogazici University - Istanbul, Turkey
Hani Hamdan, École Supérieure d'Électricité (SUPÉLEC), France
Cornel Klein, Siemens AG - Munich, Germany
Stano Krajci, Safarik University - Kosice, Slovakia
Giovanni Lagorio, University of Genova, Italy
Tsung-Chih Lin, Feng-Chia University, Taichung, Taiwan
Giuseppe Longo, Ecole Normale Supérieure Paris, France
Glenn R. Luecke, Iowa State University, USA
Elisa Marengo, Free University of Bozen-Bolzano, Italy

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Dessert, an Open-Source .NET Framework for Process-Based Discrete-Event Simulation

Giovanni Lagorio

DIBRIS - University of Genova
Genova, Italy
Email: giovanni.lagorio@unige.it

Alessio Parma

Finsa S.p.A.
Genova, Italy
Email: alessio.parma@finsa.it

***Abstract*—We present Dessert, an open-source framework for process-based discrete-event simulation, designed to retain the simplicity and flexibility of SimPy, within the strongly-typed .NET environment. Both frameworks build domain-specific languages, for simulation writing, by using existing constructs in a novel way and providing a rich library of classes. By exploiting .NET generic types and iterators, we have successfully retained, and in few places even enhanced, the lean syntax and usability of the original library, without sacrificing static type checking. Static type-safety, in addition to being a very important property by itself, facilitates runtime code optimizations; indeed, benchmarks show that our Dessert outperforms SimPy.**

Keywords–Discrete-event simulation; .NET; Python.

I. INTRODUCTION

DES (Discrete-Event Simulation) is an intuitive and flexible form of modeling that enables to represent and simulate complex systems in a wide range of application domains, from logistics and supply chain management, to health care. In this paper, we present Dessert, a process-based DES framework for .NET, explaining the rationale behind its design, and discussing the technical challenges we have faced during its development. The design of Dessert has been heavily inspired by SimPy [1] [2], which exploits Python *generators* [3], a special form of coroutines [4], for writing process-based simulations cleanly and easily.

Being written in, and consumed from, Python can be seen as a double-edged sword for SimPy, since typing errors are found at runtime and the (dynamic) typechecking overhead harms simulation running times. In designing Dessert we have striven to create a first-class "citizen" in the strongly-typed .NET environment, while retaining the lean syntax and usability of SimPy. For instance, Figure 1 shows a simple example simulation in Python, using SimPy, and Figure 2 shows the same example, written in F# using Dessert. This example is described more in Section II but, as the reader can easily verify, both listings are, with the exception of small syntactic differences, quite similar and very readable; indeed, even without knowing anything about SimPy or Dessert, the meaning of the simulation can be easily inferred.

We have developed Dessert as an open-source project, readily available via both *NuGet* [5], the package-management platform for .NET, and *GitHub* [6], one of the most popular hosting service for software development projects. *Any* .NET language can be used to write simulations to be run on our engine, since it complies with the *Common Language Specification (CLS)*, a strict subset of the .NET *Common Type System* that describes how to design types that can be manipulated by any CLS consumer [7].

The paper is organized as follows: Section II gives an overview of SimPy and Dessert, Section III analyzes design and implementation issues, and Section IV compares the performance of our framework in various environments. Finally, Section V discusses related work, while Section VI outlines some concluding remarks and further work.

```
1  import simpy
2
3  def car(env):
4   while True:
5     print('Start parking at %d' % env.now)
6     parking_duration = 5
7     yield env.timeout(parking_duration)
8     print('Start driving at %d' % env.now)
9     trip_duration = 2
10    yield env.timeout(trip_duration)
11
12 env = simpy.Environment()
13 env.process(car(env))
14 env.run(until=15)
```

Figure 1. A simple example of SimPy (Python).

```
1  open Dessert
2
3  let rec car(env:SimEnvironment)=seq<SimEvent> {
4     printfn "Start parking at %g" env.Now
5     let parkingDuration = 5.0
6     yield upcast env.Timeout(parkingDuration)
7     printfn "Start driving at %g" env.Now
8     let tripDuration = 2.0
9     yield upcast env.Timeout(tripDuration)
10    yield! car(env)
11 }
12
13 let env = Sim.NewEnvironment()
14 env.Process(car(env)) |> ignore
15 env.Run(until = 15.0)
```

Figure 2. A simple example of Dessert (F#).

II. OVERVIEW OF SIMPY AND DESSERT

As mentioned in Section I, SimPy exploits Python *generators* for writing process-based simulations. Indeed, in SimPy a *process* is simply a generator function, which is used to model active components like customers, vehicles or agents.

 ISBN: 978-1-61208-344-5

All processes live in an *environment*, and interact with it and with each other via *events*. This is shown in Figure 1, where the function `car` is used to model a process where a car alternates between being parked and driving. More in detail, an environment `env` is created (line 12), then a process is created by passing `car(env)` to the method `process` of the environment (line 13) and the simulation is run for 15 units of time, by calling `run(until=15)` (line 14). The process defined by the function `car` enters in an "infinite" (that is, until the simulation runs) loop that consists in:

- "parking the car", simulated by suspending the process for five units of time by yielding a *timeout event*, created by calling `env.timeout` (line 7);
- "driving the car", simulated by suspending for two units of time (line 10).

The environment `env` is used both to create new events and to get the *current time*, given in simulation units, by accessing `env.now` (lines 5 and 8). It is up to the simulation writers to decide what a *unit of time* corresponds to; for some simulations using seconds is a sensible choice, for others it makes more sense to use minutes and so on. While in SimPy simulations can be performed "as fast as possible", in real time or by manually stepping through the events, the current version of Dessert always run simulations at "full speed", so the simulation time never corresponds to the real (wall clock) time.

Figure 2 shows the same simulation of Figure 1, but written in F# using Dessert. As the reader can see, the former is just a little more verbose and, more importantly, contains type annotations (for instance, `env : SimEnvironment`, which declares that the parameter `env` must comply with the type `SimEnvironment`) that are statically checked by the compiler.

While *events* are obviously the central topic of DES, Dessert also provides some utility types for representing:

- *resources* (modeled by classes `Resource` and `PreemptiveResource`), which can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps);
- *containers* (`Container`), which model the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples);
- *stores* (`Store<T>` and `FilterStore<T>`), which are resources that enable the production and consumption of discrete objects of type `T`.

Moreover, other classes aid in gathering statistics about resources and processes. Given the available space we cannot detail all features, so we give an overview of the key concepts by means of the small, yet feature packed, following example.

Figure 3 contains a stripped down version of a process representing a *network switch*, which is used in the peer to peer simulation presented in Section IV-C. In this simulation, the switch waits quietly for incoming frames and, when one arrives, the switch delivers it to the right target. However, to perform its work, the switch needs to temporarily store incoming frames inside the buffer `_buffer`, which can only store `G.BufferSize` frames. When the buffer is full, any incoming frame is simply dropped, that is, thrown away; this fact is logged, inside method `Receive`, by invoking `G.Stats.DroppedFrame()`. We point out that, except for syntactic differences, this code is analogous to the one that it could have been written for SimPy. This is not just a matter of name similarity: the key point is that the usage of generators is *fully* preserved. Consider, for instance, the method `Run`, at line 7, which implements the behavior of the switch as an *infinite* generator. The body of the method just consists of an infinite loop, which contains the instructions that "animate" the switch. In particular, the first `yield return` yields an event that corresponds to the wait for an incoming frame. When an incoming frame `f` arrives, method `Receive` puts `f` in the buffer `_buffer`, awakening `Run`, that continues its execution at line 11. From there, the process calls a subroutine, `WaitForSend` (not shown), that stops the switch for the required time to send the frame `f`; then, as the final step, the frame is really sent to the proper target.

```
 1  sealed class Switch : Entity {
 2     readonly Store<Frame> _buffer;
 3     Switch(SimEnvironment e, G g) : base(e, g) {
 4        var cap = G.BufferSize;
 5        _buffer = Sim.NewStore<Frame>(e, cap);
 6     }
 7     IEnumerable<SimEvent> Run() {
 8        while (true) {
 9           var getFrame = _buffer.Get();
10           yield return getFrame;
11           var f = getFrame.Value;
12           var w = WaitForSend(f, f.Len);
13           yield return Env.Call(w);
14           Send(f);
15        }
16     }
17     void Receive(Frame f) {
18        if (_buffer.Count == G.BufferSize)
19           G.Stats.DroppedFrame();
20        else
21           _buffer.Put(f);
22     }
23     void Send(Frame f) {
24        if (f.Type == FrameType.Request)
25           G.ServerOSes[p.Dst].Receive(f);
26        else
27           G.ClientOSes[p.Dst].Receive(f);
28     }
29  }
```

Figure 3. The switch process in (C#).

In this example, the buffer is represented as store of `Frame`, `Store<Frame>`, allowing us to use its blocking operations (`Get`, in this case), to stop the process until the buffer contains something to get. While the API of Dessert resembles the one of SimPy, there are some important differences. On the one hand, as we detail in Section II, everything, from events to stores, is strongly typed in Dessert so, for instance, local variable `f`, in line 12, has (inferred) static type `Frame`, since `_buffer` has static type `Store<Frame>`.

On the other hand, our goal was not to make a *straight* "clone" of SimPy, but keeping what we liked (*a lot* of design choices and features) while trying to improve the usability even more, by making some changes and additions. One addition is the introduction of a new type of events, the *call events*. In the example, a call event is used at line 14, and expresses a "call" to a *subgenerator*. While newer versions of Python, since version 3.3, elegantly handle this situation by using the new `yield from` expression [8], previous versions of Python and all mainstream .NET languages do not offer such a feature.

 ISBN: 978-1-61208-344-5

```
def f(): # f is a generator function
  two = yield 1 # two gets the argument of send
  yield 3
  # ...

g = f() # gets the generator
one = g.next() # gets the 1st yielded value
three = g.send(2) # gets the 2nd yielded value
```

Figure 4. Example of Python generators.

A possible workaround, not particularly elegant nor intuitive, is to iterate through the result of a subgenerator call, $v_1, v_2, \ldots$, yielding each value v_i. We think that expressing these calls through our *call events* makes the code more readable and intuitive.

III. Design and implementation issues

In this section, we first describe a couple of prerequisites, common to any implementation of a DES engine, and then we focus on typing issues.

The common prerequisite are: an efficient priority queue, to store the *(pending) event set*, and a random number generator, able to deal with various probability distributions. Curiously, both are absent in the .NET standard library. While there is not a single data structure that is the best choice for storing the event set in all situations, an *heap* is a fairly reasonable choice [9]. For this reason, we have implemented, and experimented with, various kinds of heaps (array, binary, binomial, Fibonacci, and pairing) and finally settled with a *skew heap* [10] that, in our experiments, outperformed all the other kinds. The standard .NET `System.Random` class only provides the uniform number distributions; fortunately, we have found, and used, an excellent free library [11], that supports four different random number generators and many discrete and continuous probability distributions.

When "translating" the idea of modeling a process as *generator function* yielding *events*, one of the major issues we had to face has been the fact that in Python the *yield* construct is an expression, while in .NET the corresponding construct is a statement. Moreover, in our settings, the type of such a value should be statically determined. More in detail, in Python a function f containing an `yield` expression e, that is, $e \equiv$ `yield` e', is a *generator function*, which returns an iterator, known as a *generator* g, which is an automatically generated object that permits to iterate through the values generated by evaluating the yield-expressions. The evaluation of e suspends the execution of f, which is then resumed when a method (as `next` or `send`) is invoked on g. The resulting value of e depends on the method which resumed the execution; for instance, consider the snippet of code shown in Figure 4: the call `g.next()` evaluates the body of `f` until it yields the value `1`, which is assigned to `one`. The subsequent call `g.send(2)` resumes the evaluation of `f`, that assigns `2` to `two` and yields the value `3`, which is finally assigned to `three`.

This passing values back-and-forth works very well for SimPy, where triggered events can "return" values (by `send`ing them to the generator). We tried to translate this idea in .NET as close as possible; unfortunately, in all mainstream .NET languages there are some critical differences in how generators work. Terminology aside (we stick with the Python terminology, double-quoting Python terms when used in place of the .NET terms, to make the comparison easier to follow), a method m containing a `yield return` statement s is a "generator function", whose invocation returns a "generator" g. As in Python, the evaluation of s suspends the execution of m, which is resumed when the parameterless method `MoveNext()` is called on g. The key difference is that `yield return` is a *statement*, so there is no way to pass a value v to be used as the "resulting value" of evaluating s (since s, being a statement, does not evaluate to a value!). Since a process yields only events, we introduced the (read-only) property `Value` in `SimEvent`, the supertype of all event types in Dessert, to emulate the Python behavior: when the execution of a generator function f is resumed, after an invocation of `MoveNext()` on the corresponding generator, f can retrieve the "returned/sent" value by reading the property `Value` of the yielded event object; see, for instance, lines 10 and 11 of Figure 3.

In order to statically type these values, we would have liked to introduce a generic type `SimEvent<TVal>`, exposing a property `Value` of type `TVal`, to represent events that "return" values of type `TVal`. Unfortunately, the situation is more complex than that: each event type E must also expose a collection of callbacks `Callbacks`, which are invoked when the event is triggered. In .NET the standard type to model a strongly-typed callback, that gets an object of type E as the only argument, is `Action<`E`>`. If we try to implement this common interface in `SimEvent<...>` we stumble in an inherent recursion: if E is an event type returning values of type T, then E should be a subtype of `SimEvent<`T`>`, which, in turn, should expose a collection of `Action<`E`>`. This is a known and recurring situation [12] that can be solved by introducing a second type-argument; indeed, we have defined `SimEvent<TEv,TVal>` where `TEv` is the type of the event, and `TVal` is the type of the "returned" values, as described above. In this way, inside `SimEvent<TEv,TVal>`, we can declare a collection of strongly-typed callbacks as `ICollection<Action<TEv>>`. For instance, consider `Timeout<T>`, which represents events that are scheduled with a certain delay and return values of type `T`. Such a type (indirectly) extends `SimEvent<Timeout<T>,T>`; so when we create a timeout event of type, say, `Timeout<double>` we obtain an object that exposes the collection `Callbacks` of type `ICollection<Action<Timeout<double>>>`. This guarantees that the callbacks of `Timeout<double>` are, correctly, a collection of `Action<Timeout<double>>`.

In SimPy, and so in Dessert, events can be combined together to form *event conditions*, that is, events that are triggered when some *condition* becomes true. For instance, given two events e_1 and e_2, we could create a new condition event $e_{\texttt{AND}}$ that is triggered when both e_1 and e_2 are triggered, or create another event $e_{\texttt{OR}}$ that is triggered when *any* of them is triggered, and so on. In general, any number of events and any predicate p can be specified, allowing simulation authors to build arbitrarily complex conditions that are triggered whenever p becomes true on the given events. A common use of event combinations is implementing timeout policies; for instance, given a certain event e, obtaining a new event that corresponds to waiting for e *or* the expiration of a timeout event.

Differently from SimPy, in our strongly-typed settings, the result of combining arbitrary events, of types $t_1,\ldots,t_n$,

 ISBN: 978-1-61208-344-5

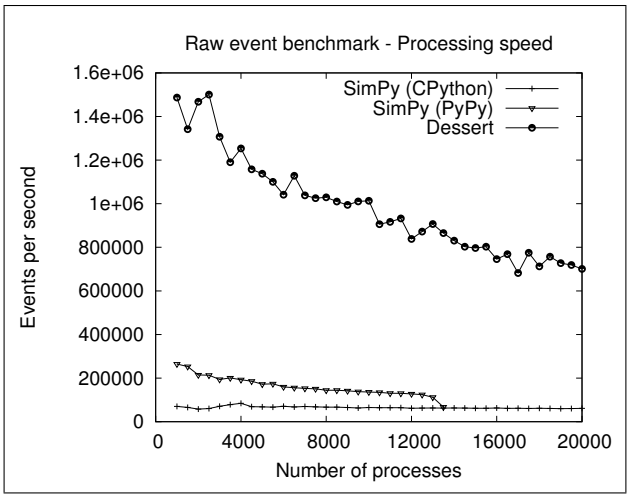

Figure 5. Events per second in timeout benchmark.

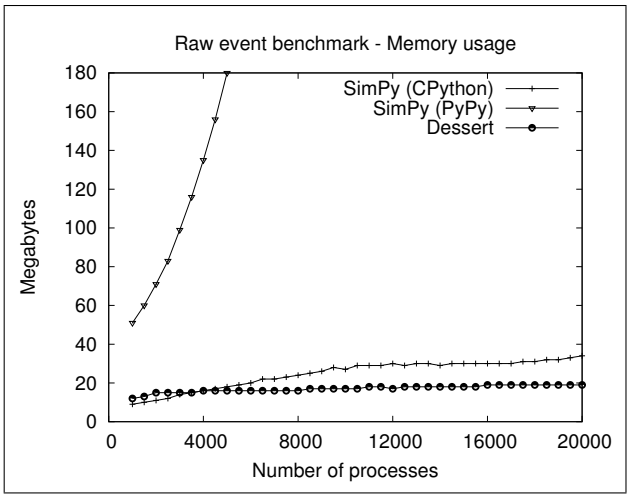

Figure 7. Memory usage in timeout benchmark.

must have a type that "remembers" the types t_1,...,t_n. That is, if an event has some type t_1, and another event has type t_2, then the resulting type of combining them must include (some encoding of) both t_1 and t_2. For this reason, we use the "variadic-generic type" `Condition<`$t_1,\ldots,t_n$`>` to encode the type of condition events built from events of type $t_1,\ldots t_n$. Inside an object of type `Condition<`$t_1,\ldots,t_n$`>` the source events are available through the read-only properties named `Ev`N, where $N \in \{1,\ldots,n\}$. Note that `Condition` cannot really be single generic type, since in .NET each generic type is constrained to have a fixed number of type arguments. Handling the combination of an arbitrary number of events would require a mechanism analogous to C++ variadic templates [13] which, at the moment, is not available in C# (and the other mainstream .NET languages). So, we had to use a family of generic types (`Condition<T1>`, `Condition<T1,T2>`, `Condition<T1,T2,T3>` and so on); fortunately, this family of types can be automatically generated, for any arbitrary number of type arguments, by exploiting the T4 [14] (Text Template Transformation Toolkit) offered by Visual Studio.

IV. BENCHMARKS

In this section, we describe the benchmarks used to assess the relative performance of our Dessert, with respect to SimPy. We start, in Section IV-A, with the specifications of the machines used to run the benchmarks and the general description of the benchmark environment. Then, we describe the two kinds of benchmarks we carried out. The former, described in Section IV-B, is an artificial simulation, akin to a stress-test, where we obtain the average raw event processing time of the engines. The latter, described in Section IV-C, consists in running a real simulation of a peer-to-peer (P2P) system, thus measuring how the different engines perform on a "real-world" simulation.

```
def timeoutBenchmarkProcess(env, counter):
  while True:
    yield env.timeout(randomDelay())
    counter.increment()
```

Figure 6. Benchmark process.

A. Benchmark environment

Every benchmark has been run under a dedicated virtual machine (VM), created and run by VirtualBox [15] 4.3.2, hosted on Ubuntu 13.10 on a Intel Core 2 Duo E4700 with 4 GB or RAM. We created a Windows VM, with Windows 7 SP1 and the .NET Framework 4.5.1, and a GNU/Linux one, with Lubuntu [16], a lightweight variant of the more famous Ubuntu, and Mono 3.2. Both VMs share the same hardware profile: 2 CPU cores and 2 GB of RAM. Since DES is a strongly CPU-bound process, neither Dessert nor SimPy use secondary storage, we do not detail storage specifications. As concerns Python interpreters, we tried out both CPython 2.7, the "default" language implementation, and PyPy [17] 2.2, a recent and highly optimized alternative Python implementation. Both implementations have been run with full optimizations enabled (`-OO` flag).

In order to time our benchmarks, we started a virtual stopwatch at the beginning of each run and we stopped it at the end. On .NET we used a standard dedicated class, `System.Diagnostics.Stopwatch`, while on Python we used the facilities exposed by the general `time` module. To evaluate memory usage, we sampled the *resident set size* (RSS) of the (operating system) process at fixed intervals, by taking advantage of the standard class `System.Diagnostics.Process` on .NET, and of the library `psutil` [18] on Python.

In the following sections, for lack of space, we thoroughly analyze only the benchmarks on Windows. Anyway, the tests on GNU/Linux confirmed what we found on Windows, with a caveat: since Mono is not as optimized as .NET, Dessert still outperforms SimPy on CPython, but PyPy becomes an interesting competitor, yielding better results in the P2P simulation tests, but consuming an enormous quantity of memory, as it does on Windows. Given Mono continuous improvements, the performance of Dessert on Linux can only get better, so we hope to achieve soon the same results we already obtain on Windows.

B. Raw event processing benchmark

The goal of this benchmark is to measure the raw event processing speed of the DES engines. To do this, we designed a rather artificial simulation, in which we spawn an increasing number of extremely simple processes, as shown in Figure 6.

 ISBN: 978-1-61208-344-5

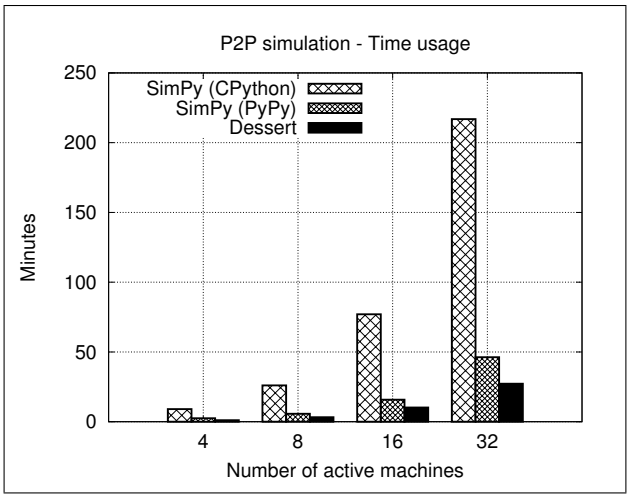

Figure 8. Time usage in P2P simulation.

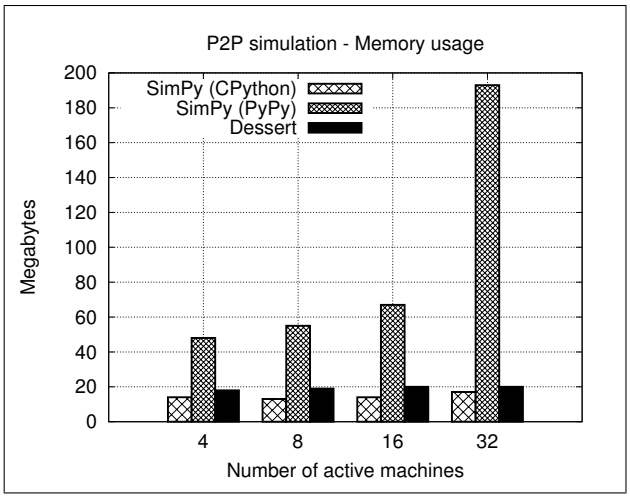

Figure 9. Memory usage in P2P simulation.

Each of those processes simply awaits a random timeout and, when woken up, increases a shared counter, which records the total amount of timeout events properly handled by the engine. So, by dividing that counter by the simulation execution (wall) time, we obtain a good approximation of the average event handling speed. Analogously, we measure how much memory each engine consumes by repeatedly running the same simulation and varying the number of processes from an already significant $1,000$, to a rather big $20,000$. In this way, we evaluate how the engines perform when heavily loaded. For any given process count, we have run twenty simulations and averaged the results.

Before discussing the results, we would like to emphasize that these benchmarks, by themselves, cannot tell us which is the fastest engine, but only which engine has the potential of being the fastest. As it is shown Figure 5, the raw event processing speed of Dessert is impressive, especially when the number of processes is (relatively) low. Anyway, as the graph clearly shows, in this benchmark Dessert *always* outperforms SimPy, even when it is run by PyPy. We also note that the graph shows the results of PyPy only to $13,000$ processes because, given its huge memory consumption, PyPy crashes after a period of uninterrupted swapping activity (that is, thrashing). This fact can be clearly seen in Figure 7, where PyPy memory consumption goes off the charts even with a small number of processes. The same figure shows that Dessert and CPython follow, more or less, the same curve, demonstrating that Dessert potentially allows users to run simulations faster without incurring on higher memory consumption. Moreover, on higher loads Dessert is faster (Figure 5) *and* consumes less memory (Figure 7).

C. P2P simulation

While the benchmarks previously discussed are useful to understand how fast the simulation engines could perform, they could not answer to a crucial question: what is the fastest engine on common, "real-world", simulations? To answer such a question, we have simulated the execution of a peer to peer protocol based on linear network encoding [19]. Since the protocol was created solely for teaching purposes, we will describe it here very briefly.

Suppose we have n machines, each one running both a client and a server process, and set $k = \lfloor \frac{n}{2} \rfloor$. At first, each file that must be shared is first split into k parts, then other $k-1$ parts are created by linear combinations of the first k parts. Thus, every file is encoded in $n-1$ parts, and each part is stored on a different machine. The rest of the simulation consists in seeing whether the clients, that try to retrieve (parts of) the files from the servers, saturate the whole network, since all communications are routed through a single switch. In particular, each client needs to request at least k parts to recover a file, but it could do more requests to reduce wait times. Therefore, one of the goals of the simulation is to understand how many extra requests give the lowest wait times. For each combination of machine count n and extra-request count r, we run twenty simulations, so that results are pretty accurate and reliable. Therefore, since r lies in the interval $[0, k-1]$, for each n we execute $20 \cdot k = 10 \cdot n$ simulations.

As it is shown in Figure 8, Dessert can execute these simulations faster than SimPy, especially when the number of machines gets higher. Under these particular settings, Dessert is *five* times faster than SimPy run on CPython, and twice as fast than SimPy when run on PyPy, which we deem as a good result. Results on memory consumption, shown in Figure 9, confirm PyPy memory problems and the fact that Dessert and SimPy, when run on CPython, have nearly the same footprint, although in this case the one of Dessert is slightly higher.

V. Related work

On the one hand, many libraries enable to write discrete-event simulations, using a variety of programming languages and environments. Indeed, as we have already said, our work has been greatly inspired by SimPy [1][2][20], which is written in, and usable from, Python. On the other hand, the .NET framework has been somewhat neglected by DES library authors, so there are very few free options (some commercial options are: Micro Saint® Sharp [21] and Sage®, the successor of HighMAST™ [22]) to choose from.

In particular, as far as we know, our Dessert is the *first* open-source (complete) project, on the .NET framework, for writing discrete event simulations following the *process oriented paradigm*. In this paradigm, simulations consist of interacting *processes*, that is sequences of events and activities.

 ISBN: 978-1-61208-344-5

This approach allows users to write simple and readable simulation code; the relationships between various paradigms and, especially, the challenges associated with modeling problems with different aspects best represented by different paradigms is a topic of on-going research [23].

Focusing on the .NET framework, the only free options that we have found implement a different paradigm or are incomplete and, apparently, abandoned. SharpSim [24] is an open-source library, written in C#, that implements the *event-oriented paradigm*. In this paradigm, users model the systems in terms of events. Implementations of this paradigm can be very efficient, but simulation code following this style is less modular, and harder to write and understand [25]. React.NET [26] is another open-source library written in C#, which shares our paradigm and general goals. Unfortunately, the project seems dead, since there are no stable releases and it has not been updated since 2006. Finally, DotNetSim [27] is described as a prototype that exploits .NET for writing fully object-orientated components that cross programming languages, packages and platforms and link them in a single application. However, this seems to be another dead project, since we could not find any prototype to download and evaluate.

VI. Conclusion and Future Work

We have presented Dessert, a fully managed .NET engine for process-based discrete-event simulation. On the one hand, Dessert has been heavily inspired, and tries to follow, the simplicity and leanness of SimPy, in the strongly typed .NET world. On the other hand, Dessert is not, and was not supposed to be, a *straight* "clone" of SimPy: we kept what we liked, which is *a lot*, but we have also tried to improve the usability even more, by making some changes and additions. Moreover, by leveraging the .NET framework, and adhering to Common Language Specification, Dessert allows user to write simulations in a variety of different programming languages. For these reasons, Dessert yields better performances both at *development* and *execution* time, since static typing is beneficial for both catching many problems early on, and permitting the use of refactoring and context-aware code completion tools, like Visual Studio IntelliSense. Indeed, the speed-up offered by Dessert is impressive, especially when SimPy is interpreted through CPython, the "default" Python interpreter, and has nearly the same memory footprint. We also benchmarked SimPy on PyPy, a recent and highly optimized alternative Python implementation, which closes the gap in running times, but at the cost of a huge memory consumption. For this reason, SimPy on PyPy crashes on "big" simulations that our Dessert handles effortlessly.

Since Dessert is completely open-source, its future developments are somewhat unpredictable. We plan to develop a library of higher level abstractions, like network components and elements of stocking chains, to ease the development of complex simulations. Moreover, we would like to address the loss of performance when running on Mono. Another direction for further work is the development of a proper domain-specific language, to write simulations even more easily, which could be compiled and run on our engine.

References

[1] "SimPy," 2014, URL: https://pypi.python.org/pypi/simpy [accessed: 2014-03-08].

[2] K. Müller, "Advanced systems simulation capabilities in SimPy," 2004, europython 2004, URL: http://simpy.sourceforge.net/old/images/Advanced_Systems_Simulation_Capabilities_in%20SimPy_Fallback_Last.pdf [accessed: 2014-03-09].

[3] N. Schemenauer, T. Peters, and M. L. Hetland, "Simple generators," 2001, python Enhancement Proposal 255 URL: http://www.python.org/dev/peps/pep-0255/ [accessed: 2014-03-08].

[4] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 31, no. 2, Feb. 2009, pp. 6:1–6:31.

[5] "Dessert on NuGet," 2014, URL: https://www.nuget.org/packages/Dessert/ [accessed: 2014-03-08].

[6] "Dessert on GitHub," 2014, URL: https://github.com/pomma89/Dessert [accessed: 2014-03-08].

[7] J. Hamilton, "Language integration in the common language runtime," ACM Sigplan Notices, vol. 38, no. 2, 2003, pp. 19–28.

[8] "What is new in Python 3.3," 2014, URL: http://docs.python.org/3.3/whatsnew/3.3.html [accessed: 2014-03-08].

[9] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 7, no. 2, 1997, pp. 157–209.

[10] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," SIAM Journal on Computing, vol. 15, no. 1, 1986, pp. 52–69.

[11] S. Troschuetz, ".NET random number generators and distributions," 2014, URL: http://www.codeproject.com/articles/15102/net-random-number-generators-and-distributions [accessed: 2014-03-08].

[12] J. O. Coplien, "Curiously recurring template patterns," C++ Report, vol. 7, no. 2, 1995, pp. 24–27.

[13] D. Gregor and J. Järvi, "Variadic templates for C++0x." Journal of Object Technology, vol. 7, no. 2, 2008, pp. 31–51.

[14] "Code generation and t4 text templates," 2014, URL: http://msdn.microsoft.com/en-us/library/bb126445.aspx [accessed: 2014-03-08].

[15] "VirtualBox," 2014, URL: https://www.virtualbox.org/ [accessed: 2014-03-08].

[16] "Lubuntu," 2014, URL: http://lubuntu.net/ [accessed: 2014-03-08].

[17] "PyPy," 2014, URL: http://pypy.org/ [accessed: 2014-03-08].

[18] "psutil," 2014, URL: https://code.google.com/p/psutil/ [accessed: 2014-03-08].

[19] S.-Y. Li, R. W. Yeung, and N. Cai, "Linear network coding," Information Theory, IEEE Transactions on, vol. 49, no. 2, 2003, pp. 371–381.

[20] K. Müller and T. Vignaux, "SimPy: Simulating systems in Python," 2003, ONLamp.com Python DevCenter, URL: http://www.onlamp.com/pub/a/python/2003/02/27/simpy.html [accessed: 2014-03-08].

[21] W. K. Bloechle and D. Schunk, "Micro saint® sharp simulation software," in Proceedings of the 35th conference on Winter simulation: driving innovation. Winter Simulation Conference, 2003, pp. 182–187.

[22] P. C. Bosch, "Simulations on .NET using HighPoint's highmast™ simulation toolkit," in Simulation Conference, 2003. Proceedings of the 2003 Winter, vol. 2. IEEE, 2003, pp. 1852–1859.

[23] S. K. Heath, A. Buss, S. C. Brailsford, and C. M. Macal, "Cross-paradigm simulation modeling: challenges and successes," in Proceedings of the Winter Simulation Conference, 2011, pp. 2788–2802.

[24] "SharpSim," 2014, URL: http://sharpsim.codeplex.com/ [accessed: 2014-03-08].

[25] N. Matloff, "Introduction to Discrete-Event Simulation and the SimPy language," 2008, URL: http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf [accessed: 2014-03-08].

[26] "React.NET," 2014, URL: http://reactnet.sourceforge.net/ [accessed: 2014-03-08].

[27] M. Pidd and A. Carvalho, "Simulation software: not the same yesterday, today or forever," Journal of Simulation, vol. 1, no. 1, 2006, pp. 7–20.

 ISBN: 978-1-61208-344-5

Tests as Documentation: a First Attempt at Quality Evaluation

Maura Cerioli and Giovanni Lagorio

DIBRIS - University of Genova
Genova, Italy
Email: {maura.cerioli,giovanni.lagorio}@unige.it

***Abstract*—We present a novel method, and its associated supporting tool, for automatically singling out *sloppy tests*; that is, tests that run successfully on (some) *incorrect* implementations, that violate the property they are expected to verify. Our freely available tool is written in C#, but the technique is language agnostic and can be easily applied to other languages.**

Keywords–Testing; Debugging.

I. INTRODUCTION

Test methods, for instance those written using a framework of the *x*Unit family [1], simply called *tests* from now on, have initially been introduced in software development process for *unit testing*, that is, testing of small units of code during their development, more than thirty years ago [2].

More recently, tests have been also used to capture information about the code to be developed, playing in some sense the role of *running specifications*. This is the case, for instance, of the test-driven approach [3], where tests are developed along with the code, and used to improve the developer understanding of the required code, making it explicit. Though tests in the test-driven approach are aimed more at knowledge capture than code improvement, they are still *white box*, written by the software developers taking advantage of private code structures for the set up.

A more innovative use of tests is in refactoring and/or migration of legacy systems [4], or in iterative development processes. Indeed, in those cases, tests are defined and verified against a version of the system, be it the system to be refactored or the current iteration prototype, but are intended to be run on the *next versions*, still to be developed at the moment such tests are written. In this way, tests capture observable behaviors of the system, approved by the stakeholders on the current version, and guarantee the next version to preserve such behaviors, complementing (or altogether replacing) the corresponding documentation.

To put the system in the required state, before the call of the method to be tested, this approach requires the tests not to rely on the internal structure of the system, that is going to change. Instead, each operation has to go through the *interface* of the system [5][6][7], in a *black box* style. Moreover, the design of the overall test suite cannot be driven by the current implementation, as it is going to change, and all the adequacy criteria based on code coverage are unreliable.

Analogous problems arise for tests distributed along the specification of components/services, as convincing evidence of their correctness [8]. Indeed, when the tests are used to capture knowledge about the functionalities of a system, they cannot rely on its internal structure when preparing the initial state for the *call under test*. Otherwise, they would risk undue disclosures of the system implementation to users, who have full access to the tests, and the approach would be brittle against changes to the implementation.

Moving from white box tests used to improve the system implementation, to black box tests used to document the system, requires to change the definition of test quality, as well as the techniques to evaluate it. Indeed, when tests are aimed at improving the technical quality of the system under test, it may suffice that the overall test suite is capturing enough bugs. Thus, in literature, we find plenty of techniques to assess the quality of a test suite, by measuring how extensively the test suite, as a whole, exercize the system. The exact meaning of *extensively* may vary, giving rise to different quality criteria. For instance, statement/branch/multiple condition coverage [9], or mutation testing [10].

However, in our target cases, tests are used as *living documentation* [11]. Thus, the description of each *individual* test must correspond to its implementation, because stakeholders and maintenance staff will rely on those test descriptions to understand the system behavior. Therefore, in this setting, we need to assess the quality of *each individual test*, as opposite to the quality of the overall test suite. Moreover, the meaning of quality is also different w.r.t. standard approaches, because a test has a *high quality* when it strictly conforms to its description, disregarding both its capability to spot bugs, and the portion of system it exercises.

Therefore, standard evaluation techniques are not appropriate, and we propose here a different approach.

The first step to get high-quality tests is to verify their *correctness*, that is, that they run successfully on a correct implementation of the system. In other words, tests, as any other software, need to be tested. Such a necessity is partially reduced by their intrinsically limited complexity. However, even when writing small tests, it is rather easy to introduce mistakes or to misinterpret their goals. There are basically two approaches to test verification: inspection by a human reader, as peer review can improve the quality of tests [12], and the execution on a reference implementation, known to be correct. The former method permits, in a single pass, to verify the correctness of the tests and evaluate their quality, in terms of correspondence to their definition. However, it is extremely time consuming [13], hence expensive. Moreover, as tests are many and often quite similar, the attention level of the human inspector and, accordingly, the number of detected

 ISBN: 978-1-61208-344-5

imperfections/mistakes may decrease. Finally, such costs have to be sustained whenever tests are updated.

The automatic verification by a reference implementation, on the other hand, is widely used in practice whenever a reference system is available during the implementation of the tests, as in the cases we are addressing. In such settings, the reference system is assumed to be a correct *oracle* so, if a particular test fails, then it is known *a priori* to be incorrect. Once tests appear to be *correct*, i.e., they successfully run on the oracle, their *quality* has to be evaluated.

To carry on this task in automatic verification style, we need slight variations of the system. Each of these variations, that we call *anti-oracles*, intentionally violates the description of a specific test t, and hence it is a prospective victim of the t, that should be able to kill it. Then, from the outcome of t on that anti-oracle, we can get precise information about the adequacy of t w..r.t. its description, that is, on its *quality*.

To keep the design effort of such anti-oracles sustainable, we propose a method to instrument the oracle. The instrumented oracle, I, can *behave* both as a correct implementation *and* as the needed anti-oracles, in different runs.

Our method is supported by a lightweight tool, which takes care of differentiating the runs on I. Note that, in our target scenarios, tests, in their setup code, must (only) use the very same elements of the public interfaces under test. Thus, our anti-oracles, to be effective, should *behave correctly on all calls, except for the call under test*. This fact imposes further requirements on the design of the supporting tool.

Our approach is reminiscent of the mutation testing technique, in that our anti-oracles are variation of the oracle. But, while mutants are randomly generated and used to estimate the probability of the whole test suite to detect technical bugs, each anti-oracle is *designed* to target an individual test and verify the adequacy to its description. Thus, the mutation testing technique cannot address the problem we are interested into. We introduce our method in Section II, and we briefly sketch its implementation in Section III. A preliminary evaluation of our method is provided in Section IV, and in Section V we discuss its relations to mutation testing, while some conclusions are drawn in Section VI.

II. Proposed Method

We consider basic standard test methods consisting of three parts: the *setup*, usually few lines of code to initialize the status of the system, the *call under test*, that is the specific method invocation whose behaviour is verified by the test, and, finally, an *assertion* stating properties about the result of the call, in terms of both the yielded value, if any, and the resulting state of the system.

Our method assumes the existence of a working system, to be used as the oracle, and of the specifications of the tests to be implemented. Such specifications should be as accurate as possible, but cannot be expected to be formally expressed in a rigorous specification language, like, for instance, some kind of logic. Indeed, in most realistic cases it is not possible, or highly inconvenient, to formalize the properties to be checked.

This limitation rules out the possibility of automatically generating tests from their formal specification (such tests would be obviously consistent with their specification *by construction*, provided the generator to be correct).

Let us clarify the expected level of formality of test specifications on a toy example, written in C# and using NUnit [14] (however, the idea is independent from both). The class `IntStack`, shown in Figure 1, implements a very basic stack of integers as a tiny wrapper on the standard generic class `Stack<>`.

```
public class IntStack {
  private Stack<int> _stack = new Stack<int>();
  public int Size(){
    return this._stack.Count;
  }
  public void Push(int i){
    this._stack.Push(i);
  }
  public void Pop(){
    this._stack.Pop();
  }
  /* ... */ }
```

Figure 1. A stack of integers.

Using this prototype, we want to polish a set of tests, for instance, targeting the method `Push`:

- `PushDoesNotAffectPreviousElements`
- `AfterPushSizeIsPositive`
- `PushIncreasesSizeFrom3To4`
- `PushAddsElement`

`PushDoesNotAffectPreviousElements` could be specified by: "*after pushing a number on a stack, already containing some items, they will be still on the stack and in the same order*".

The goal is verifying the individual tests to be adequate w.r.t. their specification. The oracle (i.e., the reference implementation) is used first to verify that all the tests appear to be *correct*, that is, successfully running on the oracle. The next step of our method is to verify that they are also *sufficiently strict*. To this end, we first derive, from each test specification, a list of possible mistakes, which the test should be able to detect accordingly to its specification. For instance, for `PushDoesNotAffectPreviousElements`, the list of possible mistakes includes: one of the original items is dropped; one of the original items is replaced by another number; two of the original items are swapped.

Then, each mistake from such a list is implemented by an *anti-oracle*, which should replace the correct oracle implementation to answer the *call under test*, and only *that* particular call, making the test fail (if it is sufficiently strict).

For instance, `AfterPushSizeIsPositive` should be able to capture anti-oracles where the size is zero after an element has been pushed. So, the body of `Push`, in such a anti-oracle, could simply empty `this._stack`.

Finally, we instrument the oracle to derive an enriched system able to make the call under test fail for each test. At this aim we define a utility class `FindCaller` that, depending on a global switch (for instance, the value of an environment variable), may operate in two modes: *record* and *evaluate*.

The basic idea is that in *record mode* the enriched system behaves like the oracle, so all tests must be successful, while our utility class logs some information about the execution, which are needed for subsequent runs in *evaluate mode*. In such a mode, instead, for each test t, the anti-oracle(s) designed for t is used to answer the call under test by t, while all other calls are answered by the (correct) oracle methods. Thus, in

```
public void Push(int i) {
  string caller = FindCaller.GetTestName();
  if (caller == "AfterPushSizeIsPositive" ||
       caller == "PushIncreasesSizeFrom3To4") {
    this._stack.Clear();
    return;
  }
  if (caller ==
      "PushDoesNotAffectPreviousElements") {
    int previous = this._stack.Pop();
    this._stack.Push(previous+123);
    // continue as if nothing has happened
  }
  if (caller == "PushAddsElement") {
    this.Push(i+1);
      // the *direct* caller for this recursive
      // call is not PushAddsElement, so we get
      // normal behavior for this call
    return;
  }
// everything as before...
// (that is, the behaviour of the oracle)
```

Figure 2. Instrumented `Push` method.

```
[Test]
public void PushIncreasesSizeFrom3To4() {
  IntStack s = new IntStack();
  for (int i=0; i<=2; i++)
    s.Push(i);
  int sizeBeforePush = s.Size();
  s.Push(3);
  Assert.That(s.Size(),
              Is.Not.EqualTo(sizeBeforePush));
}
```

Figure 3. An example of a sloppy test.

this mode, all tests should fail.

The class `FindCaller` offers the method `GetTestName`, whose behavior changes dramatically depending on the operating mode:

- in *record* mode, `GetTestName` always returns **`null`**, but also logs some method call information, to be later used in *evaluate mode*;
- in *evaluate* mode, `GetTestName` returns the name of a test-method t if it is invoked by the call under test by t; otherwise it yields **`null`**.

Thus, to decide if the standard implementation of a method, or its anti-oracle, failing on a test-method t, has to be used, we can simply check if `GetTestName()` returns the string t. Hence, we can inject the failing anti-oracle into our oracle and have a single software product P to maintain (we will address how to keep the actual implementation and its anti-oracles separate in Section VI). This product P behaves, in *record* mode, as the original oracle, while in *evaluate* mode behaves as the needed sophisticated anti-oracles previously discussed.

Figure 2 shows how to instrument the method `Push` in P.

In *evaluate* mode, the instrumented code behaves as the provided failing anti-oracle for each test-method, on its call under test. Hence, all tests should fail. Yet, if we ran the test shown in Figure 3, we would discover that it still passes.

This points out an inadequacy of the test (i.e., the test is sloppy): indeed, instead of asserting that the `Size()` of the stack, after the `Push(3)`, is *different* than before, it should assert that the size *has increased by one*.

Notice that, in the code of anti-oracles, all the internal structure of the oracle can be used, including calls to the public methods, because such calls will be automatically answered by the oracle code. Thus, each anti-oracle is usually implemented by very few lines of code, in most cases just a single one.

Every time a verification fails, that is, tests pass in *evaluate mode*, a refinement step is needed. However, our method does not prescribe how to perform it. Thus, it can be plugged on different processes for improving test quality.

III. Implementation of the Utility Class

As described in the previous section, our utility class `FindCaller` offers the static method `GetTestName` that allows any implementation method m to know the name of the running test t, when the current call of m is the call under test of t (otherwise, `GetTestName` simply returns **`null`**).

The simplest way to detect if the current call of m is the call under test, for some test t, would be to require call under test to be annotated in some way (e.g., by some attribute, and use such information via reflection). However, we want to be able to *evaluate existing tests as they are*, without having to tamper with them, so the implementation of `GetTestName` is more challenging.

Here, we just sketch the idea, since our C# implementation takes into consideration some technical details that are not particularly relevant. We refer interested readers to the freely available source code [15].

In order to understand if a call to a method m, of the implementation, may be the call under test for some test t, `GetTestName` simply rules out the cases that cannot be a call under test, which are:

1) m has been called by another implementation method m' (possibly coinciding with m, in case of recursion), instead than directly by some test;
2) there is a (temporally) subsequent direct call to m, by the same test t, hence the current call is just part of the setup.

Thus, `GetTestName` returns t for the last call of *any* implementation method m directly called by t (or one of its auxiliary methods), even if m is not the one tested by t, say m_t. However, this is not a problem, as the condition `FindCaller.GetTestName()=="t"` is only checked inside the instrumented version of m_t, so that returning t, instead of **`null`**, to the call of some other method goes undetected, and is immaterial.

Therefore, the tasks of method `GetTestName` are to identify:

1) the name of the currently running test, say t;
2) if m has been called by another method of the implementation;
3) if this call to m by t is its (temporally) last call to m.

Since almost every programming language keeps the information about the (direct and indirect) callers of a method in the (machine) call-stack, we can address tasks 1 and 2, by performing a stack walk. The remaining task, 3, needs another technique, which is discussed below.

In languages offering reflection/introspection features, like C# and Java, the call-stack is readily available. For instance, in

 ISBN: 978-1-61208-344-5

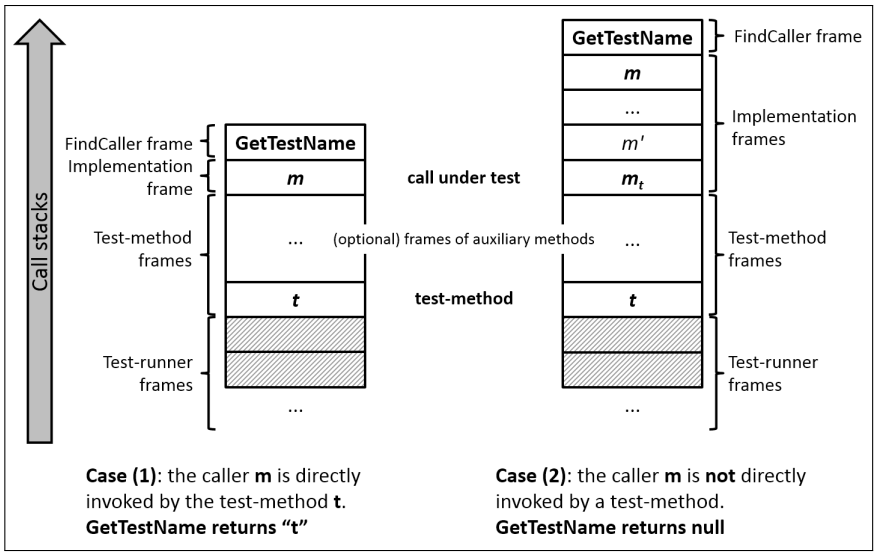

Figure 4. Two examples of call stacks.

C#, using the standard class `StackTrace` we can easily obtain an array of `StackFrame` objects.

Figure 4 shows two examples of the kind of call stacks that `GetTestName` has to deal with. The topmost frame is always for `GetTestName`. The rest of the stack consists of three groups of frames (from the bottom): those of the test runner, those of the test methods, and, finally, those of the implementation methods. The topmost of this latest group, that is, the second frame from the top, belongs to the method m, that has to decide whether to behave as the oracle or some anti-oracle.

The lowest frame of the test method group is the one for t (task 1) and is identified by our utility class by checking whether the method is annotated by one of the custom attributes of NUnit [14]. Of course, this can be easily generalized to other testing-framework.

Task 2 corresponds to checking whether, between the frame of m and the one of t, there are some other frames belonging to implementation methods. Figure 4 shows, side by side, two possibilities:

- left: m is directly called by t (or some of its auxiliary methods), so `GetTestName` must return t (unless Task 3 detects a subsequent call to m);
- right: t invokes m_t, which could for instance be the method under test, and then m_t calls some other implementation methods $(m', \ldots, m)$. Each of these methods, being instrumented, will call `GetTestName`, that must recognize that the current execution of its direct caller (m, in the figure) does not correspond to the call under test. Thus, even if its direct caller coincided with m_t, that is, $m = m_t$, `GetTestName` should return **`null`**.

These two cases are also exemplified in the sequence diagram shown in Figure 5, where the test runner invokes the test `PushAddsElement`, which, in turn, invokes `Push`, which invokes `GetTestName`. In this case, `GetTestName` returns the string `"PushAddsElement"`, so the instrumented `Push` method misbehaves by pushing `(i+1)`, instead of `i` (see Figure 2), by recursively invoking itself. In this second activation, `GetTestNames` returns **`null`**, so no anti-oracle is activated and the call is answered by running the oracle code. Notice that the call of `Top` invokes `GetTestName` too, and gets the string `"PushAddsElement"` as result. But, having no instrumentation for that test, `Top` behaves as the oracle.

Task 3, that is, detecting if this call to m by t is its (temporally) last call to m, can be tackled by exploiting the following idea: since we want any failing implementation (for t) to differ from the oracle only on the call under test c, the execution flow of t, until it reaches c, has to be exactly the same on both the oracle and the anti-oracle.

Because all tests pass on the oracle, a single run of all tests in *record mode* allows our class `FindCaller` to collect the information about the order of all the calls that any test makes to any implementation method. After these information have been collected and persisted, they can be used in *evaluate mode* to discern, for each test t, which call is, indeed, the temporally last one.

IV. Preliminary evaluation

The proposed method has been experimented in the context of the project evaluation of an undergraduate course on component based development. Such a project consisted of two independent phases: the development of tests against the specification of a toy component for the management of an auction site, and the implementation of the component itself. The component specification consisted of five small interfaces for about 20 methods and 12 properties, and a few exceptions. The semantics of each method/property was expressed by few lines of text in natural language, as in our running example, and the reference implementation was about 600 lines of code.

The first phase was a collaborative activity by 16 groups of 5 persons each, with the goal of redundantly implementing 150 test specifications. Each test specification, given in natural language, fixed the method to be tested, the call parameters (if any), the required setup of the system, and the expected result.

Each group member was required to individually develop 10 tests and inspect those written by the other group members. Students were equally penalized by errors made during the

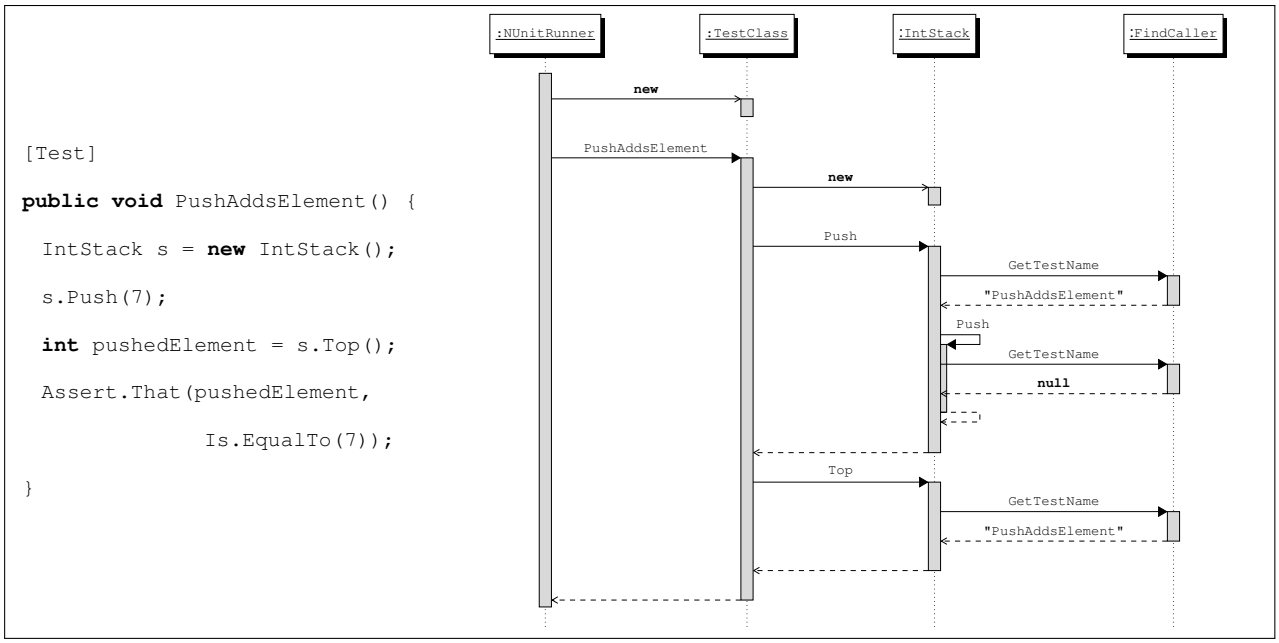

Figure 5. `PushAddsElement` and its sequence diagram.

development and peer review. Therefore, they were motivated to carefully read the tests by other members of their team. Indeed, from the discussions going on in a forum for intra-group communications, we know that most students took the assignment seriously and devoted energy and time to get it done at the best of their abilities.

At the end of the review phase, we evaluated the tests using our reference implementation, and 428 out of 590 passed (note that not all the enrolled students completed the project; so only 590 out of the expected 800 tests were submitted for evaluation). Then, we applied our method to those apparently correct, in order to detect sloppy tests: 13 tests out of 428 did not fail as they should have been. That is, about the 3% of the peer-reviewed tests were still slack. Notwithstanding the apparently low value, it is worth noting that:

- 50% of the groups delivered at least one sloppy test, and a group even produced 5 sloppy tests out of 30, as it can be seen in Table I (only groups with at least one sloppy tests have been inserted);
- the tests had been already manually inspected by other members of the group to improve their quality [12];
- for each test specification we implemented just the most obvious failure, hence, capturing only a part of the sloppy tests;
- the given test specifications have been kept very simple to simplify the students' work. With more complex test specifications a higher number of sloppy tests should be expected.

Group	# test methods	Failed	Correct	Sloppy
16	40	10	25	5 (16,67%)
1	30	7	21	2 (8,70%)
4	40	9	30	1 (3,23%)
5	30	6	23	1 (4,17%)
6	40	7	32	1 (3,03%)
7	40	2	37	1 (2,63%)
8	40	8	31	1 (3,13%)
13	30	7	22	1 (4,35%)

TABLE I. EVALUATION RESULTS.

The sloppy tests detected by our experiment can be roughly categorized into three classes:

- Verifying a property weaker than the one expressed by their informal specification. For instance, though required to verify that the result R of some operation is $S = \{a, b, c\}$, they just check that R has three elements, or that $R \subseteq S$, or viceversa. This is by far the most common error, and corresponds to the intuition of sloppy test.
- Verifying the thrown exception to be the one required, but without discriminating if it has been thrown by the method under test, or by some previous call during the test setup. This sloppiness may easily go undetected when system exceptions, like, for instance, `InvalidOperationException` or `ArgumentNullException`, are expected, since they may be thrown in many different situations.
- Making blatantly stupid mistakes, like, for instance, invoking a different method in place of the one to be tested. It may sound unlikely that such evident mistakes are overlooked by reviewers. But, it does happen since their attention is often focused on checking small details, or the logical flow of the test to make sense *per se*, forgetting to check it against its specification.

A threat to the validity of this experiment might be that the subjects were students instead of professionals. Thus, the evaluation could be biased by their limited skills. We plan to apply our technique to the tests of some open-source project in order to estimate its usefulness in a real world context.

V. Related work

To the best of our knowledge, our method is the first one proposed in literature for revealing sloppy test cases. While our work uses an idea similar to *mutation testing* [16][17], there are substantial differences.

Mutation testing is a technique for evaluating the ability of a test suite in detecting faults, and can also be used as a tool to add new test cases to obtain higher coverage scores. The technique consists of two steps: the creation of mutants and their execution. First, mutants, i.e., clones of the original program with the exception of one random atomic change, are created. For example, a mutant could be produced by changing a binary operator (e.g., "+") into another (e.g., "*") to create a faulty version of the original program. The "rule" that changes an operator with another is called *mutation operator*. Then, the target test suite is executed against all the produced mutants. A mutant is said *killed* if at least a test case belonging to the suite is able to reveal the performed mutation. The test suite adequacy is computed by dividing the number of killed mutants by the total number of mutants.

Although mutation testing is largely recognized as a satisfactory technique for the improvement of a test suite, the aim is revealing parts of system code not exercised by any test, where randomly mutations go undetected. This allows to improve the test suite as a whole, by adding test cases targeting the unexplored parts. But, mutation testing is inadequate for our goal, that is, individual test adequacy against its specification, in a setting where testing must go through the public interfaces of the system. Indeed, mutation testing

- evaluates and improves test suites as a whole instead of individual tests;
- addresses a different concept of quality, without any connection to the users' expectations about the kind of bugs that should be detected by the tests, accordingly to their description;
- could yield a false positive, if mutants are killed by a failing test setup involving the very same methods to be tested; this cannot happen with white-box testing, where the setup explicitly accesses the internal structure of the system, but it is quite common when also tests must go through the public interfaces.

VI. Conclusion and Future Work

We have proposed a method to verify the adequacy of individual tests to their specification. Our method requires to elaborate minimal changes to a reference implementation, making a well written test fail on the resulting anti-oracle. Moreover, our method is supported by a tool, that takes care of having such changes executed only on that test.

Currently, the changes are manually injected into the oracle, except for those tests expecting an exception, where the tool can automatically throws an exception of unexpected type to verify that the test is correctly strict. We plan to use aspect-oriented programming [18] techniques in order to keep separate the code to be injected from the reference implementation. Indeed, we are currently evaluating PostSharp Express [19] for .NET, as a supporting tool.

A further enhancement is allowing several different anti-oracles for the same test t, implementing different bugs t should be able to detect. At this aim, the test-runner should be made aware that some tests need to be run several times, and method `GetTestName` should yield, on the call under test, not only the test name, but also the number of its run, in order to possibly change the behaviour.

The current version of the tool has been preliminarily evaluated by an experiment on the projects of a course. The results were quite encouraging, as we captured 3% of sloppy tests on a population already improved by a preliminary peer-review process. However, they were also obviously limited, being based on the performance of students instead of professionals. Further applications to some industrial sized project are needed in order to estimate its real usefulness.

Acknowledgment

We warmly thank Filippo Ricca for the helpful discussions on mutation testing.

References

[1] P. Hamill, Unit Test Frameworks: Tools for High-Quality Software Development. O'Reilly, 2004.

[2] G. Myers, The Art of Software Testing, ser. A Wiley-Interscience publication. Wiley, 1979.

[3] K. Beck, Test-Driven Development by Example, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[4] A. Marchetto and F. Ricca, "From objects to services: toward a stepwise migration approach for Java applications," International Journal Software Tools Technological Transfer, vol. 11, no. 6, 2009, pp. 427–440.

[5] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-Based automatic test case generation for web services testing," in SOSE '05: Proceedings of the IEEE International Workshop. Washington, DC, USA: IEEE Computer Society, 2005, pp. 215–220.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Systematic generation of XML instances to test complex software applications," in RISE, 2006, pp. 114–129.

[7] H. M. Sneed and S. Huang, "The design and use of WSDL-Test: a tool for testing web services: Special issue articles," J. Softw. Maint. Evol., vol. 19, no. 5, 2007, pp. 297–314.

[8] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in FASE, 2005, pp. 34–48.

[9] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," Communications of the ACM, vol. 6, no. 2, Feb. 1963, pp. 58–63.

[10] W. E. Wong, Mutation Testing for the New Century. Springer, 2001.

[11] G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications, 2011.

[12] F. Lanubile and T. Mallardo, "Inspecting automated test code: A preliminary study," in Proc. of 8th International Conference on Agile Software Development (XP 2007). Springer-Verlag, 2007.

[13] T. Thelin, H. Petersson, P. Runeson, and C. Wohlin, "Applying sampling to improve software inspections," Journal of Systems and Software, vol. 73, no. 2, October 2004, pp. 257–269.

[14] "NUnit," 2014, URL: http://www.nunit.org [accessed: 2014-03-09].

[15] "FindCaller," 2014, URL: http://www.disi.unige.it/person/LagorioG/FindCaller.cs [accessed: 2014-03-09].

[16] R. G. Hamlet, "Testing programs with the aid of a compiler," IEEE Transactions on Software Engineering, vol. 3, no. 4, Jul. 1977, pp. 279–290.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, vol. 11, no. 4, Apr. 1978, pp. 34–41.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-m. Loingtier, and J. Irwin, "Aspect-oriented programming," in ECOOP. Springer-Verlag, 1997.

[19] "PostSharp Express," 2014, URL: http://www.postsharp.net/ [accessed: 2014-03-09].

 ISBN: 978-1-61208-344-5

Hardware Realization of Embedded Control Algorithm on FPGA

Róbert Krasňanský, Branislav Dvorščák and Štefan Kozák
Institute of Automotive Mechatronics, Faculty of Electrical Engineering and IT,
Slovak University of Technology in Bratislava,
Bratislava, Slovakia
{robert.krasnansky, branislav.dvorscak, stefan.kozak}@stuba.sk

***Abstract*—This paper explores an efficient algorithm for design and implementation of Proportional-Integral-Derivative (PID) controller on the Field Programmable Gate Array (FPGA) technology. To create a synthesizable control algorithm, the Very High Speed Integrated Circuits Hardware Development Language (VHDL) was used as a programming tool. The paper points to the possibilities of parallel computation with the aim of speeding up the control implementation. The practical application of proposed control algorithm is illustrated by a test performed on a real laboratory Direct Current (DC) motor system. The results confirm the legitimacy of using the FPGA methodology for design of control algorithms, since it improves speed, accuracy and compactness. In addition, it is cost effective and has a low power consumption, which are desirable attributes in embedded control applications.**

Keywords-FPGA; PID controller; Spartan 6; DC motor; VHDL language

I. INTRODUCTION

Motivated by the practical success of conventional control methods applied in industrial process control, there has been an increasing amount of work on development of effective hardware realizations of these control algorithms. Despite the numerous control design methods that have been proposed in the literature, it is estimated that PID controllers are still employed in more than 92% of the industrial processes today and many control systems using PID control have proved its satisfactory performance [1].

Recently, it has been shown that FPGAs can pose an alternative solution for the realization of digital control systems, previously dominated by the microprocessor systems [2]. The motivation behind using FPGAs to implement a PID controller, rather than microcontrollers or digital signal processors (DSPs), is that they provide a good balance between performance and cost. On the other hand, although the microcontrollers may be cheaper, they do not provide enough processing power to effectively perform complex calculations in real-time. Digital signal processors can implement complex algorithms quickly; however, these implementations are expensive. In addition, the systems designed on FPGA are flexible and can be reprogrammed an unlimited number of times. Unlike processors, FPGA circuits use dedicated hardware for processing commands. FPGAs logical structures can be arranged to execute in a truly parallel manner unlike the inherent sequential execution in microcontrollers, so different processing operations do not have to compete for the same resources. This functionality also makes it possible for multiple control loops to run on a single FPGA device at different rates. Execution time may be this way dramatically reduced, since parallel architectures allow FPGA-based controllers to reach the level of performance of their analog counterparts without their main drawbacks as parameter drifts or lack of flexibility [7]. These features make FPGAs very interesting for rapid prototyping.

The objective of this work is to design and implement a digital PI controller algorithm on FPGA platform and verify its performance as well as assess the FPGA suitability for control application.

The paper is organized as follows. Section II presents the overview to the FPGA architecture and functionality as well as VHDL language features and applications. Section III introduces the technical background of the PID algorithm followed by an approach for designing and implementation of the control system extended with the anti-windup on FPGA technology. In Section IV, an application of the proposed design to a laboratory DC motor system is presented and the experimental results on Xilinx FPGA chip are discussed. Comparisons are made between the implementation on a real system and the simulation results. The conclusion and future work are provided in Section V.

II. BACKGROUND

A. FPGA Architecture

The Field Programmable Gate Array (FPGA) represents an integrated circuit containing a two-dimensional array of configurable logic blocks whose interconnection and functionality can be reprogrammed depending upon the requirement of the user [8]. A typical FPGA architecture depicted in the Fig. 1 consists of three major elements:

- Programmable logic blocks, which consist of Configurable Logic Blocks (CLBs) arranged in an array that provides the functional elements and implements most of the logic in an FPGA. Each logic block has two flip flop and can realize any 5-input combinational logic function.
- Programmable interconnect resources provide routing path to connect between individual CLBs and between CLBs and input-output blocks.
- Input-Output Blocks (IOBs) provide the interface between the package pins and internal signal lines and thus the interconnection of external signals and

 ISBN: 978-1-61208-344-5

internal signals in an array of CLBs. It can be programmed and configured as input, output or bidirectional port.

The CLBs, IOBs and their interconnectors are managed by a configuration program stored in a memory chip.

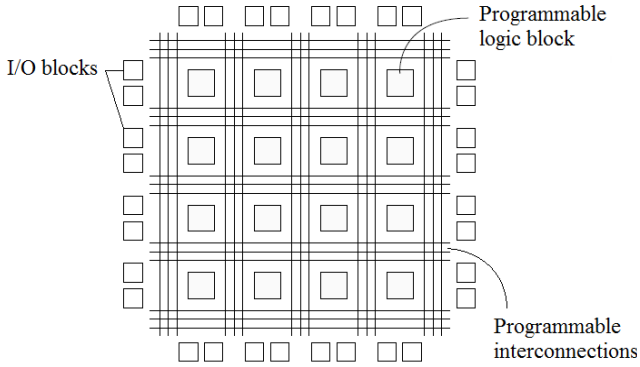

Figure 1. FPGA internal structure

A custom design can be implemented by specifying the function of each logic cell and setting the connection of each programmable switch. The CLBs structures include 2, 4 or more logic cells, also called logic elements. The structure of a logic cell, as the basic grain of the FPGA, is presented in Fig. 2. It consists of a Look-up Table (LUT), which can be configured either as a ROM, RAM or a combinatorial function.

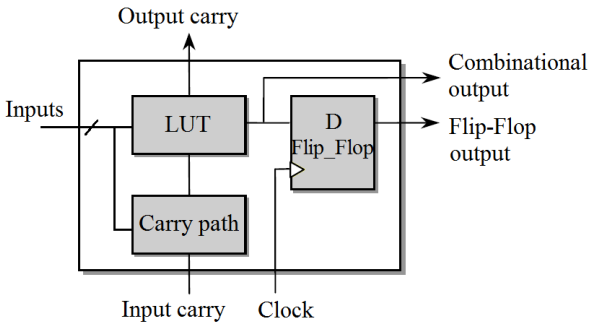

Figure 2. Logic cell [7]

Also, a carry look-ahead data path is included in order to build arithmetic operators and a D-Type Flip-Flop with all its control inputs, allowing registering the output of the logic cell.

B. *VHDL Programming Language*

FPGAs can be programmed using Very High Speed Integrated Circuits Hardware Development Language (VHDL) [1] specifically developed to describe the behavior and structure of a digital circuit and its attributes. It uses significantly different principles than C language; for instance, the commands in the code are not executed sequentially, from the top to the bottom but in parallel way. VHDL describes the connections of the logic gates together to form adders, multipliers, registers and so on. A custom design can be implemented by specifying the function of each logic cell and setting the connection of each programmable switch.

A circuit design process can be carried out as shown in the Fig. 3. Once a FPGA is programmed, the internal circuitry is connected in a way that creates a hardware implementation of the application defined in the software. The big advantage of FPGA-based algorithms design is the possibility to employ the modular approach. Since there are a lot of I/O ports, it is theoretically possible to design more algorithms on one chip without influencing one another.

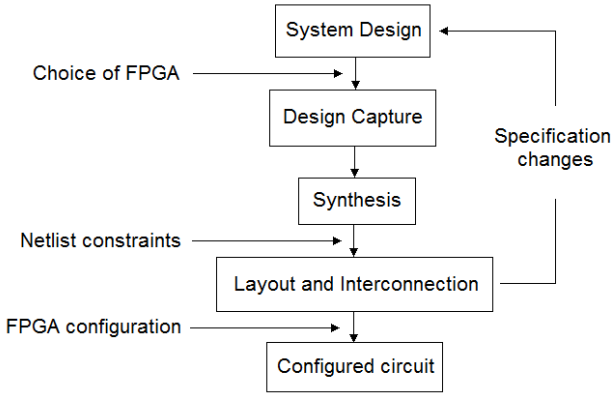

Figure 3. Design process of the circuit

The result is a user programmable piece of hardware with the reliability of dedicated hardware circuitry and the speed of modern microprocessor. Finally, FPGAs are Join Test Action Group (JTAG) compliant, thus the test data can be serially loaded into the device and the test results can be serially read out.

III. IMPLEMENTATION OF CONTROL ALGORITHM ON FPGA

A. *Digital PI Controller*

In this paper, the PID algorithm is applied for closed loop control. Among the control structures used in the industrial segment, the classic parallel PID controller depicted in Fig. 4 is one of the most widely used due to its well established practical implementation and tuning. The controller output is computed in continuous time as follows:

$$u(t) = k_p \left\{ e(t) + \frac{1}{T_i} \int_0^t e(t)dt + T_d \frac{de(t)}{dt} \right\} \qquad (1)$$

where the adjustable parameters are the proportional gain k_p, the reset time T_i and the derivative time T_d, while $u(t)$ is the control output and $e(t)$ is the error signal (setpoint response level – measured response). The compensation parameters allow an increase in the system performance in a variety of ways.

Proportional control increases gain margin and stabilizes a potentially unstable system. Integral control, on the other

 ISBN: 978-1-61208-344-5

hand, minimizes steady-state error and derivative control increases system speed by increasing system bandwidth.

For a small time sample *T*, (1) can be transformed to a difference equation by discretization using Euler integration method – rectangular integration.

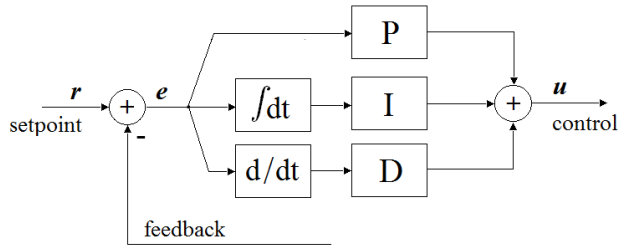

Figure 4. PID controller structure

A difference equation can be implemented by digital systems, either in hardware or software, where the derivative term is replaced by a first-order difference expression and the integral part by a sum, so the equation is given as:

$$u(n) = k_p e(n) + k_i \sum_{j=0}^{n-1} e(j) + k_d (e(n) - e(n-1)) \quad (2)$$

where n is a discrete time instant, $k_i = k_p T/T_i$ is the integral coefficient, $k_d = k_p T_d/T$ is the derivative coefficient and T is sampling time. Using this algorithm called the "position form", all past errors $e(0)$ - $e(n)$ have to be stored to compute the sum. In this paper, we prefer the "incremental form" of the PI algorithm, where the recursive equation describing this algorithm is obtained when (2) for the time instant n-1 is subtracted from the same equation for the time instant n. Thus, the expression for $u(n-1)$ is calculated in the following way:

$$u(n-1) = k_p e(n-1) + k_i \sum_{j=0}^{n-1} e(j) + k_d (e(n-1) - e(n-2)) \quad (3)$$

and the correction term as

$$\begin{aligned} \Delta u(n) &= u(n) - u(n-1) \\ &= k_0 e(n) + k_1 (e(n-1) + k_2 e(n-2)) \end{aligned} \quad (4)$$

Subsequently, for the PI controller, the current control input is in the form

$$u(n) = u(n-1) + \Delta u(n) = u(n-1) + k_0 e(n) + k_1 e(n-1) \quad (5)$$

where

$$k_0 = k_p$$

$$k_1 = -k_p + k_i$$

and $k_i = k_p T/\mathrm{Ti}$ is the integral coefficient. The big advantage of this approach is that in software implementation, (5) avoids accumulation of all past errors.

The PI incremental form (5) has to be decomposed into basic arithmetic operations:

$$e(n) = w(n) - y(n) \quad (6)$$

$$p_0 = k_0 e(n) \quad (7)$$

$$p_1 = k_1 e(n-1) \quad (8)$$

$$s_1 = p_0 + p_1 \quad (9)$$

The current control output is then calculated as

$$u(n) = s_1 + u(n-1) \quad (10)$$

B. *Parallel Design*

For the implementation of the proposed PI algorithm onto FPGA, the parallel design [3] has been used. This design is mainly composed of combinational logic, so each operation has got its own arithmetic unit – adder or multiplier. Such modified control algorithms are then feasible on FPGA circuits. The parallel design architecture of the PI incremental algorithm is depicted in Fig. 5. The design requires a total of 2 combinational logic multipliers, 3 adders and 3 registers [6]. The clock signal *clk* is used to control sampling frequency. The negation of *y* is generated using bit-wise complementing and subsequently adding 1. The difference *w* - *y* generates current error $e(n)$.

Registers are used to store the intermediate results obtained. Multipliers and adders are used for multiplication and addition of input signals according to arithmetic operations described in the previous section A. The block REG stores error values $e(n)$ and $e(n-1)$. Hence, at the rising edge of control, signal $e(n)$ of the last cycle is latched at register REG, thus becomes $e(n-1)$ of this cycle. Similarly, $u(n-1)$ are recorded at REGs by latching $u(n)$ respectively [10].

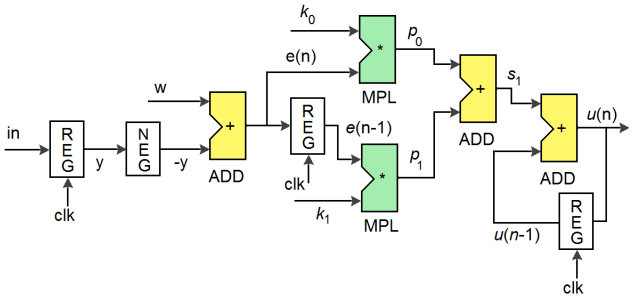

Figure 5. Parallel design of incremental PI algorithm

The values of $e(n)$ and $e(n-1)$ with their polarity indicating whether the calculated value is positive or negative are fed to PI equation (10) and the current control

output is calculated. From Fig. 5, it can be seen that the register blocks (REG) depend on clock frequency. That means that the functionality of these blocks shall be within the process, which responds to the rising edge of *clk* signal. At the same time, registers can be set to initial values of 0 after the first start or by asserting the reset signal. The process code example can be developed as depicted in Fig. 6 below.

```
Regist_process : PROCESS (clk, reset)
BEGIN
    IF  reset = '1' THEN
          en1 <= to_sfixed (0, en1);
          un1 <= to_sfixed (0, un1);
          e <= to_sfixed (0, e);
    ELSIF  clk'EVENT AND clk = '1' THEN
          en1 <= e;
          e <= to_sfixed (to_integer (unsigned (w)) –
                to_integer (unsigned (y)), e);
           un1 <= "0" & Add3 (16 downto -9);
    END IF;
END PROCESS Regist_process;
```

Figure 6. The laboratory model of DC motor

Once the signal *reset* is in logical state 1, the variables $e(n)$, $e(n\text{-}1)$ and $u(n\text{-}1)$ are reset. When the signal *reset* is in logical state 0 and the rising edge of the signal *clk* occurs at the same time, the program assigns to the variable $e(n\text{-}1)$ the value of the variable $e(n)$, similarly to the variable $u(n\text{-}1)$ the value of the variable $u(n)$ and calculates the difference of input signals $w - y$.

C. Implementation of Anti-windup Control

In the motor speed control, the maximum control output from a PI controller is determined by the converter protection, magnetic saturation and motor overheating. Hence, the saturation is applied even at the cost of introduction a non-linearity into the system. This phenomenon, called windup effect, can lead to a large overshoot, long settling time or even unstable closed-loop system.

The goal of the implementation of anti-windup in the incremental form of the PI controller is to eliminate the wind-up in the error integrator and to provide a strictly aperiodic step response even in case with large input disturbance. The implementation of anti-windup system is easy using incremental PI algorithm. The control action value is being checked and u_{out} is determined according to the following equation [4]:

$$u_{out} = \begin{cases} u_{in} & \text{if } u_{\max} > u_{in} > u_{\min} \\ u_{\max} & \text{if } u_{in} \geq u_{\max} \\ u_{\min} & \text{if } u_{in} \leq u_{\min} \end{cases} \quad (11)$$

where $u_{in}(n)$ represents the control output before saturation and $u_{out}(n)$ is the saturated control output variable.

IV. CASE STUDY

A. Laboratory DC Motor System

In this section, the proposed algorithm is applied to control a real laboratory DC motor system (Fig. 7) to demonstrate its high performance and efficacy. The system consists of two co-operating real DC servomotors, where the first one is connected as a drive motor and the other one as a generator. The manipulated variable is the input voltage of DC motor and the output controlled variable is the angular speed represented by the output voltage in range of 0-10V. To obtain a model of the system, input-output relations of the plant have been identified with the help of the software LABREG [5]. The interconnection of the laboratory model with the software LABREG is assured by the Advantech data acquisition card type PCI 1711.

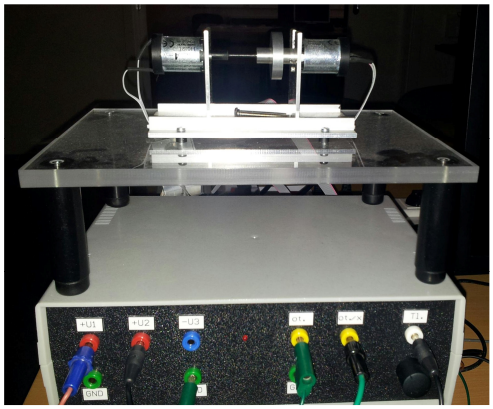

Figure 7. The laboratory model of DC motor

The discrete transfer function we obtained with the selected sampling rate T_s = 0.1s has been converted to the following continuous-time model:

$$G(s) = \frac{0.08047s + 1.677}{0.4142s^2 + 1.053s + 1} \quad (12)$$

The control objective was to drive the angular speed of the motor to track the desired reference signal.

B. Design of Control Algorithm

The first control algorithm has been developed using VHDL language in the Xilinx ISE Design Suite 14.4 software environment according to incremental PI from (Fig. 5). Firstly, a software implementation was developed and tested to verify the algorithm functionality. By choosing of appropriate sampling period and fixed point format a discrete PI controller has been developed from continuous-time PI controller, whereas the inverse dynamic tuning method has been used to tune the parameters k_p and T_i. From the parameter tuning experiment the following results were obtained: proportional gain k_p = 0.2025, integral coefficient T_i = 0.4752, derivative coefficient k_d = 0 and sample period T = 0.1s. The same discrete PI parameters were applied to the

 ISBN: 978-1-61208-344-5

hardware implementation on FPGA to perform the control tests.

Because of the fact that most FPGAs are limited to finite precision signal processing using fixed-point arithmetic, the bit word-length and radix setting of input and output signals were determined carefully to ensure the fidelity of the algorithm. Since every addition or subtraction causes adding an extra bit as well as every multiplication result will have a bit width equal to the sum of the number of bits in the inputs, this was the most important part of the design. A simple flow diagram (Fig. 8) shows the implementation of the designed algorithm using FPGA. The algorithm has 4 inputs: the motor system output, the reference signal *w* with the same bit width as signal *y*, clock and reset.

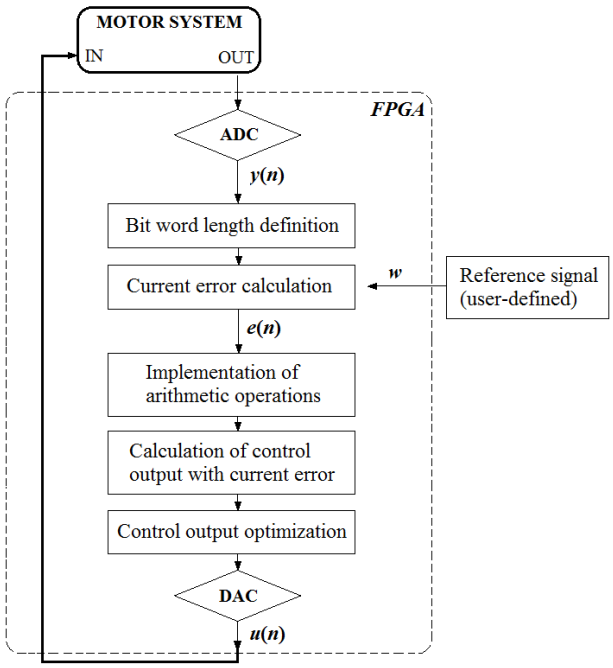

Figure 8. FPGA-based control implementation cycle

The step sequence of the algorithm can be determined as follows:

Step 1: Initialization of the system (set clock frequency, declaration of system variables);

Step 2: Setting of bit width of the signals (input and output signals according to the resolution of A/D conversion results);

Step 3: Calculation of the current error according to the reference signal defined by user ($e = w - y$);

Step 4: Calculation of the control output with the current error based on the combinatorial logic operations according to relation (10) and parallel architecture (Fig. 5);

Step 5: Optimization of the obtained control output for 8-bit D/A converter;

Step 6: The analog output signal obtained is fed back to drive the speed of DC motor system.

The second control algorithm was developed according to section C and (11) and also applied to the real-time speed control of the laboratory motor system. This algorithm is unlike the first one augmented of the anti-windup mechanism. The calculated control output is adjusted according to (10) and after that optimized and fed back to motor system through 8-bit DAC.

C. *Experimental Results*

The experimental studies were carried out to evaluate the performance of the proposed control algorithm. The algorithm was downloaded into SPARTAN-6 FPGA development kit (Fig. 9) and the complete system was reset.

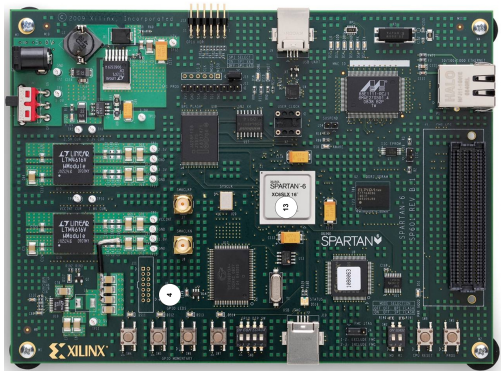

Figure 9. SPARTAN-6 development board

The comparison of the experimental results executed using FPGA with the simulation results obtained from MATLAB are illustrated in Fig. 10 below.

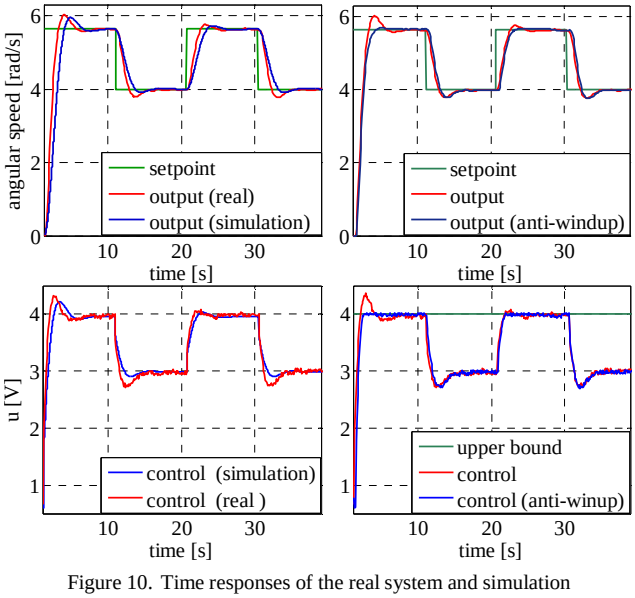

Figure 10. Time responses of the real system and simulation

The comparison of the performance of the proposed anti-windup PI control algorithm with the PI control algorithm without the anti-windup mechanism is also depicted. The

results show the effectiveness and good performance of the FPGA-based controller. The limited vector size of the signals due to the different interpretation of the fixed-point arithmetic has an effect in the calculation and therefore in the shape of the obtained time responses.

The objective evaluation of quality has been performed by meaning of various performance and quality criteria in the time domain (settling time, maximal overshoot and root mean square error (RMSE)) with results expressed in Table 1. The PID algorithm has been demonstrated to be effective for DC motor speed control.

TABLE I. QUANTIFICATION OF QUALITY CONTROL CRITERIA

Control Performance			
Controlled output	***Settling time***	***Max. Overshoot (%)***	***RMSE***
Real system	8,25	6,9449	0,8937
Simulation	9,1	5,6301	0,8992
Real system with anti-windup	5,3	0,7815	0,9217

As seen in Table 1, anti-windup mechanism has improved the control performance mostly in the way of overshoot and settling time.

D. Resource Usage

Xilinx tool device utilization summary and percentage of available resources reports, which have been used for the current design using FPGA are shown in Table 2 below.

TABLE II. DEVICES UTILIZATION SUMMARY

Device Utilization Summary			
Slice Logic Utilization	***Used***	***Available***	***Utilization***
Number of Slice Registers	170	18,224	1%
Number used as Flip Flops	168		
Number used as Latches	2		
Number of Slice LUTs	313	9,112	3%
Number used as logic	305	9,112	3%
Number of occupied Slices	124	2,278	5%
Number of MUXCYs used	204	4,556	4%
Number with an unused Flip Flop	186	344	54%
Number with an unused LUT	31	344	9%
Number of fully used LUT-FF pairs	127	344	36%
Number of slice register sites lost to control set restrictions	94	18,224	1%
Number of bonded IOBs	33	232	14%
Number of LOCed IOBs	13	33	39%
Number of RAMB16BWERs	0	32	0%
Number of RAMB8BWERs	0	64	0%
Number of BUFG/BUFGMUXs	5	16	31%
Number of DSP48A1s	3	32	9%

Hardware resources usage was: 168 slice flip-flops, 170 slice registers and 313 slice LUT's. It can be seen that just 5% of the FPGA was used.

V. CONCLUSION AND FUTURE WORK

In this paper, a closed-loop PI algorithm was proposed, designed and successfully implemented on FPGA platform. The performance was verified and tested for control of laboratory DC motor system. The control algorithm has been improved by using the anti-windup structure in case of considering the input constraints. The overall control algorithm has been programmed using VHDL language and implemented on Xilinx Spartan-6 FPGA development kit. The experimental results show a good set-point tracking and demonstrate that FPGAs are well suited for implementation of complex motor control algorithms due to their high speed execution characteristics. The future work will deal with the design and implementation of more complex predictive control algorithm considering the constraints on input, state and output variables.

ACKNOWLEDGMENT

The work has been supported by the Slovak Research and Development Agency under grants APVV-0772-12 and APVV-0246-12.

REFERENCES

[1] P. J. Ashenden, The Designer's Guide to VHDL. Morgan Kaufmann, 1995.

[2] K. J. Astrom and B. Wittenmark, Computer Controlled Systems, Englewood Cliffs, NJ: Prentice-Hall, 1997.

[3] Y. F. Chang, M. Moallem, and W. Wang, "Efficient implementation of PID control algorithm using FPGA technology," Proceedings of 43 IEEE Conference On Decision and Control, vol. 5, Dec. 2004, pp. 4885–4890.

[4] L. Charaabi, E. Monmasson, and I. Slama-Belkhodja, "Presentation of an efficient design methodology for FPGA implementation of control systems: Application to the design of an antiwindup PI controller," Proc. IEEE Ind. Electron. Soc. Annu. Conf., vol. 3, Nov. 2002, pp. 1942–1947.

[5] S. Kajan and M. Hypiusová, "Labreg Software for Identification and Control of Real Processes in Matlab," Technical Computing Prague 2007: 15th Annual Conference Proceedings, Prague, Czech Republic, Nov. 2007, pp. 71.

[6] R. Krasňanský and B. Dvorščák, "Design and Implementation of FPGA-based PID controller," In ACCS'13: 3rd International Conference on Advanced Control Circuits and Systems, ERI, Luxor, Dec. 2013, pp. 43.

[7] E. Monmasson and M. N. Cirstea, "FPGA Design Methodology for Industrial Control Systems-A Review," IEEE Transactions on Industrial Electronics, vol. 54, August 2007, pp. 1824-1842.

[8] J. Oldfield and R. Dorf, Field-Programmable Gate Arrays, John Wiley & Son, 1995.

[9] Xilinx Data Book, 2006, Available online at: www.xilinx.com (accessed March 28, 2014).

[10] W. Zhao, B. H. Kim, A. C. Larson, and R. M. Voyles, "FPGA implementation of closed-loop control system for small-scale robot," In ICAR'05: 12th International Conference on Advanced Robotics, Seattle, WA, July 2005, pp. 70-77.

 ISBN: 978-1-61208-344-5

First-Order Combinatorics Presenting a Conceptual Framework for Two Levels of Expressive Power of Predicate Logic

Mikhail Peretyat'kin
Institute of Mathematics and Mathematical Modeling
Almaty, Kazakhstan
e-mail: m.g.peretyatkin@predicate-logic.org

***Abstract*—In this work, we proceed to study finitary and infinitary first-order combinatorics within the framework of a new approach intended to investigations of predicate logic. Some properties of these combinatorics are established. We present a general scheme of semantic layers of model-theoretic properties having importance in the given direction. A number of demonstrations is given showing essence of both finitary and infinitary combinatorial methods for first-order theories. The work represents a basis for further investigations on expressive power of predicate logic.**

***Keywords*—*first order logic; theory; finitely axiomatizable theory; computably axiomatizable theory; Tarski-Lindenbaum algebra; model-theoretic property; computation; first-order combinatorics*.**

I. INTRODUCTION

Constructions of finitely axiomatizable theories were created to answer questions concerning a common problem about expressive power of first-order logic. There are constructions of Church [2], Kleene [7], Hanf [5], Peretyat'kin [12], and others. Each construction represents a general method for constructing finitely axiomatizable theories that can yield a series of finitely axiomatizable theories depending on one or a few input parameters. One can manage properties of the obtained theory by choice of the parameters. Some open questions on expressive power of first-order logic can also be solved with simpler methods based on the signature reduction procedures.

The idea to introduce a first-order combinatorial terminology have arisen from the available approaches to solve a principal problem of characterization of predicate logic of a finite language by building isomorphisms between the Tarski-Lindenbaum algebras of predicate calculi of two finite rich signatures with preservation as a large semantic layer of model-theoretic properties as possible [9][10][11]. The method of constructing such an isomorphism [10] is based on *algorithmic computation* in first-order predicate logic. It uses a universal construction of finitely axiomatizable theories simulating some computation of a Turing machine carrying out the role of a computer-controller. This approach can be said to be an *infinitary first-order combinatorics*. The second method of constructing the isomorphism, [11], is based on a *finite combinatorial transformation* in predicate logic. It uses so-called finite-to-finite signature reduction procedures and can be said to be *finitary first-order combinatorics*. In the work [14], a complex of concepts and general specifications connected with first-order combinatorics was given together with some reasoning justifying the combinatorial terminology for use in this direction. In this work, we introduce a number of further concepts and formulate some claims concerning applications of the finitary and infinitary first-order combinatorial methods for construction and transformation of theories in predicate logic. Besides, a series of general statements is formulated, and a number of demonstrations are given showing essence of the concepts related with finitary and infinitary first-order combinatorics and outlining limits of their possible applications.

In the third section, we introduce definitions of semantic layers relevant in this direction, in the fourth section we introduce a concept of the relation of virtual definable equivalence between theories, the fifth section describes a common scheme of application for infinitary first-order combinatorics, the sixth section specifies possible versions of the universal construction of finitely axiomatizable theories, in the seventh section we list some common statements concerning first-order combinatorics, in the eighth section we describe main situations corresponding to first-order combinatorics. In the ninth section we give some summary to the paper.

II. PRELIMINARIES

Theories in first-order predicate logic with equality are considered. General concepts of model theory, algorithm theory, Boolean algebras, and constructive models can be found in Hodges [6], Rogers [17], and Goncharov and Ershov [4]. Basic concepts concerning first-order combinatorics can be found in [14]. Generally, *incomplete* theories of finite or enumerable signatures are considered.

A finite signature is called *rich*, if it contains at least one $n$ary predicate or function symbol for $n>1$, or two unary function symbols. The following notations are used: $FL(\sigma)$ is the set of all formulas of signature σ, $FL_k(\sigma)$ is the set of all formulas of signature σ with free variables $x_0,...,x_{k-1}$, $SL(\sigma)$ is the set of all sentences (i.e., closed formulas) of signature σ. A theory is said to be *computably axiomatizable* if it admits a computable system of axioms. By $L(T)$, we denote the Tarski-Lindenbaum algebra of theory T of formulas without free variables, while $\mathcal{L}(T)$ denotes the Tarski-Lindenbaum algebra $L(T)$ considered together with a Gödel numbering γ; thereby, the concept of a computable isomorphism is applicable to such objects.

 ISBN: 978-1-61208-344-5

Let σ be a signature, and Σ be a subset of $SL(\sigma)$. By $[\Sigma]^\star$, we denote a theory of a signature $\sigma' \subseteq \sigma$ generated by the set Σ as a set of its axioms, where σ' contains only those symbols from σ that occur in formulas of the set Σ. Let σ^∞ be a fixed enumerable maximally large infinite signature containing countably many both constant symbols, symbols of propositional variables, and predicate and function symbols of each arity $n>0$. If a theory T of signature σ^∞ is defined by the set of axioms $\{\Phi_i \,|\, i \in W_m\}$ as follows: $T=[\Sigma]^\star$ (where W_m is mth computably enumerable set in Posts's numbering), the number m is called a *weak computably enumerable index* or simply *weak index* of T, and we denote this theory by $T^\star{}_{\{m\}}$, $m \in \mathbb{N}$. This sequence represents all possible computably axiomatizable theories, up to an algebraic isomorphism of theories. Symbol $\mathfrak{P}(X_0,...,X_\mathfrak{a})$, shortly $\mathfrak{P}$, is specialized to denote a propositional formula of signature $\sigma^* = \{X_0, X_1, ..., X_k, ...; k \in \mathbb{N}\}$ (consisting of propositional variables), while $\mathfrak{a}$ specializes the number of variables occurred in the formula. By *PRO*, we denote the set of all such formulas, while $\mathfrak{P}_i(X_0,...,X_{\mathfrak{a}(i)})$, $i \in \mathbb{N}$, is a fixed Gödel numbering of the set *PRO*. For a set $A \subseteq \mathbb{N}$, record $A \models \mathfrak{P}$ denotes the value of term $\mathfrak{P}(\chi_A(0), \chi_A(1), ..., \chi_A(\mathfrak{a}))$, where $\chi_A(x)$ is characteristic function of the set A. Here, propositional formula $\mathfrak{P}$ plays the role of a table condition applicable for set $A \subseteq \mathbb{N}$.

Formulation to the *universal construction* $\mathbb{FU}$ of finitely axiomatizable theories can be found in [12, Ch.6]. Main definitions connected with semantic layers are found in [14]. We use notation *MQL* for the model quasiexact semantic layer presenting *infinitary first-order combinatorics*, [14].

III. First-order combinatorics and a scheme of semantic layers

In accordance with specifications [14], signature reduction procedures represent a basis for the concept of first-order combinatorics; they are considered as particular cases of combinatorial methods in predicate logic. Signature transformations "finite-to-finite" represent finitary combinatorial methods, while signature reduction procedures "infinite-to-finite" represent infinitary combinatorial methods. The problem is to generalize these particular methods to a maximum general approach for which it would be possible to apply such a serious term as 'combinatorics'. A principal aim of the combinatorial approach is to characterize classes of finitary and infinitary methods of transformation of theories. After that, we can define the *finitary semantic layer* as the set of those model-theoretic properties $\mathfrak{p}$ which are preserved under finitary first-order methods, and *infinitary semantic layer* as the set of those properties $\mathfrak{p}$ which are preserved under infinitary methods. For the first-order combinatorial approach, its perfection is considered as a demand of higher priority, while the maximality of the semantic layers of preserved model-theoretic properties is considered as a demand of secondary priority.

In Fig. 1, we present a scheme of inclusions between the semantic layers and similarity relations relevant for first-order combinatorics. Two relations $\approx$ and $\approx_a$ in the top are relations of isomorphism of theories, where $\approx$ means a *model isomorphism* or simply *isomorphism*, while $\approx_a$ means an *algebraic isomorphism* or $\exists\cap\forall$*-presentable equivalence* between two theories. Although $\approx$ and $\approx_a$ are not similarity relations, they are included in the scheme for the sake of completeness. Relations $\equiv_l$ and $\equiv_{al}$ are similarity relations relative to the semantic layer *ML* consisting of all model properties, and respectively, to the layer *AL* consisting of all algebraic properties. Semantic layers *MDL*, *ADL*, *MCL*, etc., and corresponding similarity relations $\equiv_{ad}$, $\equiv_d$, etc., are defined by the classes of *singleton*, *Cartesian*, and respectively, *Cartesian-quotient* interpretations [14]. Leading letter A means an algebraic version while M means a model version. A middle letter S means 'singleton', C means 'Cartesian', and

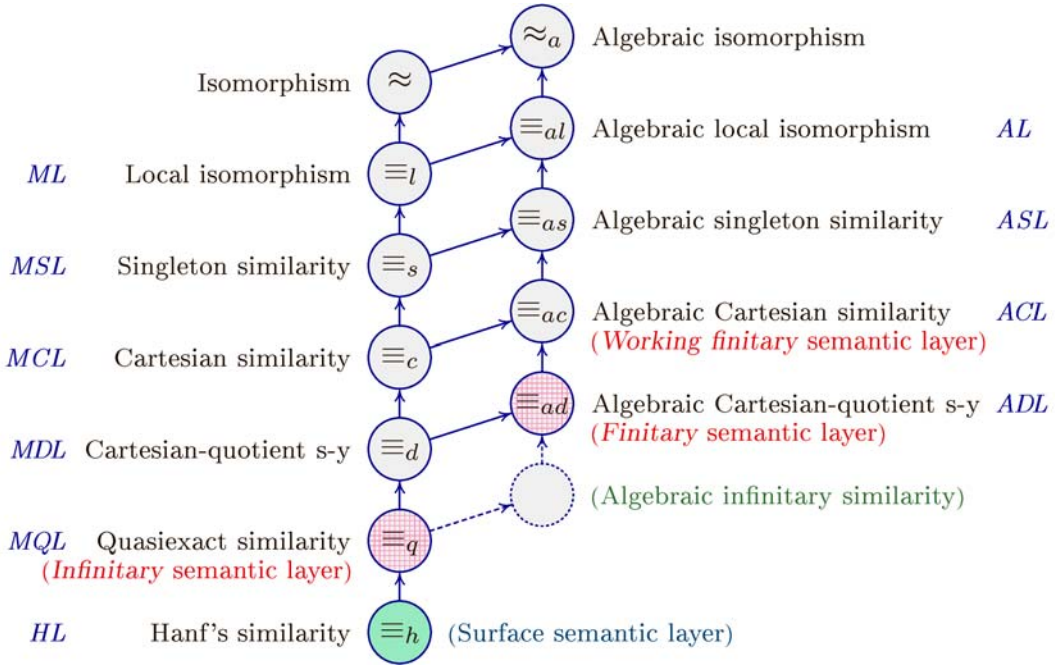

Fig. 1. A scheme of semantic layers of model-theoretic properties

D means 'Cartesian-quotient'. The Hanf layer *HL* is supposed to be $\varnothing$.

Infinitary semantic layer *MQL* [12][14] has a sophisticated definition. Therefore, it would be useful to introduce a simple rule to check whether a model-theoretic property $\mathfrak{p}$ is included in this layer. For this purpose, we consider two following classes of interpretations of theories:

$$I: T \to T\langle \varphi_1^{m_1}/\varepsilon_1, ..., \varphi_k^{m_k}/\varepsilon_k \rangle, \quad (1)$$

$$I: T \to T\langle \varphi_1^{m_1}/\varepsilon_1, ..., \varphi_k^{m_k}/\varepsilon_k \rangle \oplus SI, \quad (2)$$

where T is a computably axiomatizable theory of an enumerable signature σ, $\varkappa = \langle \varphi_1^{m_1}/\varepsilon_1, ..., \varphi_k^{m_k}/\varepsilon_k \rangle$ is a tuple of formulas of signature σ suitable for the Cartesian-quotient extensions, [14, Section 3], while *SI* is the theory of a successor relation with an initial element in signature $\{\lhd^2, c\}$. Based on this, we define two following semantic layers:

Reference_Block (3)

(a) *FinL* = the set of all model-theoretic properties $\mathfrak{p}$ of algebraic type preserved by any interpretation I of the form (1) with an arbitrary computably axiomatizable theory T and arbitrary tuple $\varkappa$ of this form,

(b) *InfL* = the set of all model-theoretic properties $\mathfrak{p}$ of algebraic type preserved by any interpretation I of the form (2) with an arbitrary computably axiomatizable theory T and arbitrary tuple $\varkappa$ of this form.

End_Ref

It can be checked that the following assertions take place:

$$FinL = ADL, \quad (4)$$

$$InfL \stackrel{\mathfrak{R}}{=} I2f\mathfrak{L}, \quad InfL \cap ML \stackrel{\mathfrak{R}}{=} I2f\mathfrak{L} \cap ML \stackrel{\mathfrak{R}}{=} Uni\mathfrak{L}. \quad (5)$$

This shows that the family (1) forms a representative class of interpretations for finitary semantic layer *ADL*. On the other hand, a simple modification (2) of the scheme (1) forms a class of interpretations for the semantic layer *InfL*, that, in view of (5), can play the role of a simple rule to check whether a model-theoretic property $\mathfrak{p}$ is included in infinitary semantic layer. The layer *FinL* is said to be the *rapid finitary* semantic layer, while *InfL* is said to be the *rapid infinitary* semantic layer. Simplicity of the definitions (3)(a) and (3)(b) ensures relevance of the layers *FinL* and *InfL* for first-order combinatorics.

As for the algebraic version of the infinitary semantic layer, it is currently not supported by any version of the universal construction of finitely axiomatizable theories. This layer is included in the scheme in Fig. 1 for completeness (shown by a dashed circle).

IV. Virtual definable extensions and finitary first-order combinatorics

There is a known in model theory method of addition to the universe imaginary elements corresponding to a definable set of elements or even to a first-order definable set of tuples of certain length modulo a definable equivalence relation (they are said to be *virtual* elements). Let's add a finite set of virtual regions to the universe. Furthermore, we have to include in signature new predicates distinguishing these areas and establishing a relation of the new elements with the old tuples modulo the equivalence relations. There is a possibility, based on predicate logic, to manipulate with first-order formulas in the extended universe containing the source universe together with the added virtual areas. For this purpose, it is required to define special rules of construction and interpretation of first-order formulas in such an extended region. This method allows us, remaining inside the old universe of models of theory T, to manipulate with language of first-order logic in models of some new theory S, which is possible said to be a *virtual first-order definable extension* of the source theory T. One can mention that, any model-theoretic properties of theories T and S should be considered as coincided since the virtually extended theory S is presented in the initial theory T. Furthermore, notice that such an operation of addition of a finite number of virtual definable regions can be performed in a general situation when the source theory T is incomplete. In this case, we obtain a computable isomorphism between the Tarski-Lindenbaum algebras $\mu: \mathcal{L}(T) \to \mathcal{L}(S)$ preserving all really model-theoretic properties.

Two theories T and S are said to be *virtually definably equivalent*, written as $T \approx\!\!\approx S$, if there are virtual definable extensions T' of T and S' of S such that T' and S' are algebraically isomorphic, $T' \approx_a S'$. This relation, close to that considered in Manders [8], seems to be the most common equivalence relation between first-order theories. Since the operation of a virtual definable extension of a theory is closely related to the operation of Cartesian-quotient extension of a theory, [14, Section 3], we obtain that this relation between theories plays the principal role within the complex of concepts for finitary first-order combinatorics.

V. Normalized scheme for infinitary first-order combinatorics

In this section, we specify some method of construction of finitely axiomatizable theories with pre-assigned model-theoretic properties. In the most common case, the target theory depends on an input parameter n. Our goal is to construct a finitely axiomatizable theory $F = F^{(n)}$ of a given finite rich signature τ. First, we build an intermediate computably axiomatizable theory $T = T^{(n)}$ using some particular method. Signature of T:

$$\sigma = \{X_i \mid i \in \mathbb{N}\} \cup \sigma', \quad (6)$$

where X_i, $i \in \mathbb{N}$, is a sequence of nulary predicates (i.e., propositional variables), and σ' depends on the aim of our construction. Axioms of T consist of three groups:

Frame: a group of axioms describing general form of a so-called skeleton of the theory; these axioms depend on the aims of the construction;

Space: formulas of the form $\mathfrak{P}(X_0, ..., X_\mathfrak{a})$, with $\mathfrak{P} \in PRO$;

Ext: formulas of the form $\mathfrak{P}(X_0, ..., X_\mathfrak{a}) \to \Psi$, with $\mathfrak{P} \in PRO$ and $\Psi \in SL(\sigma')$.

Applying the universal construction $\mathbb{FU}$, we build a finitely axiomatizable theory $F = F^{(n)} = \mathbb{FU}(T, \tau)$ of the wished finite rich signature τ together with a computable isomorphism $\mu: \mathcal{L}(T) \to \mathcal{L}(F)$ between the Tarski-Lindenbaum algebras preserving model-theoretic properties of their completions within the infinitary semantic layer *MQL*.

 ISBN: 978-1-61208-344-5

Introduce the following notation

$$\theta_i = \mu(X_i),\ i \in \mathbb{N}. \tag{7}$$

For an arbitrary set $A \subseteq \mathbb{N}$, we denote

$$T[A] = T + \{X_i \mid i \in A\} \cup \{\neg X_j \mid j \in \mathbb{N} \setminus A\}, \tag{8}$$
$$F[A] = F + \{\theta_i \mid i \in A\} \cup \{\neg \theta_j \mid j \in \mathbb{N} \setminus A\}.$$

Furthermore, we define a number m such that

$$W_m = \{k \mid T \vdash \mathfrak{P}_k(X_0, ..., X_{\mathfrak{a}(k)})\}, \tag{9}$$

and introduce the following notation

$$\Omega(m) = \{A \subseteq \mathbb{N} \mid (\forall k \in \Omega(m)) A \models \mathfrak{P}_k\}. \tag{10}$$

Any object involved in the transformation $n \mapsto T \mapsto F$ is presented via appropriate computably enumerable index or Gödel number such that the whole passage $n \mapsto T \mapsto F$ is defined by an effective operator relative to indices and/or Gödel numbers. The given complex of transformations is said to be *normalized* if the following conditions are satisfied:

$$\begin{array}{l} \text{(a)}\ T \vdash \mathfrak{P}(X_0, ..., X_{\mathfrak{a}}) \Leftrightarrow T.Spase \vdash \mathfrak{P}(X_0, ..., X_{\mathfrak{a}}), \mathfrak{P} \in \textit{PRO}, \\ \text{(b)}\ (\forall A \subseteq \mathbb{N})\ T[A] \textit{ is either complete or contradictory}. \end{array} \tag{11}$$

These conditions are, in fact, natural. If we have an arbitrary effective transformation $n \mapsto T' \mapsto F'$, where T' is a computably axiomatizable theory constructed from n, and F' is a finitely axiomatizable theory of signature τ obtained from T' by the universal construction, then, this scheme can equivalently be transformed in the form of a normalized scheme $n \mapsto T \mapsto F$. Furthermore, any normalized complex must satisfy the following properties: (a) $T[A]$, $A \in \Omega(m)$ represents the family of all complete extensions of T; (b) $F[A]$, $A \in \Omega(m)$, represents the family of all complete extensions of F; (c) isomorphism μ maps $T[A]$ to $F[A]$, for all $A \in \Omega(m)$; (d) for any $A \in \Omega(m)$ complete theories $T[A]$ and $F[A]$ have identical model-theoretic properties within the infinitary semantic layer *MQL*; (e) effectively, in the system of axioms of T, one can find $s \in \mathbb{N}$ such that function $\varphi_s^A(t)$ is characteristic for the set $\mathrm{Nom}(T[A])$, for all $A \in \Omega(m)$; a definition of $\varphi_s^A(t)$ is found in [17, p.130].

VI. Versions of the universal construction

Hereafter, we use notation *MQL* for a sublayer of the infinitary layer *MQL*.

Simplest form of the universal construction, denoted $\mathbb{F}\breve{\mathbb{U}}$, is presented by:

Statement 1. [Generic universal construction: a primitive form] *The following assertion holds* (*where MQL* $\subseteq$ *MQL*):

$$(\forall\ c.a.\ theory\ T)(\exists\ f.a.\ theory\ F)\left[T \equiv_{MQL} F\right]. \tag{12}$$

A more common formulation to the universal construction, [12, Th.0.6.1]:

Statement 2. [Generic universal construction: a normal form] *Given an arbitrary computably axiomatizable theory T and a finite rich signature σ. Effectively in a weak c.e. index of T and Gödel number of σ, one can construct a finitely axiomatizable theory $F = \mathbb{F}\mathbb{U}(T, \sigma)$ of signature σ together with a computable isomorphism $\mu : \mathcal{L}(T) \to \mathcal{L}(F)$ between the Tarski-Lindenbaum algebras preserving all model-theoretic properties within the layer MQL* (*it is supposed that MQL* $\subseteq$ *MQL*).

The following dependence statement takes place.

Lemma 3. *Having any version of the universal construction in the primitive form* (12) *with the semantic layer MQL* $\subseteq$ *MQL, one can restore the missing effectiveness requirement obtaining its normal form presented in Statement* 2 *with the same layer MQL.*

Proof. First, we introduce an operation with a sequence of theories. We use sequence $T^{\star}{}_{\{n\}}$, $n \in \mathbb{N}$, including all, up to an isomorphism, c.a. theories, cf. Preliminaries. Let $T^{\star}{}_{\{n\}}$ has signature σ_n. It is assumed that $\sigma_n \cap \sigma_k = \varnothing$ for all n, k such that $n \neq k$. Consider the following new signature

$$\sigma' = \{Z_i^0 \mid i \in \mathbb{N}\} \cup \{U^1, c\} \cup \sigma_0 \cup \sigma_1 \cup \ldots \cup \sigma_k \cup \ldots, \tag{13}$$

where Z_i^0, $i \in \mathbb{N}$, are symbols of nulary predicates. It is assumed that the symbols U, c, and Z_i, $i \in \mathbb{N}$, do not belong to $\sigma_0 \cup \sigma_1 \cup \ldots \cup \sigma_k \cup \ldots$.

Construct theory $T^u_{c.a.}$ of signature σ' defined by the following set of axioms:

1°. $U(x) \leftrightarrow (x \neq c)$,

2°. $(\exists x) U(x)$,

3°. $Z_n \to \neg Z_k,\ \ n, k \in \mathbb{N},\ n \neq k$,

4°. $Z_n \to$ (*on* $U(x)$, *axioms of* T_n *are satisfied*), $\ n \in \mathbb{N}$,

5°. $Z_n \to$ (*outside* $U(x)$, σ_n*-symbols defined trivially*),

6°. $\neg Z_k \to$ (*all* σ_k*-symbols are defined* c*-trivially*), $k \in \mathbb{N}$.

Denote this theory by $\bigotimes_{n \in \mathbb{N}} T^{\star}{}_{\{n\}}$. The statement above "defined c-trivially" means that all σ_k-predicates are identically false, each σ_k-function f^m satisfies $f(x_1, ..., x_m) = x_1$ for all its arguments, and each σ_k-constant is interpreted by c.

We can show that the following assertions hold:

(a) theory $T^u_{c.a.} = \bigotimes_{n \in \mathbb{N}} T_n$ is computably axiomatizable;

(b) for any $n \in \mathbb{N}$, theory $T^u_{c.a.} \cup \{Z_n\}$ is algebraically isomorphic to the constant extension $T^{\star}{}_{\{n\}\langle c\rangle}$ of the theory $T^{\star}{}_{\{n\}}$;

(c) there is a computable isomorphism $\mu_n : \mathcal{L}(T^{\star}{}_{\{n\}}) \to \mathcal{L}(T^u_{c.a.} \cup \{Z_n\})$ preserving all model-theoretic properties within the semantic layer *ASL*.

Part (a) is a consequence of the fact that the sequence $T^{\star}{}_{\{n\}}$, $n \in \mathbb{N}$, is computable. Part (b) is checked immediately. Part (c) is a consequence of (b).

Now, we are going to use the universal c.a. theory $T^u_{c.a.}$ to deduce the normal form of the universal construction from its primitive form. Applying the primitive form (12) of the universal construction, we find a finitely axiomatizable theory F_0 together with a computable isomorphism $\mu_0 : \mathcal{L}(T^u_{c.a.}) \to \mathcal{L}(F_0)$ preserving the layer *MQL*. After that, a construction with the effectiveness requirement is obtained as an immediate consequence of the universality condition for $T^u_{c.a.}$ stated in

 ISBN: 978-1-61208-344-5

(a)–(c); namely, we have to perform the following transformation:

$$T^{\star}{}_{\{n\}} \mapsto \underbrace{T^{u}_{c.a.}+\{Z_n\}}_{S} \mapsto \mathbb{F}\check{\mathfrak{u}}(S) \mapsto \mathrm{Redu}(\mathbb{F}\check{\mathfrak{u}}(S),\sigma), \quad (14)$$

where $\mathrm{Redu}(H,\sigma)$ denotes a signature reduction procedure from a finitely axiomatizable theory H to such a theory of finite rich signature σ. By construction, we can effectively build a computable isomorphism between the Tarski-Lindenbaum algebras of theories $T^{\star}{}_{\{n\}}$ and $\mathrm{Redu}(\mathbb{F}\check{\mathfrak{u}}(T^{u}_{c.a.}+\{Z_n\}),\sigma)$. Thus, the transformation (14) can play the role of a normal form of the universal construction with the layer *MQL*. □

The following statement represents so-called *universal-under-canonical* construction; alternatively, it is said to be the *canonical-mini* construction:

Statement 4. *There is a routine proof* (*by way of transformation of theories based on the methods of infinitary first-order combinatorics*) *that, from statement of the canonical construction,* [*12, Ch.3, Th.3.1.1*], *deduces a weak version of the universal construction with the following semantic layer of model-theoretic properties* (*denoted by M1L°*):

(a) *existence of a prime model, its strong constructivizability, and the value of its algorithmic dimension* (*relative to strong constructivizations*);
(b) *existence of a countable saturated model and its strong constructivizability*.

PROOF. Only outline of the proof is given. The canonical construction can control those model-theoretic properties which are expressible in signature $\sigma^{*}=\{P^{1}_{0},P^{1}_{1},...,P^{1}_{k},...;k\in\mathbb{N}\}$ with infinitely many unary predicates (pay an attention: Chapter 3 of [12] is titled "The construction over a unary list", where the *list* means a *layer*). On the other hand, the pointed out layer *M1L°* consists of exactly those model-theoretic properties controlled by the canonical construction, which are expressible in terms of structure of the Tarski-Lindenbaum algebras $\mathcal{L}_n(T)$, $n\in\mathbb{N}$, $n>0$, of theory T.

Let T be an arbitrary computably axiomatizable theory of an enumerable signature σ' having Gödel numbering Φ_i, $i\in\mathbb{N}$, for the set $SL(\sigma')$. The sequence of sentences Φ_i, $i\in\mathbb{N}$, is a generating set for the Tarski-Lindenbaum algebra $\mathcal{L}(T)$. Enrich σ' with propositional variables X_i, $i\in\mathbb{N}$, and add to T additional axioms $X_i\leftrightarrow\Phi_i$, $i\in\mathbb{N}$. Construct parameterized Stone space $\Omega(m)$ for T relative to generating sequence X_i, $i\in\mathbb{N}$. Considering a set $A\in\Omega(m)$ as an oracle, let's construct Boolean algebra $\mathcal{B}=\bigotimes_{0<i<\omega}\mathcal{L}_k(T[A])$ that, in fact, is a c.e. Boolean algebra relative to computation with oracle A. It is an important moment that satisfaction of each model-theoretic property $\mathfrak{p}\in M1L^{\circ}$ in theory T is expressible (in a known way) via the algebra $\mathcal{B}$. On the other hand, we can present the algebra $\mathcal{B}$ (depending on oracle A) via c.e. binary tree computable with the same oracle. This gives a value to the second parameter $s\in\mathbb{N}$ to the canonical construction. Applying the construction $\mathbb{F}\mathbb{C}$ to the obtained pair of input arguments (m,s), we finally build theory $F=\mathbb{F}\mathbb{C}(m,s,\sigma)$ that, by virtue of main statement of the canonical construction, is the required finitely axiomatizable theory. □

Mention that, an available proof for the canonical construction is essentially simpler in comparison with that for the universal construction. On the other hand, the proof given above represents a demonstration of the methods of infinitary first-order combinatorics.

VII. SUMMARY: SOME COMMON STATEMENTS CONCERNING FIRST-ORDER COMBINATORICS

In this paragraph, we formulate a series of common statements corresponding to finitary and infinitary first-order combinatorics (or not corresponding to such a combinatorics).

S1. In the case of finitary first-order combinatorics, characteristic property of the transformation between theories is availability of a one-to-one mapping between the isomorphism types of their models (this property is said to be the *model-bijectiveness*).

S2. In the case of infinitary first-order combinatorics, a characteristic property of the construction is availability of non-standard fragments in models of the target theory, whose description should be simple enough; moreover, this simplicity is a principal demand of infinitary first-order combinatorics.

S3. In the case of infinitary first-order combinatorics, our goal is to build a computably axiomatizable theory that, generally, may be incomplete; a description of the family of all complete extensions of the theory should be presented; the axioms should provide some pre-assigned properties of these extensions depending on an input parameter; applying an appropriate version of the universal construction, we obtain the target finitely axiomatizable theory.

S4. In the case of infinitary first-order combinatorics, the input parameter could be absent if our goal is to build a separate example of finitely axiomatizable theory with some pre-assigned properties; on the other hand, we can use a few input parameters (more than one) if it is necessary for the problem considered.

S5. In the case of infinitary first-order combinatorics, a complete theory may be considered as a particular case of incomplete theories; however, if the purpose is limited with complete theories only, such a construction does not correspond to specifications of infinitary first-order combinatorics or is weakly linked with it.

S6. If we consider or build an incomplete theory, but it is impossible to parameterize the family of its complete extensions, such a construction does not correspond to specifications of infinitary first-order combinatorics or is weakly linked with it.

S7. In the case of infinitary first-order combinatorics, first of all, the used methods of construction or transformation of theories are principal; as for the requirements of computability of the construction and enumerability of the signature, they ordinarily are satisfied automatically.

S8. In the case of infinitary first-order combinatorics, a sublayer of the full infinitary semantic layer *MQL* may be considered; an empty layer $\varnothing$ is also admissible.

VIII. DEMONSTRATIONS: SITUATIONS CORRESPONDING TO FIRST-ORDER COMBINATORICS

In this section, we consider a number of typical examples of applications of finitary and infinitary combinatorial meth-

 ISBN: 978-1-61208-344-5

ods; we also demonstrate some situations when methods of construction and transformation of theories does not correspond to the concept of first-order combinatorics.

1) Definitionally equivalent theories: In [16, p. 481], C. Pinter writes, "There are many common instances of theories, which may be formulated naturally in more than one way, using different sets of primitive relations and operations. For example, lattice theory may be presented as a ... theory ... with the operations $+$ and $\cdot$, or alternatively, as a theory ... whose language has only one nonlogical symbol $\leqslant$. ... When theories T and T' are related in this manner, they are said to be *definitionally equivalent*." Similar sense has the concept of relation of *synonymity of theories* introduced in Bouvere [1]. As mentioned in [18, p.130], "... synonymity requires the universe to remain unchanged," the same is also true relative to Pinter's definitional equivalence. Some additional examples: Boolean algebras can be considered in the signature either $\{\cup,\cap,-,0,1\}$, or $\{\subseteq,0,1\}$, or even $\{+,\cdot,0,1\}$; group theory can be considered in the signature either $\{\cdot,e\}$, or $\{\cdot\}$, or even $\{+,\theta\}$, etc. In these situations, we have a simplified version of finitary first-order combinatorics (because more common virtual definable extensions of theories are not used here).

2) Virtual definitionally equivalent theories: Some situations which are close to finitary first-order combinatorics were discussed by Leslaw Szczerba in [18]. At [18, p. 130] he writes, "... authors frequently use sequences of elements as new elements, members of a new universe (e.g., points may be pairs of real numbers as in the case of the Cartesian plane), universes might be restricted to definable subsets, and moreover, new elements might be equivalence classes with respect to some definable equivalence relation." These statements exactly correspond to the concept of Cartesian extension of a theory and, in other words, to the concept of a virtual definitional extension of a theory. Here, we exactly have finitary first-order combinatorics. Dale Myers in [9, p.85] calls this transformation an *interpretive isomorphism* and states that this definition was introduced in Manders [8] (author's remark: Manders' definition is based on Szczerba's ideas). Some additional examples: (a) Consider the class K of models, which are Boolean algebras in signature $\{\cup,\cap\}$ with omitted both particular elements 0 and 1; thereby, the operations are partial. Alternatively, we can consider this class of systems in signature $\{\subseteq\}$. Applying virtual definable extension to $\mathrm{Th}(K)$, we can obtain theory *BA* of Boolean algebras. From this fact, we obtain that theories $\mathrm{Th}(K)$ and *BA* have identical model-theoretic properties (namely, there is a computable isomorphism $\mu:\mathcal{L}(\mathrm{Th}(K))\to\mathcal{L}(BA)$ that preserves all model-theoretic properties). (b) Another example is a system of positive real numbers $\mathfrak{N}=(\mathbb{R}^{>0},\cdot,+,-)$ with partial operation $-$. Applying virtual definable extension to $\mathrm{Th}(\mathfrak{N})$, we can obtain theory $\mathrm{Th}(\mathfrak{M})$, where $\mathfrak{M}=(\mathbb{R},\cdot,+,-)$; thereby, theories $\mathrm{Th}(\mathfrak{N})$ and $\mathrm{Th}(\mathfrak{M})$ have identical model-theoretic properties.

In all these cases, we have a situation of finitary first-order combinatorics.

3) Particular examples of finitely axiomatizable theories: Suppose that we are going to construct a finitely axiomatizable theory F of a given finite rich signature σ satisfying the following properties: the set of all complete extensions of F consists of a countable sequence F_k, $k\in\mathbb{N}\cup\{\omega\}$, such that, each of the theories F_0, F_1, F_2, ... is ω-stable theory and is finitely axiomatizable over F, while F_ω is not finitely axiomatizable over F, it is not ω-stable and has a prime model. First, we have to find a computably axiomatizable theory T with these properties (it is a simple exercise to build such a theory). Applying the universal construction to T, we can pass a finitely axiomatizable theory $F=\mathbb{FU}(T,\sigma)$ together with a computable isomorphism $\mu:\mathcal{L}(T)\to\mathcal{L}(F)$ preserving all model-theoretic properties of the infinitary semantic layer *MQL*. Because both properties $\mathfrak{p}_1=$"theory is ω-stable" and $\mathfrak{p}_2=$"theory has a prime model" belong to *MQL*, we obtain finally that the theory F indeed satisfies the posed properties.

This example demonstrates methods of infinitary first-order combinatorics.

4) Algorithmic complexity estimates for semantic classes: Let σ be a finite rich signature, and Φ_i, $i\in\mathbb{N}$, be a fixed Gödel numbering for the set of sentences of this signature. We are going to prove the following statement.

Theorem 5. $\{n\,|\,\Phi_n$ *determines a complete theory*$\}\approx\Pi^0_2$.

PROOF. The upper estimate can be established immediately.

For the lower estimate, we consider the following m-universal in Π^0_2 set: $I=\{n\,|\,W_n$ *is infinite* $\}$, [17, Th.13-VIII, p.264]. Signature of the theory $\sigma=\{X_0,...,X_i,...\}$ consists of propositional variables (i.e., nulary predicates). Given an input parameter n. Consider computably axiomatizable theory $T=T^{(n)}$ of signature σ, determined by the following set of axioms:

1°. $X_k\leftrightarrow(\exists x_1...x_k)\bigwedge_{0<i<j\leqslant k}(x_i\neq x_j)$,

2°. X_k, $k\in W_n$.

Applying the universal construction $\mathbb{FU}$ to $T^{(n)}$, we effectively find a finitely axiomatizable theory $F=F^{(n)}=\mathbb{FU}(T^{(n)},\sigma)$ of signature σ together with a computable isomorphism $\mu:\mathcal{L}(T)\to\mathcal{L}(F)$. First, consider the case $n\in W_n$. In this case, W_n is infinite, so all models of T are infinite and the theory is ω_0-categorical; thus, T is complete by Vaught Theorem. Now, consider the case $n\notin W_n$. In this case, W_n is finite, thereby, theory T cannot be complete since it has both finite and infinite models. As a result, we have obtained that the theory T is complete if and only if $n\in I$; thus, we have

$$n\in I\Leftrightarrow T_n \text{ is complete } \Leftrightarrow F_n \text{ is complete.}$$

The theory $F^{(n)}$ is defined effectively in $T^{(n)}$. Therefore, there exists a total computable function $f(n)$ such that the sentence $\Phi_{f(n)}$ is an axiom of this theory. Finally, we obtain the required lower estimate:

$$n\in I\Leftrightarrow\Phi_{f(n)} \text{ determines a complete theory.}$$

Proof of Theorem 5 demonstrates methods of infinitary first-order combinatorics. Furthermore, there are lots of results in this direction in [12, Ch. 8].

5) Isomorphisms between predicate calculi of different finite rich signatures: We consider a problematic concerning the isomorphism type of the Tarski-Lindenbaum algebra $\mathcal{L}(PC(\sigma))$ of predicate calculus $PC(\sigma)$ of a finite rich signature σ. Methods of [11] determine a Hanf's isomorphism μ between the Tarski-Lindenbaum algebras $\mathcal{L}(PC(\sigma_1))$ and $\mathcal{L}(PC(\sigma_2))$ of

 ISBN: 978-1-61208-344-5

any two finite rich signatures σ_1 and σ_2. It is assembled from a countable set of partial mappings, which are finite-to-finite signature reduction procedures; thereby, this isomorphism μ preserves all really model-theoretic properties. The work by Myers [9] also defines such an isomorphism between the predicate calculi $\mathcal{L}(PC(\sigma_1))$ and $\mathcal{L}(PC(\sigma_2))$ that is assembled from partial mappings described by Gaifman's maps [3]. These two approaches coincide with each other from the point of view of finitary first-order combinatorics.

6) Structure of the Tarski-Lindenbaum algebras of semantic classes: There are lots of results in this direction in papers [13][15] and others. Their proofs demonstrate methods of infinitary first-order combinatorics.

7) Definability in Peano Arithmetic and set theory: Let T be a rich theory like arithmetic or set theory (for definiteness, let T be Peano Arithmetic). It is a known fact that T is not complete; moreover, any finitely axiomatizable extension of T cannot be complete. Thereby, we have obtained that the Tarski-Lindenbaum algebra $\mathcal{L}(T)$ must be countable, atomless Boolean algebra. However, axiomatic of the theory T is such that neither direct parameterization nor even understandable description of the family of all complete extensions is possible. Thus, argumentation of statement S6, cf. Section VII, is applicable to this situation concerned to rich formal systems. Some additional examples: Church construction [2], Kleene construction [7] (presenting an extension of Peano Arithmetic), first-order presentation of any universal computing system, etc.

All these examples demonstrate situations outside of the approach based on the methods of first-order combinatorics.

IX. CONCLUSION

The work presents some extra details and gives general demonstrations and specifications to the concepts of finitary and infinitary first-order combinatorics. Based on both formal substantiations and informal arguments, we show that the introduced complex of concepts and definitions for the first-order combinatorics adequately corresponds to the problems on expressive possibilities of predicate logic presenting a firm basis for Computer Science as well as for other branches of mathematics.

In Section IV, we introduced the concept of virtual definable equivalence between theories; this relation presents essence of finitary first-order combinatorics. Further, in Section V, we describe a scheme of application of infinitary first-order combinatorics. This scheme represents the most general form of a computable procedure to build a theory T from a complex C of objects of computational nature with a transformation of the obtained theory T to a finitely axiomatizable theory F together with a computable isomorphism $\mu : \mathcal{L}(T) \to \mathcal{L}(F)$ between their Tarski-Lindenbaum algebras preserving model-theoretic properties within the infinitary semantic layer *MQL* whose fundamental nature is established in [14]. In fact, a special set $X \subseteq \mathbb{N}$ is used in this construction presenting a parameterization for Stone space of the target theory. This set X plays the role of an oracle; thereby, the transformation related to infinitary first-order combinatorics represents, as a whole, is a common Turing computation (it is possible to say, computable Brute Force with an oracle).

Summarizing, we can say that, the combinatorial approach requires sophisticated definitions and is partially based on informal substantiation. Nevertheless, the concepts of finitary and infinitary first-order combinatorics adequately correspond to the posed class of problems; moreover, the informal argumentations and limitations are rather natural justifying the appropriateness of using the combinatorial terminology in this direction.

REFERENCES

[1] K. Bouvere, "Synonymous theories," The Theory of Models; Addison, Henkin and Tarski, Eds., North-Holland, Amsterdam, 1965, pp. 402-406.

[2] A. Church, "A note on Entscheidungsproblem," J. Symbolic Logic, vol. 1, no. 1, 1937, pp. 40–41; Correction: ibid, pp. 101–102.

[3] H. Gaifman, "Operations on relational structures, functors and classes. I," Proceedings of the Tarski Symposium, 1971, Proceedings of Symposia in Pure Mathematics, American Mathematical Society, Providence, R.I., First edition: 1974, Second edition: 1979, pp. 21-39.

[4] S. S. Goncharov and Y. L. Ershov, Constructive models, Plenum, New York, 1999.

[5] W. Hanf, "The Boolean algebra of Logic," Bull. American Math. Soc., vol. 31, 1975, pp. 587–589.

[6] W. Hodges, A shorter model theory, Cambridge University Press, Cambridge, 1997.

[7] S.C. Kleene, "Finite axiomatizability of theories in the predicate calculus using additional predicate symbol," Memories of American Math. Society, no. 10, 1952, pp. 27–68.

[8] K. Manders, "First-order logical systems and set-theoretic definability," Preprint, 1980.

[9] D. Myers, "An interpretive isomorphism between binary and ternary relations," Structures in Logic and Computer Science: A Selection of Essays in Honor of Andrzej Ehrenfeucht, Springer, 1997, pp. 84-105.

[10] M. G. Peretyat'kin, "Semantic universal classes of models," Algebra and Logic, 1991, vol. 30, no. 4, pp. 414–434.

[11] M. G. Peretyat'kin, "Semantic universality of theories over superlist," Algebra and Logic, 1992, vol. 30, no. 5, pp. 517–539.

[12] M. G. Peretyat'kin, Finitely axiomatizable theories, Plenum, New York, 1997.

[13] M. G. Peretyat'kin, "On the Tarski-Lindenbaum algebra of the class of all strongly constructivizable prime models," Proceedings of the Turing Centenary Conference CiE2012, Lecture notes in Computer Science, vol. 7318, Springer-Verlag: Berlin-Heidelberg, June 2012, pp. 589–598.

[14] M. G. Peretyat'kin, "Introduction in first-order combinatorics providing a conceptual framework for computation in predicate logic," Computation tools 2013, The Fourth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, IARIA, May–June 2013, pp. 31-36.

[15] M. G. Peretyat'kin, "The Tarski-Lindenbaum algebra of the class of all strongly constructivizable countable saturated models," P. Bonizzoni, V. Brattka, and B. Lowe Eds., Computability in Europe CiE 2013, Lecture notes in Computer Science, vol. 7921, Springer-Heidelberg, July 2013, pp. 342–352.

[16] C. Pinter, "Properties preserved under definitional equivalence and interpretations," Zeitschr. f. math. Logik und Grundlagen d. Math, 24, 1978, pp. 481-488.

[17] H. J. Rogers, Theory of recursive functions and effective computability, McGraw-Hill Book Co., New York, 1967.

[18] L. Szczerba, "Interpretability of elementary theories," Logic, Foundations of Mathematics and Computability Theory; Butts and Hintikka, Eds., D. Reidel Publishing Co., Dordrecht-Holland, 1977, pp. 129-145.

 ISBN: 978-1-61208-344-5

A Contextual Access Control Model for Online Social Network

Khalida Guesmia, Narhimene Boustia
Department of Computer Science
University of Saad Dahleb
Blida, Algeria
{khalida.guesmia@gmail.com nboustia@gmail.com}

***Abstract*— The sharing of personal and sensitive data has emerged as a popular activity over online social network. The availability of this information obviously raises privacy and confidentiality issues. The current access control models provided by online social network do not allow users to specify their access control on base of time, location, or under other circumstances. In this paper, we propose an access control model for social networks to express much more fine grained access control policies than the existing models, the OrBAC model is used because it provides a complete model to specify contextual and dynamic access control requirements. We also propose a logic specification of OrBAC with Temporal Logic of Actions.**

Keywords-Online social network; access control; OrBAC; TLA; context.

I. INTRODUCTION

In the last few years, Online Social Network sites (OSNs) have increasingly been used by more and more people around the world which have become integrated into the daily practices of millions of users [1]. OSNs are used to communicate with friends and family, to publish and to share different types of information with other members. Therefore, an unexpected large number of users and massive amount of data which is mainly representing a real life of the users are available in OSNs [2]. As a result, many challenges of scalability, management, and maintenance are posed in OSNs [3]. Cloud Computing paradigm emerges to face these challenges. Most of OSNs shift to Cloud Computing [4] by using different services models (Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS)) to support the huge number of users and their activities in OSNs. In the same time, it raises security and privacy concerns. It is important to review and define what we exactly mean by privacy in this context. Privacy means the right to self-determination regarding data disclosure [5]. Most OSNs provide access control system for users to configure their privacy settings by specifying who may access to their own information. In fact, the available protection settings are based primarily on relationship depth so almost of users expose their contents to more or less users than expected [6], which may lead to serious consequences in some cases [7]. Further, the dynamic developments of OSNs and the variety of data exists in social networks introduce new access control requirements for privacy management, which cannot be able to meet through the available privacy configuration. It is clear that users should be provided with more expressive and flexible mechanisms to protect their information from unwanted disclosure and unauthorized access. Defining access control policies in OSNs is a non-trivial task due to their large number of members and their connections and to the complexity of their environment. Thus, our objective in this paper is to propose appropriate access control model for Facebook that enables users to specify their privacy preferences in an expressive way without overburdening the users or the system [8].

The remainder of this paper is organized as follows: in Section II, we give briefly an overview of the social network Facebook, the most popular OSNs in the world. In Section III, we discuss some related work of access control in OSNs. In Section IV, we outline the main features of Organization Based Access Control model. In Section V, we define the Temporal Logic of Actions that we will be used to specify our proposed security policy for Facebook which be presented in Section VI. In Section VII, we show how to specify various contexts in Facebook using our formalism. In Section VIII, we apply our work to an example in Facebook, and finally, in Section IX, we summarize this paper with future works.

II. THE SOCIAL NETWORK FACEBOOK

Recently, the popularity of Facebook increased significantly. Facebook is a platform for users to interact with each other; according to statistics in December 2013, Facebook is the busiest site on the internet with more than 700 million daily active users around the world and it has built an extensive infrastructure to support this growth. When a user joins Facebook, he/she has to create a profile of himself/herself with biographical data, then sends and accepts invitations to add other users as friends. A user can directly communicate with his/her friends by messaging or poking, he/she can upload different types of information (photo, video, etc.) and share it with others; he/she can also join groups, like fan pages, and organize events [9]. All activities performed by a user are organized chronologically in his/her Timeline through which other users, as well as the user himself/herself, can check his/her past activities conveniently. A user receives his/her friend's updates on Newsfeed. When he/she finds something interesting, he/she can further perform actions, such as like, share and comment, on it. Facebook has expanded its development scope by adapting PaaS model. It opened up for third party application

 ISBN: 978-1-61208-344-5

by releasing its development Application Programming Interface (API) in May 2007 [10]. The third party applications bring value both to the platform and its users by providing new features. So, users can add these applications to their profiles and use them without having to install new hardware or software. These applications are deployed on their own servers and Facebook only acts as a proxy for integrating the application's output to its own pages. The third party applications require user's data to perform its functionality. For example, a simple horoscope application generates daily horoscope based on user's birthday. Furthermore, Facebook is a shared platform, used and managed by different entities (the provider, the third party application and the users). For that, users must carefully control what contents are visible to whom in order to preserve privacy. Therefore, users set their privacy preferences through an audience selector which supports only five modes (public, friends of friends, friends only, specific friends and only me). So, users cannot specify their access control on base of time, location, or under other circumstances. Therefore, the privacy sitting provided by Facebook is not expressive enough, it is limited somehow. In addition, users cannot control what others reveal about them such as tagging users in post, tag is option available in Facebook where users can simply tag other users by associating their profiles with post without their permission [11]. It should be noted that Facebook provider has full access to all user's personal data. Further, Facebook provide no privacy control against third party application. To overcome the limitations and challenges of privacy control in Facebook, the model Organization Based Access Control (OrBAC)[12] is expressive enough to specify access privacy based in various information and it supports in their policy different types of context, so OrBAC model is well suitable for Facebook [13].

III. RELATED WORK

Privacy is an emerging challenge in OSNs that caught much attention recently. There exist different research works that have examined different aspects of the privacy problem. Ajami et al. [7], it is confirmed that users have trouble with existing privacy controls, and they have difficulties to set their preferences. The traditional access controls models are not sufficiently flexible to specify the requirements of privacy management in OSNs [14]. Different access control models and mechanisms are proposed to support users when they set their privacy settings in OSNs. Abdessalem and BenDhia [15], it is proposed a reachability-based access control model that allows users to express their privacy preferences as constraints on existing links with other users. Wang et al. [16], it is developed an automated access control policy specification tool that helps ordinary users to specify who should have access to which part of their data. Oo [17], it is presented a fine-grained OSN access control model based on semantic web technologies in order to automatically construct access control rules for the user's privacy settings with the minimal effort from the user. However, many of these mechanisms provided solution for a certain privacy requirements but missed others. Ahmad and Whitworth [18], it is provided a distributed access control based on decentralised architecture for OSN instead the centralized architecture to avoid the single authority of the provider and that way, users have full control to manage their privacy control. It is good idea but it requires a lot of work. There is also considerable works [19] has been done in the area of access control in the Cloud Computing environment, which is completely challenging research problem and there is no complete solution for it.

IV. ORGANIZATION-BASED ACCESS CONTROL MODEL

The central entity in OrBAC model is the Organization. An organization may be viewed as any entity that has to manage a security policy. In our case study, Facebook itself corresponds to an organization. We can consider also user profile, fan page, group, application and event as organisations. There are always subject, object, action in access control model. In OrBAC model, a subject will be any active entity in a system that accesses objects. In Facebook, subject can be users (Alice, Bob, ...), application, etc. Object is any information or resource which can be accessed. For example, list of friends, photo, fan page, etc. Action is operations that subject are allowed to do on objects. For example, like, share, send message, etc. The idea of OrBAC model is to specify the security policy at the organizational level so instead of modeling the policy by using the concrete and implementation related concepts of subject, object and action, the OrBAC model suggests reasoning with their abstract concepts. The abstract concepts of subject, object and action are respectively role, view and activity. The concept of role in OrBAC is assigned to subjects with similar permissions. For example, in user profile, we can define admin, close friend, family, and colleague as roles.

If org is an organization, and r is a role, then Role_appropriate (org, r) means that role r is defined in organization org.

Role_Appropriate (user_profile, admin)

Role_Appropriate (user profile, colleague)

The concepts of view and activity are in the same way used in OrBAC model to respectively group objects and actions which similar permissions apply to, for example, the objects in user profile can be grouped in the following views: public data, limited data, and private data and for activity, we can define publishing as abstract of actions post, share, comment, and tag in Facebook.

If org is an organization, v is a view, then view_ appropriate (org, v) means that view v is defined in organization org.

View_Appropriate(user_profile, limited_data)

If org is an organization, a is an activity, then Activity_Appropriate (org, a) means that activity a is defined in organization org.

Activity_Appropriate(user_profile, publishing)

In OrBAC, specification of a security rule is not restricted to permissions, but also includes the possibility to specify prohibitions, obligations and recommendations. As we have mentioned before, Security rules in OrBAC model are specified with abstract entities as follows:

 ISBN: 978-1-61208-344-5

If org is an organization, r is a role, v is a view, a is an activity and c is a context then Permission(org, r, v, a, c) (resp. Prohibition(org, r, v, a, c), Obligation(org, r, v, a, c) or Recommendation (org, r, v, a, c)) means that organization org grants role r permission (resp. prohibition, obligation or dispensation) to perform activity a on view v within context c.

For instance, *Permission(Facebook, member, public_data, consulting, Default)*: "Facebook grants member permission to consult public data within the Default context". The Default context represents a condition which is always true.

To activate a given security rule, the subject, the object and the action must separately satisfy some conditions, these conditions are that the subject must be assigned to a given role, the object must be used in a given view and the action implements some given activity. This is represented by the following OrBAC relationships:

If org is an organization, s is a subject and r is a role, then Employ (org, s, r) means that org employs subject s in role r.

Employ (Facebook, Alice, member): "the role member is assigned to the user profile Alice in the Social Network Facebook".

Employ (Alice_Profile, Alice, admin): "Alice is admin in her own profile".

If org is an organization, o is an object and v is a view, then Use(org, o, v) means that org uses object o in view v.

Use (Fashion_Page, Pub.mp4, public_data): "the Page fan Fashion uses the video Pub.mp4 as a public data".

Use(Private_Group, Team.png ,limited_data): "the group Facebook Private uses photo Team.png as a limited data"

If org is an organization, α is an action and a is an activity, then Consider(org, α, a) means that org considers that action α implements the activity a.

Consider(Fcaebook, read, consulting): "in Facebook, we consider read as a consulting"

Consider(Alice_Profile, add_photo, publishing): "in Profile Alice, we consider add photo as a publishing"

Besides these conditions, there are extra conditions that must be satisfied to activate a security rule. These extra conditions may be related to very different notions, such as temporal or spatial requirements. We call context such extra conditions.

If org is an organization, s is a subject, o is an object, α is an action and c is a context, then Define(org, s, o, α, c) means that within organization org, context c is true between subject s, object o and action α. This issue will be detailed in section VII.

For Concrete level, security rules are specified with concrete entities as follows:

If s is a subject, o is an object and α is an action then Is_permitted(s, o, α) (resp. Is_prohibited(s, o, α), Is_obliged(s, o, α) and Is_recommended(s, o, α)) means that subject s is permitted (resp. prohibited, obliged, recommended) to perform action α on object o.

For instance, *Is_permitted(Alice, Pub.mp4, read)*: "Alice is permitted to read video Pub.mp4".

V. TEMPORAL LOGIC OF ACTIONS OVERVIEW

Generally, the choice of a formal language for specifying a security policy is based on the capabilities and richness of this language, and on the requirements of the targeted application. The Temporal Logic of Actions (TLA) is a powerful tool to specify systems and their properties, especially for interactive and concurrent systems. TLA combines two logics: a logic of actions and a standard temporal logic [20]. Variables, values, states, functions, predicate and actions are basic concepts in TLA. Values are elements of a data type. A variable has a name like *x* and *y*, and can be assigned a value. A constant is a variable that is assigned a fixed value. A state is characterized by assignment of a value *s[[x]]* to each variable *x*. A function is a nonboolean expression built from variables, operator symbols, and constants, such as $x^2 + y - 3$. The semantics *[[f]]* of a function f is a mapping from states to values. For example, $[[x^2 + y - 3]]$ is the mapping that assigns to the state s the value $s[[x]]^2+s[[y]]-3$, where *s[[x]]* and *s[[y]]* denote the values that *s* assigns to *x* and *y*. Generally, *s[[f]]* $\equiv f(\forall$ *'v': s[[v]]/v)* where f*(*$\forall$*'v': s[[v]]/v)* is the value obtained by substituting *s[[v]]* for each variable *v* in the expression. Semantically, a variable is also a function that assigns the value *s[[x]]* to the state *s*. A predicate is a boolean expression built from variables, operator symbols, and constants, such as $x = y+1$. The semantics *[[P]]* of a predicate *P* is a mapping from states to booleans. A state *s* satisfies a predicate *P* iff *s[[P]]*, the value of *[[P]]* in *s*, equals true. An action is a boolean valued expression formed from variables, primed variables, operator symbols, and constants. Formally, an action represents a relation between old states and new states, where unprimed variables refer to the old state and the primed variables refer to the new state. Formally, an action *A* is a function assigning a boolean *s[[A]]t* to a pair of states (*s, t*) , where *s* is the old state with unprimed variables, and *t* is the new state with primed variables. For example, $x' = y + 1$ has the boolean value of $t[[x]] = s[[y]] + 1$. We say that *(s, t)* is an *A* step if *s[[A]]t* equals true. Generally, *s[[A]]t* $\equiv A(\forall$ *'v': s[[v]]/v, t[[v]]/v')*. Since a predicate *P* is a boolean expression built from variables and constants, it is regarded as a special action without primed variables. A pair *(s,t)* is a *P* step iff *s[[P]]* is true. The basic temporal operator is □ (always). The semantics of a temporal action is defined using the concept of behavior. A behavior σ in TLA is an infinite sequence of states $< s_0, s1, s_2, \ldots >$ (a finite set of states can be regarded as infinite with identical repeating states).

$< s_0, s_1, s_2, \ldots >$ *[[A]]* $\equiv s_0$ *[[A]]* s_1

$< s_0, s_1, s_2, \ldots >$ *[[□A]]* $\equiv \forall n \geq 0: s_n$ *[[A]]* s_{n+1}

The same semantics can be defined for predicates since a predicate is a special form of action.

In TLA, a formula is built from predicates and actions with logical connectors and temporal operators.

VI. A SECURITY POLICY FOR ONLINE SOCIAL NETWORK SITE USING ORGANIZATION BASED ACCESS CONTROL MODEL

In this section, we present a logical approach for formalizing OrBAC adopted for Facebook. First we describe the basic components, and then we define the logic model of OrBAC with these components. A system state is a set of assignments of values to variables. In OrBAC, there are eight different kinds of entities, organization, subject, object, action, role, view, activity, and context. Each entity is specified by a finite set of attributes. We require that each entity has at least one attribute for identity, which is unique and cannot be changed. An attribute of an entity is denoted as **ent.att** where **ent** is the entity's identity and **att** is the attribute name. Hereafter, we assume that an entity name without any attribute specified denotes its identity. An attribute is a variable of a specific datatype, which includes a set of possible values, i.e., domain and operators to manipulate them. For example, the domain of attribute "gender" of entity "user profile" is {male, female}. The assignment of a value to an attribute is denoted by ent.att = value. We use *ent.att* to denote an attribute value. The constants correspond to the instances of the entities. A function is an expression built from one or more attributes and constants. For example, Alice_profile.age= Alice_profile .currentDate - Alice_profile.birthday. The variables, the functions, and the constants comprise the basic terms of our logical model. A predicate is a boolean expression built from variables, functions, and constants. A predicate can be defined with a number of attributes from a single entity, or two entities, or the system. In our model, predicate correspond to the relationships of OrBAC presented in Section IV and for the concrete permissions, prohibitions, obligations and recommendations that apply to subjects, objects and actions are represented as follows:

$\forall s \forall o \forall \alpha \forall r \forall v \forall a \forall c$

$Permission(org, r, v, a, c) \wedge$

$Employ(org, s, r) \wedge$

$Use(org, o, v) \wedge$

$Consider(org, \alpha, a) \wedge$

$Define(org, s, o, \alpha, c) \rightarrow Is\_permitted(s, o, \alpha).$

If organization *org*, within the context *c*, grants role *r* permission to perform activity *a* on view *v*, if *org* employs subject *s* in role *r*, if *org* uses object *o* in view *v*, if *org* considers that action α implements the activity *a* and if, within *org*, the context *c* is true between *s*, *o* and α then *s* has permission to perform α on *o*.

VII. SPECIFYING CONTEXT IN ONLINE SOCIAL NETWORK

Different contexts may be expressed within OrBAC model [21]. In order to show the expressiveness of our proposed model, we design several scenarios and give their corresponding formulas in our logic.

The temporal context depends on the time at which the subject is requesting for an access to the system, it should be possible to express that a given action made by a given user on a given object is authorized only at a given time/date, after or before a given time/date, or during a given time interval. To validate a given request for an access, it is necessary to be able to evaluate the current time/date, we suppose each organization have a clock. For example, the admin of group Library create new poll to choose best author for 2013.The pool is open to only members of group until 20/12/2013.

$Define(Library\_Group, s, Best\_Author\_2013, select, Before\_date\_31/01/2014) \rightarrow Employ(Library\_Group, s, member) \wedge Library\_Group.currentTime <= 20/12/2013.$

The spatial context depends on the subject location. Knowing the location from where the user makes the request can be useful to specify the access control policy. We can distinguish two different types of spatial context. The physical spatial context and the logical spatial context. The first one corresponds to the physical location of the user, namely his or her office, a security area, a specific building, the country, etc. The logical spatial context corresponds to the logical location he or she stands in. For example, it can be the computer, the network or the sub-network, the smartphone, etc. In some cases, physical and logical spatial contexts are highly correlated. The network IP address from which a user is connected probably corresponds to a specific physical place such as a department area. For example, we can specify that the participation in Marathon event is allowed only to users who are connected from the same country where the Marathon will be held.

$Define(Marathon\_Event, s, Page\_Event, join, connected\_country) \rightarrow s.\ connected\_country = Marathon\_Event.\ country.$

The prerequisite context, the permission is granted to a subject, only if some specific conditions are satisfied. For example, Bob want to share his video "How to root Samsung Galaxy S3" with other members who are not necessarily their friends but they search how to root Samsung smartphone.

$Define(Bob\_Profile, s, How\_to\_root\_Samsung\_Galaxy\_S3.mp4, share, Hashtag\_Video) \rightarrow s.search \in (rootsamsung, rootsmartphone, samsungGalaxyS3).$

The provisional context depends on previous actions the subject has performed in the system. For example, Alice wants to specify that when she adds new friend, this latter is permitted to consult her Timeline from the moment that became her friend.

$Define(Alice\_Profile, s, Timeline, read, Adding\_New\_Freind) \rightarrow Alice\_Profile.currentTime >= Alice\_Profile.dateBeFreindWith(s)$

VIII. EXAMPLE OF A SECURITY POLICY IN FACEBOOK

In this section, we show how security policy of profile user in Facebook can be expressed and deducing in our formalism. Alice in her profile Facebook defines Mary as close friend, John as member of family, Elena, Mike,Paul as friends and she specifies that her friends who work at the same work place as her, they have colleague as role in her profile. She likes page fan Zinedine Zidane, she joins group Photoshop Club, and she adds Puzzle Game as application. In our formalism, this is represented by the following instances:

 ISBN: 978-1-61208-344-5

- Subjects and Roles

Role_appropriate (Alice_Profile, family)
Role_appropriate (Alice_Profile, close_friend)
Role_appropriate (Alice_Profile, colleague)
Role_appropriate (Alice_Profile, friend)
Employ(Alice_Profile, s, colleague) →
Employ(Alice_Profile, s, friend) ∧
Alice_Profile.workplace=s. workplace.
Employ(Alice_Profile, John, family)
Employ(Alice_Profile,Mary, close_friend)
Employ(Alice_Profile, Elena, friend)
Employ(Alice_Profile, Mike, friend)
Employ(Alice_Profile,Zinedine_Zidane,page)
Employ(Alice_Profile, Photoshop_Club,group)
Employ(Alice_Profile,Puzzle_Game, application)

- Objects and Views

Alice defines the following views and objects in her profile:
View_appropriate(Alice_Profile, limited_data)
View_appropriate(Alice_Profile, private_data)
View_appropriate(Alice_Profile, public_data)
Use(Alice_Profile, List_of_ friends , private_data)
Use(Alice_Profile, Birthday, private_data)
Use(Alice_Profile, Joke , limited_data)
Use(Alice_Profile,Gender, public_data)

- Actions and Activities

Alice defines the following actions and activities in her profile:
Activity_appropriate(Alice_Profile, publishing),
Activity_appropriate(Alice_Profile, adding_friend),
Activity_appropriate(Alice_Profile, consulting),
Consider(Alice_Profile, post,publishing),
Consider(Alice_Profile, send_invitation ,adding_friend),
Consider(Alice_Profile,read, consulting).

- Hierarchies

In OrBAC, organizations, roles, views, activities can be organized hierarchically. *Sub_Role(Profile, family , friend).* The role family inherits the permissions from the role friend.

- Context

Alice wants to share joke with her *colleagues* but only with women.

Define(Alice_Profile, s, Joke, read,
Only_Women_Colleague) → Employ(Alice_Profile, s,
colleague) ∧ s.Gender=Female.

- Security policy

Alice specifies the following permissions:
Permission(Alice_Profile, friend, limited_data, consulting ,
Only_Women_Colleague)
Prohibition (Alice_Profile, friend, public_data, tagging ,
*Default):*Alice don't want to be tagged in public post.

Elena and Mike work at the same company as Alice and they want to access to joke posted by Alice. On the basis of the specified access policies defined by Alice, the system determines whether access should be granted or denied. First for Elena, we have:

Permission(Alice_Profile, friend, limited_data, consulting ,
Only_Women_Colleague) ∧
Employ(Alice_Profile, Elena, friend) ∧
Alice_Profile.workplace= Elena. workplace ∧
Employ(Alice_Profile, Elena, colleague) ∧
Use(Alice_Profile, Joke , limited_data) ∧
Consider(Alice_Profile, read, consulting) ∧
Elena.Gender=Female ∧
Define(Alice_Profile, Elena, Joke, read,
Only_Women_Colleague) → Is_permitted(Elena, Joke,
read).
So Elena is permitted to read Alice's joke.
For Mike, we have:
Permission(Alice_Profile, friend, limited_data, consulting ,
Only_Women_Colleague) ∧
Employ(Alice_Profile, Mike, friend) ∧
Alice_Profile.workplace= Mike. workplace ∧
Employ(Alice_Profile, Mike, colleague) ∧
Use(Alice_Profile, Joke , limited_data) ∧
Consider(Alice_Profile, read, consulting) ∧
Mike.Gender=Female !

The context (Only_Women_Colleague) is not satisfied for Mike so Mike is not permitted to read Alice's joke.

IX. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a contextual access control model for users to manage access to their data in Facebook with a flexible and effective way. Using our model, users can specify their privacy sittings based in various information and they can configure access control to users, as well as applications so, it give more control to the users. We have also developed a logic specification of OrBAC for Facebook with Temporal Logic of Actions. We are currently working in developing our solution to perform a user study in order to analyse the decidability and the performance of our model in real case and we will plane to develop configuration interface for users to easily specify their privacy preferences based on our proposed model.

REFERENCES

[1] M. S. Ezaleila and H. Azizah, "Online social networking: a new virtual playground", International Proceedings of Economics Development & Research, 2011, vol. 5, issue 2, pp. 314-318.

[2] J. Becker and H. Chen**,** "Measuring privacy risk in online social networks", Web 2.0 Security and Privacy Workshop, 2009.

[3] P. Dudi, "Cloud computing and social networks: a comparison study of myspace and facebook", Journal of Global Research in Computer Science, March 2013, vol. 4, no. 3, pp. 51-54.

[4] B. Yang, W. Tsai, A. Chen, and S. Ramandeep," Cloud computing architecture for social computing - a comparison study of Facebook and Google", International Conference on Advances in Social Networks Analysis and Mining, Kaohsiung, 2011, pp. 741–745.

[5] R. Iannella and A. Finden, "A privacy awareness: icons and expression for social networks", in 8th International Workshop for Technical, Economic and Legal Aspects of

Business Models for Virtual Goods Incorporating the 6th International ODRL Workshop, Namur, Belgium, 2010.

[6] Y. Liu , K. P. Gummadi, B. Krishnamurthy, and A. Mislove, "Analyzing facebook privacy settings:user expectations vs. reality", Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference IMC'11, Berlin, Germany, November 2-4, 2011, pp. 61-70.

[7] R. Ajami, N. Ramadan, N. Mohamed, and J. Al-Jaroodi, "Security challenges and approaches in online social networks: a survey", International Journal of Computer Science and Network Security IJCSNS, August 2011, vol. 11, no. 8, pp. 1-12.

[8] M. Beye, A. Jeckmans, Z. Erkin, P. Hartel, R. Lagendijk, and Q. Tang, " Literature overview - privacy in online social networks". Technical report, Centre for Telematics and Information Technology, 2010.

[9] J. Pang and Y. Zhang, "A new access control scheme for facebook-style social networks", 2013.

[10] K. Singh , S. Bhola , and W. Lee, "xBook: redesigning privacy control in social networking platforms", Proceedings of the 18th conference on USENIX security symposium, Montreal, Canada, August 10-14, 2009, pp. 249-266.

[11] M. Madejski, M. Johnson, and S. M. Bellovin. "The failure of online social network privacy settings". Technical Report CUCS-010-11, Columbia University, Feb. 2011.

[12] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte,A. Miège, C. Saurel, and G. Trouessin. "Organization Based Access Control". In 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy, June 2003.

[13] F. Cuppens,N. Cuppens-Boulahia, and E. P. Vina, "Adaptive access control enforcement in social network using aspect weaving", Proceedings,17th International Conference, DASFAA 2012, International Workshops: FlashDB, ITEMS, SNSM, SIM3, DQDI, Busan, South Korea, April 15-19, 2012, pp. 154-167.

[14] A. Ahmad and B. Whitworth, "Future directions in access control for online social networks", International Conference on Networks and Information ICNI, Bangkok, Thailand. November 24-25, 2012.

[15] T. Abdessalem and I. BenDhia. "a Reachability-Based Access Control Model for Online Social Networks". In Proceedings of the First ACM SIGMOD Workshop on Databases and Social Networks, DBSocial' 11, Athens, Greece, June 12-16, 2011, pp. 31-36.

[16] T. Wang , M. Srivatsa , and L. Liu, "Fine-grained access control of personal data", Proceedings of the 17th ACM symposium on Access Control Models and Technologies, Newark, New Jersey, USA, June 20-22, 2012, pp. 145-156.

[17] S. H. P. Oo, "Intelligent access control policies for social network site", International Journal of Computer Science & Information Technology (IJCSIT), June 2013, vol. 5, no. 3, pp. 183-190.

[18] A. Ahmad and B. Whitworth, "Distributed access control for social newtorks", in 7th International Conference on Information Assurance And Security, Malaysia, 2011, pp. 68-73.

[19] J.M.A. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray, "Toward a Multi-Tenancy Authorization System for Cloud Services", Security & Privacy, IEEE, Nov. 2010, vol. 8 , issue 6, pp. 48-55.

[20] X. Zhang , J. Park , F. Parisi-Presicce ,and R. Sandhu, "A logical specification for usage control", Proceedings of the ninth ACM symposium on Access control models and technologies, Yorktown Heights, New York, USA, June 02-04, 2004, pp. 1-10.

[21] F. Cuppens and N. Cuppens-Boulahia "Modelling Contextual Security Policies",International Journal of Information Security, 2008, vol. 7, issue 4 , pp. 285-305.

 ISBN: 978-1-61208-344-5

First Steps towards Automated Synthesis of Tableau Systems for Interval Temporal Logics

Dario Della Monica*, Angelo Montanari†, Guido Sciavicco‡ and Dmitri Tishkovsky§
* ICE-TCS, School of Computer Science, Reykjavik University, Iceland
† Department of Mathematics and Computer Science, University of Udine, Italy
‡ Department of Information Engineering and Communications, University of Murcia, Spain
§ School of Computer Science, University of Manchester

dariodm@ru.is, angelo.montanari@uniud.it, guido@um.es, dmitry@cs.man.ac.uk

***Abstract*—Interval temporal logics are difficult to deal with in many respects. In the last years, various meaningful fragments of Halpern and Shoham's modal logic of time intervals have been shown to be decidable with complexities that range from NP-complete to non-primitive recursive. However, even restricting the attention to finite interval structures, the step from model-theoretic decidability results to the actual implementations of tableau-based decision procedures is quite challenging. In this paper, we investigate the possibility of making use of automated tableau generators. More precisely, we exploit the generator METTEL^2 to implement a tableau-based decision procedure for the future fragment of the logic of temporal neighborhood over finite linear orders. We explore and contrast two alternative solutions: a *concrete* tableau system, that operates on a concrete interval structure explicitly built over a finite, linearly-ordered set of points, and an *abstract* one, that operates on an interval frame which is forced to be isomorphic to a concrete interval structure by suitably constraining its accessibility relation.**

Keywords–Interval temporal logics; satisfiability; tableau systems; automated tableau system generation.

I. INTRODUCTION

In this paper, we make some initial steps towards the automated synthesis of tableau systems for interval temporal logics. It is well-known that turning (optimal) declarative, tableau-based systems for decidable temporal logics into effective decision procedures is far from being trivial. Such a transition turns out to be particularly complex in the case of interval temporal logics. In the last years, it has been experimented for two specific logics, namely, the temporal logic of sub-intervals D, interpreted over dense linear orders [1], and the future fragment of the logic of temporal neighborhood A, interpreted over finite linear orders [2]. However, in both cases the proposed solution is tailored to the logic under consideration, and thus it lacks generality. In this paper, we explore the possibility of exploiting a general tool for the automated synthesis of tableau systems, namely, the generator METTEL^2, to deal with interval temporal logics. Even though we will apply the proposed solution to the logic A only (as Bresolin et al. did in [2]), there is no any limitation that prevents its application to other interval temporal logics.

Propositional interval temporal logics play a significant role in computer science, as they provide a natural framework for representing and reasoning about temporal properties in a number of application domains [3]. This is the case, for instance, of computational linguistics, where significant interval-based logical formalisms have been developed to represent and reason about tenses and temporal prepositions [4]. As another example, the possibility of encoding and reasoning about various constructs of imperative programming in interval temporal logic has been systematically explored by Moszkowski in [5]. Other meaningful applications of interval temporal logics can be found in knowledge representation, systems for temporal planning and maintenance, qualitative reasoning, theories of action and change, specification and design of hardware components, concurrent real-time processes, event modeling, and temporal databases. Modalities of interval temporal logics correspond to binary relations between time intervals. In particular, Halpern and Shoham's modal logic of time intervals HS [6] features one modality for each Allen interval relation [7]. In [6], the authors showed that HS is undecidable over all meaningful classes of linear orders. Since then, a lot of work has been devoted to the study of HS fragments, mainly to disclose their computational properties and relative expressiveness. The classification of HS fragments with respect to the status (decidable/undecidable) of their satisfiability problem is now almost completed. In this paper, we focus our attention on the class of finite linear orders, which comes into play in a variety of application domains, e.g., in planning problems. A complete classification of HS fragments over finite linear orders is given in [8]. It shows that there are 62 non-equivalent (with respect to expressiveness) decidable HS fragments, which can be partitioned into four complexity classes, ranging from NP-complete to non-primitive recursive. For each decidable fragment, an optimal, tableau-based decision procedure has been devised. However, since each of such procedures has been given a declarative formulation, no one of them is available as a working system, apart from the tableau-based decision procedure for the fragment A reported in [2]. The only attempt to apply a generic theorem prover to an interval temporal logic can be found in [1], where a tableau-based decision procedure for the fragment D, interpreted over dense linear orders, has been developed in LoTREC [9][10]. LoTREC is a generic prover for modal and description logics that can be used to prove validity and satisfiability of formulas. Whenever a formula is satisfiable, it returns a model for it; whenever a formula is not valid, it returns a counter-model for it. In LoTREC, a tableau is a special kind of labeled graph that is built, and possibly revised, according to a set of user-defined rules. Every node of the graph is labeled with

 ISBN: 978-1-61208-344-5

a set of formulae and can be enriched by auxiliary markings, if needed. Unfortunately, LoTREC, as well as most generic theorem provers, cannot be exploited to deal with other interval temporal logics because (i) it does not support an explicit treatment of world labels, and (ii) it manages closing conditions based on loop checks, but it does not allow explicit checks on the number of worlds generated during the construction of a tentative model. Such limitations are overcome by the current version of METTEL^2 [11], which provides the user with a flexible language for specifying propositional syntaxes and tableau calculi.

In the following, we make use of METTEL^2 to implement a tableau-based decision procedure for A over finite linear orders. We explore and contrast two alternative solutions: a *concrete* tableau system, that operates on a concrete interval structure explicitly built over a finite, linearly-ordered set of points, and an *abstract* one, that operates on an interval frame which is forced to be isomorphic to a concrete interval structure by suitably constraining its accessibility relation (using the specification language provided by METTEL^2). The main contributions of the paper can be summarized as follows: *(i)* it can be viewed as the first general attempt of using an automated generator to synthesize a tableau system for an interval temporal logic(D over dense linear orders is a very special case because, due to its properties, it bears strong resemblance to standard modal logic); *(ii)* while METTEL^2 works perfectly on a variety of other logics (see, e.g., [12] and Section III), it required a small, but not trivial, change to make it possible to formulate closing conditions for A; *(iii)* the abstract version of the tableau system, based on a suitable representation theorem, gives new insigths into the role of temporal knowledge representation and reasoning technique, and representation theorems [7][13][14].

The paper is structured as follows. In the next section, we introduce the logic A. In Section III, we provide an necessary overview of the system METTEL^2. In Section IV, we describe the proposed A-prover. Section V given an account of the experimental results. Section VI concludes the paper.

II. THE INTERVAL TEMPORAL LOGIC A

Given a linearly ordered set $\mathbb{D} = \langle D, < \rangle$, a (strict) *interval* is a pair $[a, b]$, with $a, b \in D$ and $a < b$. There are 12 different relations (excluding the identity) between two intervals on a linear order, often referred to as *Allen's relations* [7]: the six relations depicted in Fig. 1, namely $R_A, R_L, R_B, R_E, R_D, R_O$, and the inverse ones, defined in the standard way, that is, $R_{\overline{X}} = (R_X)^{-1}$, for each $X \in \{A, L, B, E, D, O\}$. Intuitively, an interval structure over a linear order $\mathbb{D}$ consists of the set of all intervals over $\mathbb{D}$, together with a set of Allen's relations. We treat interval structures as Kripke structures [15], where Allen's relations play the role of accessibility relations, and we associate a modality $\langle X \rangle$ with each Allen relation R_X. Given a modality $\langle X \rangle$ associated with the relation R_X, with $X \in \{A, L, B, E, D, O\}$, its *transpose* is the modality $\langle \overline{X} \rangle$, corresponding to the inverse relation $R_{\overline{X}}$.

Syntax and (Concrete) Semantics. Halpern and Shoham's logic HS [6] is a multi-modal logic with formulae built from a finite, non-empty set $\mathcal{AP}$ of atomic propositions, the propositional connectives $\vee$ and $\neg$, and the complete set of modalities associated with all Allen's relations. With each subset $\{R_{X_1}, \ldots, R_{X_k}\}$ of this set of relations, we associate the fragment $\mathsf{X_1X_2 \ldots X_k}$ of HS, whose formulae are defined by the grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle X_1 \rangle \varphi \mid \ldots \mid \langle X_k \rangle \varphi, \text{ with } p \in \mathcal{AP}.$$

The other propositional connectives and logical constants, e.g., $\wedge$, $\rightarrow$, and $\top$, can be derived in the standard way, as well as the dual modalities, e.g., $[A]\varphi \equiv \neg\langle A \rangle \neg\varphi$. In this paper, we focus our attention on the fragment A, whose formulae are generated by the following restricted grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle A \rangle \varphi, \text{ with } p \in \mathcal{AP}.$$

The concrete semantics of HS is given in terms of *concrete interval models*.

Definition 1: Let $\mathbb{D}$ be a linearly ordered set and $\mathbb{I}(\mathbb{D})$ be the set of all (strict) intervals over $\mathbb{D}$ (called *concrete interval structure*). A *concrete interval model* is a pair $M = \langle \mathbb{I}(\mathbb{D}), V \rangle$, where V is a *valuation function* $V : \mathcal{AP} \rightarrow 2^{\mathbb{I}(\mathbb{D})}$ that assigns to every atomic proposition $p \in \mathcal{AP}$ the set of intervals $V(p)$ on which p holds.

The *truth* of a formula is defined with respect to a concrete interval model M and an interval $[a, b]$ on it by structural induction on formulae as follows:

- $M, [a, b] \Vdash p$ iff $[a, b] \in V(p)$, for each $p \in \mathcal{AP}$;
- $M, [a, b] \Vdash \neg\psi$ iff it is not the case that $M, [a, b] \Vdash \psi$;
- $M, [a, b] \Vdash \varphi \vee \psi$ iff $M, [a, b] \Vdash \varphi$ or $M, [a, b] \Vdash \psi$;
- $M, [a, b] \Vdash \langle X \rangle \psi$ iff there is an interval $[c, d]$ such that $[a, b] R_X [c, d]$ and $M, [c, d] \Vdash \psi$, for each modality $\langle X \rangle$.

In the case of modality $\langle A \rangle$, the last semantic clause can be instantiated as follows:

$$M, [a, b] \Vdash \langle A \rangle \varphi \text{ iff there is } c > b \text{ such that } M, [b, c] \Vdash \varphi.$$

Formulae of HS can be interpreted in various interesting classes of concrete interval models, depending on the specific class of linear orders over which the models are built. As for the class of (concrete interval models built over) finite linear orders, the following small model theorem holds [16].

Theorem 1: Let φ be an A-formula. Then, φ is finitely satisfiable if and only if it is satisfiable on a model whose domain has cardinality strictly less than $2^m \cdot m + 1$, where m is the number of diamonds and boxes in φ.

The above result immediately provides a termination condition that can be used to implement a *fair* procedure that exhaustively searches for a model of size smaller than the bound.

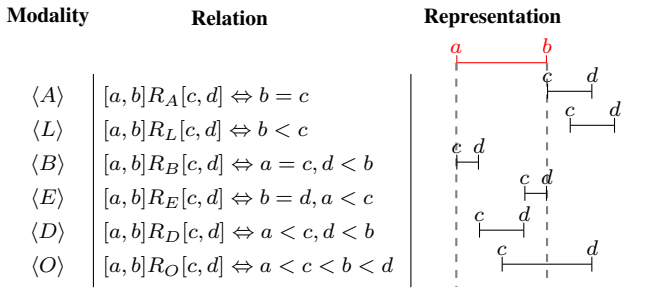

Figure 1. Allen's interval relations and the corresponding HS modalities.

Abstract Semantics. As we already pointed out, METTEL^2 is flexible enough to allow one to provide an alternative, *abstract* version of the tableau system for A, based on a different, but equivalent, set of semantic conditions. To this end, we first define a suitable class of interval frames for A, called finite abstract interval A-structures, whose distinctive features are expressed by a set of first-order conditions, and then we show that any such frame is isomorphic to a concrete interval structure. It is worth noticing that such an abstract semantics, that takes intervals as first-class citizens, is quite common in the field of interval temporal logics, but not in those of modal and point-based temporal logics. In AI, the coexistence of concrete and abstract interval structures is well known since the early stages of interval-based temporal reasoning. The variety of binary relations between intervals in a linear order was first studied systematically by Allen, Hayes, and Ferguson [7][13][14], who explored their use in systems for time management and planning. The work by Allen and colleagues was based on the assumption that time can be represented as a dense line, and that points are excluded from the semantics. Both Allen and Hayes [17] and van Benthem [18] showed that interval temporal reasoning can be formalized as an extension of first-order logic with equality with one or more relations. As pointed out in [19], the characteristics of the proposed formalizations depend on basic choices about fundamental semantic parameters, such as the class of linear orders on which the interval structure is based (all dense linear orders, the rational numbers, etc.), and the set of interval relations added to the first-order language.

Given the dual nature of time intervals, that can be represented either as ordered pairs of time points over a linear order or as suitably-constrained, first-order individual objects, *representation theorems* have an important role in interval temporal logics. They can be described as follows (with respect to a specific class of linear orders). Given an extension of first-order logic with a set of interval relations, such as, for instance, $\{meets, during\}$, is there a set of axioms which constrain abstract models in this signature to be isomorphic to concrete ones? The problem can be alternatively stated as follows: can we define an isomorphism into concrete models whose domain is the set of intervals over the considered linear order and whose relations are the concrete interval relations? A number of representation theorems for interval logics can be found in the literature, including van Benthem [18], who considers the order of rational numbers and the interval relations *during* and *before*; Allen and Hayes [17], which refer to unbounded, dense linear orders, devoid of point intervals, and to the interval relation *meets* only; Ladkin [20], who takes into consideration point-based temporal structures with a 4-argument relation that encodes the interval relation *meets*; Venema [21], who considers dense linear orders with the interval relations *starts* and *finishes*; Goranko, Montanari, and Sciavicco [22], which deal with dense linear orders with the interval relations *meets* and *met-by*; and Coetzee [23], who refers to dense linear orders with the interval relations *overlaps* and *meets*.

In the present work, we focus our attention on the class of finite linear orders and the interval relation *meets* (denoted by R_A), and we provide a representation theorem that forces any finite, suitably-constrained Kripke frame $\langle W, R_A\rangle$ to be isomorphic to a finite, concrete interval structure. As a matter of fact, some frame conditions will be explicitly forced by introducing specific first-order constrains (this is the case with irreflexivity, antisymmetry, composition, and linearity); other ones will be embedded into the definition of the tableau rules (this is the case with finiteness and connectedness).

Definition 2: Let W be a finite nonempty set and let $R_A \subseteq W \times W$ be such that for all $x, y \in W$, $x = y$ or xR_Ay or $xR_{\overline{A}}y$ or xR_Ly or $xR_{\overline{L}}y$, and so on (connectedness)[1]. The pair $\mathfrak{S} = \langle W, R_A\rangle$ is a *finite and connected, abstract interval* A*-structure* if and only if the following conditions are satisfied:

1) $\forall x \neg(xR_Ax)$ (irreflexivity);
2) $\forall x, y(xR_Ay \wedge yR_Ax \rightarrow x = y)$ (antisymmetry);
3) $\forall x, y(xR_Ay \rightarrow \exists z(\forall t(tR_Az \leftrightarrow tR_Ax) \wedge \forall t(zR_At \leftrightarrow yR_At)))$ (composition);
4) $\forall x, y, z, t((xR_Ay \wedge yR_At \wedge xR_Az \wedge zR_At) \rightarrow y = z)$ (linearity).

The next representation theorem shows that the above conditions suffice to force any finite and connected, abstract interval A-structure to be isomorphic to a finite concrete one. For the sake of readability, we introduce the relation R_A as an additional component of concrete interval structures, that is, we substitute $S = \langle \mathbb{I}(\mathbb{D}), R_A\rangle$ for $\mathbb{I}(\mathbb{D})$, Proving that any finite, concrete interval structure satisfies conditions 1–4, as well as connectedness, is trivial; proving that any finite and connected, abstract interval A-structure is isomorphic to a finite, concrete interval structure is definitely more involved. Such a result is formally stated by the following theorem, whose proof is omitted for space reasons.

Theorem 2: Every finite and connected, abstract interval A-structure is isomorphic to a finite, concrete interval structure.

Thanks to Theorem 2, we can interpret the logic A on finite and connected, abstract interval A-structures. To this end, we adapt the notion of model for A by defining it as a pair $M = \langle \mathfrak{S}, V\rangle$, where $\mathfrak{S}$ is a finite and connected, abstract interval A-structure and $V : \mathcal{AP} \mapsto 2^W$. Moreover, we accordingly revise the semantic clause for $\langle A\rangle$ as follows:

$M, i \Vdash \langle A\rangle\psi$ iff there is j such that iR_Aj and $M, j \Vdash \psi$.

In the following, we will show that one actually needs to explicitly encode conditions 1–4 only. As for finiteness, it can be forced by imposing a suitable cardinality constraint, that is, by providing an interval counterpart (that applies to $\mathfrak{S}$) of the constraint coming from Theorem 1 (that applies to concrete models). As for connectedness, it is guaraanteed by construction: all generated world are directly or indirectly connected to the initial one (no incomparable world is ever introduced).

III. AUTOMATED SYNTHESIS OF TABLEAU CALCULI AND METTEL^2

Tableau reasoning methods are powerful tools to reason about logical formalisms. They have been extensively used to develop decision procedures for description and modal logics [24][25], as well as for intuitionistic logics, conditional logics, metric and topological logics, and hybrid logics.

[1] In [17], Allen and Hayes showed that all Allen's relations are first-order definable in terms of the interval relation R_A (*meets*) only. As a matter of fact, the proof assumes the temporal domain to be dense and unbounded; however, it can be shown that such an assumption is not necessary.

 ISBN: 978-1-61208-344-5

Schmidt and Tishkovsky [26] devise a method for automatically generating tableau calculi from a first-order specification of the formal semantics of a logic. The idea is that of turning such a specification into a set of inference rules giving rise to a sound, complete, and terminating deduction calculus for the logic, provided that the logic has the finite model property.

The tableau synthesis method works as follows [26]. The user defines the formal semantics of the given logic in a many-sorted first-order language so that certain well-definedness conditions hold. The semantic specification of the logic is then automatically reduced to Skolemised implicational forms, which are subsequently transformed into tableau inference rules. Combined with a set of default closure and equality rules, the generated rules provide a sound and complete calculus for the logic. Under certain conditions, the generated set of rules can be further refined [27]. If the logic has the finite model property, the generated calculus can be automatically turned into a terminating calculus by adding a suitable blocking mechanism.

The tableau prover generator MetTeL^2 [11] has been implemented to complement the theoretical tableau synthesis framework given in [26]. MetTeL^2 produces Java code of a tableau prover from specifications of a logical syntax and a tableau calculus for a given logic. It aims at providing an easy-to-use system for non-technical users and it allows technical users to improve/extend the implementation of generated provers. MetTeL^2 has been successfully employed to produce tableau provers for modal logics, description logics, epistemic logics, and temporal logics with cardinality constraints. It is worth pointing out that prior implementations of systems for automated synthesis of tableau calculi already existed. Among them, we would like to mention LoTREC [9], [10] and The Tableau Work Bench (TWB) [28], which are the prover engineering platforms most closely related to MetTeL^2. Although MetTeL^2 does not give the user the same possibilities for programming and controlling derivations as these systems, its specification language is more expressive. As an example, Skolem terms are allowed both in premises and conclusions of rules. The expressive specification language also allows one to specify the syntax of arbitrary propositional logics and it makes MetTeL^2 able to deal with the interval temporal logic A (which we focus on in this paper) and possibly with most of the other fragments of HS.

IV. Tableau Provers for A

In this section, we describe the specifications of two tableau provers for the logic A, which are based on the concrete and the abstract semantics, respectively.

The steps for obtaining the specifications are common to both provers. They can be summarized as follows. First, we apply the tableau synthesis framework [26] to the semantics of A. Since both concrete and abstract semantics for A consist of connective definitions and the background theory, the well-definedness conditions given in [26] are trivially fulfilled for both of them. Therefore, the generated calculi are automatically sound and (constructively) complete for the logic A. Next, we apply the atomic refinement [27] to the rules of the obtained calculi by moving negated atomic formulae in the rule conclusions to its premises while changing their signs. While retaining soundness and (constructive) completeness of the calculi, this reduces branching factor of the rules and makes tableau algorithms based on the calculi more efficient. Finally, we extend the tableau languages with additional constructs which replace the first-order predicates in the original calculi. This further simplifies the calculi, making them more readable and specifiable in MetTeL^2.

The tableau specifications for the concrete and abstract semantics of A in MetTeL^2 specification language are listed in Fig. 2. The symbol / separates premises of a rule from its conclusions and the symbol || separates branches of the rule. A priority value is assigned to each rule with the keyword *priority*. The less the value the more eagerly the rule is applied during derivation.

The tableau specification for the concrete semantics of A is based on two logical sorts: the sort of points and the sort of logical formulae. Disjunction $p \vee q$ is represented in the specification as p|q, negation $\neg p$ is represented as ~p, and <A> represents the modal operator $\langle A \rangle$. Constructs which extend the language of the logic are the ordering predicate $<$ on the sort of points ($a < b$ is represented as {a<b}), the equality predicate ({{a=b}} stands for $a = b$), a Skolem function f, to generate new terms of the sort of points, and expressions of the form $[a, b] : \varphi$, which are formulae φ of A labeled by intervals $[a, b]$, where a and b are points. The rules at lines 1–8 of the concrete tableau enforce $<$ to be a strict linear ordering. The rule at line 10 ensures that all the intervals are not degenerative. The remaining rules are standard rules for modal-like logics. It is worth pointing out that the rules at lines 1–8 and at line 15 are obtained by atomic refinement from the rules generated by the tableau synthesis framework. As an example, the rule [a,b]:~(<A>p) {b < c} / [b,c]:~p is obtained by the refinement from the generated rule [a,b]:~(<A>p) / ~{b < c} || [b,c]:~p. As a consequence of the results in [27], the calculus is sound and (constructively) complete for the standard interval semantics of the fragment A.

The tableau specification for the abstract semantics is also based on two sorts: the sort of intervals and the sort of logical formulae. The additional constructs are two Skolem functions f and g, the equality predicate, and a binary relational symbol R on the sort of intervals (for the sake of simplicity, we use R for R_A). The tableau operates on labeled formulae $@_i\varphi$ (@i p in the specification), where φ is a formula of A and i is an interval. The lines 1–7 of the abstract tableau define the theory of the relation R and correspond to the conditions 1–4 in Definition 2. While the rest of the rules are similar to standard rules for modal-like logics and they can be specified in tableau development platforms like LoTREC and TWB, the four rules listed at lines 3–6 are special. All the four rules make use of the same Skolem function g; moreover, the rules at lines 3 and 5 have the Skolem function g in their premises. Allowing specifications of tableau rules where Skolem functions occur in the rule premises is a distinctive feature of MetTeL^2 prover generator, which demonstrate the expressiveness of the MetTeL^2 specification language. Similarly to the case of the concrete tableau, the rules at lines 1–7 and at line 13 are obtained by atomic refinement. Therefore, the calculus is sound and (constructively) complete for the relational semantics of the fragment A.

Termination of both provers is achieved by a modification

 ISBN: 978-1-61208-344-5

```
{a < a} / priority 0;
{a < b} {b < c} / {a < c} priority 3;
{a < b} {c < d} /
      {{c = a}} || {c < a} || {a < c} {c < b} ||
      {{c = b}} || {b < c} priority 7;
{a < b} {c < d} /
      {{d = a}} || {d < a} || {a < d} {d < b} ||
      {{d = b}} || {b < d} priority 7;
[a,b]:p [a,b]:~p / priority 0;
[a,b]:p / {a < b} priority 1;
[a,b]:~(~p) / [a,b]:p priority 1;
[a,b]:(p|q) / [a,b]:p || [a,b]:q priority 5;
[a,b]:~(p|q) / [a,b]:~p [a,b]:~q priority 3;
[a,b]:<A>p / [b,f(b,p)]:p priority 9;
[a,b]:~(<A>p) {b < c} / [b,c]:~p priority 4;
```

```
R i i / priority 0;
R i j R j i / priority 0;
R i j R k g(i,j) / R k i priority 4;
R i j R k i / R k g(i,j) priority 10;
R i j R g(i,j) k / R j k priority 4;
R i j R j k / R g(i,j) k priority 10;
R i j R j k R i l R l k / {{ j = l }} priority 6;
@i p @i ~p / priority 0;
@i ~(~p) / @i p priority 1;
@i (p|q) / @i p || @i q priority 5;
@i ~(p|q) / @i ~p @i ~q priority 3;
@i <A>p / R i f(i,p) @f(i,p) p priority 9;
@i ~(<A>p) R i j / @j ~p priority 4;
```

Figure 2. Tableau specifications for concrete (left) and abstract (right) semantics.

to the generated Java code to ignore branches which exceed the allowed limit of points or intervals (see Theorem 1).

V. Testing and Results

We have tested our implementations against the same benchmark of problems used in [2], although the absolute speed results cannot be immediately compared since the two experiments used a different hardware. These problems are divided into two classes. First, we tested the scalability of the implementation with respect to a set of combinatorial problems of increasing complexity (COMBINATORICS), where the n-th combinatorial problem is defined as the problem of finding a model for a formula that contains n conjuncts, each one of the form $\langle A\rangle p_i$ $(0 \leq i \leq n)$, plus $\frac{n(n+1)}{2}$ conjuncts of the form $[A]\neg(p_i \wedge p_j)$, with $i \neq j$. (Notice that there are $n(n+1)$ different conjuncts of the pointed out form. However, a conjunct with indices i, j is equivalent to another one with indices j, i. This is why $\frac{n(n+1)}{2}$ is posed.) Then, we considered the set of 72 purely randomized formulas used in [29] to evaluate an evolutionary algorithm for the same fragment (RANDOMIZED). Table I summarizes the outcomes of the experiments. For each class of problems, the corresponding table shows, for each instance n, the time (in milliseconds) necessary to solve the problem taking into account, when appropriate, the specific policy that has been used. In particular, the concrete version has been run under both the 'breadth first' and the 'depth first' (left branch first) policies. A time-out of 1 minute was used to stop instances running for too long.

At first sight, the relational (abstract) version of the tableau system looks more (time) efficient than the standard (concrete) one. However, the number of instances that generated a memory error indicates that the latter uses less memory, which can be considered an interesting result on its own. All the experiments were executed on Java 1.7.0_25 OpenJDK 64-Bit Server VM under the Java heap size limit of 3Gb on a hardware based on Intel® Core™ i7-880 CPU (3.07GHz, 8Mb), with a total memory of 8Gb (1333MHz), under the 64-bit Fedora Linux 17 operating system.

VI. Conclusions and Future Work

In this paper, we illustrated the outcomes of a first experiment in automated generation of tableau-based decision procedures for interval temporal logics using the automatic prover generator MetTeL^2. Thanks to its expressive power and flexibility, we explored and contrasted two alternative implementations: a concrete and an abstract one (at the best of our knowledge, this is the first tableau-based decision procedure for interval temporal logics based on an abstract frame semantics). Even though the performance of the developed systems is not particularly exciting, the use of generators like MetTeL^2 provides a general and effective way of implementing tableau systems for interval temporal logics. We believe it possible to make the concrete tableau system more efficient, provided that we represent the linear order by a list of points. This would remedy the exponential blow-up of inequality formulae in the tableau derivation, but, unfortunately, lists cannot be represented in the language of MetTeL^2 yet. The addition of such a feature to MetTeL^2 and the analysis of its actual impact are left for future work. As for the abstract tableau system, in principle, it allows us to compare alternative, but equivalent, formulations of the first-order constraints for a given fragment. Last but not least, we are going to validate the proposed approach on other, more expressive HS fragments.

Acknowledgements

The authors acknowledge the support from the Spanish fellowship program *'Ramon y Cajal' RYC-2011-07821* (G. Sciavicco), the projects *Processes and Modal Logics* (project nr. 100048021) and *Decidability and Expressiveness for Interval Temporal Logics* (project nr. 130802-051) of the Icelandic Research Fund (D. Della Monica), the Italian GNCS project *Automata, games, and temporal logics for verification and synthesis of controllers in safety-critical systems* (A. Montanari), and the research grant EP/H043748/1 of the UK EPSRC (D. Tishkovsky).

References

[1] D. Bresolin, V. Goranko, A. Montanari, and P. Sala, "Tableaux for logics of subinterval structures over dense orderings," J. of Logic and Computation, vol. 20, no. 1, 2010, pp. 133–166.

[2] D. Bresolin, D. Della Monica, A. Montanari, and G. Sciavicco, "A tableau system for Right Propositional Neighborhood Logic over finite linear orders: an implementation," in Proc. of the 22nd TABLEAUX, ser. LNCS, vol. 8123, 2013, pp. 74–80.

[3] V. Goranko, A. Montanari, and G. Sciavicco, "A road map of interval temporal logics and duration calculi," J. of Applied Non-Classical Logics, vol. 14, no. 1–2, 2004, pp. 9–54.

 ISBN: 978-1-61208-344-5

Table I. EXPERIMENTAL RESULTS (IN MILLISECONDS; '-': "OUT OF TIME"; 'M': "OUT OF MEMORY"; 'Y': "SATISFIABLE"; 'N': "UNSATISFIABLE").

COMBINATORICS

	CON		ABS	
n	DF	BF		sat
1	10	10	0	y
2	60	100	0	y
3	270	420	10	y
4	920	1360	30	y
5	2930	4010	70	y

	CON		ABS	
n	DF	BF		sat
6	7890	9850	150	y
7	19420	23670	300	y
8	47220	51220	560	y
9	-	-	1000	y
10	-	-	1790	y

	CON		ABS	
n	DF	BF		sat
11	-	-	3440	y
12	-	-	4660	y
13	-	-	7600	y
14	-	-	11560	y
15	-	-	17170	y

	CON		ABS	
n	DF	BF		sat
16	-	-	25160	y
17	-	-	35610	y
18	-	-	50740	y
19	-	-	-	-
20	-	-	-	-

RANDOMIZED

	CON		ABS	
n	DF	BF		sat
1	-	-	-	-
2	0	0	0	y
3	10	0	0	y
4	0	10	0	y
5	-	-	-	-
6	0	10	0	y
7	-	-	-	-
8	10	10	0	y
9	20	20	10	y
10	10	10	0	y
11	-	-	-	-
12	10	10	0	y
13	10	10	0	y
14	10	10	0	y
15	-	-	-	-
16	10	20	0	y
17	30	50	10	y
18	-	-	-	-

	CON		ABS	
n	DF	BF		sat
19	30	50	0	y
20	-	-	-	-
21	20	50	10	y
22	-	-	-	-
23	-	-	-	-
24	20	20	0	y
25	-	-	-	-
26	-	-	-	-
27	-	-	-	-
28	-	-	-	-
29	-	-	-	-
30	-	-	-	-
31	10	10	10	n
32	-	-	-	-
33	-	-	M	-
34	60	70	10	y
35	-	-	-	-
36	-	-	-	-

	CON		ABS	
n	DF	BF		sat
37	-	-	M	-
38	-	-	M	-
39	-	-	M	-
40	-	-	M	-
41	-	-	-	-
42	-	-	-	-
43	-	-	-	-
44	-	-	-	-
45	-	-	M	-
46	-	-	-	-
47	-	-	-	-
48	-	-	-	-
49	-	-	-	-
50	-	-	M	-
51	-	-	M	-
52	-	-	-	-
53	-	-	M	-
54	-	-	-	-

	CON		ABS	
n	DF	BF		sat
55	-	-	M	-
56	-	-	M	-
57	-	-	M	-
58	-	-	-	-
59	-	-	M	-
60	-	-	M	-
61	-	M	M	-
62	-	-	-	-
63	M	-	-	-
64	-	-	-	-
65	-	-	M	-
66	-	-	-	-
67	M	-	-	-
68	-	-	-	-
69	M	-	-	-
70	-	-	M	-
71	M	M	M	-
72	-	-	-	-

[4] I. Pratt-Hartmann, "Temporal prepositions and their logic," Artificial Intelligence, vol. 166, no. 1-2, 2005, pp. 1–36.

[5] B. Moszkowski, "Reasoning about digital circuits," Tech. Rep. STAN-CS-83-970, Dept. of Computer Science, Stanford University, Stanford, CA, 1983.

[6] J. Halpern and Y. Shoham, "A propositional modal logic of time intervals," J. of the ACM, vol. 38, no. 4, 1991, pp. 935–962.

[7] J. Allen, "Maintaining knowledge about temporal intervals," Communications of the ACM, vol. 26, no. 11, 1983, pp. 832–843.

[8] D. Bresolin, D. Della Monica, A. Montanari, P. Sala, and G. Sciavicco, "Interval temporal logics over finite linear orders: the complete picture," in Proc. of the 20th ECAI, 2012, pp. 199–204.

[9] F. del Cerro et al., "LoTREC: the generic tableau prover for modal and description logics," in Proc. of the 1st IJCAR, ser. LNCS, vol. 2083. Springer, 2001, pp. 453–458.

[10] O. Gasquet, A. Herzig, D. Longin, and M. Sahade, "LoTREC: Logical Tableaux Research Engineering Companion," in Proc. of the 14th TABLEAUX, ser. LNCS, vol. 3702, 2005, pp. 318–322.

[11] D. Tishkovsky, R. A. Schmidt, and M. Khodadadi, "The tableau prover generator MetTeL^2," in Proc. of the 13th JELIA, 2012, pp. 492–495.

[12] M. Khodadadi, R. A. Schmidt, D. Tishkovsky, and M. Zawidzki, "Terminating tableau calculi for modal logic K with global counting operators," 2012, technical report. Available at http://www.mettel-prover.org/papers/KEn12.pdf.

[13] P. J. Hayes and J. F. Allen, "Short time periods," in Proc. of the 10th IJCAI, Milano, Italy, 1987, pp. 981–983.

[14] J. F. Allen and G. Ferguson, "Actions and events in interval temporal logic," J. Log. Comput., vol. 4, no. 5, 1994, pp. 531–579.

[15] P. Blackburn, M. de Rijke, and Y. Venema, Modal Logic. Cambridge University Press, 2002.

[16] D. Bresolin, A. Montanari, and G. Sciavicco, "An optimal decision procedure for Right Propositional Neighborhood Logic," J. of Automated Reasoning, vol. 38, no. 1-3, 2007, pp. 173–199.

[17] J. F. Allen and P. J. Hayes, "A common-sense theory of time," in Proc. of the 9th IJCAI, Los Angeles, CA, USA, 1985, pp. 528–531.

[18] J. Benthem, The Logic of Time, 2nd ed. Kluwer Academic Press, 1991.

[19] W. Conradie and G. Sciavicco, "On the expressive power of first order logic extended with allenÕs relations in the strict case," in Proc. of the 14th CAEPIA, ser. LNAI, vol. 7023. Springer, 2011, pp. 173–182.

[20] P. Ladkin, "The logic of time representation," Ph.D. dissertation, University of California, Berkeley, 1987.

[21] Y. Venema, "A modal logic for chopping intervals," Journal of Logic and Computation, vol. 1, no. 4, 1991, pp. 453–476.

[22] V. Goranko, A. Montanari, and G. Sciavicco, "Propositional interval neighborhood temporal logics," J. of Universal Computer Science, vol. 9, no. 9, 2003, pp. 1137–1167.

[23] C. J. Coetzee, "Representation theorems for classes of interval structures," Master's thesis, Department of Mathematics, University of Johannesburg, 2009.

[24] F. Baader and U. Sattler, "An overview of tableau algorithms for description logics," Studia Logica, vol. 69, no. 1, 2001, pp. 5–40.

[25] R. Goré, "Tableau methods for modal and temporal logics," in Handbook of Tableau Methods. Springer Netherlands, 1999, pp. 297–396.

[26] R. A. Schmidt and D. Tishkovsky, "Automated synthesis of tableau calculi," Logical Methods in Computer Science, vol. 7, no. 2:6, 2011, pp. 1–32. [Online]. Available: http://arxiv.org/abs/1104.4131

[27] D. Tishkovsky and R. A. Schmidt, "Refinement in the tableau synthesis framework," CoRR, vol. abs/1305.3131, 2013.

[28] P. Abate and R. Goré, "The Tableau Workbench," Electronic Notes in Theoretical Computer Science, vol. 231, 2009, pp. 55 – 67.

[29] D. Bresolin, F. Jiménez, G. Sánchez, and G. Sciavicco, "Finite satisfiability of propositional interval logic formulas with multi-objective evolutionary algorithms," in Proc. of the 12th FOGA, 2013, pp. 25–36.

 ISBN: 978-1-61208-344-5

Semi-Automated Task Planning in Metric Propositional Interval Neighborhood Logic

Laura González-García
Polytechnic University of Cartagena
Cartagena, Spain
Email: lgg2@alu.uptc.es

Guido Sciavicco
Dep. of Information Engineering and Communications
University of Murcia, Murcia, Spain
Email: guido@um.es

***Abstract*—Planning is the process of thinking about and organizing the activities required to achieve a desired goal. It involves the creation and maintenance of a plan. As such, planning is a fundamental property of intelligent behaviour. This process is essential to the creation and refinement of a plan, or integration of it with other plans. In logistics planning, which is a fundamental part of every engineering projects, planning is a usually handcrafted activity, often supported by high-level commercially available software. These techniques are error-prone as they relay on the expertise of the responsible engineer, and suffer of limited reasoning capabilities, being usually based on temporal constraint networks in Allen's style. Recently, interval temporal logics have been studied that allow one to describe temporal situations at a higher level, and yet with a decidable satisfiability problem. We propose here the use of Metric Interval Temporal Neighbourhood Logic, a decidable fragment of Halpern and Shoham's Modal Logic for Time Intervals (HS), as a tool for task planning. The main characteristics of this proposal is that the language, whose syntax heavily restricts HS, and whose proposed applications so far have been limited to theoretical and abstract situations, is still expressive enough to cope with the complexity of a realistic case study.**

Keywords-Automated Planning; Interval Temporal Logics.

I. INTRODUCTION

Task management is the process of managing tasks through its life cycle. It involves planning, testing, tracking and reporting. Task management can help either individuals achieve goals, or groups of individuals collaborate and share knowledge for the accomplishment of collective goals [1]. Tasks are also differentiated by complexity, from low to high. Effective task management requires managing all aspects of a task, including its status, priority, time, human and financial resources assignments, recurrence, notifications and so on. These can be lumped together broadly into the basic activities of task management. Managing multiple individual or team tasks may require specialised task management software. Specific software dimensions support common task management activities. These dimensions exist across software products and services and fit different task management initiatives in a number of ways. In fact, many people believe that task management should serve as a foundation for project management activities. Task management may form part of project management and process management and can serve as the foundation for efficient work-flow in an organisation. Project managers adhering to task-oriented management have a detailed and up-to-date project schedule, and are usually good at directing team members and moving the project forward.

As a discipline, task management embraces several key activities. Various conceptual breakdowns exist, and these, at a high-level, always include creative, functional, project, performance and service activities. *Creative* activities pertain to task creation. These should allow for task planning, brainstorming, creation, elaboration, clarification, organization, reduction, targeting and preliminary prioritization. *Functional* activities pertain to personnel, sales, quality or other management areas, for the ultimate purpose of ensuring production of final goods and services for delivery to customers. These should allow for planning, reporting, tracking, prioritizing, configuring, delegating, and managing of tasks. *Project* activities pertain to planning and time/costs reporting. These can encompass multiple functional activities but are always greater and more purposeful than the sum of its parts. Project activities should allow for project task breakdown, task allocation, inventory across projects, and concurrent access to task databases. *Service* activities pertain to client and internal company services provision, including customer relationship management and knowledge management. These should allow for file attachment and links to tasks, document management, access rights management, inventory of client and employee records, orders and calls management, and annotating tasks. *Performance* activities pertain to tracking performance and fulfillment of assigned tasks. Finally, *report* activities pertain to the presentation of information regarding the other five activities listed, including graphical display.

Task management software tools abound in the marketplace [2][3]. Some are free; others exist for enterprise-wide deployment purposes. Some boast enterprise-wide task creation, visualization and notifications capabilities - among others - scalable to smaller, medium and bigger size companies, from individual projects to ongoing corporate task management. Project management and calendaring software also often provide task management software with advanced support for task management activities and corresponding software environment dimensions, reciprocating the myriad project and performance activities built into most good enterprise-level task management software products. Nevertheless, most of such software lack truly intelligent capabilities, as they are based on *algebraic* networks in Allen's style [4][5]. The main limits of algebraic, constraint-based reasoning, compared to *logical* reasoning are discussed in [6], and include the fact that algebraic networks: *(i)* are purely existential, and do not allow,

 ISBN: 978-1-61208-344-5

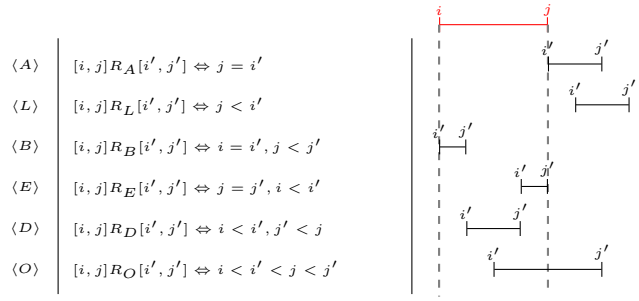

Figure 1: Allen's interval relations and the corresponding HS modalities.

in general, the specification of universal properties; *(ii)* do not allow, in general, the specification of negative information; *(iii)* are not designed to easily integrate and/or compare two or more plans.

The fact that constraint-based networks are usually preferred over logic-based reasoning systems in planning languages and software is explained by the usually bad computational behaviour of interval-based temporal logics. Interval temporal logics provide a natural framework for temporal representation and reasoning on interval structures over linearly ordered domains. They take time intervals as the primitive ontological entities and define truth of formulae with respect to them instead of to time instants. Modal operators of interval temporal logics correspond to binary relations between pairs of intervals In the realm of interval temporal logics, a prominent role is accorded to Halpern and Shoham's modal logic of time intervals (HS) [7], whose modalities make it possible to express all Allen's binary interval relations. Unfortunately, most of them, including HS and the majority of its fragments, turn out to be undecidable (a comprehensive survey on interval logics can be found in [8]; more recent contributions include [9][10]). Focusing our attention to the class of models built on the set of the integers, in [10] it has been shown that there exists exactly 44 expressively different fragments of HS with a decidable satisfiability problem, with complexities from NP-complete to EXPSPACE-complete, and 62 are decidable in the finite case [11]. Among these, a metric extension of the fragment that features $\langle A \rangle$ and $\langle \overline{A} \rangle$ only (known as $\mathsf{A\overline{A}}$ or PNL) has been developed by Bresolin et al. in [12]. The resulting interval temporal logic, called Metric PNL (MPNL for short), pairs PNL modalities with a family of special proposition letters expressing integer constraints (equalities and inequalities) on the length of the intervals over which they are evaluated. The authors show that the satisfiability problem for MPNL, interpreted over finite linear orders and the natural numbers, is decidable, and, in particular, EXPSPACE-complete.

In this paper, we propose the use of MPNL as a task planning reasoning tool. In the next section, we provide the necessary preliminaries on MPNL and interval temporal logics in general. In Section 3, we consider the problem of representing a plan in MPNL, and in Section 4 we apply our technique to a practical case-study, before concluding.

II. PRELIMINARIES

Let $\mathbb{D} = \langle D, < \rangle$ be a linearly ordered set. An *interval* over $\mathbb{D}$ is an ordered pair $[i, j]$, where $i, j \in D$ and $i < j$ (*strict semantics*). There are 12 different non-trivial ordering relations (excluding equality) between any pair of intervals in a linear order, often called *Allen's relations* [4]: the six relations depicted in Figure **??** and the inverse ones. We interpret interval structures as Kripke structures and Allen's relations as accessibility relations, thus associating a modality $\langle X \rangle$ with each Allen's relation R_X. For each operator $\langle X \rangle$, its *inverse* (or *transpose*), denoted by $\langle \overline{X} \rangle$, corresponds to the inverse relation $R_{\overline{X}}$ of R_X (that is, $R_{\overline{X}} = (R_X)^{-1}$). Halpern and Shoham's logic HS is a multi-modal logic with formulas built on a set $\mathcal{AP}$ of proposition letters, the boolean connectives $\vee$ and $\neg$, and a modality for each Allen's relation. We denote by $\mathsf{X_1} \ldots \mathsf{X_k}$ the fragment of HS featuring a modality for each Allen's relation in the subset $\{R_{X_1}, \ldots, R_{X_k}\}$. Formulas of $\mathsf{X_1} \ldots \mathsf{X_k}$ are defined by the grammar:

$$\varphi ::= p \mid \neg\psi \mid \psi \vee \tau \mid \langle X_1 \rangle \psi \mid \ldots \mid \langle X_k \rangle \psi,$$

where $p \in \mathcal{AP}$ is a propositional letter. The other boolean connectives can be viewed as abbreviations, and the dual operators $[X]$ are defined, as usual, as $[X]\varphi \equiv \neg\langle X \rangle \neg p$. The semantics of HS is given in terms of *interval models* $\mathcal{M} = \langle \mathbb{I}(\mathbb{D}), \mathcal{V} \rangle$, where $\mathbb{I}(\mathbb{D})$ is the set of all intervals over $\mathbb{D}$ and $\mathcal{V} : \mathcal{AP} \mapsto 2^{\mathbb{I}(\mathbb{D})}$ is a *valuation function* that assigns to every $p \in \mathcal{AP}$ the set of intervals $\mathcal{V}(p)$ over which p holds. The *truth* of a formula over a given interval $[i, j]$ in an interval model $\mathcal{M}$ is defined by structural induction on formulas:

$\mathcal{M}, [i,j] \Vdash p$	iff	$[i,j] \in \mathcal{V}(p)$
$\mathcal{M}, [i,j] \Vdash \neg\psi$	iff	$\mathcal{M}, [i,j] \nVdash \psi$
$\mathcal{M}, [i,j] \Vdash \psi \wedge \tau$	iff	$\mathcal{M}, [i,j] \Vdash \psi$ and $\mathcal{M}, [i,j] \Vdash \varphi$
$\mathcal{M}, [i,j] \Vdash \langle X_k \rangle \psi$	iff	$\mathcal{M}, [i',j'] \Vdash \psi$ for some $[i,j] R_{X_k} [i',j']$.

Formulae of HS can be interpreted over a class of interval models (built on a given class of linear orders). Among others, we mention the following important classes of (interval models built on important classes of) linear orders: *(i)* the class of *all* linear orders; *(ii)* the class of (all) *dense* linear orders, that is, those in which for every pair of distinct points there exists at least one point in between them; *(iii)* the class of (all) *discrete* linear orders, that is, those in which every element, apart from the greatest element, if it exists, has an immediate successor, and every element, other than the least element, if it exists, has an immediate predecessor *(iv)* the class of (all) *finite* linear orders, that is, those having only finitely many points. In the recent years, a great effort has been devoted to the study of decidability of fragments of HS. Ever since HS was introduced, it was immediately clear that its satisfiability problem is undecidable when interpreted on every interesting class of linearly ordered sets [7], including all of the above mentioned ones. While this sweeping result initially discouraged further research in this direction, recent results showed that the situation is slightly better then it seemed. Given the set of HS modalities that correspond to the set of Allen's relations $\{R_{X_1}, \ldots, R_{X_k}\}$, we call *fragment* $\mathcal{F} = \mathsf{X_1 X_2} \ldots \mathsf{X_n}$ any subset of such modalities, displayed in alphabetical order. There are 2^{12} such fragments. Some of these are expressively equivalent to each other; in [9] (respectively, [13]) it is possible

 ISBN: 978-1-61208-344-5

to find all possible inter-definability in the class of all linearly ordered sets (respectively, all dense linearly ordered sets), giving rise to 1347 (respectively, 966) expressively different fragments. The number of different fragments on other classes of linear orders has not been determined yet, but it is believed that the situation in the finite or discrete case should be similar. Out of these fragments, it has been possible to prove that exactly 62 are decidable in the finite case [11], and 44 in the (strongly) discrete case (and in the case of $\mathbb{Z}$) [10], all of which with complexities that range from NP-complete (in very simple cases) to NEXPTIME-complete, EXPSPACE-complete, to non-primitive recursive.

Motivated by the (potential) applicability of these logics, in the discrete and the finite case the possibility of adding *length constraints* has been studied. Following [14] we introduce a set of pre-interpreted atomic propositions referring to the length of the current interval. Given a *distance* function $\delta : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$, defined as $\delta(i, j) = |i - j|$, for each $\sim \in \{<, \leq, =, \geq, >\}$, we introduce the length constraint $\mathsf{len}_{\sim \mathsf{k}}$, with the following semantics:

$$\mathcal{M}, [i, j] \Vdash \mathsf{len}_{\sim \mathsf{k}} \text{ iff } \delta(i, j) \sim k.$$

As studied in [14][12], the language of $\mathrm{A}\overline{\mathrm{A}}$ can be extended with length constraints when interpreted over $\mathbb{N}$, $\mathbb{Z}$, or finite models without loosing the decidability of the fragment itself; its complexity, though, worsen from NEXPTIME to EXPSPACE. Equality and inequality constraints are mutually definable, although there is a increase in formula length if we consider, for example, only constraints of form $\mathsf{len}_{=\mathsf{k}}$ as primitive. Length constraints can be expressed in HS in a direct way. The simplest way to achieve this is to make use of $\langle B \rangle$ or $\langle E \rangle$. For example, under the discreteness hypothesis, we have that:

$$\mathcal{M}, [i, j] \Vdash \mathsf{len}_{=\mathsf{k}} \text{ iff } \langle B \rangle^{k-1} \top \wedge [B]^k \bot,$$

which proves that $\mathrm{A}\overline{\mathrm{A}}$ plus metric constraints (known as MPNL) is a proper fragment of HS.

Finding an optimal balance between expressive power and computational complexity is a challenge for every knowledge representation and reasoning formalism. Interval temporal logics are not an exception in this respect; in [14] the applicability of MPNL has been advocated. To recall some of the arguments, MPNL has been proved expressive enough to to encode (*metric versions* of) basic operators of point-based linear temporal logic (LTL) as well as interval modalities corresponding to Allen's relations. In addition, it allows one to express limited forms of fuzziness. Limiting ourselves to a few examples, we show that MPNL is expressive enough to encode the strict *sometimes in the future* (respectively, *sometimes in the past*) operator of LTL:

$$\langle A \rangle (\mathsf{len}_{>0} \wedge \langle A \rangle (\mathsf{len}_{=0} \wedge p))$$

Moreover, length constraints allow one to define a metric version of the *until* (respectively, *since*) operator. For instance, the condition: '*p is true at a point in the future at distance k from the current interval and, until that point, q is true (pointwise)*' can be expressed as follows:

$$\langle A \rangle (\mathsf{len}_{=\mathsf{k}} \wedge \langle A \rangle (\mathsf{len}_{=0} \wedge p)) \wedge [A](\mathsf{len}_{<\mathsf{k}} \to \langle A \rangle (\mathsf{len}_{=0} \wedge q)).$$

MPNL can also be used to constrain interval length and to express metric versions of basic interval relations. First, we can constrain the length of the intervals over which a given property holds to be at least (respectively, at most, exactly) k. As an example, the following formula constrains p to hold only over intervals of length l, with $k \leq l \leq k'$:

$$[G](p \to \mathsf{len}_{\geq \mathsf{k}} \wedge \mathsf{len}_{\leq \mathsf{k}'}) \quad (bl)$$

where the *universal modality* $[G]$ (*for all intervals*) is expressible in the language, and its corresponding formula depends on the class of models over which the formula is interpreted. By exploiting such a capability, metric versions of almost all Allen's relations can be expressed (the only exception is the *during* relation). As an example, we can state that: '*p holds only over intervals of length l, with $k \leq l \leq k'$, and any p-interval begins a q-interval*' as follows:

$$(bl) \wedge [G] \bigwedge_{i=k}^{k'} (p \wedge \mathsf{len}_{=\mathsf{i}} \to \langle \overline{A} \rangle \langle A \rangle (\mathsf{len}_{>\mathsf{i}} \wedge q)).$$

Finally, MPNL makes it possible to express some forms of 'fuzziness'. As an example, the condition: '*p is true over the current interval and q is true over some interval close to it*', where by 'close' we mean that the right endpoint of the p-interval is at distance at most k from the left endpoint of the q-interval, can be expressed as follows:

$$p \wedge (\langle A \rangle \langle \overline{A} \rangle (\mathsf{len}_{<\mathsf{k}} \wedge \langle \overline{A} \rangle \langle A \rangle q) \vee \langle A \rangle (\mathsf{len}_{<\mathsf{k}} \wedge \langle A \rangle q)).$$

III. Task Planning in MPNL

It is generally accepted that *task planning* in Engineering is a fundamental phase of the design, organization, and control of any realistic work organization plan. In its simplest version, it includes, at least:

1) A list of each *atomic* task, along with its properties (including its temporal duration);
2) A set of *precedence* relations among tasks.

The purpose of a systematic organization of such set of task is to answer the following question: *Is the plan possible, and, if so, what is its minimal temporal duration?* In view of these considerations, we may define a plan as follows.

Definition 1: A *plan* is a finite collections of *tasks*, each one of which with a finite and univocally determined *duration*, and such that they are placed over a finite temporal line respecting a finite collection of *precedence requirement*.

The typical practical approach to the problem of finding a plan from the collection of its requisite is twofold. On the one side, engineers are trained to organize tasks in a systematic network of precedence (for example, with the so-called *critical path method* [15]), and to compute (by hand) the viability of the entire network. On the other side, commercially available software, such as, for example, Microsoft Project© are used to aid this process. Now, it is easy to observe that:

1) Atomic tasks can be logically treated as propositional letters;
2) The precedence relation can be modeled as Allen's relation *meets*;

3) On a dicrete/finite temporal line, durations are exactly *length constraints*.

These considerations allow us to conclude that MPNL is a suitable logical counterpart of a task planning network. Plans are, by definition, temporally finite, and by seeing the plan the logical conjunction of the (formulas corresponding to) set of all constraints, *plan viability* corresponds to (MPNL-) *formula satisfiability*. Moreover, the notion of *minimal duration* is precisely the notion of *minimal model* of a formula.

From now on, we consider the language of MPNL interpreted in the class of *finite* (and, therefore, discrete) models. Satisfiability of MPNL-formulas in the finite case can be safely restricted to the *initial interval*, which we can denote as $[-1, 0]$. In fact, given any MPNL-formula φ, the latter is satisfiable if and only if the MPNL-formula $\langle A\rangle\varphi$ is initially satisfiable. By applying such a small technical modification, we obtain a task, represented as a propositional letter, will be placed on the interval $[i, j]$ exactly when the plan sets it to start at the moment i and to finish at the moment j. In this context, the *universal modalitity*, introduced in the previous section, can be expressed as follows:

$$[G]p \equiv [A]p \wedge [A][A]p.$$

Let us assume, now, that tasks are represented by the propositional letters $T_1, T_2, \ldots \in \mathcal{T}$, where $\mathcal{T}$ is a finite set of tasks. Similarly, it is convenient to assume that task indexes are collected in a subset of natural numbers $\mathcal{I}$. Therefore, by expressing:

$$\langle A\rangle T_l \vee \langle A\rangle\langle A\rangle T_l,$$

where $l \in \mathcal{I}$, we force the task T_1 to be part of a plan. Similarly, by adding:

$$[G](T_l \rightarrow \mathsf{len}_{=\mathsf{k}}),$$

we force T_l to have the duration of k units. The *precedence relation* can be then expressed as follows. Given a constraint of the type: *the task T_l cannot start before the task T_g has finished*, we can set:

$$[G](T_l \rightarrow \langle\overline{A}\rangle(T_g \vee \langle\overline{A}\rangle T_g).$$

To make sure that we can exclude unwanted models, we can add the following constraint that ensures tasks are unique:

$$\bigwedge_{l\in\mathcal{I}} [G]\langle A\rangle(T_l \rightarrow [A]\neg T_l).$$

Unlike algebraic networks, MPNL allows one to express more complex requirements. First of all, besides single tasks $T_1, T_2, \ldots$, we can express the concept of *task type*, by adding suitable propositions in conjunction with those that denote tasks, and, then, impose *universal* constraints over them. Suppose, for example, that $\mathcal{T}' \subset \mathcal{T}$ collects all and only those tasks of a certain type, for which we have the constraint that: *between any two successive tasks of $\mathcal{T}'$ a temporal distance of at least k units must get by*. We can deal with such a constraint by means of the following technique:

$$\begin{cases} \bigwedge_{T_l\in\mathcal{T}'}[G](T_l \rightarrow P)\wedge \\ [G](P \rightarrow \bigvee_{T_l\in\mathcal{T}'})\wedge \\ [G](P \wedge \langle A\rangle\langle A\rangle P \rightarrow \langle A\rangle(\mathsf{len}_{>\mathsf{k}-1}P)), \end{cases}$$

where by means of the first formula, we make sure that elements of $\mathcal{T}'$ are labeled by an additional proposition P, by the second one we guarantee that P labels only elements of $\mathcal{T}'$, and, finally, by last one we introduce the temporal constraint.

Other types of constraints can be expressed, such as: *the tasks T_l and T_g cannot start at the same time*:

$$[G]((\langle A\rangle T_l \wedge \langle A\rangle T_g) \rightarrow \bot).$$

Finally, it is worth noticing that more complex constraints can be expressed in MPNL. In fact, we can easily identify the *maximal temporal duration* $\overline{k}$ of any task in $\mathcal{T}$; by using this information, as we have explained in Section 2, almost every Allen's relation can be expressed over bounded intervals (all tasks are bounded), and they can be used in both existential and universal statements.

In more advanced task planning systems, one would like to be able to take into account a certain amount of *finite resources*. Indeed, in real cases, not every temporally sound plan is executable, if it requires, at any given moment of time, more resources than those that are at disposition. It is not difficult to see that this information can be expressed by using only a propositional language, such as MPNL. Let us assume that we measure our resources with a natural number n. The requirements may indicate the resources that are consumed by each task, and the total number of resources that are at the disposition for the entire plan. We assume, for each task T_l, that the propositional letter R^n_l denotes the fact that n resource units are necessary; clearly, if N is the maximum number of units that are necessary for any task, at most $|\mathcal{T}|\cdot N$ different propositional letters must be added to the language. We then have to assign the correct number of units to each task, and, since we can only compare intervals, in order to take into account *overlapping* tasks (and therefore, the combined amount of resources required at any given moment), we collect such information at the finest temporal granularity, that is, unit intervals:

$$\begin{cases} \bigwedge_{T_l\in\mathcal{T}}[G](T_l \rightarrow R^n_l)\wedge & \\ \bigwedge_{s=1,\ldots,M}\bigwedge_{l\in\mathcal{I}}\bigwedge_{k\leq\overline{k}}[G] & (\langle A\rangle(\mathsf{len}_{=\mathsf{k}} \wedge T_l \wedge R^s_l) \rightarrow \\ & \bigwedge_{k'<k}\langle A\rangle\langle A\rangle(\mathsf{len}_{=1} \wedge R^s_l)). \end{cases}$$

Now, we can easily pre-compute all and only those sequences of indexes $l_1, l_2, \ldots$ in $\mathcal{I}$, such that, for each such sequence σ, there are tasks in $\mathcal{T}$ such that, by summing all resources requested by each of them, we obtain a number greater than the maximum numbers of units available, which we can denote by M. Let Σ be the set of such sequences of indexes. If $\sigma = l_1, l_2, \ldots, l_{|\sigma|} \in \Sigma$, then there exists tasks in $\mathcal{T}$ for which we

 ISBN: 978-1-61208-344-5

Table I: A FRAGMENT OF A REALISTIC CASE STUDY. UPPER SIDE: TASKS. LOWER SIDE: GENERAL REQUIREMENTS.

Symbol	Name	Type	Duration	Preceding Task(s)	Formulas
T_1	Water well building	–	5	–	$\langle A\rangle T_1 \vee \langle A\rangle\langle A\rangle T_1$
T_2	Water convey construction work	P_1	12	T_1	$\langle A\rangle T_2 \vee \langle A\rangle\langle A\rangle T_2$, $[G](T_2 \rightarrow \mathsf{len}_{=12})$ $[G](T_2 \rightarrow \langle\overline{A}\rangle(T_1 \vee \langle\overline{A}\rangle T_1)$
T_3	Water convey construction work	P_1	14	T_2	$\langle A\rangle T_3 \vee \langle A\rangle\langle A\rangle T_3$, $[G](T_3 \rightarrow \mathsf{len}_{=14})$ $[G](T_3 \rightarrow \langle\overline{A}\rangle(T_2 \vee \langle\overline{A}\rangle T_2)$
T_4	Pumphouse construction	–	15	–	$\langle A\rangle T_4 \vee \langle A\rangle\langle A\rangle T_4$, $[G](T_4 \rightarrow \mathsf{len}_{=15})$
T_5	Pump acquisition,installation and electric connections	–	30	T_1	$\langle A\rangle T_5 \vee \langle A\rangle\langle A\rangle T_5$, $[G](T_5 \rightarrow \mathsf{len}_{=30})$ $[G](T_5 \rightarrow \langle\overline{A}\rangle(T_1 \vee \langle\overline{A}\rangle T_1)$
T_6	Ground filling-up and cleaning	–	4	T_3	$\langle A\rangle T_6 \vee \langle A\rangle\langle A\rangle T_6$, $[G](T_6 \rightarrow \mathsf{len}_{=4})$ $[G](T_6 \rightarrow \langle\overline{A}\rangle(T_3 \vee \langle\overline{A}\rangle T_3)$

General Requirement or Generic Data	Formula or Symbol
Maximal duration of a task	30
Tasks are unique	$\bigwedge_{l=1,\ldots,6}[G]\langle A\rangle(T_l \rightarrow [A]\neg T_l)$
*Between any two convey construction works, a minimum of 5 time units must be given for checking	$[G]((T_2 \vee T_3) \leftrightarrow P_1) \wedge [G](P_1 \wedge \langle A\rangle\langle A\rangle P_1 \rightarrow \langle A\rangle(\mathsf{len}_{>4} P_1))$

have used the propositional letters, corresponding to the need of resources, $R_{l_1}^{f(l_1)}, R_{l_2}^{f(l_2)}, \ldots$, and the previous constraints have placed them somewhere in the model. We need to make sure that no such tuple of propositional letter is true at any given unit interval:

$$\bigwedge_{\sigma\in\Sigma}\bigwedge_{l_1,\ldots,l_{|\sigma|}\in\sigma}[G]((\mathsf{len}_{=1}\wedge R_{l_1}^{f(l_1)}\wedge R_{l_2}^{f(l_2)}\wedge\ldots\wedge R_{l_{|\sigma|}}^{f(l_{|\sigma|})}) \rightarrow \bot).$$

Concluding, this section shows the applicability of a logical framework as an effective support tool for plan design and test. The advantages of using a logical tool in substitution of an algebraic one are well-known, and include, among others, *(i)* the possibility of specifying universal properties; *(ii)* the possibility of specifying negative information; *(iii)* the possibility of comparing, under various points of view, two or more plans. Moreover, it is worth recalling that algebraic networks feature only *limited* disjunction capabilities; as for example, the only way to encode a requirement such as *the task T_l precedes the task T_m or the task T_g* in an algebraic network is to compute two entirely separated networks, while such a requirement has an immediate logical counterpart:

$$[G]((T_m \vee T_g) \rightarrow \langle\overline{A}\rangle(T_l \vee \langle\overline{A}\rangle T_l).$$

IV. A REALISTIC CASE STUDY

We present in this section a fragment of a realistic case study that includes a planning phase. The project under analysis is the construction of a drinkable water provision system for various towns. It includes the building of two water wells, one pumphouse, an impulsion system, and one energy transformation unit. A realistic case study features various tens of different requirements, all of which fall into some of the categories explained in the previous section, and classified into different (conceptual) groups.

In order to show the applicability of MPNL as planning support system, we give in Table I an extract from the collection of task requirements and conditions for this project, and we translate it into MPNL, adding the general conditions as explained in the previous section. To the original plan, we added a further condition in order to show the capabilities of this approach. We suppose that, at some point the chief engineer requires a minimum time to check the construction work before continuing (*): instead of re-thinking the entire plan, it is enough to add the corresponding formula and re-run the satisfiability checker.

V. CONCLUSIONS

The purpose of this paper was twofold. On the one side, we present a novel technique to solve a well-known problem, that is, plan design and checking in Engineering. This problem is usually solved by means of simple algebraic methods, well-established in the field, but that suffer of intrinsic limitations. On the other side, we give a clear and simple application of a recently studied temporal logic for time intervals, that is, MPNL. Algebraic networks are historically preferred over logical formalism for planning applications; this is due to many reasons, among which we mention computational properties of the formalisms and simplicity of the approach. The recent discover of decidable, pure temporal logic for time intervals may change this perspective, allowing one to use more power formalisms without giving up the decidability and therefore the possibility of computer-assisted design. Moreover, while it is true that algebraic networks present, usually, a satisfiability problem with (non-deterministic) polynomial complexity, the plan designing and checking phase needs not to be real-time (it is usually a off-line procedure), and one can afford longer computation times.

ACKNOWLEDGMENTS

The authors acknowledge the support from the Spanish fellowship program *'Ramon y Cajal' RYC-2011-07821* (G. Sciavicco).

REFERENCES

[1] W. Bibel, "Let's plan it deductively," in *Proc. of the 15th Int. Joint Conference on Artificial Intelligence (IJCAI)*, 1997, pp. 1549–1562.

[2] U. Riss, A. Rickayzen, H. Maus, and W. van der Aalst, "Challenges for business process and task management," *Journal of Universal Knowledge Management*, vol. 0, no. 2, pp. 77–100, 2002.

 ISBN: 978-1-61208-344-5

[3] IBM. (2014, Jan.) Life cycle of human tasks. [Online]. Available: http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/index.jsp?topic=/com.ibm.websphere.bpc.610.doc/doc/bpc/ctasklifecycle.html

[4] J. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[5] J. F. Allen and P. J. Hayes, "A common-sense theory of time," in *Proc. of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, USA, 1985, pp. 528–531.

[6] G. Sciavicco, "Reasoning with time intervals: A logical and computational perspective," *ISRN Artificial Intelligence*, vol. 2012, 2002, available online. Article ID 616087.

[7] J. Halpern and Y. Shoham, "A propositional modal logic of time intervals," *J. of the ACM*, vol. 38, no. 4, pp. 935–962, 1991.

[8] V. Goranko, A. Montanari, and G. Sciavicco, "A road map of interval temporal logics and duration calculi," *J. of Applied Non-Classical Logics*, vol. 14, no. 1–2, pp. 9–54, 2004.

[9] D. Della Monica, V. Goranko, A. Montanari, and G. Sciavicco, "Expressiveness of the interval logics of allen's relations on the class of all linear orders: Complete classification," in *Proceedings of the 22th International Joint Conference Artificial Intelligence (IJCAI)*, 2011, pp. 845–850.

[10] D. Bresolin, D. Della Monica, A. Montanari, P. Sala, and G. Sciavicco, "Interval temporal logics over strongly discrete linear orders: the complete picture," in *Proc. of the 3rd International Symposium on Games, Automata, Logics and Formal Verication (GANDALF)*, 2012, pp. 155–168.

[11] D. Bresolin, D. Della Monica, A. Montanari, P. Sala, and G. Sciavicco, "Interval temporal logics over finite linear orders: the complete picture," in *Proc. of the 20th ECAI*, 2012, pp. 199–204.

[12] D. Bresolin, A. Montanari, P. Sala, and G. Sciavicco, "Optimal decision procedures for mpnl over finite structures, the natural numbers, and the integers," *Theoretical Computer Science*, no. 493, pp. 98–115, 2013.

[13] A. I. A. M. L. Aceto, D. Della Monica and G. Sciavicco, "Complete classification of the expressiveness of fragments of halpern-shoham logic over dense linear orders," in *20th International Symposium on Temporal Representation and Reasoning (TIME)*, 2013, pp. 65–72.

[14] D. Bresolin, D. D. Monica, V. Goranko, A. Montanari, and G. Sciavicco, "Metric propositional neighborhood logics on natural numbers," *Software and System Modeling*, vol. 12, no. 2, pp. 245–264, 2013.

[15] S. Shaheen, *Practical Project Management*. Wiley, 1986.

 ISBN: 978-1-61208-344-5