COMPUTATION TOOLS 2010

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

November 21-26, 2010 - Lisbon, Portugal

ComputationWorld 2010 Editors

Ali Beklen, IBM Turkey, Turkey
Jorge Ejarque, Barcelona Supercomputing Center, Spain
Wolfgang Gentzsch, EU Project DEISA, Board of Directors of OGF, Germany
Teemu Kanstren, VTT, Finland
Arne Koschel, Fachhochschule Hannover, Germany
Yong Woo Lee, University of Seoul, Korea
Li Li, Avaya Labs Research - Basking Ridge, USA
Michal Zemlicka, Charles University - Prague, Czech Republic

COMPUTATION TOOLS 2010

Foreword

The First International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking [COMPUTATION TOOLS 2010], held between November 21 and 26 in Lisbon, Portugal, inaugurated an event under the umbrella of ComputationWorld 2010 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.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2010 Technical Program Committee, as well as the numerous reviewers. The creation of such a broad and 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 2010. 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 2010 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 2010 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 also hope the attendees enjoyed the beautiful surroundings of Lisbon, Portugal.

COMPUTATION TOOLS 2010 Chairs:

Luis Gomes, Universidade Nova de Lisboa, Portugal
Radu-Emil Precup, "Politehnica" University of Timisoara, Romania
Kenneth Scerri, University of Malta, Malta

COMPUTATION TOOLS 2010

Committee

COMPUTATION TOOLS Advisory Chairs

Luis Gomes, Universidade Nova de Lisboa, Portugal
Kenneth Scerri, University of Malta, Malta
Radu-Emil Precup, "Politehnica" University of Timisoara, Romania

COMPUTATION TOOLS 2010 Technical Program Committee

Stefan Andrei, Lamar University, USA
Henri Basson, University of Lille North of France (Littoral), France
Ateet Bhalla, Technocrats Institute of Technology - Bhopal, India
Manfred Broy, Technical University of Munich, Germany
Noël Crespi, Institut Telecom, France
Brahma Dathan, Metropolitan State University - St. Paul, USA
Luis Gomes, Universidade Nova de Lisboa, Portugal
Rajiv Gupta, University of California - Riverside, USA
Haidar Harmanani, Lebanese American University, Lebanon
Raimund Kirner, University of Hertfordshire, UK
Bernd J. Krämer, FernUniversität - Hagen, Germany
Giovanni Lagorio, DISI/University of Genova, Italy
Yuan Fang Li, University of Queensland, Australia
Zhiming Liu, UNU-IIST, Macao
Tomoharu Nakashima, Osaka Prefecture University, Japan
Flavio Oquendo, European University of Brittany - UBS/VALORIA, France
Aomar Osmani, Université Paris 13, France
Radu-Emil Precup, "Politehnica" University of Timisoara, Romania
Antoine Rollet, University of Bordeaux, France
Antonino Sabetta, ISTI-CNR - Pisa, Italy
Kenneth Scerri, University of Malta, Malta
Daniel Schall, Vienna University of Technology, Austria
Sharad Sharma, Bowie State University, USA
Giovanni Semeraro, University of Bari "Aldo Moro", Italy
Bernhard Steffen, TU Dortmund, Germany
Toyotaro Suzumura, IBM Research / Tokyo Institute of Technology, Japan
Zhonglei Wang, Technical University of Munich, Germany
Marek Zaremba, University of Quebec, Canada

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

Debugging PVS Specifications of Control Logics via Event-driven Simulation

Cinzia Bernardeschi*, Luca Cassano*, Andrea Domenici* and Paolo Masci*†
*Department of Information Engineering, University of Pisa, Italy
†Information Science and Technologies Institute, National Research Council, Pisa, Italy
Email: {Cinzia.Bernardeschi,Luca.Cassano,Andrea.Domenici,Paolo.Masci}@ing.unipi.it

***Abstract*—In this paper, we present a framework aimed at simulating control logics specified in the higher-order logic of the *Prototype Verification System*. The framework offers a library of predefined modules, a method for the composition of more complex modules, and an event-driven simulation engine. A developer simulates the specified system by providing its input waveforms as functions from time to logic levels. Once the simulation experiments give sufficient confidence in the correctness of the specification, the model can serve as a basis for the formal verification of desired properties of interest. A simple case study from a nuclear power plant application is shown. This paper is a contribution to research aimed at improving the development process of safety-critical systems by integrating simulation and formal specification methods.**

***Index Terms*—PVS; simulation; formal specification; validation**

I. Introduction

Control systems are an important field of application for formal methods and rigorous engineering practices, since they combine real-time requirements and non-trivial control tasks whose failure may compromise safety. Subtle design faults, which are often difficult to avoid and tolerate, and the possibility of failures caused by the occurrence of non-obvious combinations of events, make such systems hard to certify with respect to safety requirements.

In this paper, we present a methodology aimed at simulating control logics specified in the higher-order logic of the *Prototype Verification System (PVS)* [1]. We have developed a library of (purely logic) specifications for typical control logic components, a methodology to combine them into more complex systems, and a simulation engine capable of animating the formal specifications with the PVS ground evaluator.

Section II exposes the motivations for this work. We introduce the PVS system in Section III, then we describe the theories for the logical specification of control components (Section IV) and the theory defining the simulator (Section V). In Section VI we describe a simple case study from the field of control logics for nuclear power plants (NPPs), and finally the conclusion and related work are found in Section VII.

II. Motivation

The use of formal methods is increasingly being required by international standards for the development of safety critical digital control systems (e.g., [2], [3]), but, in industrial practice, verification and validation of such systems relies heavily on simulation and testing. A rigorous development process would benefit from the combined application of formal verification, simulation, and testing. In particular, simulation would be a means to validate specifications against requirements. However, verification tools (such as theorem provers and model checkers) and simulation tools use different languages, and few designers are versed in the use of both kinds of tools.

This work is a first step in a research activity whose expected outcome is a toolset that translates specifications from an application-oriented language into a high-order logic theory that guides the execution of the simulator described in this paper. When the simulation results make developers confident that the specifications are correct, a more detailed and formal analysis may be done by theorem proving. The theorem proving approach was chosen as it may be expected to avoid the problem of state space explosion that model checking tools face in the analysis of complex real-time systems.

III. PVS and PVSio

The PVS [1] specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `integer`, `real`, among others, and the function type constructor (e.g., type `[A -> B]` is the set of functions from set `A` to set `B`). Predicates are functions with range type `bool`. The type system of PVS also includes record types, dependent types, and abstract data types.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A collection of built-in (*prelude*) theories and loadable libraries provide standard specifications and proved facts for a large number of theories. A theory can use the definitions and theorems of another theory by *importing* it.

PVS has an automated theorem prover. A less frequently used component is its *ground evaluator* [4], used to animate functional specifications by translating executable PVS constructs into efficient *Lisp* code. The *PVSio* package [5] extends the ground evaluator with a library of imperative programming language features such as side effects, unbounded loops, and input/output operations. Thus, PVS specifications can be conveniently animated within the *read-eval-print* loop of the ground evaluator that reads PVS expressions from the user interface and returns the result of their evaluation.

 ISBN: 978-1-61208-112-0

IV. MODELING CONTROL LOGICS

In this section we describe the PVS theories developed to formally model control logics. We start with the PVS theories that model time, logic levels, signals, and basic operations on signals. Then, we introduce samples of the library for the basic digital modules of a system, such as logic gates and timers. Finally, we show how to build complex components out of basic elements. The developed theories are executable: definitions always use interpreted types and quantification is always performed over bounded types. In the following sections, only the `time_th` theory will be shown in a syntactically complete form; to save space, only fragments of PVS code will be shown in the rest of the paper for the other theories.

A. Time and Logic Levels

Theory `time_th` (shown below) contains the type definition of time (modeled as ranging over the continuous domain of real numbers) and time interval.

```
time_th: THEORY
 BEGIN
  time: TYPE = real
  interval: TYPE = {t: time | t >= 0}
 END time_th
```

Besides the `zero` and `one` logical values, modeling hardware circuits requires additional levels for *unknown* values and *high impedance*. Unknown values are useful to model the logic level when the digital circuit is powered up, while high impedance represents open circuits (designed or faulty).

Theory `logic_levels_th` provides the definitions of the logic levels and of the basic logical operators over the four-valued logic (`lAND`, `lOR`, `lNOT`). In the following fragment we show the first definitions of the theory.

```
logic_level: TYPE  = below(4)
zero: logic_level = 0;
one: logic_level  = 1;
Z: logic_level    = 2;  %-- high impedance
U: logic_level    = 3;  %-- unknown value
lAND(v1, v2: logic_level): logic_level =
      IF one?(v1) AND one?(v2) THEN one
      ELSIF zero?(v1) OR zero?(v2) THEN zero
      ELSE U ENDIF
```

B. Signals

A signal describes the variation of a logic level over time, and we represent signals as functions from the domain of time to logic levels. Theory `signals_th` contains, besides the definition of `signal`, the symbolic constant for time resolution, `tres`, which models the minimum time between two observable variations of a signal, and the definition of a utility function to build periodic signals (`make_periodic`).

Basic signals provided in the theory are: `constval`, a constant logical level; `step`, a signal that goes from zero to one at time τ; `pulse`, a signal that is one only in the time interval $[\tau, \tau + d)$, where d is the interval size.

Some useful predicates on signals are defined, such as `rising_edge?`, used to detect if a signal `s` has a rising edge at time `tau`. Operations that apply logical connectives to the values of signals at each given time are defined (`sOR`, `sAND`, `sNOT`). Sample definitions of this theory follow.

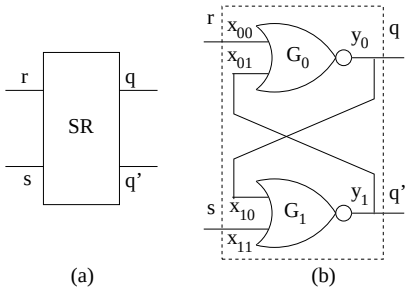


Fig. 1. An SR flip-flop.

```
IMPORTING time_th, logic_levels_th
signal: TYPE = [time -> logic_level]
tres: {t: interval | t > 0}
make_periodic(s: signal, T: interval): signal =
    % definition not shown
constval(v: logic_level): signal =
    LAMBDA (t: time): v
step(tau: time): signal =
    LAMBDA (t: time):
       IF t >= tau THEN one ELSE zero ENDIF
pulse(tau: time, d: posreal): signal =
    % definition not shown
rising_edge?(i: signal, tau: time): bool =
    zero?(i(tau - tres)) AND one?(i(tau + tres))
      AND one?(i(tau))
sAND(s1, s2: signal): signal =
    LAMBDA (t: time): lAND(s1(t), s2(t))
```

C. Digital Modules

In our framework, a control logic is a *composite digital module*, obtained by connecting *basic digital modules*.

Digital modules are characterized by a set of *ports*, a *state*, that is the collection of all signals present at its ports, and a *transition function* that specifies how the state changes according to a module's functionality. The behavior of each module in the framework is defined by its transition function.

Ports are abstractions of the terminals of physical devices. Each port is identified by its *category* (one of *input*, *output*, *internal*) and its *port number* within the category. Basic modules have only input and output ports, whereas composite modules also have internal ports. In a composite module, the input and output ports are its externally visible terminals, and its internal ports are the ports of the (basic) component modules that are not externally visible.

For example, a NOR gate is modeled as a module with two input ports, one output port, and no internal ports. Another example is an SR flip-flop, which can be modeled either as a basic module (Figure 1(a)) with two input ports, two output ports and no internal ports, or as a composite module built from two NOR gates. In the latter case, the resulting system is shown in Figure 1(b), where ports x_{00} of gate G_0 and x_{11} of gate G_1 are input ports, ports y_0 of G_0 and y_1 of G_1 are output ports, and ports x_{01} and x_{10} are internal ports.

Theory `digital_modules_th` contains type definitions for the state of a digital module (`state`) and for transition functions (`digital_module`). Type `state` the value of

 ISBN: 978-1-61208-112-0

the signals present at any time is a record that maintains the lists of signals applied at any time on its ports. It has one list of signals for each of the three port categories, and a port of the system is identified by its position in the list of the corresponding category. In the rest of this paper the term *signal* will sometimes be used instead of *port*, so that "signal x" means "the signal present at port x". The transition function type `digital_module` is time-dependent and has the signature $[time \to [state \to state]]$.

Note that the state of a module is defined as the set of `signals` applied to, or generated by, the module at a given time, and not as the set of their instantaneous values.

The theory includes also a number of auxiliary functions to build lists of ports (i.e., of signals) and to select a specific port of a module, such as `ports(`n`)`, `ports(`s, n`)`, etc. The first definitions of the theory follow.

```
IMPORTING signals_th
ports: TYPE = list[signal]
state: TYPE = [# input: ports, output: ports,
                 internal: ports #]
digital_module: TYPE = [time -> [state -> state]
%-- port constructors
ports(n: nat): RECURSIVE
               {p: ports | length(p) = n} =
  IF n = 0 THEN null
  ELSE cons(constval(U), ports(n - 1)) ENDIF
   MEASURE n
ports(s: signal, n: nat): RECURSIVE
               {p: ports | length(p) = n} =
    % definition not shown
%-- port selectors
port(p: ports, i: below(length(p))): signal =
   nth(p,i)
```

Types `state` and `digital_module` are very general, and they are refined by subtyping in the theories for basic digital modules and composite digital modules.

D. Basic Digital Modules

Basic digital modules are elements without a visible internal structure, defined only by their input and output ports and by their transition function. The state of a basic module has an empty list of internal signals, and the lists of input and output signals have a predefined length.

The theory is parametric with respect to a parameter `delay`, representing the time needed by the component to change its outputs when its inputs change.

In addition to the parameterized definitions for the state and transition function types, the theory contains a state constructor (`new_state`). Part of the theory is shown below.

```
basic_digital_modules_th[delay: nonneg_real]: THEORY
 BEGIN IMPORTING digital_modules_th
  state(nIN, nOUT: nat): TYPE =
           {s: state | length(input(s)) = nIN AND
                       length(output(s)) = nOUT AND
                       length(internal(s)) = 0 }
  basic_digital_module(nIN, nOUT: nat): TYPE =
    [time -> [state(nIN, nOUT) -> state(nIN, nOUT)]]
```

This theory is imported by other theories that define various classes of basic blocks, such as logic gates, timers, and flip-flops, presented in the following.

1) Logic gates: The `logic_gates_th` theory defines the transition functions of the basic combinatorial gates. The theory is parameterized by the propagation delay of the gates.

As the state is defined by the *signals* at the ports (and not the instantaneous values), the new state will normally be equal to the previous one, unless the environment applies different signals to the inputs (e.g., a pulse replaces a constant level). The definition for the NOR gate is shown below.

```
logic_gates_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
gateNOR: basic_digital_module(2, 1) =
  LAMBDA (t: time): LAMBDA (s: state(2, 1)):
    s WITH [output := ports(time_shift(
            sNOR(port0(input(s)), port1(input(s))),
            delay))]
```

2) Timers: The `timers_th` theory defines devices that generate a single pulse when they receive a rising edge on their input port. The pulse duration is a parameter of the device. Their response to the input depends on previous values of the output and possibly of the input(s).

```
timers_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
  timerM(d: posreal): basic_digital_module(1, 1) =
    LAMBDA (t: time): LAMBDA (s: state(1, 1)):
      IF rising_edge?(port0(input(s)), t) AND
         zero?(port0(output(s)), t)
      THEN s WITH [output := ports(pulse(t+delay, d))]
      ELSE s ENDIF
```

The theory defines also resettable timers (not shown), whose output drops to zero on receiving a rising edge at the reset port.

3) Flip-flops: The `flipflop_th` theory defines 1-bit registers, such as the SR flip-flop (Figure 1(a)). Ports s and r are the set and reset terminals, the stored bit is on the output marked q, and q' is its complement. Ports q and q' hold their previous value when s and r are both zero. If s becomes one while r is zero, then q is one, and stays at one even after s returns zero. Similarly, if r becomes one while s is zero, then q is zero, and stays at zero even after r returns zero.

```
flipflop_th[delay: nonneg_real]: THEORY
BEGIN IMPORTING basic_digital_modules_th
flipflopSR: basic_digital_module(2, 2) =
 LAMBDA (t: time): LAMBDA (st: state(2, 2)):
  LET r = port0(input(st)), s = port1(input(st)),
    q = port0(output(st)), q_prime = port1(output(st))
  IN IF zero?(s, t) AND zero?(r, t) THEN st
     ELSIF one?(s, t) AND zero?(r, t)
     THEN IF zero?(q, t) AND one?(q_prime, t)
          THEN st WITH [output := ports
                 (step(t+delay), sNOT(step(t+delay)))]
          ELSE st ENDIF
     ELSIF zero?(s, t) AND one?(r, t)
     THEN IF one?(q, t) AND zero?(q_prime, t)
          THEN st WITH [output := ports
                 (sNOT(step(t+delay)), step(t+delay))]
           ELSE st ENDIF
     ELSE st WITH [output := ports(2)] ENDIF
```

E. Composite Digital Modules

Basic digital modules can be connected together to create *composite digital modules*. The corresponding theory contains only the high-level definition for the state and the transition function, and for a state constructor (not shown).

 ISBN: 978-1-61208-112-0

```
BEGIN IMPORTING digital_modules_th
state(nIN, nOUT, nINT: nat): TYPE =
  {s: state | length(input(s)) = nIN AND
              length(output(s)) = nOUT AND
              length(internal(s)) = nINT}
composite_digital_module(nIN, nOUT, nINT: nat):
   TYPE = [time -> [state(nIN, nOUT, nINT)
                    -> state(nIN, nOUT, nINT)]]
```

1) Building Composite Digital Modules: A composite module is modeled by the composition of the transition functions of its components, whose form depends on the interconnections of the components.

In order to build the composite module, one must first define the *system* state, i.e., the union of its input, output, and internal ports. Then the subsets of the composite system state relative to the components (*component substates*) must be identified. Then the transition function is defined along the following lines: (i) Each port of the composite module is assigned a unique name by equating the port to a variable of type `signal` in a `LET` expression (e.g., `r = port0(input(st))` gives the name `r` to the first input port of state `st`); (ii) for each basic component, we define its current substate by selecting its input and output signals from the current system state; (iii) for each basic component, we define its next substate as a variable of type `state`, and we equate it to the basic component's transition function applied to the current substate defined in the previous step; (iv) the output signals of the new system state are the union of the output signals of the new substates of the basic components connected to the system output; (v) the internal signals of the next system state are the union of the internal signals of the new substates of the basic components.

As an example, we show the composite module of the SR flip-flop built from a pair of cross-coupled NOR logic gates. With reference to Figure 1(b), in this example port `x01` is renamed as `r1`, and `x10` as `s1`.

```
flipflopSR: composite_digital_module(2, 2, 2) =
LAMBDA (t: time): LAMBDA (st: state(2, 2, 2)):
 LET r = port0(input(st)), s = port1(input(st)),
  q = port0(output(st)), q_prime = port1(output(st)),
  r1 = port0(internal(st)), s1 = port1(internal(st)),
  nor0 = gateNOR[tres](t)(new_state(2, 1)
       WITH [input := ports(r, r1),
             output := ports(q)]),
  nor1 = gateNOR[tres](t)(new_state(2, 1)
       WITH [input := ports(s, s1),
             output := ports(q_prime)])
 IN st WITH [output := ports(port0(output(nor0)),
                        port0(output(nor1))),
             internal := ports(port0(output(nor1)),
                         port0(output(nor0)))]
```

In the system transition function `flipflopSR`, we let signal `r` be the signal on the first input port (`port0`) of the current system state `st`, and similarly for `s`, `q`, `q_prime`, `s1`, and `r1`. Then, substate `nor0` of gate `G0` is the result of transition function `gateNOR`. The argument of this function is a state with input signals `r` and `r1`, and output signal `q`. A similar description applies to `nor1`.

V. The Event-driven Simulator

This section describes an event-driven simulator of digital modules. First, we introduce *events*, i.e., instants when a signal may change its value. Second, we enrich the specification of the system with events. Third, we present the event-driven simulation engine, which uses the enriched specification to evaluate the system only at specific instants, instead of at periodic steps as in time-driven approaches [6].

A. Events

Theory `events_th` defines the type `event` as a record with fields `t`, the instant of a single event or of the first of a series of periodic events, and `T`, the period of the series (single events have `T=0`). The theory includes the ordering relation between events and operations on list of events. Some definitions are shown below.

```
BEGIN IMPORTING time_th
 event: TYPE = [# t: time, T: interval #];
 <(e1, e2: event): bool =
     (t(e1) < t(e2)) OR
       (t(e1) = t(e2) AND T(e1) < T(e2));
```

B. Annotated Signals

In theory `annotated_signals_th` we annotate the formal specification of signals with the list of events associated with each signal. We redefine the type `signal` as a record with the fields `val`, the functional specification of the signal, and `evts`, the set of instants when the waveform changes value. For example, the set of events associated with a constant level generator is empty, while the set of events associated with a pulse generator at time τ and duration d contains events τ and $\tau + d$, both with period $T = 0$.

The basic operators on signals are re-defined to calculate the events of the resulting signal, whose events are the union of events of the operator parameters. Some events in the resulting signal may not affect the signal value. For example, if initially one of the `sOR` inputs is a constant `one`, no set of events on the other input changes the output. Such redundant events, however, do not affect the simulation.

The following fragment shows the definition of `sNOR`.

```
BEGIN IMPORTING events_th, logic_levels_th
  sNOR(s1a, s2a: signal): signal =
    LET s1 = val(s1a), s2 = val(s2a),
      f = LAMBDA (t: time):
      IF one?(s1(t)) OR one?(s2(t)) THEN zero
      ELSIF zero?(s1(t)) AND zero?(s2(t)) THEN one
      ELSE U ENDIF,
      e = evts(s1a) + evts(s2a)
    IN (# val := f, evts := e #)
```

Annotated signals carry all the information needed by the simulator to handle events, so the specification of the digital modules is unchanged.

C. Simulator

The simulator maintains a list of events (*worklist*), initialized with the starting time of the simulation. The events are listed in ascending order without duplicates. At each simulation step, the simulator extracts the first event (*current*

 ISBN: 978-1-61208-112-0

event) from the worklist, and then it computes the next state by applying the system transition function at the time specified by the event. Then, the new events associated with the signals in the generated state are inserted in the worklist.

1) Worklist: Theory `worklist_th` defines the type `worklist` as a list of events, provides the function `get_first` that, given a current time, returns the first event associated with a set of signals and greater than the current time, and the function `update_wl` that updates the worklist. Function `update_wl` finds the new events in the next state and inserts them in the worklist. Note that, since the model of the system may contain ideal modules that update instantaneously their output ports, function `update_wl` must not remove the current event from the worklist as long as the generated state is not stable. These simple worklist manipulators are not shown.

2) Simulation Engine: The simulation engine applies the system transition function and returns the state of the system after a certain number of steps. It uses a customizable `dump` function to output a simulation trace.

```
simulate_system(n_steps: nat)
     (f: [time -> [state -> state]])
     (wl: worklist)(outf: OStream, pn: port_names):
    RECURSIVE [state -> state] =
 LAMBDA (s: state):
  IF n_steps > 0 AND length(wl) > 0
  THEN LET current_t = t(get_first(wl)),
     s_prime = update_state(s)(current_t, f),
     wl_prime = update_wl(wl)(current_t, s, s_prime),
     dbg = dump(outf, pn, s, s_prime,
                 wl, wl_prime, current_t)
    IN simulate_system(n_steps - 1)(f)(wl_prime)
                    (outf, pn)(s_prime)
  ELSE s ENDIF
```

The input parameters are the maximum number of steps, the system transition function, the worklist, the output stream for the trace, and the names of the signals. The function is called with an initial worklist containing all events of the initial state and an event for the initial time.

At each step, the function (i) gets the simulation time from the first event in the worklist, (ii) generates the next system state, (iii) updates the worklist, and (iv) outputs the system state. The simulation terminates when either the new worklist is empty, or the maximum number of steps is reached.

The following excerpt shows how the digital module `flipflopSR` is simulated. In function `sim_flipflopSR`, the initial state is constructed from the signals at the ports, the worklist is initialized, and `simulate_system` is invoked with the transition function as an argument. The *reset* port is initially fed with a constant zero signal, the *set* port with a pulse of $4s$ at time 0.3, and q (q') holds a constant zero (one).

```
sim_flipflopSR(N_STEPS: nat): bool =
 LET r = constval(zero), s = pulse(0.3, 4),
     q = constval(zero), q_prime = constval(one),
     r1 = q_prime, s1 = q,
     initial_st = new_state(2, 2, 2)
       WITH [input := ports(r, s),
             output := ports(q, q_prime),
             internal := ports(r1, s1)],
     initial_wl = worklist(initial_st, 0),
     final_s  = simulate_system(N_STEPS)(flipflopSR)
                               (initial_wl)(outf, pn)
                               (initial_st)
 IN TRUE
```

The simulation trace can be a list of event times, signal values and worklist contents at each step, or a *Value Change Dump* [7] output, readable by a visualization tool (e.g., *GTKWave* [8]).

3) Automated Execution of Test-Cases: The universal and existential quantifiers of PVS can be used to automatically set up different simulation studies, e.g., to analyze the response of the system to different input waveforms. This allows, for instance, instrumenting the framework for the execution of simulations in order to discover interesting test cases.

In the following example, the `test_flipflopSR` function uses the `FORALL` quantifier to generate all possible combinations of logical levels. Each combination defines an initial state for an SR flip-flop, and each state is used to compute a next state.

```
test_flipflop_th: THEORY BEGIN %--imports omitted
  % ...
  discrete_time: TYPE = below(2)
  test_flipflopSR: bool =
   FORALL (t_set, t_reset: discrete_time):
    FORALL (v1, v2: logic_level): v1 /= v2 =>
     (LET initial_st = new_state(2, 2, 2)
            WITH [input := ports(pulse(t_reset, 1),
                                 pulse(t_set, 1)),
                  output := ports(constval(v1),
                                  constval(v2)),
                  internal := ports(constval(v2),
                                  constval(v1))],
          initial_wl = worklist(initial_st, 0),
          final_s = simulate_system(5)(flipflopSR)
                    (initial_wl)(outf,pn)(initial_st)
      IN TRUE)
   %...
 END test_flipflop_th
```

VI. Case Study: A Stepwise Shutdown Logic

As an illustration of the practical applicability of the framework presented in this paper, we report on a simple case study from the field of Instrumentation and Control for NPPs. A high-level description of a control logic, expressed as a Function Block Diagram [9], has been manually translated into a PVS specification using the presented framework, and the specification has been animated to simulate the control logic. Simulated test cases have been automatically generated, allowing a possible malfunction to be detected at this early stage of development.

A. Description of a Stepwise Shutdown Logic

A *stepwise shutdown* process keeps process variables (such as, e.g., temperature or neutron flux) within prescribed thresholds by applying corrective actions (e.g., inserting control rods) not immediately to their full extent, but gradually, in a series of discrete steps separated by settling periods.

A Stepwise Shutdown Logic (SSL) was analyzed in [10] with a model checking approach. The framework proposed in this paper is used to analyze the same system.

 ISBN: 978-1-61208-112-0

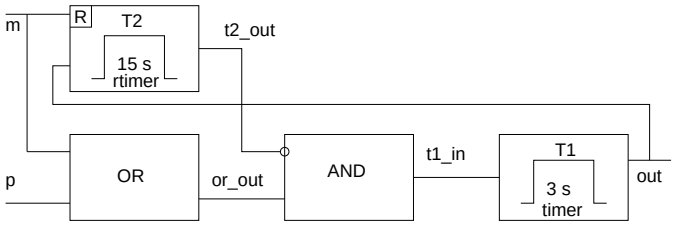


Fig. 2. A close-up view of the stepwise shutdown logic.

The requirements of the SSL, as described in [10], can be informally stated as follows: if an *alarm* signal (e.g., overpressure in a pipe) is asserted, the system must assert a control signal to drive a corrective action for 3 seconds (*active period*), then the control signal is reset for twelve seconds (*wait period*) and the cycle is repeated until either the alarm signal is reset or a complete shutdown is reached. An operator, however, by activating a *manual trip* signal, may force the wait periods to be skipped in order to accelerate the process.

Figure 2 shows the main part of one of the SSL implementations analyzed in [10], where `m` is the manual trip, `p` is an alarm signal, and `out` is the control signal. When all signals are low, the output `t2_out` of timer T2 is low, and the AND gate is enabled. When `p` is asserted, its rising edge passes through the AND gate to the input of the T1 timer that sends a $3s$ pulse to the output. The output is fed back to the input of T2, a resettable timer with a pulse duration of $15s$. The output pulse of T2 disables the AND gate that in turn resets the input of T1. Since T1 is not resettable, its output pulse lasts for three seconds, then returns to low for the remaining $12s$ of the T2 pulse. After this wait period, the output of T2 goes low, the AND gate is enabled, and T1 starts a new pulse if an input signal is still asserted.

If `p` is high, and `m` is asserted during a wait period, T2 is reset and its output enables the AND gate, allowing the trip signal to reach T1 and restart it at the end of the $3s$ pulse.

The SSL is modeled by the `systemC` transition function (see Figure 3), according to the guidelines in Section IV.

B. Simulation of the Stepwise Shutdown Logic

In this section we show some simulated situations.

Simulation 1 Signal `p` is a step function with the rising edge at $t = 0.3s$, and signal `m` is a constant zero (no manual intervention). The control logic produces a series of pulses that drive the plant towards a shutdown, as expected (Fig. 4). In the following, we show the PVS code for this simulation.

```
sim_system1: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(constval(zero), step(0.3)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Fig. 4. Output of sim_system1, displayed with GTKWave.

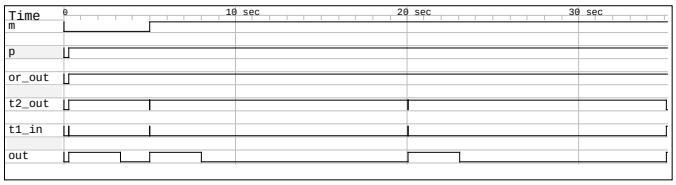


Fig. 5. Output of sim_system2, displayed with GTKWave.

Simulation 2 Signal `p` is a step function with the rising edge at $t = 0.3s$ and signal `m` is a step function with the rising edge at $t = 5s$. This means that the trip switch is pushed during the first wait period. As expected, that wait period is interrupted, a new $3s$ output pulse is generated, and the subsequent pulses are generated with the normal $15s$ cycle, since the trip switch has not been released and the resettable timer responds only to a rising edge (Fig. 5).

```
sim_system2: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(step(5), step(0.3)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Simulation 3 In this instance, signal `p` is a step function with the rising edge at $t = 1s$ and signal `m` is a pulse of duration $1s$ starting at $t = 2s$ followed by another pulse of duration $3s$ at $t = 10s$. In this case, the manual intervention occurs during the active period of the first output pulse. Contrary to expectation, after the end of this output pulse, the output is stuck at zero and no further corrective action takes place, even if the alarm (high pressure) persists and the manual trip switch is pressed again. A fundamental safety requirement is thus violated (Fig. 6).

```
sim_system3: bool =
 LET initial_st =
  new_state(nIN, nOUT, nINT)
   WITH [input := ports(sOR(pulse(2,1), pulse(10,3)),
                        step(1)),
         output := ports(constval(zero)),
         internal := ports(constval(zero), nINT)],
  initial_wl = worklist(initial_st, 0),
  final_s = simulate_system(NSTEPS)(systemC)
            (initial_wl)(outf, pn)(initial_st)
  IN TRUE
```

Test-Cases Interesting simulation examples, such as `sim_system3`, can be discovered by instrumenting the framework for the execution of test cases.

 ISBN: 978-1-61208-112-0

```
systemC: composite_digital_module(nIN, nOUT, nINT) =
 LAMBDA (t: time): LAMBDA (st: state(nIN, nOUT, nINT)):
  LET m = port0(input(st)), p = port1(input(st)), out = port0(output(st)),
      t2_in = port0(internal(st)), t2_out = port1(internal(st)),
      %- similar definitions for or_in, and_en, and_out
      rtimer = rtimerM[T1](D2)(t)(new_state(2,1) WITH [input:=ports(t2_in,m), output:=ports(t2_out)]),
      or2 = gateOR[T0](t)(new_state(2,1) WITH [input:=ports(or_in,p), output:=ports(or_out)]),
      inh_and = gateANDH[T0](t)(new_state(2,1) WITH [input:=ports(and_en,and_in), output:=ports(and_out)]),
      timer = timerM[T2](D1)(t)(new_state(2,1) WITH [input:=ports(t1_in), output:=ports(out)])
   IN st WITH [input := ports(m, p), output := ports(port0(output(timer))),
               internal := ports(port0(output(timer)), port0(output(rtimer)), m, port0(output(or2)),
                port0(output(rtimer)), port0(output(or2)), port0(output(inh_and)), port0(output(inh_and)))]
```

Fig. 3. PVS model of the Stepwise Shutdown Logic.

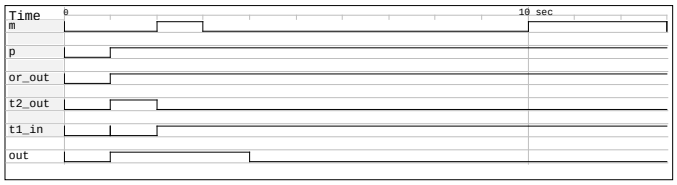


Fig. 6. Output of sim_system3, displayed with GTKWave.

In the following example, function `test_system` uses the `FORALL` quantifier to automatically generate the initial state for the different test cases. The initial states differ by the starting time of the pulse applied to the manual trip port. The ground evaluator implicitly transforms the universally quantified formula on `t0` into a loop that, at each iteration, generates a new initial state with a pulse starting at `t0 = 0,1,..,N-1` on the manual trip port, and applies the simulator for `NSTEPS` steps.

```
sim_system_test(N: nat): bool =
 FORALL(t0: below(N)):
  LET initial_state =
      new_state(nIN, nOUT, nINT)
       WITH [input := ports(pulse(t0,1), step(1)),
             output := ports(constval(zero)),
             internal := ports(constval(zero), nINT)],
     initial_wl = worklist(initial_st, 0),
     final_s = simulate_system(NSTEPS)(systemC)
               (initial_wl)(outf, pn)(initial_st)
    IN TRUE
```

VII. Conclusion and Related Work

PVS has been used in various works to describe hardware systems, e.g., in [11], [12], [13]. With our approach, the formal specifications are executable and they can be simulated with the ground evaluator of PVS. This way, once the simulation experiments give developers sufficient confidence in the correctness of the specification, the same PVS models can serve as the basis for the formal verification of properties in the theorem prover of PVS. It is known that a large share of defects in computing systems stem from errors in the formulation of specifications [14].

In the present work, a library of (purely logic) specifications for typical control logic components is presented, and an approach to define an event-driven simulator capable of executing the logic specifications is shown. The library includes theories to model logic signals over time, where time is a variable in the domain of real numbers. The simulator is based on the paradigm of event-driven-simulation, and its core component is defined as a function in the higher-order logic language of the PVS system. proving environment. The approach has been applied to a simple case study in the field of NPPs. The same case study had been previously studied by other researchers with a model checking approach [10].

This work is part of our current research activity aiming at developing a simulation and analysis framework for control logics that enables developers to rely both on simulation and theorem proving to assess the correctness of specifications and designs.

References

[1] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. on Software Engineering*, vol. 21, no. 2, pp. 107–125, 1995.

[2] "Railway applications – Software for railway control and protection systems," CENELEC, European Committee for Electrotechnical Standardization, Tech. Rep. EN 50128:2001 E, 2001, european standard.

[3] "Software for Computer Based Systems Important to Safety in Nuclear Power plants," IAEA, International Atomic Energy Agency, Tech. Rep. NS-G-1.1, 2000.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "Evaluating, testing, and animating pvs specifications," Computer Science Laboratory, SRI International, Tech. Rep., 2001.

[5] C. Muñoz, "Rapid prototyping in PVS," National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NASA/CR-2003-212418, 2003.

[6] A. M. Law and D. Kelton, *Simulation Modeling and Analysis*. McGraw-Hill, 2000.

[7] "IEEE Standard Verilog Hardware Description Language," IEEE, Tech. Rep. IEEE Std 1076-2000, 2000.

[8] "GTKWave 3.3 Wave Analyzer User's Guide," BSI, Tech. Rep., 2009.

[9] International Electrotechnical Commission, *Programmable controllers - Part 3: Programming languages, Ed 2.0, International Standard IEC 61131-3, IEC*, 2003.

[10] K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, and J. J. Hämäläinen, "Verification of Safety Logic Designs by Model Checking," in *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*, 2009.

[11] S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial on using PVS for hardware verification," in *Theorem Provers in Circuit Design (TPCD '94)*, ser. LNCS, R. Kumar and T. Kropf, Eds. Springer-Verlag, 1997, no. 901, pp. 258–279.

[12] M. Srivas, H. Rueß, and D. Cyrluk, "Hardware verification using PVS," in *Formal Hardware Verification: Methods and Systems in Comparison*, ser. LNCS, T. Kropf, Ed. Springer-Verlag, 1997, no. 1287, pp. 156–205.

[13] C. Berg, C. Jacobi, and D. Kröning, "Formal verification of a basic circuits library," in *Proc. of the IASTED International Conference on Applied Informatics, Innsbruck (AI 2001)*. ACTA Press, 2001.

[14] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *Proceedings of the IEEE International Symposium on Requirements Engineering*, 1993, pp. 126–133.

 ISBN: 978-1-61208-112-0

Sharing Ballistics Data across the European Union

Richard S Wilson
C_3RI
Sheffield Hallam University
Sheffield, United Kingdom
r.wilson@shu.ac.uk

Lukasz Jopek
C_3RI
Sheffield Hallam University
Sheffield, United Kingdom
l.jopek@shu.ac.uk

Christopher D Bates
C_3RI
Sheffield Hallam University
Sheffield, United Kingdom
c.d.bates@shu.ac.uk

Abstract - Across Europe police organisations are using numerous systems, both computerised and manual, to capture information about firearm crimes. The Odyssey platform intends to address this issue by providing police organisations with the ability to access ballistics data from other European law enforcement agencies. The Odyssey platform is a prototype system that has been developed to identify standards for the development of a European wide ballistics information system. In this paper, we outline the investigation tools found within the platform and discuss how these were developed. The prototype has been demonstrated to law enforcement communities across Europe and is in its final stages of development.

Keywords – ballistics; sharing; law enforcement; data mining; Europe.

I. INTRODUCTION

Police organisations across Europe deploy many different systems, both computerised and manual, to record information about crimes which involve the use of firearms. Ballistics crime is relatively rare – just 0.2% of all crimes in the UK involve firearms, but justifiably this is taken very seriously by Law Enforcement Agencies. The disparate systems used today are well suited to resolving crimes when they are committed locally, but problems arise when crossing jurisdictions. Although individual offences are unlikely to cross borders, guns and bullets are moved between European Countries [4]. Exchanging information so that, for example, guns can be tracked or patterns of usage followed has always been difficult. Information is often exchanged via the telephone, email or one-to-one meetings. Before information can be shared, investigators need to have some indication that the ballistic item is linked to another offence.

The Odyssey project addresses this problem by providing users with access to a plethora of information from investigations across Europe. Incidents are evaluated to find ones which are *similar* to the one under investigation. Detectives are then alerted to these similar incidents. The Odyssey platform provides police users with access to information such as expert and witness statements, images and videos, as well as details of bullets, cartridge-cases and firearms. All of this information is presented through directed graphs or a historical timeline view of an investigation. Odyssey improves crime resolution times by facilitating communication between experts.

In this paper we will describe the data structures which underpin Odyssey. We will provide an overview of some of the most recent developments in police information management systems followed by a description of the Odyssey platform prototype. The paper concludes with a discussion about the standards that were identified through the development of the prototype.

II. BEYOND STATE OF THE ART

Internationally, there are a number of crime database systems in use by different law enforcement agencies. Some of these systems include: COPLINK, NABIS (National Ballistics Intelligence Service), HOLMES (Home Office Large Major Enquiry System) 2 and I-24/7.

COPLINK is an information and knowledge management system aimed at capturing, accessing, analysing, visualising and sharing information between United States (US) law enforcement agencies. COPLINK comprises of two components COPLINK Connect (CC) and COPLINK Detect (CD).

CC is designed to integrate disparate heterogonous data sources, including legacy systems, to facilitate information sharing between police departments. CC provides police officers with access to one central data repository, which allows them to carry out four types of independent searches (person, vehicle, incidents or locations). In addition to this, police officers can carry out partial and phonetic based searching, access previous searches and upload images and documents.

CD expands the functionality of CC to automatically find associations within police databases. It is aimed at supporting detectives and crime analysts in finding associations between people, vehicles, incidents and locations. At present the system is able to find associations between individual entities, but is unable to map them onto a geographical map. The strength of an association is determined through the use of co-occurrence analysis and clustering. The system is able to search for meaningful terms in both structured (database tables) and unstructured (witness statements) data [1].

UK police forces have access to a number of independent database systems. These databases are used to record, monitor and manage offences in such areas as sex offences, gun crimes and major incident management.

 ISBN: 978-1-61208-112-0

NABIS provides ballistic examination services, for twenty UK based police forces, through three hubs, which are based in London, Birmingham and Manchester [7].

The database is fundamental to the service that NABIS provides. The NABIS database supports the recovery and analysis of ballistic items. Associations between the ballistic information, people, objects and events are formed to create tactical intelligence. Security is implemented on a per-user basis so that users are only able to access information that is relevant to their role [6].

HOLMES2 is an information management system which assists police forces in the investigation of serious crimes, such as serial murders, large scale fraud and major disasters. HOLMES2 lets police forces share information and identify links between independent incidents. HOLMES2 is an Oracle based database that resides on a UNIX system [9].

Across Europe, Interpol has implemented the I-24/7 global police communications system that allows 188 member countries to share information about criminals and criminal activities. I-24/7 provides member countries with 24 hour access to a vast array of police information. Such information is related to suspected terrorists, wanted persons, fingerprints, DNA profiles and stolen vehicles. In addition to this, the I-24/7 system provides each member country with access to other member's national databases [4].

When ballistics crimes are investigated, recovered items such as guns, bullets or cartridge-cases can be compared by forensic specialists. Test-fired bullets are examined for a range of marks made as they pass down the barrel of the gun [2]. By comparing the marks on different bullets a trained examiner can determine if two bullets come from the same weapon. The process of examining bullets under a microscope is time-consuming and difficult. A number of systems such as IBIS, Papillon and EVOFINDER have been built to automate the evaluation process; however, these systems do not inter-operate [11]. A bullet scan from one manufacturer's system cannot be used on anothers.

The Odyssey project is helping to define standards for sharing ballistics data between systems across Europe. Figure 1 provides some indication of the different ballistic matching systems in place across Europe.

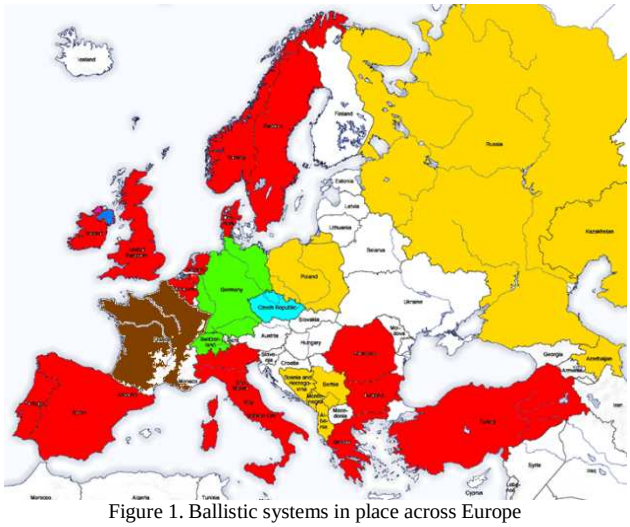


Figure 1. Ballistic systems in place across Europe

This is in contrast to the United States of America (USA), where IBIS has first mover advantage and has developed a centralised IBIS system. Furthermore, Odyssey is different to NABIS and the I-24/7 database currently in place in the UK and across Europe, as the NABIS database is specifically designed to manage the examination of ballistic items. The I-24/7 database has a European-wide database which largely retains information related to the individual. The I-24/7 system doesn't integrate data from ballistic systems due to the heterogeneous nature of data. Odyssey is different to other US and European police systems, as Odyssey seamlessly combines relational querying and data mining results from multiple different domains.

III. INTELLIGENT SEARCHING

Odyssey retains data within two main databases - a local database and a central database called CEON. The local database is maintained by the individual police organisation, whilst CEON holds ballistics data uploaded by police organisations across Europe. The data within the local database is replicated within CEON through an XML transfer structure, which is updated on a daily basis. The central database is interrogated, using supervised and unsupervised data mining techniques, to find associations with other ballistic incidents that have been committed across Europe. It is anticipated that the central database will handle at least six hundred thousand new ballistic incidents per year. This is based on the average number of firearm offences committed in the UK between 1999 and 2009, multiplied by the number of member states [3]. Each police organisation with a related ballistic incident is alerted to the fact that an association has been found. An alert is generated when a potential match is identified in the central database. An automated message is then sent back to the user to inform them that a potential match has been found. This message is delivered to an e-mail type inbox within the GUI (Graphical User Interface). In addition to this, the platform

 ISBN: 978-1-61208-112-0

allows individual police organisations to restrict access at the individual ballistic incident level to their data.

Extracting intelligence using the GUI is achieved in two stages. First, the user *defines the search item* then they *browse the results*, which can then lead to further searches. Figure 2, below, shows a search that has been carried out through the Odyssey GUI.

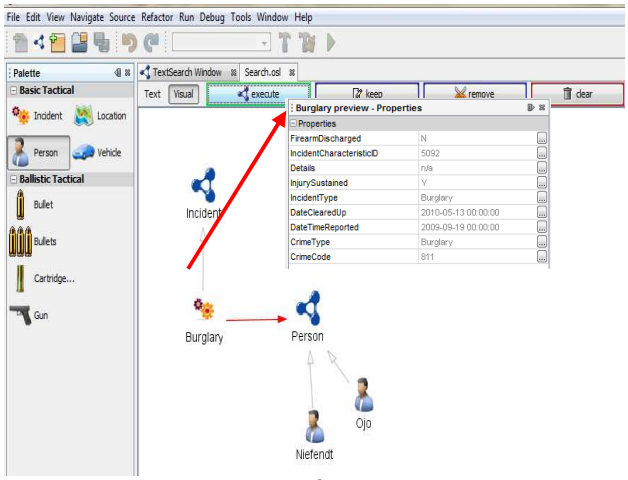


Figure 2. Odyssey GUI

First, the user selects a *basic or ballistic tactical item* and refines their search in the *search properties*. The user then has the option to add further tactical items and link them to the results of the previous search. Searches are refined, in the platform, through the search properties which converts the query into OSL (Odyssey Semantic Language). Querying with OSL allows the user to access information directly from the database and combines it with intelligence from the data mining backend. Below is an example of a simple OSL query generated through the GUI.

QUERY person WHERE weapon HAS VALUE Sig Sauer P238 AND country HAS VALUE France

Expanding the query to determine similar ballistic incidents by using the data mining backend and within a confidence limit is expressed as follows:

QUERY person WHERE weapon HAS VALUE Sig Sauer P238 AND country HAS VALUE France WITH CONFIDENCE > 0.7

Having identified the high level information, through the GUI, the user is able to drill down into it. This data is contained within the Odyssey Evidence Repository (OER). The OER provides access to some of the physical artefacts collected through the investigation process, including: images, videos and expert and witness statements.

In addition to this, Odyssey also provides historical data views of related incidents. These views are presented to the user as a timeline. This promotes transparency as the user is able to identify any updates that have occurred as the investigation has progressed, which includes the identification of additional ballistic incidents and changes made to the underlying information.

The intelligent searching services found within Odyssey's arsenal are vital to facilitating the resolution of ballistic incidents committed across Europe.

IV. ODYSSEY HUB

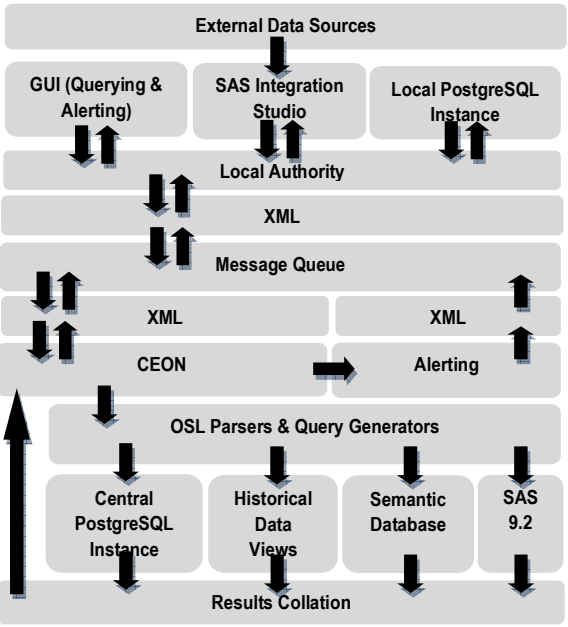


Figure 3. Architecture of the Odyssey Platform

The Odyssey platform is pragmatic software that can run on any standard off the shelf system. It was developed through the integration and repurposing of open source and common off the shelf software applications. These included Java, NetBeans, PostgreSQL, Antlr and SAS. Whilst the platform currently utilities SAS for the integration and mining of data the software could quite easily be replaced with open sourced software such as Python and WEKA. Figure 3 above provides an overview of the architecture of the Odyssey platform.

External data, from European police organisations, is extracted and transformed using SAS Integration Studio. This data is then loaded into the local authority database, which is a replication of the central database. These databases were developed using PostgreSQL and are based on the database structures currently in place at NABIS, Europol and CiFEx (Centre for Information on Firearms and Explosives) – CiFEx are ballistic experts in the UK. This helped to focus the development of the databases towards ballistic incidents. This allows the user to manage their own data and helps to insure the smooth transfer and integration of data between the local and central databases. There are over fifty tables in the database that are linked using Object Identifiers (OIDs). An OID is automatically generated when data is uploaded into the database, which is unique across

 ISBN: 978-1-61208-112-0

the entire instance of the database. Referential integrity between items is not maintained by the databases. This causes scalability problems as associations between data items have to be made outside of the database. This issue is further compounded by the predicted size of the database, given the number of ballistic incidents per year. Indeed, as the platform stores images, videos and statements, the anticipated size of the database is expected to grow into the terabytes. A binary items table was therefore implemented within the database which works as a file system that points to the stored images, videos and statements.

Data is uploaded into the local component through the GUI. This is achieved through a number of automatically generated SQL insert statements, which is then speeded up through the use of OIDs. It is at this point where the user is able to specify the initial sharing permissions. This is done through the GUI, by selecting the *ALLOW* or *DENY* option followed by the *user specified options*. The *user specified options* allow the user to grant or restrict access at five levels: country, region, police organisation, department and user. These options are then changed into numerical codes that are later translated, along with the insert statements, into OSL. These codes are then retained within the permissions table of the authorisation database. Data is loaded into the local database through the GUI, see Figure 4.

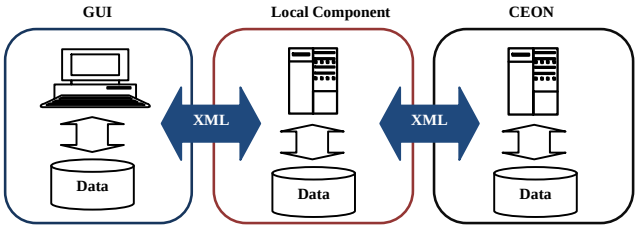


Figure 4. Odyssey XML Transfer Structure

The GUI translates the OSL into SQL which is used to pass data to the local database as XML. The data is then transferred to CEON using the same XML structure, where it is interrogated for associations. Through the use of SAS integration studio, we have demonstrated that it is possible to extract, transform and load data from ballistic and incident systems into a relational database structure.

When a match is identified an alert is then triggered and sent back to the local component. If permission is granted to view the data, the related data is also sent back to the local users through the XML structure. Following on the data is then retained in the cache of the local component where it is serialised within the platform. The platform also allows the user to modify their data, which is done through the translation of SQL update or delete statements into OSL. The process of uploading and returning a result (alert and data) is completed in twenty four hours. The user is restricted from directly changing the data in the central database due to batch data mining and user processing. A change to the central database would occur within twenty four hours of a user modifying the local database. When the update has occurred the users receive an alert that asks them to re-run their query.

The shared data is based on the options specified by the user, which is retained in the permissions table within the authorisation database. The shared data is sent back to the local component through the XML structure where it is serialised within the platform. It is presented to the user as a related result within the GUI. Figure 5 below shows the results of an Odyssey search.

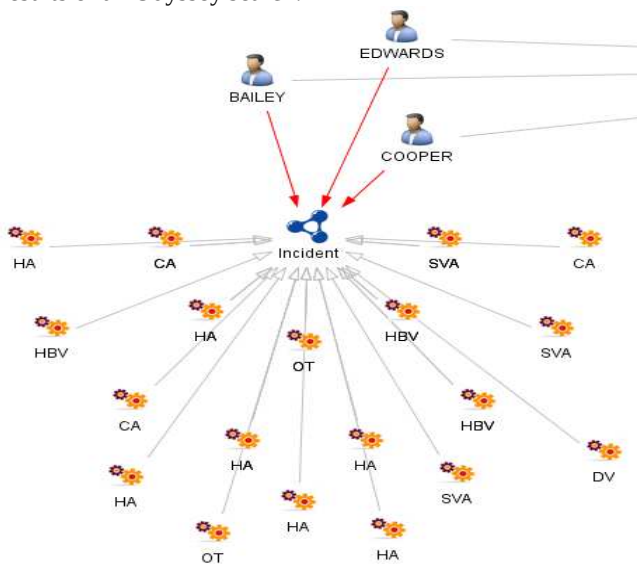


Figure 5. Search Result

This shows the related crimes committed by Edwards, Cooper and Bailey.

Querying the database is carried out through using the GUI, or by creating OSL with the built in assisted functions. Querying with the GUI, the user drags a tactical item from the tool bar and specifies the search options in the search properties. The user specified options are transformed into OSL using the backend semantic engine. OSL is a grammar that was developed in Antlr, which translates the search into SQL and is specifically designed to hide the underlying structure of the database from the user. The semantic language was created through understanding how police investigators communicate and think during investigations and by capturing domain specific knowledge about the meaning of police language. Below is an example of OSL and its translation into SQL:

```
QUERY ballistic incident WHERE weapon_manufacturer
HAS VALUE Sig Sauer AND victim_gender HAS VALUE
female
```

```
SELECT *
FROM odyssey.ballistic_incident
LEFT JOIN ballistic_incident_has_recovered_firearm ON
(ballistic_incident_has_recovered_firearm.recovered_firear
m_oid = ballistic_incident.oid)
LEFT JOIN ballistic_incident_has_recovered_firearm ON
```

 ISBN: 978-1-61208-112-0

```
(ballistic_incident_has_recovered_firearm.recovered_firear
m_oid = recovered_firearm.oid)
LEFT JOIN ballistic_incident_has_case ON
(ballistic_incident_has_case.ballistic_incident_case_oid =
ballistic_incident.oid)
LEFT JOIN ballistic_incident_has_case ON
(ballistic_incident_has_case.ballistic_incident_case_oid =
case.oid)
WHERE case.gender_of_victim = "female" AND
recovered_firearm.manufacturer = "Sig Sauer";
```

If the user specifics a confidence limit then Odyssey will return the results ranked in descending order of confidence. The confidence is calculated on a daily basis, by applying prebuilt algorithms to a merged copy of the central database tables the Odyssey data mining mart (DMM). The data is merged using SAS Data Integration Studio and is scored using prebuilt algorithms that were created in SAS Enterprise Miner. The score is returned to the ballistic incident table by matching the OID from the DMM with the OID in the ballistic incidents table. The ballistic incidents table is then sorted in descending order of score. The following is an example of an OSL query with an expressed confidence limit, which is translated into SQL:

```
QUERY ballistic incident WHERE weapon_manufacturer
HAS VALUE Sig Sauer AND victim_gender HAS VALUE
female WITH CONFIDENCE > 0.7
```

```
SELECT *
FROM odyssey.ballistic_incident
LEFT JOIN ballistic_incident_has_recovered_firearm ON
(ballistic_incident_has_recovered_firearm.recovered_firear
m_oid = ballistic_incident.oid)
LEFT JOIN ballistic_incident_has_recovered_firearm ON
(ballistic_incident_has_recovered_firearm.recovered_firear
m_oid = recovered_firearm.oid)
LEFT JOIN ballistic_incident_has_case ON
(ballistic_incident_has_case.ballistic_incident_case_oid =
ballistic_incident.oid)
LEFT JOIN ballistic_incident_has_case ON
(ballistic_incident_has_case.ballistic_incident_case_oid =
case.oid)
WHERE case.gender_of_victim = "female" AND
recovered_firearm.manufacturer = "Sig Sauer" AND
ballistic_incident.score > 0.7;
```

Documents are stored in a separate PostgreSQL database as blobs. Using local compatible software, police experts are able to share information across the platform. It helps to facilitate communication between experts, as they are able to access documents from other law enforcement agencies, along with their contact details. Such communication is believed to be vital in helping to resolve crimes that have been carried out in different locations [8].

Historical data are presented to the user as a timeline of events, which shows any updates to the incidents and any associated incidents that the user has the authority to view. An update is defined as the user deleting, updating or inserting data into an existing ballistic incident. The timeline of historical events is created through using another PostgreSQL database, which stores the delta along with the author and date/time of the change. The updated item is identified in the historical database by the OID of the initial item, combined with an SQL timestamp. The approach adopted by the platform is quite unique as other historical databases that use OIDs with timestamps have retained the historical data in the same database. The main argument to support this is that consistency and speed of data retrieval is maintained when the historical data is stored in the same database [10]. Whilst speed of retrieval and consistency of data items is important to the platform, the initial aim is to return the latest view of the ballistic incident to the user. The speed in which historical data is returned to the user is however improved, through the use of the OID and timestamp, as the requested data is easily ranked through sorting these values in ascending order.

V. TOWARDS COMPLETION

Further development of the platform will focus on the automatic identification of key words in expert and witness statements – the Odyssey Statement Miner (OSM). This will present the user with a list of key words, ranked in order of occurrence, from the associated statements. We expect that this will help the police experts to assess the usefulness of the documentation.

The OSM will be developed using SAS Text Miner in conjunction with a bespoke java programme, which removes any conjunctive adverbs from the statements. SAS Text Miner will be used to identify any words that occur more than three times within a statement. From this a dictionary of words, along with the frequency of occurrence will be created. This will then be presented to the user as a list of words in descending order of frequency. It is hoped that this will help the user to decide whether or not to view the statement.

It is noticeable and disappointing that no EU-wide standards exist for secure police data systems. Odyssey has been able to demonstrate that widely available open software can be repurposed easily to build such systems. The project will be recommending a list of such technologies.

VI. CONCLUSION

This paper provides an overview of some of the most current crime information management systems in place in the USA, UK and across Europe. It highlights that there is a need for a platform which combines data from the different ballistic systems currently in place across Europe. An overview of the functions found within the prototype are also discussed, which focuses upon accessing, loading,

 ISBN: 978-1-61208-112-0

sharing and querying of data. The development of the investigation tools within the platform are also explained and future work in relation to the completion of the prototype is outlined. The paper concludes with a brief discussion regarding the standards that will be recommended at the end of the project.

REFERENCES

[1] Chen, H., Zeng, D., Atabakhsh, H., Wyzga, W., and Shroeder, J. (2003), *COPLINK Managing Law Enforcement Data and Knowledge*, Communications of the ACM, January 2003. Vol. 46, No.1. pp. 28-34.

[2] Bundeskriminalamt. (2004), Firearm Type Determination, Available [online]: https://www.forensic-firearms.bund.de/ [17/06/2010].

[3] Gun Control Network. (no date), Firearm Offences – England and Wales, Available [online]: http://www.gun-control-network.org/GF05.htm [17/06/2010].

[4] Interpol. (no date,) Connecting Police: I-24/7, Interpol, Available [online]: http://www.interpol.int/Public/ICPO/FactSheets/GI03.pdf [17/06/2010].

[5] Leon, F.P. (2005) 'Automated comparison of firearm bullets', Forensic Science International, Vol. 156, February, pp. 40-50.

[6] National Ballistics Intelligence Services. (no date), *Database*, Available [online]: http://www.nabis.police.uk/database.asp [16/06/2010].

[7] Sims, C. (2010), National Ballistics Intelligence Service Update Report, West Midlands Police Authority, Available [online]: http://www.west-midlands-pa.gov.uk/documents/committees/public/2010/12_PerfandOps_22April2010_National_Ballistics_Report.pdf , [16/06/2010].

[8] Travis, J. (1998), 'Informal Information Sharing Among Police Agencies', National Institute of Justice, December 1998.

[9] Unisys. (2007), What is Holmes 2?, Unisys, Available [online]: http://www.holmes2.com/holmes2/whatish2/ [17/06/2010].

[10] Van Oosterom, P. (no date), '*Maintaining Consistent Topology including Historical Data in a Large Spatial Database*', Cadastre Netherlands. pp. 327-336.

[11] Yates, S., Jopek, L. Johnson Mitchell, S., and Wilson, R. (2009), '*Semantic Interoperability between Ballistic Systems through the Application of Ontology*', IADIS WWW/ Internet Conference. Vol. 2 pp. 153-157.

 ISBN: 978-1-61208-112-0

Euclides – A JavaScript to PostScript Translator

Martin Strobl, Christoph Schinko
Institut für ComputerGraphik und WissensVisualisierung
Technische Universität Graz, Austria
{ *m.strobl, c.schinko* }*@cgv.tugraz.at*

Torsten Ullrich[1], Dieter W. Fellner[2]
[1,2] *Fraunhofer Austria, Graz, Austria*
[2] *Fraunhofer IGD & TU Darmstadt, Germany*
torsten.ullrich@fraunhofer.at
d.fellner@igd.fraunhofer.de

***Abstract*—Offering an easy access to programming languages that are difficult to approach directly dramatically reduces the inhibition threshold. The Generative Modeling Language is such a language and can be described as being similar to Adobe's PostScript. A major drawback of all PostScript dialects is their unintuitive reverse Polish notation, which makes both - reading and writing - a cumbersome task. A language should offer a structured and intuitive syntax in order to increase efficiency and avoid frustration during the creation of code. To overcome this issue, we present a new approach to translate JavaScript code to GML automatically. While this translation is basically a simple infix-to-postfix notation rewrite for mathematical expressions, the correct translation of control flow structures is a non-trivial task, due to the fact that there is no concept of "goto" in the PostScript language and its dialects. The main contribution of this work is the complete translation of JavaScript into a PostScript dialect including all control flow statements. To the best of our knowledge, this is the first *complete* translator.**

***Keywords*-PostScript; JavaScript; translator; transpiler**

I. MOTIVATION

The language PostScript [1] by JOHN WARNOCK and CHARLES GESCHKE at Adobe Systems is a dynamically typed concatenative programming language which is known for its use as a page description language for desktop publishing. Beginning in the 1980s PostScript (PS) and its descendants, namely the Portable Document Format (PDF) [2], are still the standard for electronic distribution of final documents for publication. Besides desktop publishing, the programming language PostScript has been used in display [3] and window systems [4] [5] as well. Nowadays it has its revival in procedural 3D modeling. The Generative Modeling Language (GML) [6] is a programming language based on PostScript. It follows the "Generative Modeling" paradigm [7], where complex data sets are represented by algorithms and parameters rather than by lists of objects. With ever increasing computing power becoming available, generative approaches [8] [9] become more important since they trade processing time for data size. At run time the compressed procedural description can be "unfolded" on demand to very quickly produce amounts of meshes, textures, etc. that are several orders of magnitude larger than the input data.

A. PostScript in 3D

GML is very similar to Adobe's PostScript, but without any of the 2D layout operators. Instead, it provides a number of operators for creating 3D models.

PostScript and GML are interpreted, stack-based languages with strong dynamic typing, scoped memory, and garbage collection. The language syntax uses reverse Polish notation, which makes the order of operations unambiguous, but reading a program requires some practice, because one has to keep the layout of the stack in mind [10]. Most operators and functions take their arguments from the stack, and place their results onto the stack. Literals (numbers, strings, etc.) have the effect of placing a copy of themselves on the stack. Sophisticated data structures can be built on array and dictionary types, but cannot be declared to the type system. They remain arrays and dictionaries without further type information.

B. JavaScript

PostScript programs are typically not produced by humans, but by other programs, e.g., printer drivers and devices. However, it is possible to write computer programs in PostScript just like in any other programming language.

In order to simplify the GML development and 3D design process, 3D modeling tools (Autodesk Maya, 3ds Max, etc.) can be used. Unfortunately, these tools do not preserve the generative nature. They can only export the generated result.

Encoding shape as program code clearly has the greatest flexibility, but up to now it requires coding (programming), which is usually done by humans. To accelerate the GML creation process and to increase efficiency we propose a JavaScript (JS) translator to GML. JS is a structured programming language featuring a rather intuitive syntax, which is easy to read and to understand. It also incorporates features like dynamic typing and first-class functions. The most important feature of JS is that it is already in use by many non-computer scientists, namely designers and creative coders [11]. JS and its dialects are widely used in applications and on the Internet: in Adobe Flash (called ActionScript), in interactive PDF files, in Microsoft's Active Scripting, in VRML97, etc. Consequently, a lot of documentation and tutorials to introduce the language exist [12]. In

 ISBN: 978-1-61208-112-0

order to be used for procedural modeling, JS is missing some functionality, which we added via libraries.

C. Overview

Euclides (www.cgv.tugraz.at/euclides) is a transpiling framework written in Java. *Euclides* will also translate an input JS program to Java or documents its structure in HTML. It features its own integrated development environment (IDE), from which one can transpile to the supported target languages. Our translation to GML makes the rich feature set of the Generative Modeling Language accessible to a wider range of users, because it hides much of the complexity involved in writing GML programs.

In the subsequent sections, we explain the JS to GML translator. Having parsed JS using ANTLR [13], the translation process begins with a correct (according to ECMAScript, ECMA-262, ISO/IEC 16262) Abstract Syntax Tree (AST). Then we show how data types, functions and operators are translated and explain the control flow.

II. Data Types

In JS each variable has a particular, dynamic type. It may be `undefined`, `boolean`, `number`, `string`, `array`, `object`, or `function`. GML also has a dynamical type system. Unfortunately, both type systems are incompatible to each other. Therefore, translating JS data types to GML poses two particular problems: On the one hand, the dynamic types must be inferred at run time. On the other hand, GML's native data types lack distinct features needed by JS. GML-Strings, for example, cannot be accessed character-wise. We solved these problems by implementing JS-variables as dictionaries [6] in GML. Dictionaries are objects that map unique keys to values. These dictionaries hold needed metadata and type information as well as methods which emulate JS behavior. As we will show later, we will utilize GML's dictionaries for scoping as well.

The system translation library for GML (which every JS-translated GML program defines prior to actual program code) contains the function `sys_init_data`, which defines an anonymous data value in the sense of JS data.

```
/sys_init_data {
   dict begin
   /content dict def
   content begin
      /type  edef
      /value edef
      /length { value length } def
   end
   content
   end
} def
```

`sys_init_data` opens a new variable-scope by defining a new, anonymous dictionary and opening it. In this new scope, another newly created dictionary is defined by the name `content`. This content-dictionary receives three entries: `type`, `value` and the method `length`. Each entry value is taken from the top of GML's stack. The newly created dictionary is then pushed onto the stack and the current scope is destroyed by closing the current dictionary, leaving the anonymous dictionary on the stack. In GML notation, a JS-variable's content is defined by pushing the actual value and a pre-defined constant to identify the type of the variable (such as `Types.number`, `Types.array`, etc.) onto the stack, and calling `sys_init_data`. The translator prefixes all JS-identifiers with `usr_` (in order to ensure that all declarations of identifiers do not collide with predefined GML objects) and uses the following translations:

Undefined: Variables of type `undefined` result from operations that yield an undefined result or by declaring a variable without defining it. `var x;` leads to `x` being of type `undefined`. It is translated to

```
/usr_x Nulls.Types.undefined
Types.undefined sys_init_data def
```

Boolean: In JS, boolean values are denoted by the keywords `true` and `false`. The translation simply maps these values to equivalent numerical values in GML, which interprets them. The JS-statement `var x = true;` becomes

```
/usr_foo 1 Types.bool sys_init_data def
```

Number: All JS numbers (including integers) are represented as 32-bit floating point values. As GML stores numbers as 32-bit floats internally as well, we simply map them to GML's number representation. For the sake of completeness, `var x = 3.14159;` is translated to

```
/usr_x 3.14159 Types.number sys_init_data def
```

String: Although GML does support strings, they cannot be accessed character-wise. We cope with this limitation by defining strings as GML-arrays of numbers. Each number is the Unicode of the respective character. As GML allows to retrieve and to set array-elements based on indexes, this approach meets all conditions of JS-strings. The statement `var x = "Hello World";` becomes

```
/usr_x [ 72 101 108 108 111 32 87 111 114 108 100 ]
       Types.string sys_init_data def
```

Array: JS arrays allow to hold data with different types, the array's contents may be mixed. This behavior is in line with GML. The JS-example `var x = [true, false, "maybe"];` has a straightforward translation:

```
/usr_x [ 1 Types.bool sys_init_data
         0 Types.bool sys_init_data
         [109 97 121 98 101] Types.string sys_init_data ]
       Types.array sys_init_data def
```

Object: In JS an object consists of key-value-pairs, e.g., `var x = { x: 1.0, y: 2.0, z: 42};` This structure is mapped to nested GML-dictionaries. The value of a variable's content is a dictionary of its own. This dictionary contains the entries corresponding to JS-object's members, which are also defined as variable contents.

The example above defines a JS-object of name `x` with key-value-pairs `x` to be 1, `y` to be 2, and `z` to be 42:

 ISBN: 978-1-61208-112-0

```
/usr_x dict begin
       /obj dict def obj begin
         /usr_x  1.0 Types.number sys_init_data def
         /usr_y  2.0 Types.number sys_init_data def
         /usr_z 42.0 Types.number sys_init_data def
       end obj
       Types.object sys_init_data end def
```

Opening an anonymous dictionary creates a new scope. In this scope, a dictionary is created and bound to the name `/obj`. It is then opened and its members are defined, just like anonymous variables would be. The object dictionary is then closed, put on the stack, and used to define an anonymous variable. The enclosing anonymous scoping dictionary is then closed and simply discarded.

JS objects may hold functions. Our translator *Euclides* handles JS object-functions like ordinary functors (next subsection) and assigns their internal name to a key-value-pair.

Function: JS has first-class functions. Therefore, it is possible to assign functions to variables, which can be passed as parameters to other functions, for example. In the following example, a function `function do_nothing() {}` is declared and defined. Afterwards, it is assigned to a variable `var x = do_nothing;`. If we abstract away from the translation of the function `do_nothing`, the statement `var x = do_nothing;` becomes:

```
/usr_do_nothing {
   %% ... definition of function omitted ...
} def

/usr_x /usr_do_nothing Types.function sys_init_data def
```

In JS, `x` can now be used as a functor, which acts the same ways as `do_nothing`. Because such functors can be reassigned, it is necessary to handle functor calls (`x()`) differently than ordinary function calls (`do_nothing()`). In this situation *Euclides* creates a temporary array, which contains the functor parameters and passes this array as well as the variable referencing the function name to a system function `sys_execute_var`. This system function resolves the functor and determines the referenced function, unwraps the array and performs the function call.

III. FUNCTIONS

A. Translation of JS Functions

In GML, functions are defined using closures, such as `/my_add { add } def`. If this function `my_add` is executed, the closure `{ add }` is put onto the stack, its brackets are removed, and the content is executed.

To execute a GML function, its parameters need to be put on the stack prior to the function call: `1.0 2.0 my_add` The resulting number `3.0` will remain on the stack. Please note, that GML functions may produce more than one result (left on the stack) at each function call. This allows to define functions with more than one result value. Following JS, called functions return only one value by convention. The number and names of function parameters are known at compile time. Only functors (referenced functions stored in variables) may change at run time and cannot be checked ahead of time.

Translated functions and parameters are named just like their JS-counterparts (except for their `usr_` prefix).

B. Scopes

As JS uses a scoping mechanism different to GML, it has to be emulated. This is a rather difficult task, which has to take the following properties of JS scopes into account.

- JS functions may call other functions or themselves.
- Called functions may declare the same identifiers as the calling functions.
- Within functions other functions may be defined.
- Blocks might be nested inside functions, redefining symbols or declaring symbols of the same name.

The translator uses GML's dictionary mechanism to emulate JS-scopes. A dictionary on the dictionary stack can be opened and it will take all subsequent assignments to GML-identifier (variables). Since only the opened dictionary is affected, this behavior is the same as the opening and closing scopes in different scoped programming languages, such as C or Java.

Thus an assignment `/x 42 def` can be put into an isolated scope by creating a dictionary (`dict`), opening it (`begin`), performing the assignment, and closing the dictionary (`end`). The following example shows how such GML scopes can also be nested:

```
dict begin
   /x 3.141 def          %% x is 3.141
   dict begin            %%
      /x 4 def           %% x is 4.0
   end                   %% x is 3.141
end                      %% x is unknown
```

As noted before, JS supports redefinition of identifiers that were declared in a scope below the current one. Fortunately, GML exhibits just the same behavior when reading out the values of variables/keys from dictionaries of the dictionary stack. Consequently, the following example works as expected.

```
dict begin
   /x 42 def
   dict begin
      /y x 1 add def     %% y is now 43
   end
end
```

However, assignments to variables have to be handled differently in GML. The Generative Modeling Language does not distinguish between declaration and definition, any declaration must be a definition and vice versa.

The translator solves this problem. It uses a system function (which is included into all translated JS sources automatically) called `sys_def`. This function applies GML's `where` operator to the dictionary stack in order to find the uppermost dictionary, where the searched name is defined.

 ISBN: 978-1-61208-112-0

The operator returns the reference to the dictionary, in which the name was found.

C. Control Flow for Functions

The Generative Modeling Language and all PostScript dialects lack a dedicated jump operation in control flow. Imperative functions often require the execution context to jump to a different point in the program at any time - and to return from there as well.

Fortunately, GML provides an exception mechanism. A GML exception is propagated down GML's internal execution stack until a `catch` instruction is encountered. In this way it overrides any other control structure it encounters. We use GML's exception mechanism to jump outside a function as illustrated in the following empty function skeleton:

```
/usr_foo {
   dict begin
   /return_issued 0 def
   { dict begin
     %% ... function body omitted ...
     end }
   { /return_issued 1 def }
   catch

   return_issued not
   { Nulls.Types.undefined
     Types.undefined sys_init_data } if
   end
   sys_exception_return_handler
} def
```

In this empty skeleton, the function opens a new anonymous scope. Inside this scope `dict begin ... end` the local identifier `/return_issued` is set to `0`. Afterwards a GML try-catch-statement { *try_block* } { *catch_block* } `catch` contains the JS-function implementation. In this translation, the catch block redefines `/return_issued` to `1` to indicate that a JS `return` statement has been executed in the function body. JS functions without any `return` statement, automatically return `null` resp. in GML `Nulls.Types.undefined Types.undefined sys_init_data`. A corresponding JS-return statement, e.g., `return 42;`, is translated to

```
   42.0 Types.number sys_init_data end throw
```

In this example, the number `42.0` is put onto the stack. The actual function body's scope is closed `end`, and the `throw` operator is applied. The distinction of whether the end of the function body was reached by normal program flow or via a return statement determines, if a return value needs to be constructed (`null`) and put onto the stack.

Parameters to functions are simply put on the stack. The function body retrieves the expected number of parameters and assigns them to dictionary entries of the outer scope defined in the function translation. A complete example of a translated JS-function shows the interplay of all mechanisms. The simple JS-function

```
function foo(n) { return n; }
```

is translated to

```
/usr_foo {
   dict begin
   /usr_n edef
   /return_issued 0 def
   { dict begin
     usr_n
     end
     throw
     end }
   { /return_issued 1 def }
   catch

   return_issued not
   { Nulls.Types.undefined
     Types.undefined sys_init_data } if
   end
   sys_exception_return_handler
} def
```

A function call, for example `foo(3)`, yields the translation `3.0 Types.number sys_init_data usr_foo`. If we assign the function `foo` to a variable `foo_functor`, the calling convention in GML would change significantly.

```
/usr_foo_functor /usr_foo Types.function sys_init_data def
```

is called via

```
[ 3.0 Types.number sys_init_data ]
usr_foo_functor sys_execute_var
```

and represents the JS call `foo_functor(3.0);`

D. Exceptions

The language JS supports throwing exceptions; e.g., `throw "Error: unable to read file.";`. Its syntax is similar to a return statement. To implement such behavior, we also use GML's exception handling mechanism. The *Euclides* translator adds a call to the predefined system function `sys_exception_return_handler` at the end of each translated function (see example above).

Throwing an exception in JS translates into a global GML variable `exception_thrown` being set to 1, closing the current dictionary and calling GML's `throw`. The `sys_exception_return_handler` will check if an actual exception is being thrown, and if so, calls `throw` again. A catch-block inside a JS program would set `exception_thrown` to 0.

IV. Operators

The evaluation of expressions demands variables to be accessed. While GML provides operators that operate on their own set of types, they obviously cannot be used to access the translated/emulated JS-variables. For this reason, the *Euclides* translator automatically includes a set of predefined GML functions that substitute operators defined in JS.

A. Value Access

Performing the opposite operation to `sys_init_data`, `sys_get_value` will retrieve the data saved in a JS-variable resp. its GML-dictionary. For example, to retrieve `v.value` the function `sys_get_value` is applied to `v`.

```
/sys_get_value { begin value end } def
```

 ISBN: 978-1-61208-112-0

B. Element Access

The system function `sys_get` implements string, array and object access. Applied to a string / an array `Arr` and index `k`, it will return the element `Arr[k]`. If its parameters are an object `Obj` and an attribute `name`, the function `sys_get` executes `Obj.name`. This may result in a value, which is put on the stack or in a function, which is called. Conforming to JS, it returns JS `undefined` for any requested elements that do not exist.

```
/sys_get {
   dict begin
   /idx exch def /var exch def

   var.type Types.string eq {
      %% ... handling strings ...
   } if

   var.type Types.array  eq {
      %% ... handling arrays  ...
   } if

   var.type Types.object  eq {
      var sys_get_value idx known 0 eq {
         %% return null, if element doesn't exist
         Nulls.Types.undefined
         Types.undefined sys_init_data
      } if
      var sys_get_value idx known 0 ne {
         %% access element
         var sys_get_value idx get
      } if
   } if
   end
} def
```

Analogous to `sys_get`, `sys_put` inserts data into strings and arrays, or defines members of objects. If `sys_put` encounters an index k that is out of an array's range, the array is resized and filled with JS `undefined`s.

C. Functors

The already mentioned routine `sys_execute_var` inspects a given variable. If it is a function, it will retrieve the array supplied to hold all parameters and execute the function. The dynamic binding of functions to variables requires to consider two situations at run time: The functor receives the correct amount of parameters for its function, or the number of parameters does not correspond to the referenced function. In the later case, the function is not called and `null` is returned instead.

At compile time, a function is defined to expect a concrete number of parameters. This information is kept to perform parameter checks at run time. In this way, the correct number of parameters for all functors can be determined any time.

D. JS built-in Operators

To illustrate the translation of relational, arithmetical or bit-shift operators defined by JS, we discuss the equal operator `==`. It is (like all such operators) mapped to a corresponding routine `sys_eq`. Depending of the operands' types it delegates the comparison to subroutines such as `bool_eq`, `string_eq` or `array_eq` that perform the actual comparison. If the types and the values do match, `sys_eq` directly returns the JS-value `true`. If types do not match, the variable is converted to the type of the respective operand, as specified by JS, and then compared.

V. Control Flow

A. Conditional Statement

The JS if-then-else statement corresponds one-to-one to the same GML statement. Consequently, the conditional expression is translated straightforwardly. Using the expression mapping introduced in the previous section (e.g. `sys_eq` implements the equality operator), the JS statement `if(a == b) { c = a; } else { c = b; }` is translated info:

```
%% if (a==b)
usr_a usr_b sys_eq sys_get_value
{ %% then:
   dict begin {
      dict begin
         /usr_c usr_a sys_def
      end
   } exec end
}
{ %% else:
   dict begin {
      dict begin
         /usr_c usr_b sys_def
      end
   } exec end
} ifelse
```

The exec-statements (and their closures) stem from the fact that both sub-statements, the then-part and the else-part, are statement blocks `{ ... }`. These blocks are executed within their own, new scopes.

B. Loops

GML supports different types of looping control structures, which have similar names to JS-loops (e.g., both languages have a for-loop). However, the GML counterparts have different semantics (e.g., GML's for-loop has a fixed, finite number of iterations, which is known before execution of the loop body, whereas JS-loops evaluate the stop condition during execution, which may result in endless loops). The *Euclides* translator uses the GML `loop` mechanism, which is an infinite loop that can be quit using the `exit` operator.

An important problem is that control structures such as for, while and do-while are not only controlled by the loop's stop condition, but also by JS statements such as `continue` and `break` within the loop body (besides `return` and `throw` as mentioned before). The statement `break` immediately stops execution of the loop and leaves it, whereas `continue` terminates the execution of the current loop iteration and continues with the next iteration of the loop. Therefore, we translate an empty while loop `while(false) { ... }` to

```
{  /continue_called 0 def
   {  0 Types.bool sys_init_data
      sys_get_value not { exit } if
      {  dict begin
         %% ... loop body omitted ...
         end
```

 ISBN: 978-1-61208-112-0

```
      } exec
   } loop
   continue_called not { exit } if
} loop
```

GML's `exit` keyword terminates the current loop. This behavior is leveraged by the *Euclides* translator to implement `break` and `continue`. The translation uses two nested loops that will run infinitely.

Prior to the begin of the inner loop `/continue_called` is set to `0`. At the top of the inner loop, the loop condition is tested. If the condition evaluates to `false`, the inner loop is exited using GML's `exit`. Otherwise a new scope is created and the loop-statement executed within that scope.

During loop iterations, there are three scenarios under which a loop can terminate:

1) If the loop condition is met: When the condition evaluates to `false`, the inner loop is exited. Since `continue_called` is not set to `true`, the outer loop will terminate as well.
2) If the loop body encounters JS `break` (resp. GML `exit`): Again, the inner loop is left. `continue_called` will not be set to `true`, hence the outer loop will also terminate.
3) If the function returns: GML's exception throwing mechanism will unwind the stack until the catch-handler at the end of the function is encountered.

If the loop body encounters a JS-`continue` statement, `continue_called` will be set to `true` and the GML `exit` command will immediately stop the inner loop. Since `continue_called` is set, execution does not leave the outer loop, however. As a consequence, `continue_called` becomes `0` again, and execution re-enters the inner infinite loop.

The do-while-statement is translated very similar to the while-statement. The only semantic differences in JS are that execution will enter the loop regardless of the loop-condition and that the loop-condition is tested after loop body execution. *Euclides* translates an empty do-while-statement `do { ... } while (false)` as follows:

```
{  /continue_called 0 def
   {  {  dict begin
         %% ... loop body omitted ...
         end
      } exec
      0 Types.bool sys_init_data
      sys_get_value not { exit } if
   } loop
   continue_called not { exit } if
   0 Types.bool sys_init_data
   pop
} loop
```

Due to a semantic difference of JS `continue` in do-while-loops, this statement needs to be handled differently. If `continue` is encountered, the loop condition must still execute before the loop body is re-entered, because side effects inside the loop condition may occur (such as incrementing a counter). *Euclides* handles this problem by executing the condition expression a second time in the outer loop. Since expressions always return values, any value resulting from the loop-expression has to be popped off the stack.

Although GML has a `for` operator, it is semantically incompatible with JS's one. Its increment is a constant number, and so is the limit. In JS, both increment and limit must be evaluated at each loop body execution. Therefore, we translate `for` just like the previous constructs by two nested loops with the increment condition repeated in outer loop (due to `continue` semantics). Finally, *Euclides* translates the JS statement `for (var i=0; i < 1; i++) { }` to GML via

```
dict begin
%% initialization (i=0)
/usr_i 0.0 Types.number sys_init_data def
{  /continue_called 0 def
   {  %% condition (i<1)
      usr_i 1.0 Types.number sys_init_data sys_lt
      sys_get_value not { exit } if
      {  dict begin
         %% ... loop body ...
         end
      } exec
      %% increment (i++)
      usr_i
         usr_i 1 Types.number sys_init_data sys_add
      /usr_i sys_edef
      pop
   } loop
   continue_called not { exit } if
   %% increment again (i++)
   usr_i
      usr_i 1 Types.number sys_init_data sys_add
   /usr_i sys_edef
   pop
} loop
end
```

The JS for-in statement `for(var x in array) statement;` is semantically equivalent to:

```
for (var i = 0; i < array.length; i++) {
   var x=array[i]; statement;
}
```

This construction loops over the elements of an array provides the loop body with a variable holding the current element.

C. Selection Control Statement

The translation of the JS `switch` statement poses several difficulties:

- If a case condition is met, execution can "fall through" till the next `break` is encountered.
- If a `break` is encountered, the currently executed `switch` statement must be terminated.
- Of course, `switch` statements may be nested.

To develop a semantically consistent solution, we did not want to alter the translation of JS-`break` inside switch statements (compared to loops). We solve the problem of breaking outside the `switch` statement by implementing it as a loop that is run exactly once. In GML it reads like `1 { loop_instructions } repeat`. This way our translation of `break` shows semantically correct behavior, it terminates the loop. Consider the following JS-program:

 ISBN: 978-1-61208-112-0

```
var x = 0, y = 0;

function bar() { return 3; }

function foo(i) {
   switch(i) {
      case 0:
      case 1:
      case 2: x = 1;
      case 4: x = 3;
      case bar(): x = 2; break;
      default: y = 5;
} }
```

The function `foo` will be translated to:

```
/usr_foo
{  dict begin
   /usr_i edef
   /return_issued 0 def
   { dict begin
      /switch_cnd_met1 0 def
      1 { usr_i 0.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
          } if

          usr_i 1.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
          } if

          usr_i 2.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 1;
             /usr_x 1.0 Types.number
             sys_init_data sys_def
          } if

          usr_i 4.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 3;
             /usr_x 3.0 Types.number
             sys_init_data sys_def
          } if

          usr_i usr_bar sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 2;
             /usr_x 2.0 Types.number
             sys_init_data sys_def
             exit
          } if
       %% y = 5;
       /usr_y 5.0 Types.number
       sys_init_data sys_def
   } repeat
   currentdict /switch_cnd_met1 undef end
   }
   { /return_issued 1 def } catch

   return_issued not {
      Nulls.Types.undefined
      Types.undefined sys_init_data
   } if
   end
   sys_exception_return_handler
} def
```

This example shows that we introduce an internal variable `/switch_cnd_metX` for traversing the case statements. As soon as a case statement condition is met, `/switch_cnd_metX` is set to `true`, leading execution into every encountered case statement.

The *Euclides* translator takes into account that switch statements may be nested. As it traverses the AST, it keeps book of all internal variable to ensure a unique name (`switch_cnd_met1`, `switch_cnd_met2`, ..., `switch_cnd_metN`).

The example translation shows that for `foo(3)` the cases 0, 1, 2, 4 and 3 (= `bar()`) will only execute case 3, where the `1 { } repeat` statement will be broken out of with the GML `exit` operator. The default block will be executed in any case if execution is still inside the `repeat` statement, no further state is checked for `default`.

VI. EXAMPLE

To demonstrate the interplay of all translational building blocks, this section shows a non-recursive, subtraction-based version of the Euclidean algorithm to calculate the greatest common denominator and its translation to GML. It can be shown by induction that two successive Fibonacci numbers are the computational worst-case of the Euclidean algorithm. We use them as input data.

```
function fibonacci(index) {
   switch (index) {
      case  0:
      case  1: return 1;
      default: return  fibonacci(index-2)
                     + fibonacci(index-1);
   }
}

function gcd(a,b) {
   if (a == 0) return b;
   while (b != 0)
      if (a > b) a = a - b;
      else       b = b - a;
   return a;
}

var x = gcd(fibonacci(5), fibonacci(6));
```

The corresponding GML code is:

```
/usr_fibonacci {
   dict begin
   /usr_index edef
   /return_issued 0 def
   {  dict begin
      /switch_cnd_met1 0 def
      1 {usr_index 0.0 Types.number sys_init_data
         sys_eq sys_getvalue switch_cnd_met1 1 eq or {
            /switch_cnd_met1 1 def
         } if

         usr_index 1.0 Types.number sys_init_data
         sys_eq sys_getvalue switch_cnd_met1 1 eq or {
            /switch_cnd_met1 1 def
            1.0 Types.number sys_init_data
            end throw
         } if

         usr_index 2.0 Types.number sys_init_data
         sys_sub usr_fibonacci
         usr_index 1.0 Types.number sys_init_data
         sys_sub usr_fibonacci
         sys_add
         end throw
      } repeat
      currentdict /switch_cnd_met1 undef end
   }
   { /return_issued 1 def } catch
   return_issued not {
```

```
      Nulls.Types.undefined
      Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def

/usr_gcd {
   dict begin
   /usr_a edef
   /usr_b edef
   /return_issued 0 def
   {  dict begin
      usr_a 0.0 Types.number sys_init_data
      sys_eq sys_getvalue
      { usr_b end throw }
      {}
      ifelse

      { /continue_called 0 def
        {  usr_b 0.0 Types.number sys_init_data
           sys_ne sys_getvalue not { exit } if

           usr_a usr_b sys_gt sys_getvalue
           {  /usr_a usr_a usr_b sys_sub sys_def }
           {  /usr_b usr_b usr_a sys_sub sys_def }
           ifelse exec
        } loop
        continue_called not { exit } if
   } loop
   usr_a end throw
   end
   }
   { /return_issued 1 def } catch
   return_issued not {
      Nulls.Types.undefined
      Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def

/usr_x
  6.0 Types.number sys_init_data usr_fibonacci
  5.0 Types.number sys_init_data usr_fibonacci
  usr_gcd
def
```

VII. Conclusion

In this article, we presented a JS to PostScript translator. While this translation is a simple infix-to-postfix notation rewrite for mathematical expressions (`1+2` becomes basically `1 2 add`), the correct translation of control flow structures is a non-trivial task, due to the fact that there is no concept of `goto` in the PostScript language and its dialects.

The main contribution of this work is the complete translation of JS into a PostScript dialect including *all* control flow statements. To the best of our knowledge, this is the first *complete* translator. Other projects (PdB by Arthur van Hoff, pas2ps by Dulith Herath and Dirk Jagdmann) do not support, e.g., `return` statements.

As *Euclides* offers a new access to GML, all GML users will benefit from its results. The possibility to use GML via a JS-to-GML translator reduces the inhibition threshold significantly. Everyone, who knows any imperative, procedural language (Pascal, Fortran, C, C++, Java, etc.) is familiar with the language concepts in JS and can use *Euclides*. Advanced GML users, who already know how to program in PostScript style, can use *Euclides* to translate algorithms, which are often presented in a imperative, procedural (pseudo-code) style [14].

Acknowledgment

We would like to thank Richard Bubel for his valuable support on ANTLR and the JS grammar. In addition, the authors gratefully acknowledge the generous support from the European Commission for the integrated project 3D-COFORM (www.3Dcoform.eu) under grant number FP7 ICT 231809, from the Austrian Research Promotion Agency (FFG) for the research project METADESIGNER, grant number 820925/18236, as well as from the German Research Foundation (DFG) for the research project PROBADO under grant INST 9055/1-1 (www.probado.de).

References

[1] Adobe Systems, Inc., *PostScript Language Reference Manual (first ed.)*. Addison-Wesley, 1985.

[2] "Document management – Portable Document Format," 2008.

[3] Adobe Systems, Inc., *Display PostScript System*. Adobe Systems Incorporated, 1993.

[4] J. Gosling, "SunDew – A Distributed and Extensible Window System," *Methodology of Window Management (Eurographics Seminars); Proceedings of an Alvey Workshop at Cosener's House, Abingdon, UK*, vol. 5, pp. 1–12, 1986.

[5] C. Geschke, S. McGregor, J. Gosling, L. Hourvitz, and M. Callow, "Screen postscript," *International Conference on Computer Graphics and Interactive Techniques archive; ACM SIGGRAPH 88 panel proceedings*, vol. 22, pp. 1–43, 1988.

[6] S. Havemann, "Generative Mesh Modeling," *PhD-Thesis, Technische Universität Braunschweig, Germany*, vol. 1, pp. 1–303, 2005.

[7] J. M. Snyder and J. T. Kajiya, "Generative modeling: a symbolic system for geometric modeling," *Proceedings of 1992 ACM Siggraph*, vol. 1, pp. 369–378, 1992.

[8] P. Müller, P. Wonka, S. Haegler, U. Andreas, and L. Van Gool, "Procedural Modeling of Buildings," *Proceedings of 2006 ACM Siggraph*, vol. 25, no. 3, pp. 614–623, 2006.

[9] S. Havemann and D. W. Fellner, "Generative Parametric Design of Gothic Window Tracery," *Proceedings of the 5th International Symposium on Virtual Reality, Archeology, and Cultural Heritage*, vol. 1, pp. 193–201, 2004.

[10] G. C. Reid, *Thinking in Postscript*. Addison-Wesley, 1990.

[11] C. Reas, B. Fry, and J. Maeda, *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2007.

[12] E. A. Vander Veer, *JavaScript for Dummies*. For Dummies, 2004.

[13] T. Parr, *The Definite ANTLR Reference – Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[14] T. H. Cormen, C. Stein, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. B&T, 2001.

 ISBN: 978-1-61208-112-0

PS-NET - A Predictable Typed Coordination Language for Stream Processing in Resource-Constrained Environments

Raimund Kirner, Sven-Bodo Scholz, Frank Penczek, Alex Shafarenko
Department of Computer Science
University of Hertfordshire
Hatfield, United Kingdom
{r.kirner, s.scholz, f.penczek, a.shafarenko}@herts.ac.uk

***Abstract*— Stream processing is a well-suited application pattern for embedded computing. This holds true even more so when it comes to multi-core systems where concurrency plays an important role. With the latest trend towards more dynamic and heterogeneous systems there seems to be a shift from purely synchronous systems towards more asynchronous ones. The downside of this shift is an increase in programming complexity due to the more subtile concurrency issues. Several special purpose streaming languages have been proposed to help the programmer in coping with these concurrency issues. In this paper, we take a different approach. Rather than proposing a full-blown programming language, we propose a coordination language named PS-Net. Its purpose is to coordinate existing resource-bound building blocks by means of asynchronous streaming. Within this paper we introduce code annotations and synchronisation patterns that result in a flexible but still resource-boundable coordination language At the example of a raytracing application we demonstrate the applicability of PS-Net for expressing the coordination of rather dynamic computations in a resource-bound way.**

***Keywords*-stream processing; embedded systems; multi-core; resource-constrained;**

I. INTRODUCTION

Stream processing is an apt metaphor of embedded computing. Indeed, owing to the generally static nature of streams connecting processing nodes, a higher degree of predictability may be achieved in representing embedded systems as stream-processing networks than with the dynamism of imperative and object-orientated milieux, where control and data can be passed from any point in the program to a given program unit provided that it is visible in that point's name space. Traditionally, stream processing is understood through the prism of the single-instruction multiple-data (SIMD) perspective. The paradigm itself is seen as a version of the latter with a different connectivity principle (streams instead of shared memory). This understanding is upheld by a number of projects, notably Stanford-based Merrimac [1] and Brook [2]. As an extreme form of this approach, one should mention strictly time-controlled synchronous solutions such as Giotto [3], [4], [5] and Scade [6], [7]. Here, the trade-off between predictability and efficiency is tilted towards predictability.

Generally, stream processing need not to be SIMD or even synchronous. In the most abstract sense, it is a representation of a program in terms of a static network of entities, each completely encapsulated and interacting with the rest only via its input and output streams. When streaming is to be used as a construction principle for larger systems, an asynchronous approach would usually be favoured, i.e., apriori unknown production rates and message arrival times. An example of this can be found in the language StreamIt [8], which has asynchronous messages and bounded nondeterminism. The most recent offering of an asynchronous streaming language comes from the project WaveScript [9] whose aim is essentially to integrate the network view and the local, synchronous view within one language with streams as first-class entities. Since this is a general-purpose streaming language, here, too, application programming concerns (i.e., algorithm correctness, ease of software evolution and accommodation of a continually changing specification) are intertwined with a whole spectrum of distributed computing concerns, such as work division, synchronisation, and load balancing, within a single level of program representation.

In our view, a more productive approach to applying the stream processing paradigm to embedded computing is to keep the concerns separated, with predominantly computational parts of the application represented as black boxes being written in a conventional programming language and with stream communication, data synchronisation and concurrency concerns being taken care of by a coordination language. We specifically focus on S-Net [10], [11], where we believe the above programme has been realised to the fullest possible extent.

The ground level of S-Net comprises stream-processing nodes represented as C-functions (or functions written in an array processing language, such as SAC [12]). These computational entities are called "boxes" and they communicate with the S-Net world via a single input and a single output stream. Data elements on these streams are represented as non-recursive record structures.

In a way, the set of boxes for a given application represents the nodes of a specialised virtual machine. The coordination program can abstract from the box functionality, the more so that the records streamed between boxes are being completely encapsulated as well: all the coordination level can see is field labels and some auxiliary integer-valued tags. This opens up an avenue towards sensible software engineering of embedded systems, where subject experts could be engaged in writing box code and describing the computational process informally in terms of record structures and box connections, and where *concurrency engineers* could be in a position to write, debug and optimise the coordination code with the experts' minimum assistance. That is the most attractive feature of the coordina-

 ISBN: 978-1-61208-112-0

tion approach compared with the competing strategies cited earlier.

However, this is not without some new problems either. The fundamental assumption of S-Net is that the application is not resource bounded. While not unreasonable in a large-scale distributed computing domain, this assumption is completely unrealistic in most embedded systems, where, if coordination has a chance, it must be essentially resource driven. This means that the placement of boxes on the system must be governed by the availability of cores and a predictive estimate of their load, which in turn means that the coordination layer must be in possession of accurate information about how much processing and communication is required for the completion of each task. By contrast, S-Net achieves its separation of concerns by relying on asynchronous dynamic adaptation: nondeterministic stream mergers, for instance, are assumed to merge in the order of record arrival, thus economising buffering space and reducing latency. Worse still, more dynamic features of S-Net namely its serial and parallel replication facilities, are not even a priori bounded since the boxes are not assumed to have the knowledge of, or the ability to communicate, the overall application design.

We set ourselves the challenge of finding a way to reconcile the need for dynamic behaviour and with the necessity to project tight enough bounds on the platform as far as the resource requirements. The S-Net facilities must therefore be curtailed to allow for static specification of various computational bounds, such as the maximum unfolding of the replicators, the maximum production rate of the boxes, the maximum correlation between the output rates dependent on a single input stream, etc.

In this paper we examine the relevant coordination facilities of S-Net in Section II and work out in Section III what needs to be modified and how so that S-Net may become usable with embedded applications. The result is a new language, called PS-Net, which is described in Section IV. Section V shows an example of how to write resource-bounded programs in PS-Net. Section VI concludes the paper.

II. Stream-Processing with S-Net

In order to present a specialised variant of S-Net that is resource-boundable, we first give a very brief overview of the language. A detailed description of S-Net can be found in the literature [10], [11].

The central philosophy of S-Net is to separate the coordination of concurrent data streams from the computational part. Computations on data are not expressible in S-Net as such, but are written in a conventional programming language. These pieces of "foreign" code are embedded into *boxes* and are given an extremely simple API to communicate with the surrounding S-Net. The API allows them to receive data from a single input stream via the normal parameter-passing mechanism, and which provides a small number of library functions for outputting data down the single output stream, both streams being anonymous. Boxes may not have a persistent internal state and consequently can only process input data individually. Nor can they access each other's state in any way during the processing: there are no global variables or inter-box references. Instead, the output records are streamed by the coordination layer of S-Net according to a coordination program that defines the streaming topology, how the streams are split and merged, and how individual records are split, merged and routed to their intended destinations.

In order to guarantee interoperability of computational entities and all different parts of a streaming network, the coordination of data flow in S-Net is analysed by means of a type system and inference mechanism. The type system of S-Net is based on non-recursive variant records with *record subtyping*. Each record variant is a possibly empty set of named record entries, where a record entry is either a field or a tag. The values of fields are only accessible by the box implementations, while the tags are integer variables whose values can also be accessed and manipulated by both, the S-Net program and the box implementations. To separate tags from fields, the tag names are surrounded by angular brackets, e.g., `<a>`. Tags allow to use some logic operations to control the flow of data. The following is a variant record type that encompasses both rectangles and circles enhanced with a tag `<id>`: `{x,y,dx,dy,<id>} | {x,y,radius,<id>}`. Each S-Net network or subnetwork has a type signature, which is a non-empty set of variant record type mappings each relating an input type to an output type. For example, a network that maps a record `{a,b}` to either a record `{c}` or a record `{d}` or maps a record `{a}` to a record `{b}` has the following type signature: `{a,b}->{c}|{d}, {a}->{b}`. S-Net also supports subtyping. For example, `{a,b}` is a subtype of `{a}`.

As with conventional subtyping, in S-Net a network or box also accepts input data being a subtype of the network's or box's input type. Those record entries of the subtype that do not match a record of the box's input type simply bypass the network or the box and are joined with the produced output. Thus, an S-Net box with the type signature `{a}->{b}`, for example, also accepts input data of the type `{a,c}`, like a type signature `{a,c}->{b,c}` but where the record field `c` simply bypasses the box. This feature is called *flow inheritance*.

A. Stream-manipulation with Filter Boxes

In S-Net, so-called *filter boxes* are used to perform manipulations of the data stream, like elimination or copying of fields and tags, adding tags, splitting records, and simple operations on the tag values. Filter boxes are expressed in square brackets and consist of a semicolon-separated list of filter actions on the right side of the transformation arrow. For example, the filter box `[{a,b,c} -> {a};{b,<t=1>};{b=c,<t=2>}]` takes records of type `{a,b,c}` and splits them into three output records: one with the field `a`, one with the field `b` extended by a tag `<t>` with the value 1, and one with the field `c` renamed to `b` and extended by a tag `<t>` with the value 2. Though the last two output messages contain the same field name `b`, they can still be processed differently at S-Net level due to their different value of tag `<t>`.

B. Network Combinators in S-Net

S-Net consists of the following four combinators to combine networks or boxes. For the description of them we assume that we have two networks net_1 and net_2 that we want to combine.

1) **Serial Composition (net_1 .. net_2):** This allows to combine two S-Net networks or boxes in a sequential fashion. Though sequential in its dataflow, in the context of stream-processing this provides parallel

 ISBN: 978-1-61208-112-0

processing in the form of pipelined execution. The code net_1 `..` net_2 essentially forms a pipeline with the stages net_1 and net_2.

2) **Parallel Composition (net_1 `|` net_2):** This allows to combine two S-Net networks or boxes in a parallel fashion, providing concurrent execution. The code net_1 `|` net_2 describes a split of data flow between the routes of networks net_1 and net_2. If net_1 and net_2 have different type signatures then the type system of S-Net will route the data to the best-matching input type, otherwise the choice is non-deterministically.
3) **Serial Replication (net_1 `*` {`out`}):** The serial replication (subsequently also called *star operator*) creates a pipeline dynamically by replicating the given network along a series composition till the output is a (sub)type of the exit pattern, where in this case the output is forwarded as the output of the replication operator. net_1 `*` {`out`} means that the data flow through a series composion of replicas of the network net_1 till the type of the output is a (sub)type of {`out`}.
4) **Parallel Replication (net_1 `!` {`<id>`}):** The parallel replication is the dynamic variant of parallel execution, where a given network is replicated dynamically controlled by the value of a tag in the data records. net_1 `!` {`<id>`} means that for each different value of the tag `<id>` of the incoming data records an exclusive path through a replica of the network net_1 is dynamically created.

Note that the combinators `|`, `*`, `!` have an out-of-order semantics on data routing, while `||`, `**`, `!!` are their order-preserving variants.

C. Synchronisation with Synchro-cells

Above S-Net operations are all asynchronous and stateless operations, allowing for an efficient concurrent processing of data streams. To synchronise the arrival of different message types, the so-called *synchro-cell* is used, which is the only stateful box in S-Net. The synchro-cell is the only means in S-Net to combine two records into a single record. The synchro-cell consists of an at least two-element comma-separated list of type patterns enclosed in `[|` and `|]` brackets. For example, the synchro-cell `[| {a}, {b,c} |]` composes two records `{a}` and `{b,c}` into a single output record `{a,b,c}`. As its state, a synchro-cell has storage for exactly one record of each pattern. When an arriving record finds its place free in the synchro-cell, it is stored in the synchro-cell, otherwise it is simply passed through. The synchro-cell is a one-shot operation, i.e., once all record patterns are filled, the composed output record is emitted and the synchro-cell from now on behaves like a simple connector passing all further messages through. To use synchro-cells in a continuous way on the input stream, it has to be nested within replication operators as described above.

III. Discussion of Predictability

In the following, we discuss what features of S-Net are hard to bound for their resource consumption and we discuss how we address this problem in PS-Net to ensure boundability of resources. The following mechanisms of S-Net are hard to bound without doing an exhaustive whole-program analysis:

- The computational part of S-Net programs is implemented in boxes. Regarding the boundability of the dynamic resource allocations, it is, of course, necessary, that the box implementations are simple enough to bound their resource requirements. However, the box implementation is outside the scope of our design of the resource-boundable coordination language PS-Net.
- In S-Net a network or box may write an arbitrary number of output messages as the type signature does not restrict them. Thus it is not know how much system load can be created within the network. This makes it hard to bound extra-functional properties such as execution time. Our solution for PS-Net is to extend the type signature with the multiplicity of the different output messages.
- The parallel composition in S-Net (`|`,`||`) features a non-deterministic choice, whose behaviour cannot be analysed precisely at language level, which makes it challenging to bound extra-functional properties such as execution time.
- The number of parallel replications in S-Net (`!`,`!!`) depends on the possible values of the replication-controlling tag value, which is hard to bound in general. In order to bound the number of dynamically created replicas for the parallel replication operator, we have to know the possible value range of the index tag. For PS-Net we extend the parallel replication combinator with an annotation about the maximum range of the index tag.
- The number of serial replications in S-Net (`*`,`**`) depends on the dynamic creation of the exit type, which is hard to bound in general. In order to bound the number of dynamically created replicas for the serial replication operator, we have to know when latest the exit pattern is produced. For PS-Net we extend the serial replication operator with an annotation of the maximal number of created replications.

A. Synchronisation Mechanisms

The synchronisation issues deserve a special discussion. On the one side the synchro-cell of S-Net has a single-shot semantics which is no problem at all to account for its maximum resource usage. However, as already said, the synchro-cell is typically embedded into a serial replication with infinite replications. This infinite replication is not a problem in S-Net, since every replica with a synchro-cell that has already shot is automatically discarded and automatically replaced by a direct stream connection.

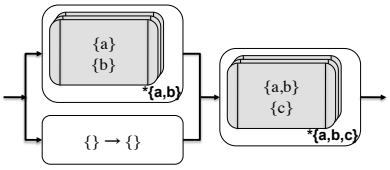
However, our general solution of making the serial replication boundable by adding an annotation about the maximal number of created replications, is unfortunately not compatible with the use of the S-Net synchro-cell, as this would rely on an infinite replication count.

Our solution for PS-Net is to avoid the combination of synchro-cell and serial replication and instead use special synchronisation constructs for use patterns of synchro-cells. We have actually identified two major use patterns for synchro-cells. They stem from the need to either synchronise a statically fixed number of records or to synchronise a dynamically varying number of records, respectively.

In the former case, the records that are to be combined can be encoded by different types. This facilitates an implementation of the synchronisation as a cascade of synchro-cells embedded in serial replications.

 ISBN: 978-1-61208-112-0

Figure 1 shows an example for such a synchronisation.

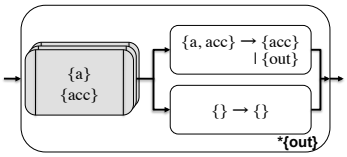



```
([|{a},{b}|]*{a,b} | []) .. [|{a,b},{c}|]*{a,b,c}
```

Fig. 1. Synchronising records of type {a}, {b}, {c}.

There, three records are being synchronised each, one record of type {a}, one of type {b}, one of type {c}. The first synchro-cell within a star combines records {a} and {b}. Any records that are neither {a} nor {b} are bypassed by means of the identity filter which is parallel to the synchroniser for {a} and {b}. Subsequently, the bypassed records of type c are synchronised with the combined records of type {a, b}. Again, this second synchro-cell is directly embedded into a star to enable repeated synchronisation.

In the second case, i.e., when we deal with a statically not determined number of records to be synchronised, a type encoding of the individual components of the record to be combined is no longer possible. Instead, a stepwise synchronisation needs to be applied to a substream of records of the same type. The emerging result record needs to be propagated from one synchronisation to the next similar to an accumulator within a folding operation. When to terminate such a folding process needs to be determined either by the folding operation itself or by the use of "separation records" of different type in the stream. Figure 2 shows an example for such a multi-synchronisation. Here, the folding box itself




```
net multi_sync {
  box fold( (a, acc) -> (acc) | (out) );
} connect ( [|{a},{acc}|] .. ( fold | [] ))*{out};
```

Fig. 2. Synchronising multiple records of type {a}.

determines when to emit a value by producing a record with a field out rather than acc. This network furthermore assumes that the initial value for each synchronisation comes in as a record containing acc. Note, that the empty filter that is parallel to the fold box serves as a by-pass for subsequent records of type {a} or type {b} so that they can be fed into subsequent unfoldings of the star combinator.

Within a range of S-Net applications [13], [14], [15], we could observe various different formes of synchro-cell uses within serial replication. However, it turns out that all of them adhere to one of the two use cases above and can be expressed by nestings of these two pattern. Therefore we capture those two pattern as two new building blocks in PS-Net, named syncq and fold, respectively.

IV. Resource-Bounded PS-Net

In this section we introduce new language constructs that are boundable. Further we introduce annotations for existing S-Net language constructs to make them boundable. These annotations might be written by the programmer or being automatically derived by program analysis. All the annotations have the form <| *AnnotExpr* |> where *AnnotExpr* can be of the following forms:

Num	...specifies a constant value
Num **:**	...specifies a lower bound
: *Num*	...specifies an upper bound
Num **:** *Num*	...specifies an interval

A. Multiplicity of Box Messages

For PS-Net we extend the type signature with an annotation about the multiplicity of messages. For example, the following box signature declaration `box foo ((a,b) -> (c)<|2|> | (d)<|1:3|>);` specifies that for each processing of on input record {a,b} the box creates exactly two output records of type {c} and between one and three output records of type {d}. Note that the records of box signatures are written in round brackets to distinguish them from network type signatures, since for the box signature the order of record entries matters.

B. Bounded Parallel Replication

The range of the index tag determines the number of different dynamically created parallel replicas. Assuming that an index range will always start from zero, we extend the parallel replication with an annotation of the upper bound of replications k, resulting in an index range from $0 \ldots k-1$. replication index range. For example, to specify that a network can be at most replicated four times (i.e., index range 0 to 3), we can write:

```
network ! <tag><|4|>;
```

Note that the total number of replications can be higher if the network is nested within another network that is replicated as well.

C. Bounded Serial Replication

We extend the serial replication operator with an annotation of the maximal number of created replications. For example, to specify that a network can be at most replicated three times (i.e., a pipeline of length three), we can write:

```
network * {out}<|:3|>;
```

D. Synchronisation with the syncq operator

The first use case of synchronisation (Figure 1) can be abstracted by means of a *synchro-queue*, which repeatedly synchronises records of two flavors defined by means of two type pattern.

Provided that the synchronic distance [16] between the two flavors is bounded, such an operator can be implemented as a finite queue whose length does not exceed that bound. We introduce synchro-queues as a new operator

```
syncq[| p1, p2 |]<| sd |>
```

where p_1 and p_2 denote type pattern to be synchronised, and sd denotes an upper bound for the synchronic distance between the pattern p_1 and p_2 on the input of this operator. For example, if we want to synchronise records of type {a}

and {b} knowing that the synchronic distance between them is at maximum 4, then we can write:

```
syncq[| {a}, {b} |]<|4|>
```

Formally, the semantics of the syncq operator is defined by the following equivalence:

$$\texttt{syncq[|}\ p_1,\ p_2\ \texttt{|]<|}\ sd\ \texttt{|>} \equiv \texttt{[|}\ p_1,\ p_2\ \texttt{|]*}\{p_1,\ p_2\}$$

Note here, that the star version on the right hand side of the equivalence potentially requires unbounded resources. Only the annotated synchronic distance sd ensures boundedness of the operator. Interestingly, a finite synchronic distance also implies that all the synchronised patterns have the same average arrival rate.

E. Synchronisation with the fold operator

The second use case of synchronisation (Figure 2) can be abstracted into a generic *folding operation*. Here, we introduce a network combinator, which transforms a folding network with a signature $(a,\ acc)$ -> (acc) into a network that subsequently synchronises a record of type $\{acc\}$ with an arbitrary number of records of type $\{a\}$, until a new record of type $\{acc\}$ arrives which triggers a new series of synchronisations.

Syntactically, we denote the fold combinator by

$$\texttt{fold[|}\ a,\ acc,\ N,\ Fold\texttt{|]}$$

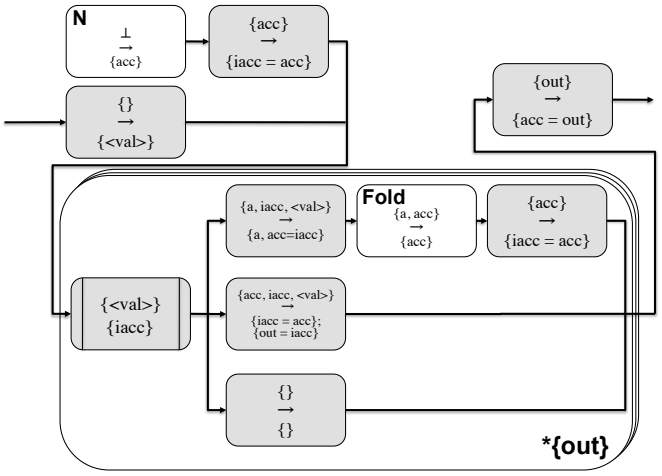
where a and acc denote the two different kinds of type pattern, N denotes a network with a type signature $\perp$ -> (acc)<|1|> that provides the initial value for acc and $Fold$ is a network of type $(a,\ acc)$ -> (acc)<|1|> which implements the folding operation itself. For example, if we want to collect partial results of type {d} of a concurrent computation into result messages of type {res}, with Init being the network to create the initial {res} message and Collect being the network name of the fold operation that merges a partial result of type {d} with the current result message of type {res}, then we can write:

```
fold[| {d}, {res}, Init, Collect |]
```

The semantics of a network fold[| a, acc, N, $Fold$|] then is defined by the S-Net shown in Figure 3.

The main complexity of this network stems from the necessity to "restart" the folding process upon arrival of a new record of type {acc}. To achieve this, all incoming data is tagged with <val> upon arrival. In the core of the network, this data, i.e., either records of type {a, <val>} or of type {acc, <val>} are synchronised with the current state of the folding operator which is kept in an internal accumulator field iacc. Depending on the type of the synchronised record, either the Fold network is applied and the internal accumulator is updated accordingly, or the current result is emitted via a record of type {out} and the internal accumulator is reset to the new value from the input field acc. Note here, that the overall fold combinator needs to be initialised with a record of type {acc} provided by the network N. Its value serves as initial state for the internal accumulator.

A key observation of this network is that for each incoming record the synchro-cell of the first incarnation of the star operator synchronises which transforms the entire inner network effectively into an identity function for the subsequent records. In combination with a multiplicity of 1 for the Fold network, this guarantees that the fold operator can be implemented in constant space.




```
( ( N .. [ {acc} -> {iacc = acc} ])
  || [{} -> {<val>}]
)
.. ( [| {<val>} , {iacc} |]
     .. ( ( [ {a, iacc, <val>} -> {a, acc=iacc} ]
              .. Fold .. [ {acc} -> {iacc=acc} ]
          )
          |  [ {acc, iacc, <val>} -> {iacc = acc};
                                     {out = iacc} ]
          |  []
        )
    ) * {out}
.. [ {out} -> {acc = out} ]
```

Fig. 3. Network implementing the fold operator

V. EXAMPLE

We evaluate the presented approach by applying it to the well-known fork-join pattern that many image processing applications expose. An image is broken down into smaller chunks and an application specific processing algorithm is run on each chunk independently in an SIMD-like fashion. A merging stage collects all processed chunks, i.e. the sub-results, and reassembles a global result image.

Where previous experiments using S-Net in its standard form have shown that this class of applications lends itself nicely to the advocated programming model we are now in a position to reformulate existing code to guarantee resource-bounded execution in PS-Net. As a representative problem of this class we implemented a ray-tracing image processing application for which we have developed an implementation in standard S-Net with performance results that compete with hand-tuned C code [13].

The implementation of the original application is intended to run on general-purpose hardware and is specified as follows:

```
net raytracing {
  box splitter( (scene, <rr_upper>, <tasks>)
        -> (scene, chunk, <rr>, <tasks>, <fst>)
        |  (scene, chunk, <rr>, <tasks> ));
  box solver  ( (scene, chunk) -> (sub_res));
  net merger  ( (sub_res, <fst>) -> (pic),
                (sub_res)        -> (pic));
  box genImg  ( (pic)          -> ());
} connect splitter .. solver!<rr> .. merger
        .. genImg;
```

The splitter divides the scene into smaller sub-scenes (chunks) and tags all chunks with the number of overall

 ISBN: 978-1-61208-112-0

produced sub-scenes. Each data element also carries an <rr> tag. This implements a round-robin scheduling using the ! combinator on the solver by tagging data elements with increasing integer values from $0, ...,$<rr_upper>-1 for <rr>. The first output is tagged with <fst> to initiate the merging process after the sub-scenes have been computed by the solver. The merging process is implemented as a sub-network of the following form:

```
net merger {
  box init  ( (sub_res, <fst>) -> (pic));
  box merge ( (sub_res, pic)   -> (pic));
} connect ( ( init .. [ {} -> {<cnt=1>} ] ) | [])
          .. ( [| {pic}, {sub_res} |]
               .. ( ( merge
                        .. [ {<cnt>} -> {<cnt+=1>}])
                    | []))*{<tasks> == <cnt>};
```

The init box is followed by a filter which adds a flag <cnt> initialised by the value 1. This flag is used to count the number of sub-scenes that have been incorporated into the result image already. Since only the first sub-scene needs to be processed by the init box, we also provide a bypass to the initialisation path for all the other records containing further sub-scenes.

After the initialisation, a star implements the merging with the remaining sub-scenes. In each unfolding (iteration) of the star the synchro-cell synchronises the accumulator held in {pic} with yet another sub-scene. The resulting joint record, containing the accumulated picture and a sub-scene to be inserted, is presented to the merge box which outputs the combined picture. The insertion of a new sub-scene is reflected in an increment of the flag <cnt> as defined by the subsequent filter. Once the counter equals the overall number of tasks, which is kept in another, flow-inherited flag <tasks>, the accumulated picture is output from the merger network.

In order to guarantee resource-boundedness of this implementation, we replace the parts of the application that make use of the general $\star$ and ! combinators by legal PS-Net constructs.

The splitting stage of the application is almost straightforwardly transformed. As we are not targeting general-purpose hardware, we use the ! <rr><|n|> combinator and annotate the maximum number n of computing resources the combinator is allowed to bind for solver instances. Because of the way we are implementing the merging process, which is detailed below, the splitter is not required to output the number of produces sub-scenes. Additionally, it also does not tag the first element. Instead, the splitter outputs the accumulator as first record for each decomposed scene.

With the PS-Net fold combinator we are able to re-implement the merging stage of the original application. The combinator's behaviour resembles the functionality of the merging stage when supplied with the merger box of the original application as fold-net argument. An initialiser network is not required, as we chose to have the splitter output all pic accumulators including the first one.

Putting it all together, the resource-bound version of the application is defined as follows (we chose 7 as an arbitrary resource limit for the ! combinator for illustration purposes):

```
net raytracing {
  box splitter( (scene, <rr_upper)) ->
                  (scene, chunk, <rr>) | (pic));
  box solver( (scene, chunk) -> (sub_res));
  box merger( (sub_res, pic) -> (pic));
  box genImg( (pic) -> ());
} connect splitter
      .. (solver!<rr><|7|> | [(pic) -> (pic)])
      .. fold[|{sub_res},{pic},_,merger|]
      .. genImg;
```

This network behaviour resembles that of the original implementation. The splitter outputs a variable number of sub-scenes and the solver is applied to these in parallel. The merging stage is wholly implemented by the fold combinator. But this implementation is guaranteed to be resource bound: The parallel replication is limited by an annotated upper bound. As the fold combinator is statically resource bound, we do not require multiplicity annotations on the splitter box.

This example has shown how the proposed coordination languages for stream processing can be used to model resource-constrained embedded applications. The stream-processing model itself has the benefit that it naturally combines the flexibility of asynchronous computation with a separation of concern between coordination and algorithmic programming.

VI. Conclusion and Future Work

In this paper we have shown the development of the resource-boundable coordination language PS-Net for stream processing, starting from the S-Net language, which has been designed for the high-performance computing domain. On the one side we had to add annotations to certain language constructs, to make them resource-boundable. Such annotations might be written directly by the developer or may be derived automatically by program analysis. Further, we have introduced the *synchro-queue* and the *folding combinator* as resource-boundable synchronisation constructs. The resulting language allows to program dynamic stream-processing applications in a resource-bound way. As a future work we will implement PS-Net within the S-Net compiler, which is quite suitable for this implementation, since the new resource-boundable constructs introduced for PS-Net can be implemented with S-Net constructs. These S-Net constructs would be non-resource-boundable in the general case, but become resource-boundable for the specific patterns derived from PS-Net constructs. Further, evaluations of resource consumption are planned to demonstrate the suitability of the PS-Net programming paradigm for embedded computing.

Acknowledgments

The research leading to these results has received funding from the IST FP-7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)".

References

[1] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labont, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *Proc. ACM/IEEE Conference on High Performance Networking and Computing (SC'03)*, Phoenix, Arizona, USA, Nov. 2003.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *Proc. ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*, Los Angeles, USA, 2004, pp. 777–786.

[3] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

 ISBN: 978-1-61208-112-0

[4] T. A. Henzinger, C. M. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.

[5] A. Ghosal, D. Iercan, C. M. Kirsch, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code," in *Online Proc. Workshop on Automatic Program Generation for Embedded Systems (APGES)*, 2007.

[6] F.-X. Dormoy, "Scade 6: A model based solution for safety critical software development," in *Proc. 4th International Conference on Embedded Real Time Software (ERTS)*, Toulouse, France, 2008.

[7] E. Technologies, "SCADE suite," web page (http://www.esterel-technologies.com/products/scade-suite/), accessed in Jul. 2010.

[8] B. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. 11th International Conference on Compiler Construction (CC'02)*. London, UK: Springer Verlag, 2002, pp. 179–196.

[9] R. Newton, L. Girod, M. C. abd Sam Madden, and G. Morrisett, "WaveScript: A case-study in applying a distributed stream-processing language," Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, Cambridge, USA, Technical Report MIT-CSAIL-TR-2008-005, Jan. 2008.

[10] C. Grelck, S.-B. Scholz, and A. Shafarenko, "A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components," *Parallel Processing Letters*, vol. 18, no. 2, pp. 221–237, 2008.

[11] A. Shafarenko, S.-B. Scholz, and C. Grelck, "Streaming networks for coordinating data-parallel programs," in *Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference (PSI'06), Novosibirsk, Russia*, ser. Lecture Notes in Computer Science, I. Virbitskaite and A. Voronkov, Eds., vol. 4378. Springer Verlag, 2007, pp. 441–445.

[12] C. Grelck and S.-B. Scholz, "SAC: A functional array language for efficient multithreaded execution," *International Journal of Parallel Programming*, vol. 34, no. 4, pp. 383–427, 2006.

[13] F. Penczek, S. Herhut, S. Scholz, A. Shafarenko, J. Yang, C. Chen, N. Bagherzadeh, and C. Grelck, "Message driven programming with s-net: Methodology and performance," in *3rd International Workshop on Programming Models and Systems Software for High-End Computing (P2S2'10), San Diego, USA*, 2010, to appear.

[14] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrere, and E. Lenormand, "Parallel signal processing with s-net," *Procedia Computer Science*, vol. 1, no. 1, pp. 2079 – 2088, 2010, iCCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B9865-506HM1Y-88/2/87fcf1cee7899f0eeaadc90bd0d56cd3

[15] C. Grelck, J. Julku, and F. Penczek, "Distributed S-Net," in *Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA*, M. Morazan, Ed. Seton Hall University, 2009.

[16] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.

 ISBN: 978-1-61208-112-0

An Application of a Domain-Specific Language Facilitating Abstraction and Secure Access to a Crime and Ballistic Data Sharing Platform

Lukasz Jopek
Cultural and Computing Research Centre (C3RI)
Sheffield Hallam University
Sheffield, United Kingdom
l.jopek@shu.ac.uk

Richard S. Wilson
Cultural and Computing Research Centre (C3RI)
Sheffield Hallam University
Sheffield, United Kingdom
r.wilson@shu.ac.uk

Christopher D. Bates
Cultural and Computing Research Centre (C3RI)
Sheffield Hallam University
Sheffield, United Kingdom
c.d.bates@shu.ac.uk

***Abstract*—Crime investigation requires controlled sharing, secure access and formalised reporting on heterogeneous datasets. This paper will focus on encapsulating data structures and services, whilst exposing abstraction, relevant only to the End-User through the application of a domain-specific language. The language is used for all interactions with the platform, enabling non-technical users to build complex queries. The language also increases the platform's security, by hiding the internal architecture of services and data structures. This solution has been demonstrated to law enforcement communities across Europe as a prototype crime and ballistic data sharing platform.**

Keywords-standardisation;data structures, domain-specific language; law enforcement; public services

I. INTRODUCTION

Odyssey is a research project looking at the difficulties in sharing information about gun-crime between Police forces across Europe [1][2]. The project is building a technology demonstrator to prove the benefits of integrating and sharing ballistic information from states across the European Union. Such sharing will support gun crime investigations and prevention activities where cross-border trafficking of weapons or ammunition is involved. The prototype will enable interoperability between existing systems and ensure compatibility with emerging European standards. The Odyssey Consortium consists of law enforcement agencies such as Europol and NABIS, the National Ballistic Intelligence Service (United Kingdom) which brings standards, broad knowledge and hands-on experience to the project. The Consortium is committed to informing the law enforcement community with the outcomes of its research, standardising data formats, integrating processes and finally prototyping a crime and ballistic data sharing platform.

Currently, law enforcement agencies across Europe rely heavily on in-house systems, which often cannot interoperate on either national, regional or departmental levels [3]. Integration, secure sharing and the use of data is highly limited due to technological boundaries; hence the process is frequently manual and costly [3][5]. The Odyssey project is seen not only as an opportunity to facilitate communication between different agencies, but to also build a common European understanding over current and future needs. The number and diversity of existing technologies, the lack of European ballistic standards and the high reliance on ballistic experts puts Odyssey into a very interesting perspective, whereby new standards, technologies and processes are expected to support current best practices. Technologically, the Odyssey project aims at prototyping a scalable platform, which is interoperable with legacy systems, processes and expertise.

In this paper, we will focus on the process of building a common interface for the system, encapsulating data structures and exposing only those abstractions that are relevant to each user. We will further explore an innovative application of a domain-specific language (DSL) in an area in which one has not been used before. The DSL is used to express every interaction within the system from getting data and defining sharing permissions, to integrating security by hiding internal structures of the platform.

II. DESIGNING A DOMAIN-SPECIFIC LANGUAGE

Domain-specific languages (DSL) express complexity at a particular abstraction tailored to both current and future needs [6]. A DSL lets non-technical people understand the overall design of a platform and interact with it, using an understandable notation that reflects their particular perspective [7].

A well-designed DSL complies with certain objective qualities. A language perceived as simple, easy, and effective is more likely to be widely accepted, even though these qualities can be subjective. However, these merits are not enough for a good, stable language. The definition and measurement of these qualities will vary in time and from one person to another. Everyone in a team developing a language can agree on where the language meets their expectations, but only when exposed to a wider community for a longer period can the quality of a language be measured. In the world of engineering, users have varied skills, preferences and needs, while usually the primary goal of designing a system is to identify base requirements and build a platform that works.

In the Odyssey project, a DSL was introduced to express the user requirements and solutions in a particular domain. A DSL promotes decoupling between components, modules and software stack layers, making the platform easily extendable and its components highly reusable. In the Odyssey project, a DSL language is used not only to convey the complexity of the domain, but also to facilitate and unify the entire communication across the platforms' components and users (Figure 1).

 ISBN: 978-1-61208-112-0

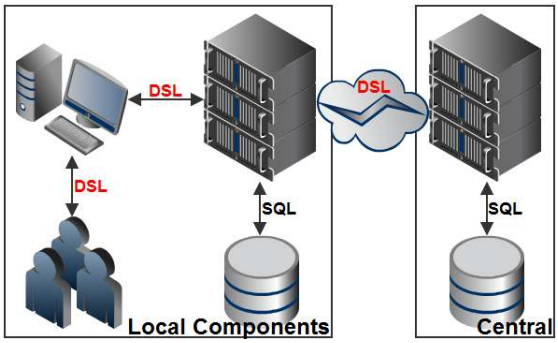


Figure 1. Role of DSL in platform's architecture

A DSL is used to express queries further translated into for example SQL. In the Figure 1, a DSL query is presented to query data from distributed databases by executing the exact statement on separate repositories. The Odysseys DSL enforces unification of queries, which enables standardisation of the results formats and therefore further reasoning. The DSL engine has the capability of retrieving and merging information from diverse data sources by transforming the query into SQL-like languages, but also integrating with for example data-mining result sets or applying reasoning using a built-in ontology.

The particular application of a DSL enhances security by providing information relevant only to the end user encapsulating data structures, abstracting services, and facilitating data manipulation. The Odyssey system enables sharing information without exposure of its structures, providing the requested up-to-date, accurate, relevant and easy to understand information.

Additionally, users are supported with tools to build and visualise DSL statements and their executions, which are also represented in the DSL.

III. THE RESEARCH CONTEXT

The Odyssey project delivers both a user interface and a domain specific language to users who have diverse, but highly specialised expertise. Their high-level cognitive abilities often relate to discovering facts in crime investigations, whilst dealing with uncertainty and different types of abstraction. A mixture of experience from former investigations, combined with an understanding of human nature and circumstances, enable investigators to reason, make decisions and act on them. Dealing with this type of situation requires an entirely different set of skills and builds an impression that the situational information with knowledge and experience is enough to solve any problem. The Odyssey project is committed to fulfilling the end user's functional requirements and expectations, whilst proposing new functionalities, which are not available in current systems due to a lack of data sharing and information exchange abilities.

Odyssey's DSL compromises between the expressiveness of both a formal and flexible semantically-enhanced language. Complexity of syntax was greatly reduced by identification and categorisation of use case scenarios, grouping of functionalities and abstraction of data sources. Moreover, in contrast to, for example SQL, the user does not need to be aware of underlying data structures, nor the platform's architecture, to enable a complex search to be undertaken. The user is asked to define the information of interest and the constraints by which the data will be filtered and sorted. In general, the user queries the system by defining the outcomes and under what conditions. One of the key requirements for the language is to facilitate access to factual information, but without taking the risk of misleading an investigator by presenting non-related data. The system reveals opportunities to the end user by facilitating discovery of new facts and collaboration on possible scenarios. Ultimately, we have identified three main qualities integrated to our solution. They are shown in Figure 2.

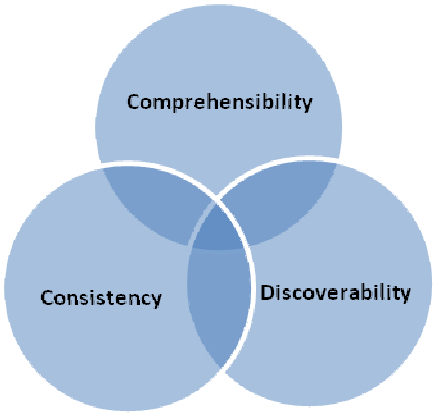


Figure 2. Principles of a scalable DSL design

Comprehensibility: Communication is pivotal to design domain-specific languages. In the domain of law enforcement agencies, a comprehensive, self-explanatory language acts as a bridge between a platform and a non-technical end user. A consistent and well-established syntax builds trust and guarantees time invested in mastering functionality will be applicable across other aspect of the platform in the present and future releases. Furthermore, we identified discoverability as the next most important characteristic of a well – designed language, which in this case means the ease of discovering features based on what we already know and the tools provided.

Consistency: The language and its controlled syntax encapsulate functionality, architecture and may even determine the entire system's performance, by for example, optimising queries and merging results. These algorithms are developed for abstracted use case models and a user is not allowed to make any modifications or optimise queries per case. The great advantage at this stage was the active involvement of the user community (especially West Midlands Police, United Kingdom), who carefully gathered both user requirements and developed an iterative process of language evaluation. Each partner in the Odyssey project has a different perception of the problems we are addressing and has contributed to the design of the platform in different ways. Sheffield Hallam University representatives visited Northern Ireland Police and West Yorkshire Police in England identifying needs and getting a hands-on experience of the current state of the art crime and ballistic ICT systems.

Discoverability: In addition, these visits provided an insight into daily activities and processes, the Odyssey project is dedicated to improve. Satisfied at this stage, the Odyssey Consortium proposed a language that would enable the modelling of crime investigations and play a

 ISBN: 978-1-61208-112-0

key role in creating data access rules and the enhancement of security to the entire platform. The user requirements stated explicitly that we should develop a language which provides a simplified method to retrieve data and to operate on datasets, but to also facilitate access to services, enable management of users' roles and maintain data sharing rules. We require a language that would facilitate every interaction, with the entire system, in a controlled, structured and concise way.

IV. ODYSSEY SEMANTIC LANGUAGE

We propose the development of a syntax and a semantic language which supports modelling of active crime investigations by operational detectives that will link general crime to ballistic data. Its innovative features are associating data retrieval techniques with data-mining results and encapsulating multiple services. Moreover, the language facilitates modelling of investigation processes and is an integral part in the platform's security. Furthermore, it was developed in an open-source grammar development environment, ANTLR. A structural Piggyback [8] design pattern was introduced to facilitate transparency between languages and services being a hosting base to which the DSL is translated. The DSL was designed on the top of a SQL bearing in mind abstraction, data mining and process modelling capabilities.

The ANTLR output is further integrated with the NetBeans Rich Client Platform (RCP) and the combination produces a fully-featured editor that seamlessly integrates with the graphical representation of search and results. The features correspond syntax colouring, error highlighting, code completion, etc.

The example below (Figure 3) presents a query expressed in Odyssey Semantic Language (OSL) that retrieves firearms with a twenty-two calibre (0.22 inch):

```
QUERY firearm WHERE calibre HAS VALUE 0.22
```

Figure 3. Selecting a firearm of a specific calibre

With a very similar query structure we can apply sharing rules on a set of data (See Figure 4 below).

```
ALLOW firearm WHERE calibre HAS VALUE 0.22
```

Figure 4. Applying sharing rules on a dataset

In the diagram below, we introduce a few entities from the Odyssey database structure that will be used in the next example to present how OSL abstracts and simplifies access to relational datasets. The database itself contains over 50 tables to model crime and ballistic evidence or retains user accounts and their roles. The diagram shows how a location, a central concept of the database structure, can be linked to a firearm. A location is only one of an incident's characteristics; other descriptors include documents or other related incidents. An incident effectively links locations with ballistic items that are a generalisation of firearms, cartridge cases, bullets, ammunition, projectiles, and other categories of ballistic and crime evidence.

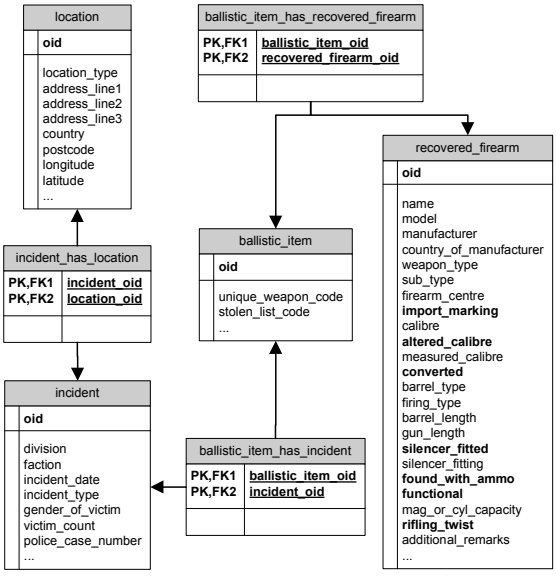


Figure 5. A partial E-R diagram presenting how a firearm and a location can be linked together

In Figure 6, we show how easy and straightforward it is to build a complex SQL-like JOIN across multiple tables from the above using the OSL. In fact, the task is almost effortless and does not require any understanding of the structure. The user does not need to be aware of a number or even classes of the tables that require joining.

```
QUERY firearm location WHERE
calibre HAS VALUE 0.22 AND
country HAS VALUE United Kingdom
```

Figure 6. Selecting a firearm linked to a location

What the user is asked to specify are concepts representing types of data and the constraints wants to apply onto the dataset. Hence, joining tables and merging resources is performed without the user's attention. In addition, what the OSL implementation enables is the integration of results from various data sources and services. This means a user can perform even more complex tasks with very similar effort; for example overlaying data with data-mining results. This level of abstraction creates a very powerful environment for non-technical users interacting with the system.

In contrast to Figure 6, the example below is of the similar expressiveness, but represented in a pure PL-SQL. According to the E-R diagram from Figure 5 the query would look like this:

```
SELECT rfa.oid, rfa.*, loc.oid, loc.*
FROM recovered_firearm rfa
LEFT JOIN ballistic_item_has_recovered_firearms bit_rfa ON
(bit_rfa.recovered_firearms_oid = rfa.oid)
LEFT JOIN ballistic_item ba ON
(ba.oid = bit_rfa.ballistic_items_oid)
LEFT JOIN ballistic_item_has_incidents bit_inc ON
(bit_inc.incident_oid = ba.oid)
LEFT JOIN incident inc ON
(inc.oid = bit_inc.incident_oid)
LEFT JOIN incident_has_locations inc_loc ON
(inc_loc.incident_oid = inc.oid)
LEFT JOIN location loc ON
(loc.oid = inc_loc.location_oid)
```

 ISBN: 978-1-61208-112-0

```
WHERE rfa.calibre = 0.22 AND loc.country = "United Kingdom";
```

Figure 7. SQL representation of the example

In the project, we prototyped a standalone client based on NetBeans RCP, enriched with visual features of the embedded Visual Library. This implementation fully supports the OSL and provides a graph-based visualisation facilitating search, browsing and what is more, reuse of search results in further investigations. We provide a user with a set of functionalities to visually manage multiple searches and results on a single screen at the same time. Besides, there is a text view available to the user, which is a document-based representation of graph content that seamlessly integrates with the visualisation.

Figure 8 below illustrates a search, whereby a user is looking for incidents linked to at least one of a previously identified person, under conditions such as location, timeframe, or an incident type.

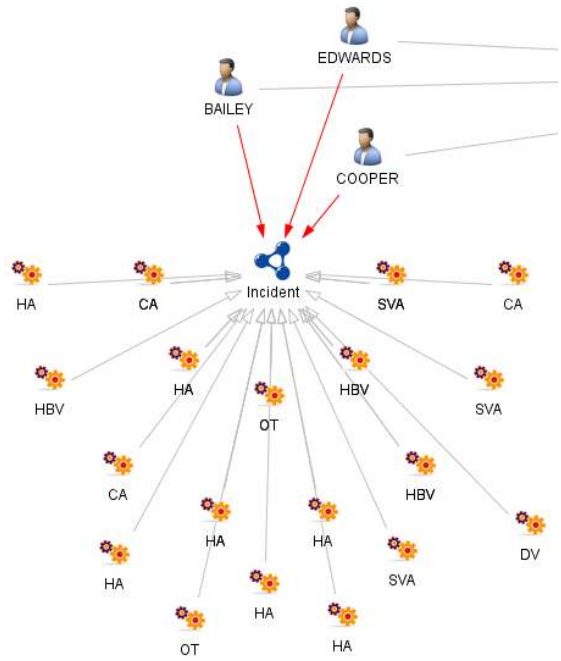


Figure 8. Visual representation of search and results with dependencies set between previous results and search

There are two types of widgets, with the first one used to build an OSL search query (labelled “Incident” in the example above) by setting properties of the class of data and dependencies from previous results (in this example, representing persons). The second type of widget represents results, which are data entries in the system, for example persons and incidents in the diagram above.

This graphical information retrieval and visualisation tool was introduced to guide and assist the end user in building, editing and executing OSL queries. The client provides a windows-based user interface that the end user is familiar with. Moreover, it offers a visualisation which is used to search, browse, but also receive alerts on new entries or updates in the database.

V. FUTURE WORK

Formalisation and standardisation of the OSL specification is one of the key areas which will be addressed after the language is presented to the user community and approved by the Odyssey Standardisation Committee.

Additionally, we will be exploring further possibilities of data mining techniques in order to extract and index information for further processing and analysis. Existing technologies recognise a vast potential in text-mining of personal statements and other plain-text documents gathered during an investigation. This would lead to further modification and new extensions to the language, which could not be addressed in this paper. For example, text-mining could result in entities such as person, location and vehicles extraction and mapping of the results on a timeline for further investigation and incident sequential analysis.

The OSL is currently under development and at this stage does not entirely cover all of the user needs and requirements; ideally, the user would be able to model processes and sequences of events that lead to or follow a crime. This is not a usual use of a domain-specific language and it might even seem to contradict with common practices. In general, DSLs model per-case solutions and do not explore the benefits of sequencing actions, events or outcomes. Therefore, usual DSL implementations are limited to a linear communication with a system rather than enabling the user to reason on data and automate the interaction with a system. Currently, the platform does not model nor visualises sequential data, but the need was widely discussed with the end user community. Furthermore, a solution based on mapping of crime and ballistic incidents on a timeline was proposed.

In summary, the future work will focus on formalisation and standardisation of solutions and practices described above, such as the DSL and processes the language is compatible with. Moreover, we will also investigate the potential of text-mining in the domain of crime investigations, which could potentially lead to changes in the OSL. Finally, we will research on how sequential data can be used of benefit and expressed in the OSL, in order to enable modelling of crime investigation processes and modelling of crime cases as such.

VI. CONCLUSIONS

The Odyssey project key result areas are the standardisation of data collection, storage and sharing, the facilitation of interoperability between existing systems and the provision of an infrastructure to both securely collaborate on cross national investigations and also extracting information through various data-mining and knowledge extraction techniques. These objectives of the Odyssey project lead to cost saving and increased efficiency, but also promote collaboration between law enforcement agencies across Europe through the use of information and communication technologies (ICTs).

In this paper, we presented how a domain-specific language can facilitate access to a platform by encapsulating data structures, enhancing security, but most importantly, enabling a non-technical user to interact with the platform through the use of a language suitable for the field of expertise.

We have designed a language according to the user requirements and prototyped a platform that makes the full use of its features. The OSL is used to access and manipulate data from multiple sources, collected by

 ISBN: 978-1-61208-112-0

various techniques and of different investigation value. Additionally, the OSL manages access to the user permissions and the sharing of data. The presented solution enables the end user to interact with the platform seamlessly switching between the OSL text- and the graph-based editor.

[1] Chau, M., Atabakhsh, H., Zeng, D., and Chen, H. (2001), 'Building an Infrastructure for Law Enforcement Information Sharing and Collaboration: Design Issues and Challenges', University of Arizona

[2] Su, S., Fortes, J., and Kasad, T.R. (2005), 'Transnational Information Sharing, Event Notification, Rule Enforcement and Process Coordination', International Journal of Electronic Government Research (IJEGR), pp. 52-62, 2005

[3] Travis, J. (1998), 'Informal Information Sharing Among Police Agencies', National Institute of Justice, December 1998

[4] Redmond, M. and Baveja, A. (2001), 'A data-driven software tool for enabling cooperative information sharing among police departments', European Journal of Operational Research, pp. 660-678, June 2001

[5] Chen, H., Schroeder, J., and Hauck, R. (2002), 'COPLINK Connect: information and knowledge management for law enforcement', University of Arizona, Decision Support Systems, pp. 271-285, 2003

[6] Yu, L. (2008), 'Prototyping, Domain Specific Language, and Testing', Engineering Letters, February 2008

[7] Mernik, M., Heering, J., and Sloane, A. (2005), 'When and how to develop domain-specific languages', ACM Computing Surveys, pp. 316-344, December 2005

[8] Spinellis, D. (1999), 'Notable design patterns for domain-speci®c languages', Department of Information and Communication Systems, University of the Aegean, pp. 91-99, December 1999

 ISBN: 978-1-61208-112-0