Modular Verification of Programs with Effects and Effect Handlers in Coq

Thomas Letan[1,2], Yann Régis-Gianas[3,4], Pierre Chifflier[1], and Guillaume Hiet[2]

[1] French Network Information Security Agency (ANSSI)
[2] CentraleSupélec, Inria Rennes – Bretagne Atlantique, IRISA-D1
[3] Univ Paris Diderot, Sorbonne Paris Cité, IRIF/PPS, UMR 8243 CNRS
[4] PiR2, Inria Paris-Rocquencourt

Abstract. Modern computing systems have grown in complexity, and the attack surface has increased accordingly. Even though system components are generally carefully designed and even verified by different groups of people, the *composition* of these components is often regarded with less attention. This paves the way for "architectural attacks", a class of security vulnerabilities where the attacker is able to threaten the security of the system even if each of its components continues to act as expected. In this article, we introduce FreeSpec, a formalism built upon the key idea that components can be modelled as programs with algebraic effects to be realized by other components. FreeSpec allows for the modular modelling of a complex system, by defining idealized components connected together, and the modular verification of the properties of their composition. In addition, we have implemented a framework for the Coq proof assistant based on FreeSpec.

1 Introduction

A typical computing platform is made of dozens of hardware components, and some of them execute complex software stacks. In this context, building a secure computing system with respect to a given security policy remains challenging, because attackers will leverage any vulnerability they can find. Both local component flaws and components composition inconsistencies, that is, a mismatch between requirements assumed by some client components and the actual guarantees provided by others, can be used by attackers.

The latter scenario may lead to a situation where every component seems to be working as expected, but their composition creates an attack path. We name this class of security vulnerabilities "architectural attacks" [1]. Over the past decade, many critical vulnerabilities affecting computing systems, in particular those relying on the x86 architecture, have raised awareness about the threat posed by architectural attacks. Figure 1 summarizes several significant attacks [2,3,4,5,6,7] inside an idealized view of an x86 computing platform. In all cases, the vulnerability was rooted in an inconsistency in the components' composition.

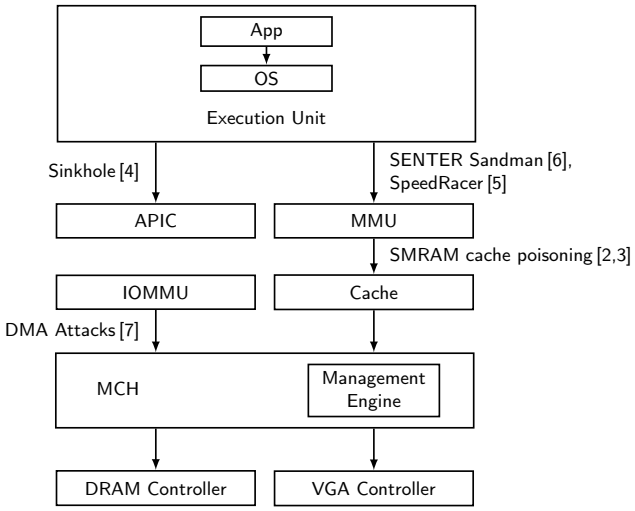

Fig. 1: Idealized x86 Computing Platform

The isolation of the System Management Mode (SMM) code by the hardware architecture is a good example to illustrate the threat posed by architectural attacks. The SMM is the most privileged execution mode of x86 CPU. Its purpose is to provide an "isolated processor environment that operates transparently to the operating system" [8] to execute so-called SMM code provided by the computer manufacturer. Since the SMM code is the most privileged software component executed by an x86 CPU, it is a desirable target for an attacker. The SMM code is stored in a dedicated memory region within the system memory, called the SMRAM; it is expected that only a CPU in SMM can access the SMRAM. In particular, the Memory Controller Hub (MCH) [9] provides a security mechanism to that end. In 2009, Duflot *et al.* [3] and Wojtczuk *et al.* [2] have independently shown that the cache could be used to circumvent this protection. The countermeasure required adding a new security mechanism to the CPU, meaning only computers produced after 2009 are protected against this particular vulnerability.

For many years, industrial manufacturers [10,11] and researchers [12,13] have aimed to formally specify and verify hardware architectures. However, verifying properties of existing computing platforms poses significant challenges, because they tend to be both complex and under-specified; we are not aware of any model of an existing and broadly used computing system that is comprehensive in terms of its hardware and software components. Unfortunately, such a model is a prerequisite to verify a computing platform in terms of architectural attacks. The Coq proof assistant [14] has proven to be effective to model specific hardware components [15,16,13,17]. It provides a rich specification language, tools to write machine-checked proofs and a mechanism to derive executable programs to experimentally validate models. The scale of the task dictates several require-

ments regarding the formalism to adopt in this particular case. It must allow for considering independently each component of our system, before composing them to conclude about the properties of the system as a whole.

This objective is reminiscent of the programming language problematic to model and verify large programs with side effects. Reasoning about side effects in purely functional languages such as Gallina, the Coq specification language, is difficult, firstly because they require somehow taking into account an outer *stateful* environment and secondly, because the composition of stateful computations is not well-handled by traditional (monadic) approaches. Algebraic effects and handlers [18] are a generic approach overcoming this double challenge. They allow modelling large classes of effects (e.g., exception, state, non-determinism) and to compose effects within purely functional programs, while deferring the realizations of these effects to dedicated handlers.

In this paper, we aim to show how a variant of algebraic effects based on Free monads can be used to reason about large systems, by modelling their components as effect handlers. Our contribution is threefold:

- We propose an approach which leverages the key concepts of algebraic effects and handlers to model and to verify each component of a computing system independently, while providing the necessary abstractions to compose these components in order to verify the properties of the resulting system.
- We have implemented FreeSpec [5], a framework for the Coq proof assistant to *modularly* verify programs with effects and effects handlers with our approach.
- We have modelled and verified a simplified MCH with FreeSpec, in a first step towards illustrating how our formalism can be leveraged to tackle our initial objective, that is modelling and verifying computing platforms.

The rest of the paper proceeds as follows. We describe how we model components in terms of programs with effects and effect handlers (Section 2) and we introduce so-called abstract specifications to verify their respective properties (Section 3). We also discuss the current limitations of our approach in terms of components connection; FreeSpec works well with tree of components, when more complex connection patterns require more investigation. To illustrate our definitions, we use a running example in the form of the specification and verification of a simplified MCH. Then, we discuss how the different levels of composition of FreeSpec can be leveraged to scale the modelling and verification work for larger systems (Section 4). Finally, we detail how FreeSpec is in line with an ongoing effort to modularly verify both large systems and programs with effects (Section 5).

2 Modelling Programs with Effects

The first objective of FreeSpec is to incrementally model a complex system, one component at a time. To do so, we use the key concepts of algebraic effects

[5] FreeSpec has been released as a free software, under the terms of the GPLv3. `https://github.com/ANSSI-FR/FreeSpec`

and effect handlers, implemented with a variant of the Free monad called the Program monad as defined in the `operational` package of Haskell [19].

This section and the one afterwards proceed through a running example: a minimalist Memory Controller Hub (MCH) of the x86 hardware architecture. The MCH acts as a dispatcher for the CPU memory accesses; in our case, to the VGA controller or the DRAM controller. The MCH takes part in the isolation of the SMRAM, that is the special-purpose memory region inside the system memory which contains the SMM code. If correctly configured, the MCH will reroute any memory access which targets the SMRAM to the VGA controller, if this access is done by a CPU in another execution mode than SMM.

2.1 Interface of Effects

Within a computing system, interconnected components communicate through interfaces. A component which exhibits an interface receives computational requests from other components; it handles these requests by computing their results and sending the latter back to the client component. In FreeSpec, a computational request is modelled with an effect, that is a symbolic value which represents the request and its potential result.

Thereafter, we often define sets of values, and interfaces in particular, in terms of functions to construct these values. These functions are called "constructors" [6], and they have mutually exclusive images, i.e. it is not possible to construct the same value with two different constructors. For $\mathcal{I}$ an interface, we denote by $\mathcal{I}|_{\mathcal{A}} \subseteq \mathcal{I}$ the subset of effects whose results belong to a set $\mathcal{A}$.

Example 1 (MCH Interfaces) *The VGA and the DRAM controllers exhibit a similar interface which allows reading and writing into a memory region. Their interfaces are denoted by $\mathcal{I}_{\text{VGA}}$ and $\mathcal{I}_{\text{DRAM}}$ respectively. Let $\mathcal{Loc}$ be the set of memory locations and $\mathcal{Val}$ the set of values stored inside the memory region. We use the value $()$ to model effects without results (similarly to the `void` keyword in an imperative language). We define $\mathcal{I}_{\text{DRAM}}$ (respectively $\mathcal{I}_{\text{VGA}}$) with two constructors:*

- $\mathbf{Read}_{\text{DRAM}} : \mathcal{Loc} \to \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}}$
- $\mathbf{Write}_{\text{DRAM}} : \mathcal{Loc} \to \mathcal{Val} \to \mathcal{I}_{\text{DRAM}}|_{\{()\}}$

Then, $\mathcal{I}_{\text{DRAM}} = \mathcal{I}_{\text{DRAM}}|_{\{()\}} \cup \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}}$, and $\mathbf{Read}_{\text{DRAM}}(l) \in \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}}$ is an effect that describes a memory access to read the value $v \in \mathcal{Val}$ stored at the location $l \in \mathcal{Loc}$.

The MCH interface is similar, but it distinguishes between privileged and unprivileged accesses. It also provides one effect to lock the SMRAM protection mechanism, i.e. it enables the SMRAM isolation until the next hardware reset. We define the set $\mathcal{Priv} \triangleq \{\, \texttt{smm}, \texttt{unprivileged} \,\}$ to distinguish between privileged memory accesses made by a CPU in SMM and unprivileged accesses made the

[6] In this article, functions are written in bold. In addition, constructors begin with a capital letter.

rest of the time. The MCH interface, denoted by $\mathcal{I}_{\text{MCH}}$*, is defined with three constructors:*

- $\mathbf{Read}_{\text{MCH}} : \mathcal{Loc} \to \mathcal{Priv} \to \mathcal{I}_{\text{MCH}}|_{\mathcal{Val}}$
- $\mathbf{Write}_{\text{MCH}} : \mathcal{Loc} \to \mathcal{Val} \to \mathcal{Priv} \to \mathcal{I}_{\text{MCH}}|_{\{()\}}$
- $\mathbf{Lock} : \mathcal{I}_{\text{MCH}}|_{\{()\}}$

2.2 Operational Semantics for Effects

An effect corresponds to a computational request made to an implementation of a given interface. For a given computational request, we define its *operational semantics* to compute its result. Ultimately, we will model a component as an operational semantics for all the effects of its interface. Since operational semantics are defined using a purely functional language, they always compute the same result for a given effect, which is inconsistent with the stateful aspect of hardware components. Thus, an operational semantics produces not only a result, but also a new operational semantics, which encapsulates the new state of the component.

Definition 1 (Operational Semantics). *We write* $\Sigma_{\mathcal{I}}$ *for the set of operational semantics for a given interface* $\mathcal{I}$*, defined co-inductively as*

$$\Sigma_{\mathcal{I}} \triangleq \{\, \sigma \mid \sigma : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{A} \times \Sigma_{\mathcal{I}} \,\}.$$

An operational semantics $\sigma \in \Sigma_{\mathcal{I}}$ *is a function which, given any effect of* $\mathcal{I}$*, produces both a result which belongs to the expected set and a new operational semantics to use afterwards.*

A component may use more than one interface. For instance, the MCH of our running example can access the system memory and the memory shared by the VGA controller. But an operational semantics is defined for only one interface. In FreeSpec, we solve this issue by composing interfaces together to create new ones.

Definition 2 (Interfaces Composition). *Let* $\mathcal{I}$ *and* $\mathcal{J}$ *be two interfaces.* $\oplus$ *is the interface composition operator, defined with two constructors:*

- $\mathbf{InL} : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to (\mathcal{I} \oplus \mathcal{J})|_{\mathcal{A}}$
- $\mathbf{InR} : \forall \mathcal{A}, \mathcal{J}|_{\mathcal{A}} \to (\mathcal{I} \oplus \mathcal{J})|_{\mathcal{A}}$

The resulting interface $\mathcal{I} \oplus \mathcal{J}$ contains the effects of both $\mathcal{I}$ and $\mathcal{J}$, wrapped into either **InL** or **InR** constructors, defined to preserve the effects results sets.

Example 2 (VGA and DRAM Composition) *We consider* $\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}}$*. Then,* $\mathbf{InL}(\mathbf{Read}_{\text{DRAM}}(l)) \in (\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}})|_{\mathcal{Val}}$ *is an effect that describes a read access targeting the DRAM controller, whereas* $\mathbf{InR}(\mathbf{Write}_{\text{VGA}}(l, c)) \in (\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}})|_{\{()\}}$ *is an effect that describes a write access targeting the VGA controller.*

Using $\oplus$, we can compose several interfaces together. We then need another composition operator, this time for operational semantics. We compose operational semantics together to construct a new operational semantics for the composed interface.

Definition 3 (Operational Semantics Composition). *Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces, $\sigma_i \in \Sigma_\mathcal{I}$ and $\sigma_j \in \Sigma_\mathcal{J}$ be two operational semantics dedicated to these interfaces. In this article, we use the λ-calculus abstraction notation for functions. $\otimes$ is the composition operator for operational semantics, defined as*

$$\sigma_i \otimes \sigma_j \triangleq \lambda e. \begin{cases} (x, \sigma_i' \otimes \sigma_j) \text{ when } e = \mathbf{InL}(e_i) \text{ and } \sigma_i(e_i) = (x, \sigma_i') \\ (x, \sigma_i \otimes \sigma_j') \text{ when } e = \mathbf{InR}(e_j) \text{ and } \sigma_j(e_j) = (x, \sigma_j') \end{cases}$$

The definition of $\otimes$ has an important impact over what we can specify in FreeSpec. Handling an effect of $\mathcal{I}$ (respectively $\mathcal{J}$) does not interfere with σ_j (respectively σ_i). As a consequence, *we can only specify as-is trees of components*, while graphs with, for instance, cycles or forward edges are still out of scope. This is the main limitation of FreeSpec, but its incidence is abated because computing platforms are often designed as a hierarchical succession of layers.

2.3 The Program Monad

Modelling programs with side effects in purely functional languages such as Gallina (the Coq specification language) or Haskell is usually achieved thanks to monads [20]. FreeSpec leverages a variant of the Free monad called the Program monad [19] to model programs with effects. Operational semantics play the role of `operational` [19] interpreters. We write $P_\mathcal{I}(\mathcal{A})$ for the set of programs with effects which belongs to $\mathcal{I}$, modelled thanks to the Program monad, and whose result belongs to a set $\mathcal{A}$.

Definition 4 (Program Monad). *$P_\mathcal{I}(\mathcal{A})$ is defined with three constructors:*

- **Pure** : $\mathcal{A} \to P_\mathcal{I}(\mathcal{A})$
- **Bind** : $\forall \mathcal{B}, P_\mathcal{I}(\mathcal{B}) \to (\mathcal{B} \to P_\mathcal{I}(\mathcal{A})) \to P_\mathcal{I}(\mathcal{A})$
- **Request** : $\mathcal{I}|_\mathcal{A} \to P_\mathcal{I}(\mathcal{A})$

These constructors allow for the construction of values which act similarly to abstract syntax trees to model programs with effects. On the one hand, **Pure** and **Request** are comparable to the leaves of a syntax tree and model atomic computations; **Pure** models local computations, whereas **Request** models deferring a computational request to a handler and waiting for its result. On the other hand, **Bind** (denoted by the infix operator $\gg\!=$ afterwards) models the control flow of a program with effects, like the abstract syntax tree nodes would. It defines how the result of one computation determines the following ones.

Example 3 (Copy) *We define* **copy** : $\mathcal{Loc} \to \mathcal{Loc} \to P_{\mathcal{I}_{\text{DRAM}}}(\{()\})$ *such that* $\mathbf{copy}(l, l')$ *models a program with effects that returns no result, but copies the value v stored at the memory location l inside the memory location l'.*

$$\mathbf{copy}(l, l') \triangleq \mathbf{Request}(\mathbf{Read}_{\text{DRAM}}(l)) \gg\!= \lambda v. \mathbf{Request}(\mathbf{Write}_{\text{DRAM}}(l', v))$$

Given $l \in \mathcal{Loc}$ and $l' \in \mathcal{Loc}$, $\mathbf{copy}(l, l')$ symbolically models a program with effects. To assign an interpretation of this program, it must be completed with an operational semantics which realizes the interface $\mathcal{I}_{\mathrm{DRAM}}$.

Definition 5 (Program With Effects Realization). *Let $\mathcal{I}$ be an interface, $\sigma \in \Sigma_{\mathcal{I}}$ an operational semantics for this interface and $\rho \in P_{\mathcal{I}}(\mathcal{A})$ a program with effects which belong to this interface. $\sigma[\rho] \in \mathcal{A} \times \Sigma_{\mathcal{I}}$ denotes the realization of this program by σ, defined as:*

$$\sigma[\rho] \triangleq \begin{cases} (x, \sigma) & \text{if } \rho = \mathbf{Pure}(x) \\ \sigma(e) & \text{if } \rho = \mathbf{Request}(e) \\ \sigma'[f(y)] & \text{if } \rho = q \gg\!= f \text{ and } (y, \sigma') = \sigma[q] \end{cases}$$

2.4 Components as Programs with Effects

With the interfaces, their operational semantics, the $\oplus$ and $\otimes$ operators to compose them and the Program monad to model programs with effects which belong to these interfaces, we now have all we need to model a given component which exposes an interface $\mathcal{I}$ and uses another interface $\mathcal{J}$. We proceed with the following steps: modelling the component in terms of programs with effects, then deriving one operational semantics for $\mathcal{I}$ from these programs, assuming provided an operational semantics for $\mathcal{J}$.

The behaviour of a component is often determined by a local, mutable state. When it computes the result of a computational request, not only a component may read its current state; but it can also modify it, for instance to handle the next computational request differently. This means we have to model the state of a component with a set $\mathcal{S}$ of symbolic state representations. We map the current state of the component and effects of $\mathcal{I}$ to a program with effects of $\mathcal{J}$. These programs must compute the effect result and the new state of the component.

Definition 6 (Component). *Let $\mathcal{I}$ be the interface exhibited by a component and $\mathcal{J}$ the interface it uses. Let $\mathcal{S}$ be the set of its states. The component C, defined in terms of programs with effects of $\mathcal{J}$, is of the form*

$$\forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{S} \to P_{\mathcal{J}}(\mathcal{A} \times \mathcal{S})$$

Hence, C specifies how the component handles computational requests, both in terms of computation results and state changes.

Example 4 (Minimal MCH Model) *Let C_{MCH} be the MCH defined in terms of programs with effects of $\mathcal{I}_{\mathrm{DRAM}} \oplus \mathcal{I}_{\mathrm{VGA}}$, then C_{MCH} is of the form*

$$\forall \mathcal{A}, \mathcal{I}_{\mathrm{MCH}}|_{\mathcal{A}} \to \mathcal{S}_{\mathrm{MCH}} \to P_{\mathcal{I}_{\mathrm{DRAM}} \oplus \mathcal{I}_{\mathrm{VGA}}}(\mathcal{A} \times \mathcal{S}_{\mathrm{MCH}})$$

where $\mathcal{S}_{\mathrm{MCH}} \triangleq \{\mathtt{on}, \mathtt{off}\}$ means the SMRAM protection is either activated (`on`) or deactivated (`off`).

One the one hand, the **Lock** *effect will activate the isolation mechanism of the MCH, setting its state to* `on`*. On the other hand, the effects constructed with*

$\mathbf{Read}_{\mathrm{MCH}}$ *and* $\mathbf{Write}_{\mathrm{MCH}}$ *will use the current state of the MCH, the privileged parameter of the effect and the memory location to lookup to determine if it uses the DRAM or the VGA controller. By default, it fetches the memory of the DRAM controller, except if the isolation mechanism is activated, the access is unprivileged and the targeted memory location belongs to the SMRAM. In such a case, it reroutes access to the VGA controller.*

A component C defined in terms of programs with effects cannot be used as-is to compute the result of a given effect. To do that, we need to derive an operational semantics for $\mathcal{I}$ from C.

Definition 7 (Deriving Operational Semantics). *Let C be a component which exhibits an interface $\mathcal{I}$, uses an interface $\mathcal{J}$ and whose states belong to $\mathcal{S}$. Let $s \in \mathcal{S}$ be the current state of the component and $\sigma_j \in \Sigma_{\mathcal{J}}$ be an operational semantics for $\mathcal{J}$. We can derive an operational semantics for $\mathcal{I}$, denoted by $\langle C, s, \sigma_j \rangle$, defined as*

$$\langle C, s, \sigma_j \rangle \triangleq \lambda i.(x, \langle C, s', \sigma'_j \rangle) \text{ where } ((x, s'), \sigma'_j) = \sigma_j[C(i, s)]$$

The resulting operational semantics models a system made of interconnected components, and can then be used to derive another component model into an operational semantics which models a larger system. For instance, we can proceed with the following steps to comprehensively model our running example: (i) defining the operational semantics for the DRAM and VGA controllers; (ii) using these operational semantics to derive an operational semantics from C_{MCH}. The resulting operational semantics can take part in the derivation of a cache defined in terms of programs with effects of $\mathcal{I}_{\mathrm{MCH}}$, to model a larger part of the system pictured in the Figure 1.

3 Modular Verification of Programs with Effects

The first objective of FreeSpec is to provide the required tools to model each component of a system independently, and to compose these components to model the whole system. Its second objective is to verify that the composition of several components satisfies a set of properties. To achieve that, we introduce the so-called abstract specifications, which allows for specifying, for each interface, expected properties for the effect results, independently of any underlying handler. Abstract specifications can be used to emphasize the responsibility of each component of a system regarding the enforcement of a given security policy. Verifying a component is done against abstract specifications of the interfaces it directly uses, even if it relies on a security property enforced by a deeper component in the components graph. In this case, we have to verify that every single component which separate them preserve this property. This procedure can help to prevent or uncover architectural attacks.

In this section, we proceed with our running example by verifying that the MCH correctly isolates the SMRAM. In order to do that, we define an abstract

specification which states that privileged reads targeting the SMRAM returns the value which has previously been stored by a privileged write. It models the SMRAM isolation: unprivileged writes cannot tamper with the content of the SMRAM, as read by a privileged CPU.

3.1 Definition

In FreeSpec, an abstract specification dedicated to an interface $\mathcal{I}$ is twofold. It defines a precondition over the effects that a caller must satisfy; and, in return, it specifies a postcondition over the effects results that an operational semantics must enforce. Since both the precondition and the postcondition may vary in time, we parameterize an abstraction specification with an abstract state and a step function to update this state after each effect realization.

Definition 8 (Abstract Specification). *We write A for an abstract specification dedicated to an interface $\mathcal{I}$, defined as a tuple $\langle \Omega, \mathbf{step}, \mathbf{pre}, \mathbf{post} \rangle$ where*

- *Ω is a set of abstract states*
- *$\mathbf{step} : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{A} \to \Omega \to \Omega$ is a transition function for the abstract state.*
- *$\mathbf{pre} \subseteq \mathcal{I} \times \Omega$ is the precondition over effects, such that $(e, \omega) \in \mathbf{pre}$ if and only if the effect e satisfies the precondition parameterized with the abstract state ω (denoted by $\mathbf{pre}(e, \omega)$).*
- *$\mathbf{post} \subseteq \bigcup_{\mathcal{A}} (\mathcal{I}|_{\mathcal{A}} \times \mathcal{A} \times \Omega)$ is the postcondition over effects results, such that $(e, x, \omega) \in \mathbf{post}$ if and only if the results x computed for the effects e satisfies the postcondition parameterized with the abstract state ω (denoted by $\mathbf{post}(e, x, \omega)$).*

By defining an abstract specification of an interface $\mathcal{I}$, it becomes possible to abstract away the effect handler, i.e. the underlying component. As a consequence, reasoning about a program with effects can be achieved without the need to look at the effect handlers. An abstract specification is dedicated to one verification problem (in our context, one security property), and it is possible to define as many abstraction specifications as required.

We write $\mathbf{run_{step}} : \forall \mathcal{A}, \Sigma_{\mathcal{I}} \to P_{\mathcal{I}}(\mathcal{A}) \to \Omega \to (\mathcal{A} \times \Sigma_{\mathcal{I}} \times \Omega)$ for the function which, in addition to realize a program with effects, updates an abstract state after each effect. Using $\mathbf{run_{step}}$, we can determine both the precondition over effects and the postcondition over effects results while an operational semantics realizes a program with effects.

Example 5 (MCH Abstract Specification) *Let A_{MCH} be the abstract specification such that $A_{\mathrm{MCH}} = \langle \Omega_{\mathrm{MCH}}, \mathbf{step}_{\mathrm{MCH}}, \mathbf{pre}_{\mathrm{MCH}}, \mathbf{post}_{\mathrm{MCH}} \rangle$. A_{MCH} models the following property: "*privileged reads targeting the SMRAM return the value which has been previously stored by a privileged write*":*

- *Let $\mathcal{Smram} \subseteq \mathcal{Loc}$ be the set of memory locations which belong to the SMRAM. We define $\Omega_{\mathrm{MCH}} \triangleq \mathcal{Smram} \to \mathcal{Val}$, such that $\omega \in \Omega_{\mathrm{MCH}}$ models a view of the SMRAM as exposed by the MCH for privileged reads.*

– *We define* $\mathbf{step}_{\mathrm{MCH}}$ *which updates the view of the MCH (modelled as a function) after each privileged write access targeting any SMRAM location* l*, that is*

$$\mathbf{step}_{\mathrm{MCH}}(e,x,\omega) \triangleq \begin{cases} \lambda l'.\ (\textit{if } l = l' \textit{ then } v \textit{ else } \omega(l')) \\ \qquad \textit{if } e = \mathbf{Write}_{\mathrm{MCH}}(l, v, \texttt{smm}) \textit{ and } l \in \mathcal{Smram} \\ \omega \qquad \textit{otherwise} \end{cases}$$

– *There is no precondition to the use of the MCH effects, so*

$$\forall e \in \mathcal{I}, \forall \omega \in \Omega_{\mathrm{MCH}}, \mathbf{pre}_{\mathrm{MCH}}(e,\omega)$$

– *The postcondition enforces that the result* x *of a privileged read targeting the SMRAM (*$\mathbf{Read}(l, \texttt{smm})$*) has to match the value stored in* A_{MCH} *abstract state, i.e. the expected content for this memory location* $\omega(l)$*.*

$$\mathbf{post}_{\mathrm{MCH}}(e,x,\omega) \triangleq \forall l \in \mathcal{Loc}, e = \mathbf{Read}_{\mathrm{MCH}}(l, \texttt{smm}) \wedge l \in \mathcal{Smram} \Rightarrow x = \omega(l)$$

3.2 Compliance and Correctness

The verification of a component C, which exhibits $\mathcal{I}$ and uses $\mathcal{J}$, consists in proving we can derive an operational semantics σ_i for $\mathcal{I}$ from an operational semantics σ_j for $\mathcal{J}$. This semantics σ_i enforces the postcondition of an abstract specification $A_{\mathcal{I}}$ dedicated to $\mathcal{I}$ (compliance). As C is defined in terms of programs with effects of $\mathcal{J}$, the latter needs to make a legitimate usage of $\mathcal{J}$ with respect to an abstract specification $A_{\mathcal{J}}$ dedicated to $\mathcal{J}$ (correctness).

First, σ_i complies with $A_{\mathcal{I}}$ if, (1) given any effect which satisfies $A_{\mathcal{I}}$ precondition, σ_i produces a result which satisfies its postcondition, and if (2) the new operational semantics σ_i' also complies with $A_{\mathcal{I}}$. The precondition and the postcondition are parameterized by an abstract state, so is the compliance property.

Definition 9 (Operational Semantics Compliance). *Let* A *be an abstract specification for an interface* $\mathcal{I}$*, defined as* $\langle \Omega, \mathbf{step}, \mathbf{pre}, \mathbf{post} \rangle$*,* $\omega \in \Omega$*, then* $\sigma \in \Sigma_{\mathcal{I}}$ *complies with* A *in accordance with* ω *(denoted by* $\sigma \models A[\omega]$*) iff.*

$$\forall e \in \mathcal{I}, \mathbf{pre}(e,\omega) \Rightarrow \mathbf{post}(e,x,\omega) \wedge \sigma' \models A[\mathbf{step}(e,x,\omega)] \textit{ where } (x,\sigma') = \sigma(e)$$

Secondly, programs with effects of C make a legitimate usage of an operational semantics $\sigma_j \in \Sigma_{\mathcal{J}}$ which complies with $A_{\mathcal{J}}$ if they only use effects which satisfy $A_{\mathcal{J}}$ precondition. As for the compliance property, correctness is parameterized with an abstract state.

Definition 10 (Program With Effects Correctness). *Let* A *be an abstract specification for an interface* $\mathcal{I}$*, defined as* $\langle \Omega, \mathbf{step}, \mathbf{pre}, \mathbf{post} \rangle$*,* $\omega \in \Omega$*, and* $\rho \in P_{\mathcal{I}}(\mathcal{A})$*, then* ρ *is correct with respect to* A *in accordance with* ω *(denoted by* $A[\omega] \models\!\!\!| \ \rho$*), iff.*

$$A[\omega] \models\!\!\!| \ \rho \triangleq \begin{cases} \textit{True} & \textit{if } \rho = \mathbf{Pure}(x) \\ \mathbf{pre}(e,\omega) & \textit{if } \rho = \mathbf{Request}(e) \\ \forall \sigma \in \Sigma_{\mathcal{I}} \textit{ such that } \sigma \models A[\omega], & \\ \qquad A[\omega] \models\!\!\!| \ q \wedge A[\omega'] \models\!\!\!| \ f(x) & \textit{if } \rho = q \gg\!= f \\ \textit{where } (x, \_, \omega') = \mathbf{run}_{\mathbf{step}_{\mathcal{J}}}(\sigma, q, \omega) & \end{cases}$$

Every local computation (**Pure**) is correct with respect to A in accordance with ω. A computation which uses an effect $e \in \mathcal{I}$ (**Request**) is correct with respect to A in accordance with ω if and only if e satisfies the precondition of A for the abstract state ω. Finally, the chaining of two programs with effects (**Bind**) is correct with A in accordance with ω if the first program is correct with A in accordance with ω, and the second program is correct in accordance with the abstract state reached after the realization of the first program.

Properties, inferred from an abstract specification, of a correct program with effects only hold if it is realized by a compliant operational semantics. Besides, we prove that correct programs preserve operational semantics compliance.

Theorem 1 (Compliance Preservation). *Let A be an abstract specification dedicated to an interface $\mathcal{I}$. σ a compliant operational semantics for $\mathcal{I}$ produces a compliant operational semantics σ' if it realizes a correct program ρ, that is*

$$\sigma \models A[\omega] \land A[\omega] =\!| \ \rho \Rightarrow \sigma' \models A[\omega'] \text{ where } \mathbf{run_{step}}(\sigma, \rho, \omega) = (x, \sigma', \omega')$$

As for interfaces (with $\oplus$) and operational semantics (with $\otimes$), we have also defined an abstract specification composition operator $\odot$. We do not detail its definition in this article, but it has the significant property of allowing for reasoning about the composition of interfaces and composition of operational semantics.

Theorem 2 (Congruent Composition). *Let $\mathcal{I}$ (respectively $\mathcal{J}$) be an interface. Let $A_\mathcal{I}$ (respectively $A_\mathcal{J}$) be an abstract specification and $\sigma_i \in \Sigma_\mathcal{I}$ (respectively $\sigma_j \in \Sigma_\mathcal{J}$) be an operational semantics for this interface.*

$$\sigma_i \models A_\mathcal{I}[\omega_i] \land \sigma_j \models A_\mathcal{J}[\omega_j] \Rightarrow \sigma_i \otimes \sigma_j \models (A_\mathcal{I} \odot A_\mathcal{J})[\omega_i, \omega_j]$$

With the Compliance Preservation, we know that as long as we follow the abstract specification precondition related to the effects we use, compliant operational semantics keep enforcing the postcondition. With the Compliance Preservation and Congruent Composition, we know we can reason locally, that is component by component.

3.3 Proofs Techniques to Show Compliance For Components

We have dived into the mechanisms which allow for composing together compliant operational semantics, but little has been said about how to prove the compliance property to begin with. In a typical FreeSpec use case, operational semantics are not built as-is, but rather derived from a component model (Definition 7). How to prove the resulting operational semantics complies with an abstract specification depends on how the component is connected to the rest of the system. We have already discussed the consequences of the operational semantics composition operator $\otimes$ (Definition 3). Notably, a graph of components which connects two nodes with more than one path cannot be easily modelled and verified in FreeSpec. In its current state, FreeSpec provides some theorems to

verify the properties of a component model in terms of an abstract specification, depending on the composition pattern.

The most composition connection pattern consists of one component which uses many components, and is only used by one other component. Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces and let C, a component with a set of possible states $\mathcal{S}$, which exhibits $\mathcal{I}$ and uses $\mathcal{J}$. Let $A_{\mathcal{I}}$ be an abstract specification dedicated to $\mathcal{I}$. Deriving an operational semantics from C which complies with $A_{\mathcal{I}}$ in accordance with $\omega_i \in \Omega_I$ requires to show the existence of $s \in \mathcal{S}$ and $\sigma_j \in \Sigma_{\mathcal{J}}$ such that

$$\langle C, s, \sigma_j \rangle \models A_{\mathcal{I}}[\omega_i].$$

However, proving this statement would not be very satisfying, as it ties our verification results to one specific operational semantics σ_j, and by extension one specific component. As a consequence, we define an abstract specification $A_{\mathcal{J}}$ to generalize our statement and abstracting away σ_j. We now need to prove it exists $\omega_j \in \Omega_{\mathcal{J}}$ such that given an operational semantics σ_j which complies with $A_{\mathcal{J}}$ in accordance with ω_j, the operational semantics derived from C, s and σ_j complies with $A_{\mathcal{I}}$ in accordance with ω_i, that is

$$\forall \sigma_j \in \Sigma_{\mathcal{J}},\, \sigma_j \models A_{\mathcal{J}}[\omega_j] \Rightarrow \langle C, s, \sigma_j \rangle \models A_{\mathcal{I}}[\omega_i]$$

The combinatorial explosion of cases introduced by ω_i, s and ω_j, modified as the component handles effects, makes inductive reasoning challenging. The FreeSpec framework provides a useful theorem to address these challenges, which leverages a so-called predicate of synchronization. The latter is defined by the user on a case-by-case basis, to act as an invariant for the induction, and a sufficient condition to enforce compliance.

Theorem 3 (Derivation Compliance). *Let* $\mathbf{sync}$*, a relation between abstract states of* $\Omega_{\mathcal{I}}$ *and* $\Omega_{\mathcal{J}}$*, states of* $\mathcal{S}$*, be a predicate of synchronization. Then, it is expected that,* $\forall \omega_i \in \Omega_{\mathcal{I}}$*,* $s \in \mathcal{S}$ *and* $\omega_j \in \Omega_{\mathcal{J}}$ *such that* $\mathbf{sync}(\omega_i, s, \omega_j)$ *holds, then* $\forall \sigma_j \in \Sigma_{\mathcal{J}}$ *such that* $\sigma_j \models A_{\mathcal{J}}[\omega_j]$ *and* $\forall e \in \mathcal{I}$ *such that* $\mathbf{pre}_{\mathcal{I}}(e, \omega_i)$*,*

1. *C preserves the synchronization of states, that is* $\mathbf{sync}(\omega_i', s', \omega_j')$
2. *C is defined in terms of programs with effects which are correct with respect to $A_{\mathcal{J}}$ in accordance with ω_j, that is* $A_{\mathcal{J}}[\omega_j] \Rightarrow C(e, s)$
3. *C computes a result for e which satisfies $A_{\mathcal{I}}$ postcondition, that is* $\mathbf{post}_{\mathcal{I}}(e, x, \omega_i)$

where $((x, s'), \sigma_j', \omega_j') = \mathbf{run}_{\mathbf{step}_{\mathcal{J}}}(\sigma_j, C(e, s), \omega_j)$ *and* $\omega_i' = \mathbf{step}_{\mathcal{I}}(e, x, \omega_i)$*. Should these three properties be verified, then we show that*

$$\mathbf{sync}(\omega_i, s, \omega_j) \wedge \sigma_j \models A_{\mathcal{J}}[\omega_j] \Rightarrow \langle C, s, \sigma_j \rangle \models A_{\mathcal{I}}[\omega_i].$$

Example 6 (MCH Compliance) *We want to prove we can derive an operational semantics from* C_{MCH} *(Example 4) which complies with* A_{MCH} *(Example 5).*

We define $A_{\mathrm{DRAM}} \triangleq \langle \Omega_{\mathrm{DRAM}}, \mathbf{step}_{\mathrm{DRAM}}, \mathbf{pre}_{\mathrm{DRAM}}, \mathbf{post}_{\mathrm{DRAM}} \rangle$ *an abstract specification dedicated to* $\mathcal{I}_{\mathrm{DRAM}}$ *to express the following property: "*a read access to a memory location which belongs to the SMRAM return the value

which have been previously written at this memory location." *In particular,* $\Omega_{\text{DRAM}} = \Omega_{\text{MCH}}$*, i.e. they are two views of the SMRAM, as exposed by the DRAM controller or by the MCH. In this context, the behaviour of VGA is not relevant. Let* $\top$ *be the abstract specification which has no state and such that its precondition and postcondition are always satisfied (meaning every operational semantics always complies with it). Therefore, the abstract specifications dedicated to the interface used by* C_{MCH}*, that is* $\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}}$*, is* $A_{\text{DRAM}} \odot \top$ *whose abstract state is* Ω_{DRAM}*.*

We define the predicate of synchronization $\mathbf{sync}_{\text{MCH}}$ *such that*

$$\mathbf{sync}_{\text{MCH}}(\omega_i, s, \omega_j) \triangleq s = \texttt{on} \wedge \forall l \in \mathcal{Smram}, \omega_i(l) = \omega_j(l)$$

Hence, we start our reasoning from a situation where the SMRAM isolation is already activated and the states of the two abstract specifications are the same, meaning the two views of the SMRAM (as stored in the DRAM, and as exposed by the MCH) coincide. We prove $\mathbf{sync}_{\text{MCH}}$ *satisfies the three premises of the Theorem 3. We conclude we can derive an operational semantics from* C_{MCH} *which complies with* A_{MCH}*.*

Another common composition pattern consists of a component which is used by more than one other component. FreeSpec provides a theorem which allows for extending the result obtained with the Theorem 3, in the specific case where concurrent accesses do not lead to any change of the abstract state.

4 Discussion

For two sections, we have introduced the FreeSpec key definitions and theorems so that we could model a minimal MCH component and verify its properties in the presence of a well-behaving DRAM controller. This example has been driven by a real mechanism commonly found inside x86-based computing platforms. We now discuss how FreeSpec can be leveraged to model and verify larger systems.

4.1 FreeSpec as a Methodology

The typical workflow of FreeSpec can be summarized as follows: specifying the interfaces of a system; modelling the components of the system in terms of programs with effects of these interfaces; identifying the abstract specifications which express the requirements over each interface; verifying each component in terms of compliance with these abstract specifications.

Independent groups of people can use FreeSpec to modularly model and verify a system, as long as they agree on the interfaces and abstract specifications. If, during the verification process, one group finds out a given interface or abstract specification needs to be updated, the required modifications may impact its neighbours. For instance, modelling a x86-based computing system, as pictured in Figure 1, using FreeSpec requires to take into account the CPU cache, and to verify it complies with an abstract specification similar to the one defined in

Example 5. Thus, FreeSpec could have helped uncover the attack mentioned in Section 1 [2,3], and other similar architectural attacks.

The abstract specifications are defined in terms of interfaces, i.e. independently from components. It has two advantages. First, for a given verification problem modelled with a set of abstract specifications, two components which exhibit the same interface can be proven to comply with the same abstract specification. In such a case, we can freely interchange these components, and the verification results remain true. This is useful to consider the challenge posed by components versioning, i.e. a new version of a component brings new features which could be leveraged by an attacker. Then, it is possible to verify a given component in terms of several abstract specifications. This means we can independently conduct several verification works against the same component.

4.2 FreeSpec as a Framework

FreeSpec includes about 8,000 lines of code: 6,000 for its core, 2,000 for the experiments. It has been built upon three objectives: readability of models, automation of proofs, opportunity to extract these models for experimental validation.

To achieve readability, FreeSpec borrows several popular concepts to modern functional programming language, such as Haskell. We have used the `Notation` feature of Coq to add the do-notation of Haskell to Gallina. This allows for writing monadic functions that can be read as if it were pseudo-code. The readers familiar with the monad transformers mechanism [21] may also have recognized the definition of the transformer variant of the State monad in the Definition 6. FreeSpec takes advantage of the State monad mechanism to seamlessly handle the local state of the component.

To achieve automation of proofs, we have developed specific Coq tactics. Some definitions of FreeSpec can be pretty verbose, and the proofs quickly become difficult to manage as the program grows in complexity. FreeSpec provides two tactics to explore the control flow of programs with effects.

Finally, to achieve model extraction, we have defined the key concepts of FreeSpec so that they remain compatible with the extraction mechanism of Coq. As a consequence, component models can be derived into executable programs. For a hardware component, it means we could, for instance, compare its behaviour with its concrete counterpart. For a software component, it means we can fill the gap between the model and the implementation.

5 Related Work

FreeSpec falls within two domains of research: the verification of large systems made of components, and the modular verification of programs with effects.

FreeSpec follows our previous work named SpecCert [1], whose lack of modularity complexified scalability. Kami [13] shares many concepts with FreeSpec, but implements them in a totally different manner: components are defined as labelled transition systems and can be extracted into FPGA bitstreams. Kami

is hardware-specific, thus is not suitable to reason about systems which also include software components. However, it allows for expressing more composition pattern than FreeSpec (e.g. components cycle). Thomas Heyman *et al.* [22] have proposed a component-based modelling technique for Alloy [23], where components are primarily defined as semantics for a set of operations; a component is connected to another when it leverages its operations. Alloy leverages a model finder to verify a composition of these components against known security patterns, and to assume or verify facts about operations semantics; however, it lacks an extraction mechanism, which makes it harder to validate the model.

Algebraic effects and effect handlers led to a lot of research about verification of programs with side effects [18,24], but to our surprise, we did not find any approach to write and verify programs with effects and effect handlers written for Gallina. However, other approaches exist. Ynot [25] is a framework for the Coq proof assistant to write, reason with and extract Gallina programs with side effects. Ynot side effects are specified in terms of Hoare preconditions and postconditions parameterized by the program heap, and does not dissociate the definition of an effect and properties over its realization. To that extent, FreeSpec abstract specification is more expressive (thanks to the abstract state) and flexible (we can define more than one abstract specification for a given interface). Claret *et al.* have proposed Coq.io, a framework to specify and verify interactive programs in terms of use cases [26]. The proofs rely on *scenarios* which determine how an environment would react to the program requests. These scenarios are less generic and expressive than FreeSpec abstract specifications, but they are declarative and match a widely adopted software development process. As a consequence, they may be easier to read and understand for software developers.

Previous approaches from the Haskell community to model programs with effects using Free monads [19,27] are the main source of inspiration for FreeSpec. In comparison, we provide a novel method to verify these programs, inspired by the interface refinement of the B-method [28]. It also had some similiraties with FoCaLiZe [29], a proof environment where proofs are attached to components.

6 Conclusion & Future Work

We have proposed an approach to model and verify each component of a computing system independently, while providing the necessary abstractions to compose these components together in order to verify the properties of the resulting system. We have implemented FreeSpec, an open-source framework for the Coq proof assistant which implements our approach. Finally, we applied our approach to a simplified x86-based computing platform in terms of programs.

We would like to consider more composition patterns. We also anticipate abstract specifications may become harder to understand as their complexity grows. We want to make them more declarative, so they could be more easily understood by software developers who are less familiar with functional programming and formal verification.

References

1. Letan, T., Chifflier, P., Hiet, G., Néron, P., Morin, B.: SpecCert: Specifying and Verifying Hardware-based Security Enforcement. In: 21st International Symposium on Formal Methods (FM 2016), Springer (2016)
2. Wojtczuk, R., Rutkowska, J.: Attacking SMM Memory via Intel CPU Cache Poisoning. Invisible Things Lab (2009)
3. Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM Reloaded. CanSecWest, Vancouver, Canada (2009)
4. Domas, C.: The Memory Sinkhole. In: BlackHat USA. (July 2015)
5. Kallenberg, C., Wojtczuk, R.: Speed Racer: Exploiting an Intel Flash Protection Race Condition. Bromium Labs (January 2015)
6. Kovah, X., Kallenberg, C., Butterworth, J., Cornwell, S.: SENTER Sandman: Using Intel TXT to Attack Bioses. Hack in the Box (2015)
7. Stewin, P., Bystrov, I.: Understanding DMA Malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer (2012) 21–41
8. Manual, I.P.: Intel IA-64 Architecture Software Developer's Manual. Itanium Processor Microarchitecture Reference for Software Optimization (Aug. 2000) (2000)
9. Intel: Intel 5100 Memory Controller Hub Chipset
10. Reid, A.: Who Guards the Guards? Formal Validation of the Arm v8-M Architecture Specification. Proceedings of the ACM on Programming Languages **1**(OOPSLA) (2017) 88
11. Leslie-Hurd, R., Caspi, D., Fernandez, M.: Verifying Linearizability of Intel® Software Guard Extensions. In: International Conference on Computer Aided Verification, Springer (2015) 144–160
12. Chong, S., Guttman, J., Datta, A., Myers, A., Pierce, B., Schaumont, P., Sherwood, T., Zeldovich, N.: Report on the NSF Workshop on Formal Methods for Security. arXiv preprint arXiv:1608.00678 (2016)
13. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., et al.: Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. Proceedings of the ACM on Programming Languages **1**(ICFP) (2017) 24
14. Inria: The Coq Proof Assistant https://coq.inria.fr/.
15. Braibant, T.: Coquet: A Coq Library for Verifying Hardware. CPP **7086** (2011) 330–345
16. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: ACM SIGPLAN Notices. Volume 47., ACM (2012) 395–404
17. Jomaa, N., Nowak, D., Grimaud, G., Hym, S.: Formal Proof of Dynamic Memory Isolation Based on MMU. In: Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on, IEEE (2016) 73–80
18. Bauer, A., Pretnar, M.: Programming with Algebraic Effects and Handlers. Journal of Logical and Algebraic Methods in Programming **84**(1) (2015) 108–123
19. Apfelmus, H.: The `operational` package `https://hackage.haskell.org/package/operational`.
20. Hoareetal, C.: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Engineering theories of software construction (2001)
21. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1995) 333–343

22. Heyman, T., Scandariato, R., Joosen, W.: Reusable Formal Models for Secure Software Architectures. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland, August 20-24, 2012. (2012) 41–50
23. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT press (2012)
24. Brady, E.: Resource-dependent algebraic effects. In: International Symposium on Trends in Functional Programming, Springer (2014) 18–33
25. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Dependent Types for Imperative Programs. In: ACM Sigplan Notices. Volume 43., ACM (2008) 229–240
26. Claret, G., Régis-Gianas, Y.: Mechanical Verification of Interactive Programs Specified by Use Cases. In: Proceedings of the Third FME Workshop on Formal Methods in Software Engineering, IEEE Press (2015) 61–67
27. Kiselyov, O., Ishii, H.: Freer Monads, More Extensible Effects. In: ACM SIGPLAN Notices. Volume 50., ACM (2015) 94–105
28. Abrial, J.R., Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
29. Pessaux, F.: FoCaLiZe: inside an F-IDE. arXiv preprint arXiv:1404.6607 (2014)