Reactive Semantics for User Interface Description Languages

Basile Pesin
basile.pesin@enac.fr

Celia Picard
celia.picard@enac.fr

Cyril Allignol
cyril.allignol@enac.fr

Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

User Interface Description Languages (UIDLs) are high-level languages that facilitate the development of Human-Machine Interfaces, such as Graphical User Interface (GUI) applications. They usually provide first-class primitives to specify how the program reacts to an external event (user input, network message), and how data flows through the program. Although these domain-specific languages are now widely used to implement safety-critical GUIs, little work has been invested in their formalization and verification.

In this paper, we propose a denotational semantic model for a core reactive UIDL, Smalite, which we argue is expressive enough to encode constructs from more realistic languages. This preliminary work may be used as a stepping stone to produce a formally verified compiler for UIDLs.

1 Introduction and Context

With the democratization of interactive devices, the general interaction paradigm has changed. Users now expect to interact with their systems through tactile interactions or advanced interactive devices. This extends to critical systems too, notably aviation related ones. For instance, aircraft cockpits are now digital and tactile [13] and paper strips for air traffic controllers have been replaced by elaborate digital dashboards [10].

Using traditional programming languages to implement the interactive parts of systems implies the use of numerous callbacks. The resulting code quickly becomes spaghetti code and gets particularly difficult to analyze and maintain [16, 17]. The use of dedicated languages has shown to be beneficial in such circumstances [18]. Such languages are called User Interface Description Languages (UIDLs). These languages can efficiently describe both the appearance of the system (its scene graph) and the interactive behavior (mainly activation propagation). They are becoming more popular, including for the development of critical systems [2, 1]. However, these languages have not been formally defined, particularly not their semantics, which hinders any formal reasoning on these safety-critical programs. In this paper, we tackle this problem by proposing denotational semantics for a minimal declarative UIDL.

There has been a few other attempts to give a formal semantics to interactive languages or libraries such as React [14]. However, React is based on a functional language and as such, has very different mechanisms than other UIDL/interactive languages. Indeed, the behavior of a React program is described using small-step operational semantics which specify the order in which components are rendered. In this paper, we focus on a declarative language with denotational semantics where the order of updates is left implicit.

Previous work has paved the way towards a minimal common abstract syntax for Smala, a declarative UIDLs [15]. A first formal semantic model based on bigraphs has been proposed [20],

To appear in EPTCS.

but in this work we explore an alternative expression of the semantics, which is simpler and should facilitate compiler verification in the fashion of CompCert [12] or Vélus [4]. To give more confidence in those semantics, we also aim at proving the equivalence between the two semantics, as future work.

Hence, we propose a new denotational semantics for Smalite, a declarative UIDL which includes a minimal set of constructs from Smala. As a preliminary step to a compiler correctness proof, we mechanized the language and its semantics in the Rocq prover [1] [23], and implemented a prototype compiler that generates imperative code. In section 2, we give an informal presentation of our language through an example program implementing a simple GUI program. Then, in section 3, we describe our formalization of the reactive semantics of the language. Finally, in section 4, we discuss the future steps towards extending the language and building a formal proof of correctness for its compiler.

2 Specifying interactions with Smala

The language we propose closely resembles Smala [15], a UIDL used in safety-critical applications [2, 7]. In fig. 1, we present an example Smala program where two buttons control a counter by either decrementing it until it reaches 0 or setting it back to 3.

Smala programs are composed of named processes. The root process is a `Component` (`line 1`), usually named `root`, which contains child processes. The counter is implemented as a property `count` (`line 2`), which is declared with a type and an expression giving its initial value. Then, the program contains a `Spike` (`line 3`), which represents an event that may be triggered from inside or outside the program, and reacted to. This particular `Spike` represents the event "the counter has just reached 0". Indeed, it is triggered by the binding on the same line, whose left-hand side is a condition checking that `count` equals 0. Conditions are checked only at reactions where the value of one of the properties involved changes; therefore, `zero` is triggered only at reactions where the value of `count` changes to 0.

The next process is a `Frame`, which is a special graphical component (`line 6`). For the purpose of our reactive semantics, a `Frame` is just syntactic sugar for a component with properties `title`, `width` and `height`, passed as parameters between parentheses, as well as a `Spike` `close`. On the other hand, our prototype compiler generates code which uses the `Frame` and its parameters to open a window. The compiler also binds user requests to close the window (e.g. by clicks on the ⊗ button) to triggering the `close` `Spike`. Therefore, the program can define custom reactions to user actions.

The first child of the `Frame` defines the `Font` used to display text in the interface (`line 7`). Since the rest of the program does not need to refer to this particular component, its name is unspecified (`_`). Then, the frame contains three main components: two buttons, and a text label. Each button is implemented by a component `FillColor` (with parameters `red`, `green` and `blue`), which sets the color of its `Rectangle` child (with parameters `x`, `y`, `width` and `height`). Text labels are implemented by a `FillColor` which sets the color of its `Text` child (with parameters `text`, `x` and `y`).

`Rectangle` components have two predefined spikes, `press` and `release`, which are respectively triggered when the mouse is pressed/released while the mouse pointer is on top of the rectangle. The logic of each button is implemented as a reaction to these spikes: on `press`, the

[1] formerly known as Coq

```
1  Component root {
2    Int count 3;
3
4    Spike zero; (count == 0) -> zero;
5
6    Frame f ("ICE␣2025", 300, 50) {
7      Font _ ("arial.ttf", 20) {
8        FillColor btn1 (150,150,150) {
9          Rectangle r (0,0,100,f.height) {
10           FillColor _ (0,0,0) {
11             Text _ ("decr", r.x + 30, 13) {}
12           };
13         };
14         r.press -> hg; hg: 255 =: green;
15         r.release -> dhg; dhg: 150 =: green;
16       };
17       btn1.r.release -> dec; dec: last count - 1 =: count;
18       zero ->! btn1.r; (count > 0) -> btn1.r;
19
20       FillColor btn2 (150,150,150) {
21         Rectangle<d> r (btn1.r.width+10,0,100,f.height) {
22           FillColor _ (0,0,0) {
23             Text t ("restart", r.x + 20, 13) {}
24           };
25         };
26         r.press -> hg; hg: 255 =: green;
27         r.release -> dhg; dhg: 150 =: green;
28       };
29       btn2.r.release -> rst; rst: 3 =: count;
30       (count < 3) -> btn2.r; (count == 3) ->! btn2.r;
31
32       FillColor _ (255,255,255) {
33         Text t ("rem:␣" + str(count), btn2.r.x+btn2.r.width+10, 13) {
34           count -> chtext; chtext: "rem:␣" + str(count) =: text;
35         };
36       }
37     }
38   };
39
40   Exit e (0) { f.close -> trigger };
41 }
```

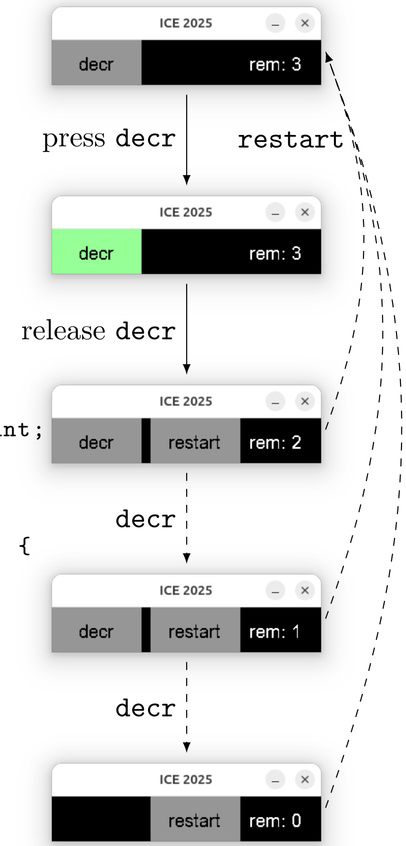

Figure 1: An example Smala program

button is highlighted by increasing the `green` property of its `FillColor` (`lines 14, 26`). On `release`, the `green` component is set back to its default value (`lines 15, 27`). The graphical effect of these first two reactions are shown at right of fig. 1.

The `release` `Spike` of each button is bound to an action on the counter. Releasing `decr` decreases the counter (`line 17`) by setting its new value to its previous value (accessed with `last`) minus one. Releasing `restart` sets the counter back to 3 (`line 29`). Buttons may be activated or deactivated depending on the value of the counter. If the counter equals 0, it is not possible anymore to decrease it, and therefore the `decr` button is deactivated (first binding `->! btn1.r` on `line 18`). As soon as the counter goes back above 0, it is reactivated. Conversely,

$$
\begin{aligned}
p ::=\ & ty\ x\ e \\
\mid\ & \texttt{Spike}\ x \\
\mid\ & x\texttt{:}\ e\ \texttt{=:}\ path \\
\mid\ & x\texttt{:}\ lhs\ \texttt{-><}ia\texttt{>}\ rhs \\
\mid\ & \texttt{Component<}ia\texttt{>}\ x\ \{p^*\}
\end{aligned}
\qquad
\begin{aligned}
path &::= x \mid x\texttt{.}path \\
e &::= c \mid path \mid \texttt{last}\ path \mid \diamond e \mid e \oplus e \\
ia &::= \texttt{a} \mid \texttt{d} \\
lhs &::= \texttt{T?(}path\texttt{)} \mid \texttt{A?(}path\texttt{)} \mid \texttt{D?(}path\texttt{)} \mid \texttt{C?(}path\texttt{)} \mid \texttt{(}e\texttt{)?} \\
rhs &::= \texttt{T!(}path\texttt{)} \mid \texttt{A!(}path\texttt{)} \mid \texttt{D!(}path\texttt{)}
\end{aligned}
$$

Figure 2: Abstract Syntax of Smalite

the `restart` button is deactivated when the counter equals 3, and reactivated otherwise (`line 30`). Normally, the activation of a parent automatically activates all of its children. However, since the counter starts at 3, the `restart` button must be initially deactivated, which is specified by `<d>` on `line 21`.

The last child component of the `Frame` is a `Text` label displaying the value of `count`. It is updated every time the value of `count` changes (binding `count ->` on `line 34`).

Finally, the program contains an `Exit` component, which takes an exit code as a parameter, and exposes a `Spike` `trigger` which halts the program. The `close` `Spike` of the `Frame` is bound to `trigger`, which allows the user to close the program by closing the window.

3 Formalizing Smalite's semantics

In this paper, we define the formal semantics of Smalite, a more restricted form of Smala. Smalite is a reactive language: the execution of a program can be seen as a series of reactions to external events. We saw in the example of fig. 1 that the main type of external event is the triggering of a `Spike`, such as `close` (`line 40`) or `release` (`line 15`). Another possible external event not showcased in the example would be an outside modification of a property. For instance, an adaptive GUI program could listen for the resizing of a `Frame` which would be encoded as a change of its `width` or `height` property.

A reaction may then update the state of the process by activating or deactivating permanent processes, or assigning values to properties. A reaction may also trigger spikes which, as seen in the example, may have an effect on the view. Therefore, we generalize external events to also account for events triggered by the program during a reaction, as defined below.

$$ev ::= \texttt{Trigger}\ path \mid \texttt{Assign}\ v\ path \mid \texttt{Activate}\ path \mid \texttt{Deactivate}\ path$$

First, an event can be the triggering of a "transient" process, which does not retain its activation between cycles (`Spike` or assign). Second, it can be an assignment to a property. Last, it can be the activation or deactivation of a "permanent" process, which retains its activation between cycles (`Component` or binding).

3.1 Abstract syntax of Smalite

We now detail the abstract syntax of Smalite as we formalize it. A minimalized version of Smala has been proposed in previous work [15]. It includes a small set of core elements that constitutes the heart of the language, expressive enough to describe the full set of components of the whole language while minimizing the number of constructs to actually formally define and verify. The

Concrete LHS	Concrete RHS	Abstract LHS	Abstract RHS	Event
`path ->`	`-> path`	`T?`(*path*)	`T!`(*path*)	`Trigger` *path*
`path ->`	N/A	`C?`(*path*)	N/A	`Assign` *v path*
`(e) ->`	N/A	(*e*)`?`	N/A	`Assign` *v path*, *path* $\in$ free(*e*)
`path ->`	`-> path`	`A?`(*path*)	`A!`(*path*)	`Activate` *path*
`path !->`	`->! path`	`D?`(*path*)	`D!`(*path*)	`Deactivate` *path*

Figure 3: Correspondence between concrete syntax, abstract syntax and events

translation from Smala to this minimal set is done through a transpilation pass not described here. Our syntax is presented in fig. 2, and builds upon the one proposed previously [15].

A program is a process *p*. There are five different processes detailed hereafter. A process may be the declaration of a property with its type *ty*, name *x* and initial value *e*. There are two kinds of transient processes, which are triggered during a single cycle: spikes and assignments. The latter assigns the result of the evaluation of an expression *e* to a property specified by an absolute *path*. An expression is either a constant *c*, the *path* to access the value or the `last` value of a property, or a unary ($\diamond$) or binary ($\oplus$) operation. Types, constants and operations are those of the host language that the compiler targets (in the case of our prototype compiler, CompCert C).

Finally, there are two kinds of permanent processes, which retain their activation between cycles. Both kinds may be initially activated or deactivated following the *ia* flag. The first permanent process is the binding, denoted by an arrow `->`. It binds the event specified by its left-hand-side to the event specified by its right-hand-side. An event detected by a left-hand-side may be the triggering (`T?`) of a transient process (`Spike`, assign), the activation (`A?`) or deactivation (`D?`) of a permanent process (binding, `Component`), the change of value of a property (`C?`), or a condition being true after one of its free variable changes ((*e*)`?`). Right-hand-sides may trigger a transient process (`T!`) or activate (`A!`) or deactivate (`D!`) a permanent process. The second permanent process is the component, which contains a list of sub-processes.

This abstract syntax is sligtly different from the concrete syntax used for the example. First, each process is explicitly named. Second, there is only one generic `Component`. It can then be instantiated to create all other components, particularly the graphical components (`Frame`, `Rectangle`, etc), as those do not play a special role in the semantics. Third, as we have seen, the left- and right-hand sides of bindings distinguish between triggering of transient processes, activation/deactivation of permanent processes, and change of property value. Figure 3 recapitulates the correspondence between concrete left- and right-hand-sides, as seen in the example, and their abstract counterpart. Last, processes refer to each other using their absolute paths, which are made up of a sequence of identifiers indicating the path from the root process to the process of interest.

In practice, our prototype compiler parses the source program of fig. 1 into an Abstract Syntax Tree (AST) represented as an inductive type in Rocq. Then, a sequence of functions recursively elaborates it into a second AST representing the more restricted syntax of fig. 2, in three passes. First, the elaborator fills-in missing names by generating globally unique identifiers. Then, it expands graphical components into generic `Component` by adding the predefined properties and spikes (e.g. `title`, `width`, `frame` for a `Frame`). Finally, it type-checks the program, makes relative paths absolutes, differentiates transient and permanent processes in bindings,

$$\boxed{\vdash^{\text{activ}}_{\text{init}} p_r \Downarrow A}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (ty\ x\ e)_{r_x} \Downarrow \emptyset}\ \text{IAP} \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Spike}\ x)_{r_x} \Downarrow \emptyset}\ \text{IAS} \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{:}\ e\ \texttt{=:}\ p)_{r_x} \Downarrow \emptyset}\ \text{IAA}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{:}\ lhs\ \texttt{-><d>}\ rhs)_{r_x} \Downarrow \emptyset}\ \text{IAB}_1 \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{:}\ lhs\ \texttt{-><a>}\ rhs)_{r_x} \Downarrow \{r_x.x\}}\ \text{IAB}_2$$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Component<d>}\ x\ \{ps\})_{r_x} \Downarrow \emptyset}\ \text{IAC}_1 \qquad \frac{\forall i.\ \vdash^{\text{activ}}_{\text{init}} (ps_i)_{r_x.x} \Downarrow A_i}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Component<a>}\ x\ \{ps\})_{r_x} \Downarrow (\bigcup_i A_i) \cup \{r_x.x\}}\ \text{IAC}_2$$

(a) Initialization of activation

$$\boxed{E \vdash^{\text{env}}_{\text{init}} p_r \Downarrow E'}$$

$$\frac{}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Spike}\ x)_{r_x} \Downarrow E}\ \text{IES} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (x\texttt{:}\ e\ \texttt{=:}\ p)_{r_x} \Downarrow E}\ \text{IEA} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (x\texttt{:}\ lhs\ \texttt{-><\_>}\ rhs)_{r_x} \Downarrow E}\ \text{IEB}$$

$$\frac{\emptyset, E \vdash_{\text{exp}} e \Downarrow v}{E \vdash^{\text{env}}_{\text{init}} (ty\ x\ e)_{r_x} \Downarrow E[r_x.x \mapsto v]}\ \text{IEP} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Component<\_>}\ x\ \{\varepsilon\})_{r_x} \Downarrow E}\ \text{IEC}_1$$

$$\frac{E \vdash^{\text{env}}_{\text{init}} p_{(r_x.x)} \Downarrow E' \qquad E' \vdash^{\text{env}}_{\text{init}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x} \Downarrow E''}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Component<\_>}\ x\ \{p;ps\})_{r_x} \Downarrow E''}\ \text{IEC}_2$$

(b) Initialization of environment

$$\boxed{\vdash_{\text{init}} p \Downarrow E, A}$$

$$\frac{\vdash^{\text{activ}}_{\text{init}} p_\varepsilon \Downarrow A \qquad \emptyset \vdash^{\text{env}}_{\text{init}} p_\varepsilon \Downarrow E}{\vdash_{\text{init}} p \Downarrow E, A}\ \text{I}$$

(c) Program initialization

Figure 4: Initialization rules

and builds the restricted AST.

3.2 Semantics of program initialization

We now describe the rules that specify the initialization of the system, presented in fig. 4. The first judgment, $\vdash^{\text{activ}}_{\text{init}} p_r \Downarrow A$, specifies the initial activation for a process p rooted at path r, where A is a set of paths of permanent processes. It is defined by the rules in fig. 4a. Non-permanent processes (property, spike, assignment) never appear in this set. Bindings and components appear in the set if-and-only-if they are declared with `<a>`. For a component marked with `<a>`, the sub-processes are initially activated according to the same rules.

The second judgment, $E \vdash^{\text{env}}_{\text{init}} p_r \Downarrow E'$, specifies how the initial property values are added to initial environment E to form E', where environments are represented by finite maps from prop-

$$\boxed{E, E' \vdash_{\text{exp}} e \Downarrow v}$$

$$\frac{}{E, E' \vdash_{\text{exp}} c \Downarrow c^{\#}}\ \text{EC} \qquad \frac{}{E, E' \vdash_{\text{exp}} x \Downarrow E'(x)}\ \text{EV} \qquad \frac{}{E, E' \vdash_{\text{exp}} \texttt{last}\ x \Downarrow E(x)}\ \text{EL}$$

$$\frac{E, E' \vdash_{\text{exp}} e_1 \Downarrow v_1 \qquad \diamond^{\#}(v_1) = \lfloor v \rfloor}{E, E' \vdash_{\text{exp}} \diamond\, e_1 \Downarrow v}\ \text{EU} \qquad \frac{E, E' \vdash_{\text{exp}} e_1 \Downarrow v_1 \qquad E, E' \vdash_{\text{exp}} e_2 \Downarrow v_2 \qquad \oplus^{\#}(v_1, v_2) = \lfloor v \rfloor}{E, E' \vdash_{\text{exp}} e_1 \oplus e_2 \Downarrow v}\ \text{EB}$$

Figure 5: Expression evaluation rules

erty paths to values. It is defined by the rules in fig. 4b. As expected, spikes, assignments and bindings have no effect on the initial environment. For properties (rule IEP), the initialization expression is evaluated under the starting environment E and the resulting value is added to E. A value is either a numerical value from the C language (integer, floating-point number), a boolean value, or a string of characters. The rules for expression evaluation specifying $E, E' \vdash_{\text{exp}} e \Downarrow v$ are unsurprising, and presented in fig. 5. The notation $c^{\#}$ represents the semantic denotation of constant c (respectively $\diamond^{\#}$ for unary operator $\diamond$ and $\oplus^{\#}$ for binary operator $\oplus$). The current value of a variable is searched in E', the updated environment (rule EV), while its `last` value is searched in E, which represents the environment at the end of the last step, and is empty at initialization (rule EL). The evaluation of operators may fail (e.g. in case of a division by zero), which is denoted on their return value by $\lfloor v \rfloor$.

Rules IEC_1 and IEC_2, for components, highlight an interesting design choice. In our definitions, the environment is updated sequentially, following the order the sub-processes are written in the program. One simpler definition would have been to update the same environment E in parallel for each sub-process ps_i into an environment E_i, and to take the union of these environments, but this would have prevented writing initial expressions that depend on the values of other properties, as seen for the button text position in fig. 1 (`line 21`). Conversely, a more expressive semantics would allow the initialization of a property to depend on the values of properties later in the program, as long as there is no cycle in the definitions. This would however require (1) a more complicated semantic judgment for initialization, (2) an analysis of the absence of cycle at the source level, and (3) a way to schedule property initialization in the compiled code. Although these are all feasible, we believe that the marginal expressivity gains are not sufficient to justify this additional complexity.

Finally, the two judgments described above are combined in rule I, presented in fig. 4c. It specifies the initialization of a Smalite program: $\vdash_{\text{init}} p \Downarrow E, A$ indicates that the process p, rooted at the empty path ε, is initialized with initial environment E and initial activation A.

3.3 Semantics of reaction

To describe how a program reacts to an external event, let us start with the main reaction rule presented in fig. 6. The judgment $E, A \vdash_{\text{react}} p(ev) \Downarrow E', A', T'$ indicates that, with initial environment E and activation A, the process p reacts to an external event ev by updating its environment to E', its activation to A', and emitting a set T of external events. The relation between these parameters are specified by three premises, which we now detail.

$$\boxed{E, A \vdash_{\text{react}} p(ev) \Downarrow E', A', T}$$

$$\frac{T = \{ev_0\} \cup \{ev \mid E, A, T, E', A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev\} \qquad E, T, E', A' \vdash^{\text{proc}}_{\text{safe}} p_\varepsilon \qquad E, A \vdash_{\text{update}} T \Downarrow E', A'}{E, A \vdash_{\text{react}} p(ev_0) \Downarrow E', A', (T \cap \{\texttt{Trigger}\ path \mid \texttt{Spike}\ path \in p\})}\ \text{R}$$

Figure 6: Main reaction rule

3.3.1 Event propagation

The first premise determines the set T of events that are emitted during the reaction step. T contains the external event that triggered the reaction, ev. It also contains any additional event propagated by the process: this is represented by judgment $E, A, T, E', A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev$ which can be read as "when process p reacts to events T, it also emits event ev". The rules defining this judgment are presented in figs. 7 and 8. These rules are not directly syntax-directed: instead, each one asserts the existence of a sub-process that may propagate events. We write the judgement $p[path] = \lfloor p' \rfloor$ for "there is a process p' rooted at $path$ in p".

The first set of rules, in fig. 7, describes how bindings propagate events. Rule PB, presented at the bottom, specifies that a binding only propagates events if it is activated. The third premise $E, T, E' \vdash^{\text{lhs}}_{\text{activ}} lhs$ of the rule assumes that the left-hand side is activated by a matching event. In particular, if the left-hand side is a condition, it must evaluate to `true`. The final premise $A, A' \vdash^{\text{rhs}}_{\text{prop}} rhs \Downarrow ev$ specifies which event ev the right-hand side emits. All rules specifying this judgment mandate that the parent of the process involved in the event is active. Activation (resp. deactivation) events are only emitted if the process was previously deactivated (resp. activated): in other words, "non-events" that do not modify the activation state are not propagated.

The last rule in fig. 7, covers the assignment. It states that an assignment produces an `Assign` v y event when (1) its parent is activated, (2) the parent of y is activated, (3) the assignment has been triggered by another event, and (4) the expression of the assignment evaluates to value v. Here, we made one interesting choice: the second premise implies that it is not possible to assign to a property whose parent is deactivated. This constraint was added because the behavior of programs where a property is assigned to when its parent is deactivated was not clear: should the assignment event be propagated during the reaction, or when the parent becomes active again, or not at all ? We chose to eliminate this question by forbidding this case entirely, as we believe it is not useful in real-world programs. In practice, our prototype compiler statically checks that this situation can never arise.

The last set of rules, in fig. 8, describes how components propagate events to their children. Rule PC_1 specifies the propagation of activation events, while rule PC_2 specifies the propagation of deactivation events. Each is specified by a judgment $\vdash^{\text{(de)activ}}_{\text{prop}} p \Downarrow ev$, which follows the same logic as the judgment for initial activation. The only difference between these two sets of rules is that activation is only propagated to children for components/bindings marked with `<a>`, while deactivation is always propagated.

3.3.2 Safe reactions

The propagation rule by itself is not enough to ensure the right semantics is given to propagation of events. Indeed, consider the pair of processes `a: 0 =: y; (x/y > 10) -> t` and suppose that

$$\boxed{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \mathit{lhs}}$$

$$\frac{\texttt{Trigger}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{T?(}x\texttt{)}}\ \text{PLT} \qquad \frac{\texttt{Activate}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{A?(}x\texttt{)}}\ \text{PLA} \qquad \frac{\texttt{Deactivate}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{D?(}x\texttt{)}}\ \text{PLD} \qquad \frac{\texttt{Assign}\ \mathrm{v}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{C?(}x\texttt{)}}\ \text{PLC}$$

$$\frac{x \in \mathsf{free}(e) \qquad \texttt{Assign}\ \mathrm{v}\ x \in T \qquad E,E' \vdash_{\text{exp}} e \Downarrow \texttt{true}}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{(}e\texttt{)?}}\ \text{PLI}$$

$$\boxed{A,A' \vdash^{\text{rhs}}_{\text{prop}} \mathit{rhs} \Downarrow \mathit{ev}}$$

$$\frac{r_x \in A'}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{T!(}r_x.x\texttt{)} \Downarrow \texttt{Trigger}\ r_x.x}\ \text{PRT} \qquad \frac{r_x \in A' \qquad r_x.x \notin A}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{A!(}r_x.x\texttt{)} \Downarrow \texttt{Activate}\ r_x.x}\ \text{PRA}$$

$$\frac{r_x \in A' \qquad r_x.x \in A}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{D!(}r_x.x\texttt{)} \Downarrow \texttt{Deactivate}\ r_x.x}\ \text{PRD}$$

$$\boxed{E,A,T,E'A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \mathit{ev}}$$

$$\frac{p[r_x.x] = \lfloor x\texttt{:}\ \mathit{lhs}\ \texttt{-><\_>}\ \mathit{rhs} \rfloor \qquad r_x.x \in A' \qquad E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \mathit{lhs} \qquad A,A' \vdash^{\text{rhs}}_{\text{prop}} \mathit{rhs} \Downarrow \mathit{ev}}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \mathit{ev}}\ \text{PB}$$

$$\frac{p[r_x.x] = \lfloor x\texttt{:}\ e\ \texttt{=:}\ r_y.y \rfloor \qquad r_x \in A' \qquad r_y \in A' \qquad \texttt{Trigger}\ r_x.x \in T \qquad E,E' \vdash_{\text{exp}} e \Downarrow v}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Assign}\ \mathrm{v}\ r_y.y}\ \text{PA}$$

Figure 7: Propagation rule for bindings and assignments

assignment `a` is triggered. According to our semantics, it generates an event `Assign 0 y`. In turn, this triggers the conditional binding, as the value of `y` as changed. However, the behavior of `x/y` is undefined because it divides by 0. This means that the predicate for expression evaluation does not apply, and therefore the premises of the rule for conditional left-hand side activation does not hold. In the end, our semantics say that this program evaluates fine, but does not trigger `t`. This is wrong: actually, if we compile and/or execute this program, it will crash. Therefore, this program should not admit a semantics at all. To correct for these cases, we introduce an additional premise in fig. 6, $E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} p_r$, which ensures that all expressions that need to be evaluated in the process p rooted at r evaluate without undefined behavior.

The definition of this judgment is presented in fig. 9. Execution of a process that does not propagate events (property, `Spike`) is always safe. An assignment is safe if, when triggered, the evaluation of its expression is safe (rule SA). Both bindings and components are safe if they are inactive (rules SB_1 and SC_1). If an assignment is active, then its left-hand side must be safe: if the left-hand side is a condition, either there is no assignment to its free variables, in which case it does not need to be evaluated (rule SLI_1), or it evaluates to a boolean value (rule SLI_2). An active component is safe if all its children processes are safe (rule SC_2).

$$\boxed{\vdash^{\text{activ}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{prop}} (x\texttt{:}\ lhs\ \texttt{-><a>}\ rhs)_{r_x} \Downarrow \texttt{Activate}\ r_x}\ \text{PAB}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{prop}} (\texttt{Component<a>}\ x\ \{ps\})_{r_x} \Downarrow \texttt{Activate}\ r_x.x}\ \text{PAC}_1 \qquad \frac{\vdash^{\text{activ}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow ev}{\vdash^{\text{activ}}_{\text{prop}} (\texttt{Component<a>}\ x\ \{ps\})_{r_x} \Downarrow ev}\ \text{PAC}_2$$

$$\boxed{\vdash^{\text{deactiv}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{}{\vdash^{\text{deactiv}}_{\text{prop}} (x\texttt{:}\ lhs\ \texttt{-><\_>}\ rhs)_{r_x} \Downarrow \texttt{Deactivate}\ r_x.x}\ \text{PDB}$$

$$\frac{}{\vdash^{\text{deactiv}}_{\text{prop}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x} \Downarrow \texttt{Deactivate}\ r_x.x}\ \text{PDC}_1 \qquad \frac{\vdash^{\text{deactiv}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow ev}{\vdash^{\text{deactiv}}_{\text{prop}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x} \Downarrow ev}\ \text{PDC}_2$$

$$\boxed{E,A,T,E'A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{p[r_x.x] = \lfloor \texttt{Component<a>}\ x\ \{ps\} \rfloor \qquad \texttt{Activate}\ r_x.x \in T \qquad \vdash^{\text{activ}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow \texttt{Activate}\ y}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Activate}\ y}\ \text{PC}_1$$

$$\frac{p[r_x.x] = \lfloor \texttt{Component<\_>}\ x\ \{ps\} \rfloor \qquad \texttt{Deactivate}\ r_x.x \in T \qquad \vdash^{\text{deactiv}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow \texttt{Deactivate}\ y}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Deactivate}\ y}\ \text{PC}_2$$

Figure 8: Propagation rules for components

3.3.3 State update

The two judgments described above specify what events are emitted during the reaction. It remains to specify how the state of the reactive system is updated by these events. This is the role of the judgment $E,A \vdash_{\text{update}} T \Downarrow E',A'$, which specifies that "when it receives events T, the state of a process updates from (E, A) to (E', A')". The unique rule specifying this judgment is presented in fig. 10.

Its first two premises specify the updated environment E': for each path p, either there is an `Assign` event to p, in which case $E'(x)$ is set to its value, or there is none, in which case $E'(x)$ keeps the same value as $E(x)$. The first premise implies a strong constraint of Smalite programs, that was not explicit until now: two assignments to the same property with different values may not occur during the same reaction. This is a stronger constraint than was chosen in the original implementation of Smala (where the value of the "final" assignment was kept), but it simplifies the semantic model and facilitates the generation of simple and efficient imperative code. In practice, this semantic constraint is satisfied for any program that respects a Reactive Static Single Assignment (RSSA) property: each reaction only contains one assignment to a given property. This property is checked by our prototype compiler.

The remaining premises specify the updated activation A': an `Activate` p event adds p in A', a `Deactivate` p event removes it. If neither type of events are emitted, then p keeps its activation

$$\boxed{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} lhs}$$

$$\frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} x}\ \text{SLT} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{A?}(x)}\ \text{SLA} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{D?}(x)}\ \text{SLD} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{C?}(x)}\ \text{SLC}$$

$$\frac{\forall x \in \mathsf{free}(e),\ \forall v,\ \texttt{Assign}\ v\ x \notin T}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} (e)\texttt{?}}\ \text{SLI}_1 \qquad \frac{E,E' \vdash_{\text{exp}} e \Downarrow v \qquad v \in \{\texttt{true},\texttt{false}\}}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} (e)\texttt{?}}\ \text{SLI}_2$$

$$\boxed{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} p_r}$$

$$\frac{}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (ty\ x\ e)_{r_x}}\ \text{SP} \qquad \frac{}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Spike}\ x)_{r_x}}\ \text{SS} \qquad \frac{\texttt{Trigger}\ r_x.x \in T \implies E,E' \vdash_{\text{exp}} e \Downarrow v}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (x\texttt{:}\ e\ \texttt{=:}\ y)_{r_x}}\ \text{SA}$$

$$\frac{r_x.x \notin A'}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (x\texttt{:}\ lhs\ \texttt{-><\_>}\ rhs)_{r_x}}\ \text{SB}_1 \qquad \frac{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} lhs}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} x\texttt{:}\ lhs\ \texttt{-><\_>}\ rhs}\ \text{SB}_2$$

$$\frac{r_x.x \notin A'}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x}}\ \text{SC}_1 \qquad \frac{\forall i,\ E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} ps_{i_{r_x.x}}}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x}}\ \text{SC}_2$$

Figure 9: Safe reaction rules

$$\boxed{E,A \vdash_{\text{update}} T \Downarrow E',A'}$$

$$\frac{\begin{array}{c}\forall p\ v,\ \texttt{Assign}\ v\ p \in T \implies E'(p) = \lfloor v \rfloor \qquad \forall p,\ (\forall v,\ \texttt{Assign}\ v\ p \notin T) \implies E'(p) = E(p) \\ \forall p,\ \texttt{Activate}\ p \in T \implies p \in A' \qquad \forall p,\ \texttt{Deactivate}\ p \in T \implies p \notin A' \\ \forall p,\ (\texttt{Activate}\ p \notin T \wedge \texttt{Deactivate}\ p \notin T) \implies (p \in A' \iff p \in A)\end{array}}{E,A \vdash_{\text{update}} T \Downarrow E',A'}\ \text{U}$$

Figure 10: State update rule

status. These premises imply a second constraint: `Activate` p and `Deactivate` p events may not occur during the same reaction. Like the previous constraint, this choice simplifies both the semantics and compilation scheme, and is checked statically during compilation.

3.3.4 Discussion

Going back to the rule that composes them in fig. 6, we can now get a high-level view of how these three judgments interact interact to specify how a Smalite process reacts to an event. As discussed, the *prop* and *safe* rules specify which events are produced, while the *update* rule specifies how state is updated. These definitions appear to be mutually recursive. On the one hand, in *prop*, the updated state E',A' is used as argument to compute generated events. On the other hand, in *update*, the set of events T is used as an argument to compute the updated state E',A'. To make these semantics executable, one would most likely need to implement them as a pair of mutually-defined fixed-points.

4 Conclusion and Future work

In this paper, we have presented Smalite, a minimal language capable of encoding more fully-featured UIDLs such as Smala. The denotational semantics of the language, along with a prototype compiler, have been implemented in the Rocq prover: this is a first stepping stone towards a verification framework for UIDLs in general and Smala in particular. We propose here some leads for future work.

4.1 A formally verified compiler for Smalite

We have implemented a prototype compiler for Smalite programs that generates C code which can be linked against the SDL library [22] to define reactive GUI programs. This compiler generates executable code for Smalite programs by finding a static scheduling of instructions that implement events (setting the value of property, (un)setting activation flag), guarded by conditions that emulate the left-hand side of bindings. This involves three major passes between separate Intermediate Representations (IRs):

1. Explicitly compute the event propagation graph of the program, where vertices are instructions and edges are guarded by conditions that correspond to the left-hand side of bindings
2. Checking that this graph does not contain any cycle, and flattening it by transforming its transitive closure into direct associations between external events and lists of guarded instructions
3. For each external event, schedule the guarded instructions according to dependencies (e.g. an assignment to `x` must be processed before an instruction that depends on `x`), and generate a function that implements the reaction to the event

Our prototype compiler targets the Obc IR of the Vélus compiler [5]. Obc is an imperative object-oriented language where each class has fields and methods. In our compiler, we use fields to store the values of properties and the activation state of permanent processes, and generate one method for each external event. Then, we reuse the Obc-to-Clight pass of Vélus to produce a Clight program. Clight is one of the frontend language of the CompCert verified compiler [12] on which Vélus, and therefore our prototype compiler, are based.

In the future, we hope to reuse the correctness proofs of Vélus and CompCert to build an end-to-end semantics preservation proof from the denotational semantics presented in this paper down to the assembly semantics provided by CompCert. The missing piece is, of course, a correctness proof that relates our denotational semantics for Smalite to the operational semantics of Obc [5, §4.1.2]. We expect this proof to be difficult, for three reasons. First, our denotational semantics model is clearly more abstract than the operational semantics used for Obc: the former asserts the existence of objects on which a set of relations hold, while the latter describes precisely how these objects are computed. Second, the correctness of our compilation scheme heavily depends on the well-formedness of the source program (absence of dependency loops, absence of contradictory assignments, etc.) which might be difficult to specify precisely and reason about. Last, the rule for reactions described in fig. 6 relies on a complex predicate to define the content of the set of events T. We are afraid that reasoning about this predicate in semantics preservation proofs will require proving implications in two directions (source-to-target and target-to-source), which might incur a significant amount of work for each compilation pass.

4.2 Static analysis, simplifications and optimizations

Our prototype compiler is, in some respects, very naïve, and future work will focus on improving it so that it generates more efficient imperative code, and accepts a larger set of source Smalite programs.

Indeed, compilation to imperative code with a fixed schedule places a restriction on which programs may be accepted: some programs with well-defined and deterministic semantics may be rejected during compilation. For instance, consider a program with only two bindings: `x -> y; y -> x`. In theory, this program is not schedulable, as there is a dependency loop between the triggering of `x` and `y`. In practice however, the semantics of this program are clear: either `x` and `y` are both triggered, or they both are not. To generate imperative code, we need to somehow cut the dependency cycle, but it is not obvious how to do so in general.

Another, more complex loop is actually showcased in our example, on `lines 29-30`. `Line 29` specifies that releasing `btn2` sets `count` back to `3`. `Line 30` specifies that changing `count` triggers a test that activates `btn2` if `count < 3`. How should these two bindings be scheduled? On the one hand, before checking the first binding, it must be determined whether or not `btn2` is active, so `line 30` should be scheduled before `line 29`. On the other hand, `line 29` sets `count`, so it should be scheduled before `line 30` that listens to a change on `count`. This looks like a dependency loop, but on closer inspection the second scheduling is never useful: indeed, `line 29` sets `count` to `3`, therefore, `count < 3` will never be true on a cycle where this assignment is activated. A general way to filter this type of false dependency loops would be to "cut" chains of bindings that include conditions that will never evaluate to true. To do so, we could use some type of static analysis such as abstract interpretation [6]. Moreover, statically simplifying conditions and cutting useless bindings would also make the generated code more efficient.

4.3 Extending Smalite

The example presented in section 2 could be simplified with a few higher-level constructs. First, we see a very similar code pattern of declaring an assignment and having a unique binding to that assignment at several places (`lines 14, 15, 17, 29, ...`). These could be simplified with a meta-process that encodes "binding to an assignment". In Smala, this is implemented by assignment sequences, with syntax `press -> { 255 =: green }`. This feature could be added as part of a more general source language than the one we present in this paper, and compiled down to the simpler constructs of Smalite.

Another unwelcome repetition appears in the definitions of the buttons `btn1` and `btn2`, which are essentially identical bar a few parameter. The full Smala language allows the user to define parameterized components which can then be instantiated in more complex programs. It is not yet clear what would be the best way to treat such a feature in our formalization: either user-defined components could be inlined into a single Smalite program, or they could be compiled separately, making Smalite and the other intermediate representations more complex.

4.4 Graphical semantics and properties

The semantic model we propose in this paper describes how the internal state of a program is updated in reaction to an event. It does not specify how this internal state is related to the observable behavior of the program. In particular, all the components used in the example (`Frame`, `Rectangle`, etc) have graphical semantics: their activation, and the value of their property affects what is displayed on the screen. Furthermore, the events that are modeled by spikes (`close`, `released`, etc) correspond to user actions. The structure of graphical components could be formalized as a scene graph, while the behavior of interactions could be specified using low-level events (click on the mouse at specific coordinates, etc). Modeling these graphical and interactive semantics at the level of the Smala source language would facilitate two high-level goals.

First, mechanize a compilation correctness proof for the code that binds the reactive program to the system libraries that implement user interactions; for our prototype compiler, that would be SDL3. Writing this proof would first require axiomatizing the behavior of all the SDL3 functions in use: drawing functions would affect the scene graph, while event-listening functions would be related to low-level events.

Second, it would be possible to reason on graphical properties of the system. Indeed, many safety-critical interactive systems, such as airplane cockpit GUIs are bound by strict norms. For instance, the ED 143 [9] specifies the behavior of the Traffic Alert and Collision Avoidance System (TCAS), which prevents aircraft from crashing into each other by alerting the pilot of imminent collisions and requesting altitude change. In particular, it specifies how incoming aircraft should be displayed on the screen.

In [21], the authors present a technique to formally verify that this specification holds for a given implementation, by generating Weakest Preconditions (WPs) and verifying them using the Z3 SMT Solver [8]. We could ground these results by proving the correspondence of the WP generation algorithm with our semantics model, and using a tool such as SMTCoq [3] to transport the proof of correctness generated by Z3 into Rocq logic. This would give us a mechanized Rocq proof that these properties hold for the source program and, by applying the compiler correctness proof mentioned in section 4.1, that they hold for the generated binary.

Another approach would be to reuse the work proposed in [20, 19], which proposes a denotational semantic model based on bigraphs for a UIDL very similar to ours. It then uses the PRISM model checker [11] to prove that the resulting bigraphical reactive system implies graphical properties of interest. To take advantage of these results, we would first need to prove that our own relational semantic model is equivalent to the bigraphical model.

References

[1] Govindaraj Anand et al. "Avionics display for Two-Seated aircraft using OpenGL". In: Artificial Intelligence and Evolutionary Computations in Engineering Systems: Computational Algorithm for AI Technology, Proceedings of ICAIECES 2020. Springer. 2022, pp. 255–265. doi: `10.1007/978-981-16-2674-6_20`.

[2] Philippe Antoine and Stéphane Conversy. "Volta: The First All-Electric Conventional Helicopter". In: MEA 2017, More Electric Aircraft. Bordeaux, France, Feb. 2017. url: `https://enac.hal.science/hal-01609233`.

[3] Michael Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: Certified Programs and Proofs. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150. isbn: 978-3-642-25379-9. doi: `10.1007/978-3-642-25379-9_12`. url: `http://link.springer.com/10.1007/978-3-642-25379-9_12` (visited on 04/03/2025).

[4] Timothy Bourke, Basile Pesin, and Marc Pouzet. "Verified Compilation of Synchronous Dataflow with State Machines". In: ACM Trans. Embed. Comput. Syst. 22 (5s Sept. 9, 2023), 137:1–137:26. issn: 1539-9087. doi: 10.1145/3608102. (Visited on 11/19/2024).

[5] Lélio Brun. "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset". PhD thesis. PSL Research University, June 2020. url: https://theses.hal.science/tel-03068862/.

[6] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs". In: Proceedings of the Second International Symposium on Programming. Dunod, Paris, France, 1976, pp. 106–130.

[7] Mathieu Cousy et al. "AEON: Toward a concept of operation and tools for supporting engine-off navigation for ground operations". In: Toward Sustainable Aviation Summit (TSAS) 2022. Toulouse, France, Oct. 2022, pp. 1–12. url: https://enac.hal.science/hal-03859908.

[8] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: Tools and Algorithms for the Construction and Analysis of Systems. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. isbn: 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24.

[9] EUROCAE. ED 143 - Minimum Operational Performance Standards For Traffic Alert And Collision Avoidance System II (TCAS II). url: https://standards.globalspec.com/std/1609213/eurocae-ed-143.

[10] Stephan Huber, Johanna Gramlich, and Tobias Grundgeiger. "From Paper Flight Strips to Digital Strip Systems: Changes and Similarities in Air Traffic Control Work Practices". In: CSCW. New York, NY, USA: Association for Computing Machinery, May 2020. doi: 10.1145/3392833.

[11] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[12] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: Communications of the ACM 52.7 (July 2009), pp. 107–115. issn: 0001-0782, 1557-7317. doi: 10.1145/1538788.1538814.

[13] Yixiang Lim et al. "Avionics Human-Machine Interfaces and Interactions for Manned and Unmanned Aircraft". In: Progress in Aerospace Sciences 102 (2018), pp. 1–46. issn: 0376-0421. doi: 10.1016/j.paerosci.2018.05.002.

[14] Magnus Madsen, Ondřej Lhoták, and Frank Tip. "A Semantics for the Essence of React". In: LIPIcs, Volume 166, ECOOP 2020 166 (2020). Ed. by Robert Hirschfeld and Tobias Pape, 12:1–12:26. issn: 1868-8969. doi: 10.4230/LIPICS.ECOOP.2020.12.

[15] Mathieu Magnaudet et al. "Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming". In: Proc. ACM Hum.-Comput. Interact. 2 (EICS June 19, 2018), 12:1–12:27. doi: 10.1145/3229094.

[16] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. Tech. rep. EPFL-REPORT-148043. Lausanne: EPFL, 2010.

[17] Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. “Causette: user-controlled rearrangement of causal constructs in a code editor”. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. 2022, pp. 241–252. doi: 10.1145/3524610.3527885.

[18] Brad A. Myers. “Separating application code from toolkits: Eliminating the spaghetti of call-backs”. In: Proceedings of the 4th annual ACM symposium on User interface software and technology. 1991, pp. 211–220. doi: 10.1145/120782.120805.

[19] Nicolas Nalpon. “Vers la vérification des langages de description d’interface utilisateur”. PhD thesis. INSA de Toulouse, Mar. 13, 2023. url: https://theses.hal.science/tel-04139613 (visited on 11/20/2024).

[20] Nicolas Nalpon, Cyril Allignol, and Celia Picard. “Towards a User Interface Description Language Based on Bigraphs”. In: Theoretical Aspects of Computing – ICTAC 2022. Ed. by Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu. Vol. 13572. Cham: Springer International Publishing, 2022, pp. 360–368. isbn: 978-3-031-17715-6. doi: 10.1007/978-3-031-17715-6_23.

[21] Daniel Prun and Pascal Béger. “Formal Verification of Graphical Properties of Interactive Systems”. In: Proceedings of the ACM on Human-Computer Interaction 6 (EICS June 14, 2022), pp. 1–30. issn: 2573-0142. doi: 10.1145/3534521.

[22] Simple DirectMedia Layer project. SDL Wiki. url: https://wiki.libsdl.org/SDL3/FrontPage.

[23] The Coq Development Team. The Coq Proof Assistant. Version 8.20. Zenodo, Sept. 4, 2024. doi: 10.5281/zenodo.14542673.