Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 41–56
www.elsevier.com/locate/entcs

Convincing Proofs for Program Certification

Manuel Garnacho,[1] Michaël Périn[2]

Université de Grenoble, INPG, CNRS

Abstract

At the highest level of formal certification, the current research trend consists in providing evaluators with a formal checkable proof produced by automatic verification tools. The aim is to reduce the certification process to verifying the provided proof using a proof-checker. However, to date, no certified proof-checker has emerged. In addition, checkable proofs do not eliminate the need to validate the formalization of the verification problem. In this paper we consider the point of view of evaluators. We elaborate criteria that must be fulfilled by a formal proof in order to convince skeptical evaluators. Then, we present a methodology based on this notion of *convincing proofs* that requires simple formalizations to reach the level of confidence of formal certification. The key idea is to build a certified proof-checker – in collaboration with the evaluators – which is finally used to validate the proof provided by developers. We illustrate our approach on the correctness proof of a buffering protocol written in C that manages the data exchanges between concurrent tasks in avionics control systems.

Keywords: certification, formal proofs, certified proof-checker, reduction of the trusted computing base.

At the highest level of certification many norms in avionics (DO-178B, IEC61508) or security (Common Criteria) highly recommend the use of formal methods [10,16]. Additionally, in some fields, computer aided verification is mature enough to rewards the developers with a rich feedback and an increase in confidence. Formal methods would then become worthwhile if the verdict of the verification tools could be reused in the certification process. However, this is not the case since evaluators cannot assume that the implementation of a verification tool is correct. Even approaches in which verification tools are instrumented to produce formal proofs of their verdict [2,11,13,14,15,17] can be rejected. Indeed, even though it is commonly thought that a formal proof is the highest reachable level of confidence one can expect, evaluators put strong conditions before accepting a formal proof as certificate. For instance, *an overlooked issue that arises in formal verification of processors is the model validation, which is crucial to ensure that formal analysis applies to the actual machine (M. Wildings [21]).*

[1] Email: manuel.garnacho@imag.fr
[2] Email: michael.perin@imag.fr

doi:10.1016/j.entcs.2009.09.005

We illustrate our understanding of the evaluators' requirements[3] for the use of formal proofs in certification on a realistic case study: the proof of correctness of a buffering protocol written in C that manages the data exchanges between concurrent tasks in avionics control systems [19]. In order to convince evaluators that a program P carries out a property Φ, by the means of a proof: **(*i*)** the proof must be checkable by a machine using a proof-checker that the evaluators have already certified; **(*ii*)** the program P addressed in the proof must be the actual program to certified, not a simplified or abstract representation of P; **(*iii*)** the property Φ addressed in the proof must be in a setting on which the evaluators agreed; **(*iv*)** the semantics of the language in which P is written and the semantics of the logic that is used to state the property Φ must be explicit to sustain the validation process; **(*v*)** all of these definitions must be obvious and, if possible, be kept close to the standard background of computer scientists in mathematics; **(*vi*)** finally, it should be obvious that the proof exactly addresses the verification problem and not another similar problem. We introduce the terminology *"convincing proofs"* to denote checkable proofs that fulfill these six requirements. As a starting point we review related work on checkable proofs with respect to these criteria.

Related work In theory, the verification methods that return a witness could be enhanced to produce convincing proofs [17]. Let us stress the points that require some extra work to meet the criteria of convincing proofs. All algorithmic methods that determine if a program fulfills a property apply many transformations to the initial verification problem. Among them, the most common are abstraction (*e.g.*, from C to boolean programs [11]), reduction to so-called small models [2], and reformulation in another framework (*e.g.*, from μ-calculus to games theory [13], from imperative to functional programs [7]). Several tools use external theorem provers and decision procedures that must then be certified or instrumented [4,14]. Criteria (*ii*) and (*iii*) require that all of these transformations and external computations appear with justification in the final proof. Most verification tools do not fulfill criterion (*iv*) since the semantics of their framework (the programming language and its associated logic) are hard-coded into the tools and not easily available for validation.

Closer to our work is the use of deductive methods based on the calculus of weakest preconditions. The tools CADUCEUS/WHY [8] and CAVEAT [16] are used in verification of critical applications written in C. They offer a good tradeoff between automation and confindence. However they do not produce a proof of correctness of the original program that could be checked by an independent certified prover: CAVEAT uses internal decision procedures and simplification rules and CADUCEUS/WHY transforms the original program before generating verification conditions which entails the correctness. The generation of checkable proofs is addressed in the Proof-Carrying Code architecture (PCC [14]) and its foundational extension (FPCC [22]): the formal proof of an untrusted program is checked before

[3] The following list of evaluator requirements was elaborated in the industrial project EDEN that aims at certifying security applications at the highest level of the Common Criteria (see `http://www.eden-rntl.org`) and through participation to several working groups with the French certification authority for security (DCSSI), and with the formal methods teams of EADS and RATP (the French metro company).

its execution, thus preventing users from the possible damage due to downloaded applications. These frameworks do not fulfill all of the evaluators' requirements: PCC still depends on a generator of verification conditions (VCG) to connect the proof, the program and the property. Moreover the definition of the program and property semantics is part of 23 000 lines of C that implement the VCG. That semantics is explicit in FPCC but it must be given in the minimal logic used in the proof (based on lambda calculus). Experience has shown that this results in numerous definitions and very complex semantics models [9]. Finally, it is worth noticing that, to date, no proof-checker has been certified. Hence, criterion (*i*) remains unsatisfied, despite an attempt to minimize and prove the COQ proof-checker in the COQ proof assistant [3].

Approach In this paper, we focus on the evaluator requirements. We present a framework for certification that is illustrated on a buffering protocol written in C that manages the data exchanges between concurrent tasks in avionics control systems. The resulting proof of correctness fulfills the criteria of convincing proofs. More precisely, our framework conforms to the following guideline: **(*i*)** the certification process leads to a trusted proof-checker which is built in collaboration with the evaluators; **(*ii*)** the proof is done on the abstract syntax tree of the program. The original program can be filtered out from the proof-term using a simple and easy to validate pretty printer (not presented in this paper); **(*iii*)** evaluators and developers must agree on a logical framework in which the correctness property can be stated; **(*iv*)** they must validate all the logical and semantic rules of the proof-system; **(*v*)** these derivation rules must then be carefully chosen: they must be obvious and as few in number as possible to ease the validation process. In this paper, we use extended transition systems, symbolic history of variables and symbolic equalities. The program is given an operational semantics as derivation rules which define the effect of an instruction in terms of basic predicates of the logic. Contrarily to general purpose verification tools, only the instructions used in the program to be certified must be addressed. **(*vi*)** the program and the property appear explicitly in the correctness statement. The proof-term consists in the derivation tree of that statement which then appears at the root.

Our work has been inspired by studies on FPCC [1,9] but it differs on two important points that were dictated by criteria (*i*) and (*vi*) of convincing proofs: 1) we avoid the definition of an operational semantics for all constructions and operators of C; 2) optimized proof-checkers for compact proofs are very complex and will be difficult to certify. For this reason, this paper focuses on transparency of the proof-checker. The compactness of proofs is left for future work. As far as we know it is the first attempt to produce a checkable proof in foundational logic of a non trivial program (a protocol for event driven system) reasoning directly on non-transformed C code and reducing the TCB close to minimum: a compiler and a machine.

Outline of the paper The rest of the paper is devoted to the design of a framework for convincing proofs. It is based on foundational proofs and depends on the construction of a certified proof-checker in collaboration with evaluators.

Since the certification process requires that evaluators validate the semantics rules, we show in Section 1 how to take advantage of this validation phase to produce a certified proof-checker by a straightforward translation of the semantics rules. In Section 2, we briefly present the protocol and the correctness statements that will serve as a basis to illustrate our framework. The formalization of the verification problem is explained in Section 3. The derivation rules that capture the semantics of C programs, of concurrent tasks, of priority and preemption are given in Section 4. Their translation into PROLOG is straightforward and defines the recursive proof-checker function. Improvements and future work are discussed in Section 5.

1 Construction of a certified proof-checker

Our reduction of the trusted computing base (TCB) is a consequence of the fact that our methodology produces a *certified proof-checker* that is finally used to validate the proof. The idea is the following: we explain each derivation rule of the proof system to the evaluators who must accept and validate each rule – to succeed, the rules must be simple and obvious. The proof-checker is then a straightforward translation of these rules into a programming language – to be trusted, the translation must be simple and obvious. The proof-checker can then be removed from the TCB which is then reduced to a compiler and a machine. Finally, the proof of correctness is validated using the certified proof-checker which checks that it is a combination of valid derivation steps.

1.1 Foundational first-order logic on uninterpreted predicates

We consider the first-order logic over uninterpreted predicates represented by syntactical terms. The formulæ are built using the standard connectors $(\wedge, \vee, \Rightarrow, \neg)$ and the quantifiers $(\forall, \exists)$. In our framework, all predicates and connectives are uninterpreted in the sense that they are not associated to *models* (the mathematical structures on which the predicates can be interpreted to determine their validity). The intended meaning of each term of the logic is captured in the set of derivation rules that define the proof system. We recall that a proof system does not give a semantic meaning to a formula ; it only defines the provability of statements as derivability in the following sense: *a statement is provable if and only if it can be obtained by the application of the derivation rules* [20]. The distinction between validity with respect to a model and provability is not significant to evaluators: they must either validate the model of a logic or its proof system. Our claim is that it is easier to validate derivation rules.

The logical proofs are drawn up in the proof-system of the natural deduction for first-order logic. It is known to be sound and complete [20]. Thanks to its strong connection with logical deduction steps used in mathematical proofs, it is easy to obtain its validation by evaluators. The derivation rules are given in Figure 1 in addition to the principle of induction on natural numbers and Leibniz's elimination of equality. We adopt a presentation of statements as sequents of the form $\mathcal{H} \vdash \Phi$ which means that Φ can be derived under the hypothesis $\mathcal{H}$. Compared

$$\frac{\mathcal{H} \vdash \Phi_2}{\mathcal{H} \vdash \Phi_1 \vee \Phi_2}\ \vee_{intro1} \qquad \frac{\mathcal{H} \vdash \Phi_1}{\mathcal{H} \vdash \Phi_1 \vee \Phi_2}\ \vee_{intro2} \qquad \frac{\mathcal{H}, \Phi_1 \vdash \Psi \quad \mathcal{H}, \Phi_2 \vdash \Psi}{\mathcal{H}, \Phi_1 \vee \Phi_2 \vdash \Psi}\ \vee_{elim}$$

$$\frac{\mathcal{H} \vdash \Phi_1 \quad \mathcal{H} \vdash \Phi_2}{\mathcal{H} \vdash \Phi_1 \wedge \Phi_2}\ \wedge_{intro} \qquad \frac{\mathcal{H} \vdash \Phi_1 \wedge \Phi_2}{\mathcal{H} \vdash \Phi_2}\ \wedge_{elim1} \qquad \frac{\mathcal{H} \vdash \Phi_1 \wedge \Phi_2}{\mathcal{H} \vdash \Phi_1}\ \wedge_{elim2}$$

$$(\Phi \in \mathcal{H})\ \frac{}{\mathcal{H} \vdash \Phi}\ hyp \qquad \frac{\mathcal{H}, \Phi \vdash \Psi}{\mathcal{H} \vdash \Phi \Rightarrow \Psi}\ \Rightarrow_{intro} \qquad \frac{\mathcal{H} \vdash \Phi \quad \mathcal{H} \vdash \Phi \Rightarrow \Psi}{\mathcal{H} \vdash \Psi}\ \Rightarrow_{elim}$$

$$\frac{\mathcal{H}, \Phi \vdash \bot}{\mathcal{H} \vdash \neg\Phi}\ \neg_{intro} \qquad \frac{\mathcal{H} \vdash \neg\neg\Phi}{\mathcal{H} \vdash \Phi}\ \neg_{elim} \qquad \frac{\mathcal{H} \vdash \Phi[x \leftarrow \tilde{x}]}{\mathcal{H} \vdash \forall x,\ \Phi}\ \forall_{intro} \qquad \frac{\mathcal{H} \vdash \forall x,\ \Phi}{\mathcal{H} \vdash \Phi[x \leftarrow t]}\ \forall_{elim}(t)$$

$$\frac{\mathcal{H} \vdash \Phi \wedge \neg\Phi}{\mathcal{H} \vdash \bot}\ \bot \qquad \frac{\mathcal{H} \vdash \Phi[x \leftarrow t]}{\mathcal{H} \vdash \exists x,\ \Phi}\ \exists_{intro} \qquad (\tilde{x} \notin Var(\Psi))\ \frac{\mathcal{H} \vdash \exists x,\ \Phi \quad \mathcal{H}, \Phi[x \leftarrow \tilde{x}] \vdash \Psi}{\mathcal{H} \vdash \Psi}\ \exists_{elim}$$

$$\frac{\mathcal{H} \vdash \Phi \quad \mathcal{H} \vdash e = e'}{\mathcal{H} \vdash \Phi[e \leftarrow e']}\ =_{elim} \qquad \frac{x = y}{y = x}\ =_{sym} \qquad \frac{}{x = x}\ =_{refl} \qquad \frac{}{e = \mathsf{simpl}(e)}\ simpl$$

$$\frac{\mathcal{H} \vdash \Phi(0) \quad \mathcal{H} \vdash \forall k{:}\mathrm{NAT},\ \Phi(k) \Rightarrow \Phi(k+1)}{\mathcal{H} \vdash \forall n{:}\mathrm{NAT},\ \Phi(n)}\ ind_{\mathbb{N}} \qquad \mathcal{Q} \in \{\forall, \exists\}\ \frac{\mathcal{H} \vdash \mathcal{Q}\, x{:}\mathrm{TYP},\ \Phi}{\mathcal{H} \vdash \mathcal{Q}\, x,\ \mathrm{TYP}(x) \Rightarrow \Phi}\ type(\uparrow\downarrow)$$

Φ, Ψ denotes formulæ, $\tilde{x}$ is a fresh symbol, t denotes a term that can be chosen, $Var(\Phi)$ is the set of free variables of Φ and $\Phi[x \leftarrow y]$ denotes the formula Φ where all free instances of x are replaced by y. Rule $(=_{elim})$ is also known as Leibniz's replacement of an expression e by an equivalent one e'. The transitivity of equality is a consequence of $(=_{elim})$. The rule (*simpl*) uses the simplification function of Section 3.2. Rule $(ind_{\mathbb{N}})$ is the induction principle on natural numbers. Rule (*type*) translates type constraints as a predicate. The other rules are the standard ones of the natural deduction.

Fig. 1. Rules of the natural deduction and induction for first-order logic in a sequent presentation

to other proof systems, sequents simplify the management of hypotheses and this reveals useful to obtain a simple proof-checker by a straightforward translation of the derivation rules in Section 1.3.

1.2 The representation of proof-terms

A *proof* of a statement $\mathcal{H} \vdash \Phi$ is a *finite derivation tree* whose conclusion – the root of the tree[4] – is the statement to prove, and which is only formed of *valid applications of the derivation rules of the proof system*, meaning that each particular application of a rule is obtained by instantiating its variables while respecting the side condition (if there is one). In our framework, derivation trees are represented as terms (denoted by $\nabla, \nabla_1, \ldots$) which conform to the following BNF syntax where Φ is a formula and $\mathcal{H}$ is a list of formulæ.

$$\nabla ::= \textsc{apply}(\textit{rule name}, [\nabla_1, \ldots, \nabla_n],\ \mathcal{H} \vdash \Phi)$$

Since a proof is a finite tree, its leaves can only be applications of derivation rules with no premises, *e.g.*, $\textsc{apply}(hyp, [\,], \mathcal{H} \vdash \Phi)$, the invocation of the (*hyp*) rule of Figure 1 that can exploit a hypothesis Φ if the side-condition $\Phi \in \mathcal{H}$ holds.

[4] In the literature on proof systems a proof-term is usually represented as a tree with hypotheses on leaves and the statement to prove at its root.

1.3 Construction of the proof-checker

Consider the derivation rule $(\wedge_{intro})$ of Figure 1. A derivation tree $\nabla = \text{APPLY}(\wedge_{intro}, [\nabla_1, \nabla_2], \mathcal{H} \vdash \Phi_1 \wedge \Phi_2)$ is a proof of the statement $\mathcal{H} \vdash \Phi_1 \wedge \Phi_2$ if the three following conditions are satisfied: the last step of ∇ is a valid instantiation of $(\wedge_{intro})$, ∇_1 is a proof of $\mathcal{H} \vdash \Phi_1$ and ∇_2 is a proof of $\mathcal{H} \vdash \Phi_2$. These conditions can easily be written as a PROLOG clause where the statement $\mathcal{H} \vdash \Phi$ and the formula $\Phi_1 \wedge \Phi_2$ are denoted by the terms `sequent(`$\mathcal{H}$`,`Φ`)` and `and(`Φ_1`,`Φ_2`)`. We remind the reader that, in PROLOG, terms are not interpreted, meaning that they are not functions but syntactic representations which can be inspected using pattern-matching.

The $(\wedge_{intro})$ derivation rule:

$$\frac{\mathcal{H} \vdash \Phi_1 \quad \mathcal{H} \vdash \Phi_2}{\mathcal{H} \vdash \Phi_1 \wedge \Phi_2} \wedge_{intro}$$

and the corresponding PROLOG *clause:*

```
check(∇,Seqt):- Seqt = sequent(ℋ, and(Φ1,Φ2)),
   ∇ = apply(and_intro, [∇1,∇2], Seqt),
   check(∇1, sequent(ℋ,Φ1)),
   check(∇2, sequent(ℋ,Φ2)).
```

The predicate `check(`∇`,Seqt)` holds if ∇ is a valid proof of the statement `Seqt` where `Seqt` $= \mathcal{H} \vdash \Phi_1 \wedge \Phi_2$. The proof-checking consists in verifying that: ∇ is an application of the rule $(\wedge_{intro})$ to two sub-proofs ∇_1 and ∇_2 with `Seqt` as conclusion, and the sub-proofs (∇_1, resp. ∇_2) are valid proofs of the premises ($\mathcal{H} \vdash \Phi_1$, resp. $\mathcal{H} \vdash \Phi_2$) of the rule $(\wedge_{intro})$. This explains the recursive calls to `check(`∇_1`, sequent(`$\mathcal{H}, \Phi_1$`))` and `check(`∇_2`, sequent(`$\mathcal{H}, \Phi_2$`))`.

The proof-checker is obtained in a similar way by applying the straightforward translation of Figure 2 to each derivation rule. The result is a set of PROLOG clauses that define a recursive predicate `check(`∇`,`$\mathcal{H} \vdash \Phi$`)` which holds if and only if ∇ is a valid derivation of the statement $\mathcal{H} \vdash \Phi$ with respect to the rules of the proof system. This predicate uses only a subset of PROLOG for derivation rules (recursion, a limited form of unification and no backtracking). This choice is defended in [22]. Actually, any programming language with recursion and pattern-matching can be used but the formulation in PROLOG had our preference since it is close to mathematical definitions.

$$\nabla = \text{APPLY}\left(rule \left((condition) \frac{[\nabla_1 \quad \ldots \quad \nabla_n]}{\dfrac{\mathcal{H}_1 \vdash \Phi_1 \quad \ldots \quad \mathcal{H}_n \vdash \Phi_n}{\mathcal{H} \vdash \Phi}} \right) \mathcal{H} \vdash \Phi \right)$$

```
check(∇,Seqt):- Seqt= sequent(ℋ, Φ),
   ∇ = apply(rule, [∇1,...,∇n], Seqt),
   condition,
   check(∇1, sequent(ℋ1,Φ1)),
   ..., check(∇n, sequent(ℋn,Φn)).
```

Fig. 2. Translation of derivation rules into PROLOG clauses that define the proof-checker recursive function

The rest of the paper describes our use of this framework to produce a *convincing proof* of correctness for a complex communication protocol written in C. First, the logical proof-system is enriched with derivation rules that define the semantics of the protocol. Second, the validation of these rules by the evaluators and their translation in PROLOG furnish a certified proof-checker. Finally, evaluators deliver

the certificate if and only if the proof-term provided by the developpers is accepted by the proof-checker.

2 Case study: correctness of a communication protocol

In this section we briefly present the protocol that will serve to illustrate our notion of convincing proofs. The protocol is used to implement multi-tasking real-time data-flow applications on an event-based operating system featuring priority and pre-emption [19]. It has been designed for an avionics control system that consists in a pool of tasks running on a single processor. Each task has an id i and is triggered by the environment on arrival of an event $\mathbb{t}_i$. It then reads available inputs from other tasks, does some computation and outputs its results to all others tasks.

The development of the control system is conducted under the assumption that computations take no time, that is, the output of a task is available instantaneously after its triggering. This so-called synchronous or 0-delay assumption drastically simplifies the development since engineers need only focus on the data-flow between tasks [5]. Therefore, the actual implementation must ensure that a task r (a reader) which uses the output of a task w (a writer) gets the correct output with respect to the data-flow ordering independently of the computation time of each task (see Section 2.2 and [19] for details).

2.1 The protocol

The protocol of Figure 3 garantees this property despite *any number of temporary suspension* (preemption) of any task by other tasks of higher priority and whatever the priorities of the tasks. It consists in two C procedures that manage a series of two-place buffers. A pair of buffers (`Bh2l[`w`][`r`]`,`Bl2h[`w`][`r`]`) is required for each pair of read and write tasks (say Task w, Task r). Actually, only one of these buffers is significant depending on the priority order of the reader and the writer. For instance, data exchanges between a high-priority writer w and a low-priority reader r are done by the means of the buffer `Bh2l[`w`][`r`]`.

An execution of a Task i consists in a sequence of two calls: a call to `os(`i`)` that updates the reading ($inst.\, I_1$) and writing ($inst.\, I_2$) locations of a Task i, followed by a call to `task(`i`)` that collects the inputs from the buffers ($inst.\, I_3$), calls the computation function of Task i ($inst.\, I_4$) and writes the results back in the buffers ($inst.\, I_5$). Since the rest of the paper does not require a deeper understanding of the behavior of the protocol, we refer the interested reader to [19] for a more detailed description. We now turn to the formalization of the correctness problem.

2.2 The correctness property

In order to state the correctness property, following [19], we define three events that govern the run of a Task: $\mathbb{t}_i$ denotes a triggering of Task i that begins with the execution of `os(`i`)`. Triggering events come from the environment through sensors. The others events $\mathbb{s}_i$ and $\mathbb{f}_i$ are produced by the operating system. $\mathbb{s}_i$ indicates

```
int prio[T]; // prio is a static array that associates a priority level to a task id

bool lwl[T][T]; // lwl[w][r] = location where a low writer w must write for a high reader r
bool lwh[T][T]; // lwh[w][r] = location where a high writer w must write for a low reader r
bool lrl[T][T]; // lrl[r][w] = location where a low reader r must read from a high writer w
bool lrh[T][T]; // lrh[r][w] = location where a high reader r must read from a low writer w

data Bl2h[T][T][2]; // Bl2h[w][r] = two-place buffer for low (w) to high (r) data flow
data Bh2l[T][T][2]; // Bh2l[w][r] = two-place buffer for high (r) to low (w) data flow

data inp[T][T]; // inp[r][w] = input of task r from task w
data out[T]; // out[i] = output of task i
```

I_1: *updates reading locations of Task* i

I_2: *updates writing locations of Task* i

```
void os(taskid i){
  taskid w,r ;
  for(w = 0; w < T; w++){
   lrh[i][w] = !lwl[w][i];
   lrl[i][w] =  lwh[w][i];
  }
  for(r = 0; r < T; r++){
   lwl[i][r] = !lwl[i][r];
   if (lrl[r][i] == lwh[i][r])
     lwh[i][r] = !lwh[i][r];
  }
}
```

I_3: *reads inputs of Task* i

I_4:

I_5: *writes outputs of Task* i

```
void task(taskid i){
  taskid w,r ;
  for(w = 0; w < T; w++){
   if(prio[i] > prio[w])
     inp[i][w] = Bl2h[w][i][ lrh[i][w] ];
   else
     inp[i][w] = Bh2l[w][i][ lrl[i][w] ];
  }
  out[i] = computation(i,inp[i]);
  for(r = 0; r < T; r++){
   Bl2h[i][r][ lwl[i][r] ] = out[i];
   Bh2l[i][r][ lwh[i][r] ] = out[i];
  } }
```

Fig. 3. The global variables and C procedures of the protocol for a given number T of tasks

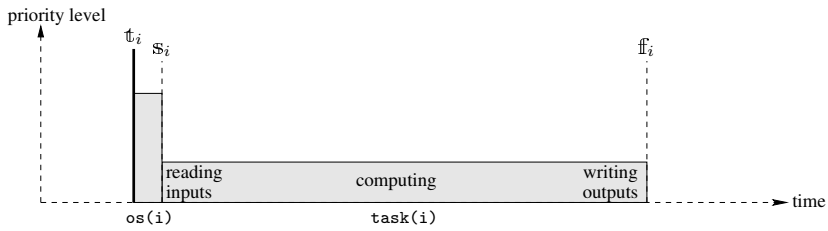

Fig. 4. Representation of the run of a Task with id i: it executes os(i) then task(i)

the starting point of `task(`i`)` which ends by sending the $\mathbb{f}_i$ event (see Figure 4). Each event is annotated by its occurence number such that the sequence of events $\mathbb{t}_i^k .. \mathbb{s}_i^k .. \mathbb{f}_i^k$ refers to the k^{th} run of Task i. The correctness of the protocol for high-to-low data-flows can now be stated precisely:

> Consider a task w (the writer) and a task r (the reader) such that `prio[`w`]` $\geq$ `prio[`r`]`. For each possible sequence of events σ such that the k^{th} triggering of Task w (event $\mathbb{t}_w^k$) is the latest run of w before the p^{th} triggering of Task r (event $\mathbb{t}_r^p$) the p^{th} input of task r from w must be the k^{th} output of task w.

The formal statement of the high-to-low correctness property is given in Figure 6. The correctness property for low-to-high data-flows is a bit more subtle but very similar. The meaningful part of Figure 6 is $\mathbb{t}_w^k .. \mathbb{t}_r^p .. \mathbb{t}_w^{k+1} \subseteq \sigma \Rightarrow (\texttt{inp[}r\texttt{][}w\texttt{]})^p = (\texttt{out[}w\texttt{]})^k$ where $(x)^k$ is the k^{th} value of x, and $e..e'$ denotes a *loose sequence* where the dots represent an unspecified sequence of events, and $\sigma' \subseteq \sigma$ means that the

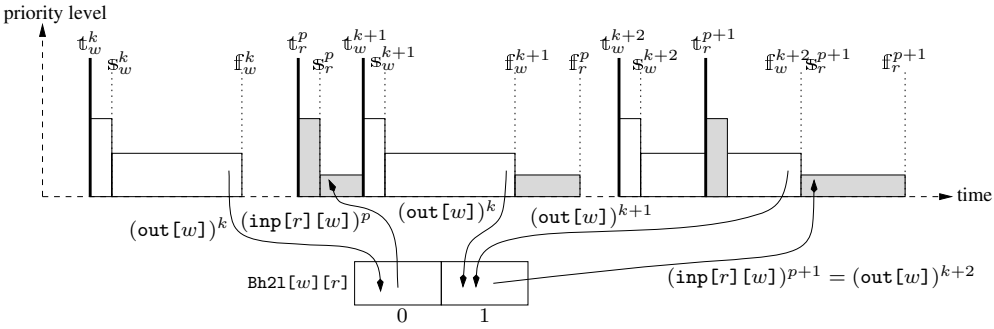

Fig. 5. Illustration of the protocol behavior for high-to-low data-flows

$\forall T$:NAT, $\forall k$:NAT, $\forall p$:NAT, $\forall r$:NAT, $\forall w$:NAT, $\forall \sigma$:SEQ, ... *the variables are typed*
$w \leq T \ \wedge \ r \leq T$ ..... *T is the size of the pool of tasks*
$\wedge$ DECL(`os(int i){...T...}`) $\wedge$ DECL(`task(int i){...T...}`) ..... *declarations of the protocol procedures*
$\wedge$ PCSE(`os(i)`, $\mathbb{t}_i$) $\wedge$ PCSE(`task(i)`, $\mathbb{s}_i\,;\mathbb{f}_i$) ..... *associates procedure calls and seq. of events*
$\wedge$ WF(σ) ..... *σ is a well-formed sequence of events*
$\wedge$ `prio[w]` $\geq$ `prio[r]` ..... *we consider the case of a high-to-low data-flow*
$\wedge$ $\mathbb{f}_r^p \subseteq \sigma$ ..... *the p^{th} execution of task r completed its execution*
$\wedge$ $\mathbb{t}_w^k .. \mathbb{t}_r^p .. \mathbb{t}_w^{k+1} \subseteq \sigma$ ..... *$\mathbb{t}_w^k$ is the latest triggering of Task w before the p^{th} triggering of Task r*
$\Rightarrow$ (`inp[r][w]`)^p = (`out[w]`)^k ..... *the data-flow respects the triggering order of the Tasks r and w*

Fig. 6. The formal correctness property for high-to-low data-flow

sub-sequence σ' can be extracted from σ, that is, all events of σ' appear in σ and their ordering in σ' is preserved in σ. The loose sequence $\mathbb{t}_w^k .. \mathbb{t}_r^p .. \mathbb{t}_w^{k+1}$ captures the scenario addressed in the correctness property, then the required data-flow can easily be specified as an equality using history of variables. The predicate PCSE associates a procedure call to a sequence of events: PCSE(`os(`i`)`, $\mathbb{t}_i$) captures the fact that `os(`i`)` is atomic (its starting and ending events are collapsed into a single event $\mathbb{t}_i$). On the contrary, PCSE(`task(`i`)`, $\mathbb{s}_i\,;\mathbb{f}_i$) points out that `task(`$i$`)` is preemptable.

Finally, notice that the implementation of Figure 3 is parameterized by the number T of tasks to manage. This parameter is universally quantified in the correctness property meaning that the proof covers all possible instantiations of T.

Figure 5 shows the data-flows obtained between a high-priority Task w and a low-priority Task r for a scenario illustrating two cases of pre-emptions. Notice that the high-to-low buffer `Bh2l[`w`][`r`]` achieves the data-flows required to satisfy the correctness property for both inputs of Task r.

3 Formalization using extended transition systems

In order to prove the correctness of the protocol, we must provide formal definitions of: the semantics of C programs, the priority and preemption mechanisms, the predicates that are used to express the property. Moreover, the protocol makes several assumptions concerning the sequences of events emitted by the system under control ; they must appear in our formalization.

3.1 Event-based extended transitions systems

All the states along a run of the system are named by a symbol such as $\mathcal{V}, \mathcal{V}_1, \ldots$. Intuitively, they refer to memory states. To represent the behavior of the control system as a transition system, we introduce three predicates that relate source and target states to: transitions on the execution of a sequential program ; transitions on execution of an interleaving of programs ; and transitions on a event. The predicate $\langle \mathcal{V}, P, \mathcal{V}' \rangle$ holds if all terminating runs of the program P, starting in the initial state $\mathcal{V}$, results in a state $\mathcal{V}'$. Similarly, $\langle \mathcal{V}, P_1 | P_2, \mathcal{V}' \rangle$ holds if all interleaved executions of programs P_1, P_2 make the system evolve from state $\mathcal{V}$ to $\mathcal{V}'$. Finally, the predicate $\mathcal{V} \xrightarrow{e} \mathcal{V}'$ holds if e is the only event that occurs between the states $\mathcal{V}$ and $\mathcal{V}'$. Therefore, the behavior of the control system *wrt.* a sequence of events $\sigma = e_1 ; e_2 ; \ldots ; e_n$ can be represented as a conjunction of predicates that link events and states $\mathcal{V}_0 \xrightarrow{e_1} \mathcal{V}_1 \wedge \mathcal{V}_1 \xrightarrow{e_2} \mathcal{V}_2 \wedge \ldots \wedge \mathcal{V}_{n-1} \xrightarrow{e_n} \mathcal{V}_n$.

3.2 Semantics of assignments as a conjunction of symbolic equalities

The proof requires us to reason on sequences of assignments independently of the actual values of the variables. Therefore, we extend the domain of values with symbols made of a variable annotated with the number of times it has been updated. Thus, x^0 denotes the value of variable `x` at the beginning of the execution and x^n its value after n assignments of `x`. In this setting, the semantics of a sequence of assignments is exactly captured by a conjunction of symbolic equalities. For instance, the sequence of instructions `t = x ; x = y ; y = t` have the semantic property $t^1 = x^0 \wedge x^1 = y^0 \wedge y^1 = t^1$ from which it is easy to prove that $x^1 = y^0 \wedge y^1 = x^0$, meaning that the instructions correctly swap the values of `x` and `y`.

Formally, the value of a program variable `x` in a state $\mathcal{V}$ is represented by the term $\textsc{eval}(\mathcal{V},\mathtt{x})$. In our framework formulæ and expressions are represented as *uninterpreted terms*, meaning that they are not functions but syntactical structures which can be explored. Expressions can be simplified by the mean of Rule (*simpl*, Fig. 1) using a recursive function (simpl) over the structure of terms that evaluates the arithmetic part and preserves the symbolic part of an expression. For an atomic expression e, $\mathsf{simpl}(e)$ is equal to e if e is a constant or a symbolic value x^n or a logical variable i; it is undefined if e is a program variable. We now define the effect of simpl on term constructors of expressions.

$$
\begin{aligned}
\mathsf{simpl}(\textsc{eval}(\mathcal{V},\mathtt{x})) &= x^k \quad \textit{a symbolic value with } k = \textsc{ac}(\mathcal{V},\mathtt{x}) \quad \textit{if } \mathtt{x} \textit{ is a program variable} \\
\mathsf{simpl}(\textsc{eval}(\mathcal{V},i)) &= i \quad \textit{if } i \textit{ is a logical variable (i.e. not sensitive to state)} \\
\mathsf{simpl}(\textsc{eval}(\mathcal{V},c)) &= c \quad \textit{if } c \textit{ is a constant} \\
\mathsf{simpl}(\textsc{eval}(\mathcal{V},\mathtt{t}[e_1,\ldots,e_n])) &= \textsc{eval}(\mathcal{V},\mathtt{t}[\mathsf{simpl}(\textsc{eval}(\mathcal{V},e_1)),\ldots,\mathsf{simpl}(\textsc{eval}(\mathcal{V},e_n))]) \textit{ if } \mathtt{t} \textit{ is an array} \\
\mathsf{simpl}(\textsc{eval}(\mathcal{V},\textsc{op}(e_1,\ldots,e_n))) &= \mathsf{simpl}(\textsc{op}(\textsc{eval}(\mathcal{V},e_1),\ldots,\textsc{eval}(\mathcal{V},e_n))) \\
\mathsf{simpl}(\textsc{op}(e_1,\ldots,e_n)) &= \begin{cases} \widehat{\mathsf{op}}\,(\mathsf{simpl}(e_1),\ldots,\mathsf{simpl}(e_n)) & \textit{if } \mathsf{simpl}(e_i) \textit{ is a constant, for all } i \\ \textsc{op}(\mathsf{simpl}(e_1),\ldots,\mathsf{simpl}(e_n)) & \textit{otherwise} \end{cases}
\end{aligned}
$$

The symbol $\widehat{\mathsf{op}}$ denotes the mathematical operator on values while OP is a term constructor. The term $\textsc{ac}(\mathcal{V},\mathtt{x})$ plays a central role ; it denotes the assignment counter of `x` at state $\mathcal{V}$. It is updated at each assignment of `x`, see Rule (*asg*$_3$, Fig. 9).

4 The semantic rules of the proof-system

The proof system that must be validated by the evaluators consists in the logical rules of Figure 1, the semantic rules of Figures 7, 8, 9, 10, additional rules to deal with operators $(+,-,<,\leq)$ on natural numbers, the rule $\frac{\text{BOOL}(b)}{\neg\neg b = b}$ on booleans and one typing rules per operator. The mathematical and typing rules are not provided. They are very simple since the protocol set boolean and above all the management of buffer doesn't depend on the domain of the data stored in it.

For the sake of clarity, the derivation rules are written using simplifying notations: all modifications of the set of hypotheses are due to the logical rules $(\vee_{elim}, \Rightarrow_{intro}, \neg_{intro}, \exists_{elim})$ of Figure 1. Therefore, we omit the hypotheses and write Φ instead of $\mathcal{H} \vdash \Phi$ in the semantic rules since none of these rules introduce or remove hypotheses. The symbol $(\uparrow\downarrow)$ means that the rule can be used in both directions and summarizes two derivation rules. The side-condition of a rule is a constraint that governs its application. It is evaluated by the proof-checker instead of being part of the proof. A typing side condition $\text{TYP}(x)$ on a variable x corresponds to checking that $\text{TYP}(x)$ belongs to the current hypotheses. The typing constraints ($\text{EVT}(e)$, $\text{SEQ}(\sigma)$,...) are omitted when the type of variable is clear from the context.

The intended meaning of the semantic rules is given in the figures and only the most significant will be commented here. Figure 7 defines well-formed traces of events. Due to the side condition $\text{WF}(\sigma)$, these rules can only be used with a sequence σ which has been declared as a *well-formed sequence* in the hypotheses. Successive runs of a task i form a sequence of events which can be defined by the regular expression $(\mathbb{t}_i\,;\mathbb{s}_i\,;\mathbb{f}_i)^*$. Then, a well-formed sequence is an interleaving of such sequences (for all tasks from 0 to T) that complies with the priorities of tasks. This definition is captured by the rules of Figure 7 which follow the original formulation of assumptions on the control systems given in [19].

Figure 8 formally defines properties of the operators $(..)$, $(;)$ and $(\subseteq)$ on traces of events. Rule $(=_{seq}$, Fig. 8) which derives equality on sub-sequences of a well-formed trace σ is valid since each event of σ is unique (events of a well-formed trace are annotated with an occurrence number). Figure 9 gives an operational semantics to the C constructions used in the protocol. The effect of an instruction is defined in terms of properties of its source and target states. The semantics of simple `for` loops $(for_1, for_2$, Fig. 9) is defined in terms of a sequence of simpler instructions. This form of equivalence-based semantics is easily validated by programmers. Our semantics of assignments – in terms of assignment counters $\text{AC}(\mathcal{V}, \mathtt{v})$, see $(asg_3$, Fig. 9) – is precise enough to express that a program P has no effect on a variable `v`: this holds if and only if the assignment counter of `v` hasn't been changed during the execution of P. Rule (*independency*) elaborates on this remark: if P_1 has no effect on a variable `v` then the effect of the execution of $P_1|P_2$ on `v` is that of P_2. No other property of the interleaving of programs is needed in the correctness proof. Finally, the rules of Figure 10 relate sequences of events, sequences of programs and transitions between states on the basis of associations $\text{PCSE}(P, \sigma)$ between a procedure call P and its sequence of events σ.

$$\mathrm{WF}(\sigma)\ \frac{\mathbb{s}_i^k \,..\, \mathbb{s}_j^p \subseteq \sigma \quad \texttt{prio[}i\texttt{]} > \texttt{prio[}j\texttt{]}}{\mathbb{s}_i^k \,..\, \mathbb{f}_i^k \,..\, \mathbb{s}_j^p \subseteq \sigma}\ \mathit{priority}_1 \qquad \mathrm{WF}(\sigma)\ \frac{\mathbb{s}_i^k \,..\, \mathbb{f}_j^p \subseteq \sigma \quad \texttt{prio[}i\texttt{]} > \texttt{prio[}j\texttt{]}}{\mathbb{s}_i^k \,..\, \mathbb{f}_i^k \,..\, \mathbb{f}_j^p \subseteq \sigma}\ \mathit{priority}_2$$

$$\mathrm{WF}(\sigma)\ \frac{\mathbb{s}_i^k \subseteq \sigma}{\mathbb{t}_i^k \,..\, \mathbb{s}_i^k \subseteq \sigma}\ \mathit{sched}_s \qquad \mathrm{WF}(\sigma)\ \frac{\mathbb{f}_i^k \subseteq \sigma}{\mathbb{s}_i^k \,..\, \mathbb{f}_i^k \subseteq \sigma}\ \mathit{sched}_f \qquad \mathrm{WF}(\sigma)\ \frac{\mathbb{t}_i^k \subseteq \sigma \quad k > 1}{\mathbb{f}_i^{k-1} \,..\, \mathbb{t}_i^k \subseteq \sigma}\ \mathit{sched}_t$$

$$\mathrm{EVT}(e)\ \frac{}{\exists i,k\ e = \mathbb{t}_i^k \vee e = \mathbb{s}_i^k \vee e = \mathbb{f}_i^k}\ \mathit{evt} \qquad \mathrm{EVT}(e),\mathrm{WF}(\sigma)\ \frac{e_i^k \,..\, e_i^p \subseteq \sigma}{k < p}\ \mathit{inc} \qquad \mathrm{WF}(\sigma)\ \frac{\mathcal{V} \xrightarrow{\sigma} \mathcal{V}'}{\forall v\ \mathrm{AC}(\mathcal{V}, v) = 0}\ \mathit{init}$$

The definition of well-formed traces of events captures the assumptions about the control system. A well-formed trace respects the priorities. If the priority of Task i is greater than that of Task j and Task i starts before Task j then it finishes its execution (event $\mathbb{f}_i^k$): before Task j starts ($\mathit{priority}_1$) and before the end of Task j ($\mathit{priority}_2$). Each starting event $\mathbb{s}_i^k$ is preceded by a triggering event $\mathbb{t}_i^k$ (sched_s). Each finishing event $\mathbb{f}_i^k$ is preceded by the corresponding starting event $\mathbb{s}_i^k$ (sched_f). A triggering event $\mathbb{t}_i^k$ is taken into account by the system (and appears in the trace) if and only if the current task ended its previous execution (sched_t). Rule (evt) says that events are elements of $\{\mathbb{t}_i^k, \mathbb{s}_i^k, \mathbb{f}_i^k \mid i, k \in \mathbb{N}\}$. The occurrence number of each event increases along a well-formed trace (inc). Finally, a well-formed trace of events starts with an initial state where the variables have not been assigned: their counters of assignment equal 0 and their values are unknown.

Fig. 7. The definition of well-formed traces of events

$$\frac{\sigma' \subseteq \sigma}{\exists \sigma_1,\sigma_2\ \sigma_1\,;\sigma'\,;\sigma_2 = \sigma}\ \mathit{def}_{\subseteq}(\uparrow\downarrow) \qquad \frac{\sigma_1 \,..\, \sigma_2 \subseteq \sigma}{\exists \sigma'\ \sigma_1\,;\sigma'\,;\sigma_2 \subseteq \sigma}\ \mathit{def}_{..}(\uparrow\downarrow) \qquad \mathrm{WF}(\sigma)\ \frac{e\,;\sigma_1\,;e' \subseteq \sigma \quad e\,;\sigma_2\,;e' \subseteq \sigma}{\sigma_1 = \sigma_2}\ =\!\mathit{seq}$$

$$\frac{\sigma_1\,;\sigma_2 \subseteq \sigma}{\sigma_1 \,..\, \sigma_2 \subseteq \sigma}\ \mathit{weakening} \qquad \frac{\sigma_1 \,..\, \sigma_2 \subseteq \sigma}{\sigma_1 \subseteq \sigma}\ \mathit{sub}_l \qquad \frac{\sigma_1 \,..\, \sigma_2 \subseteq \sigma}{\sigma_2 \subseteq \sigma}\ \mathit{sub}_r \qquad \frac{\sigma_1 \,..\, \sigma_2 \,..\, \sigma_3 \subseteq \sigma}{\sigma_1 \,..\, \sigma_3 \subseteq \sigma}\ \mathit{sub}_m$$

$$\frac{\sigma \,..\, e_1 \subseteq \sigma' \quad \sigma \,..\, e_2 \subseteq \sigma'}{\sigma \,..\, e_1 \,..\, e_2 \subseteq \sigma \ \vee\ \sigma \,..\, e_2 \,..\, e_1 \subseteq \sigma'}\ \mathit{merge} \qquad \frac{\sigma_1 \,..\, e \subseteq \sigma \quad e \,..\, \sigma_2 \subseteq \sigma}{\sigma_1 \,..\, e \,..\, \sigma_2 \subseteq \sigma}\ \mathit{join}$$

Rule ($\mathit{def}_{\subseteq}$) defines the sub-sequence predicate ($\subseteq$). Rule ($\mathit{def}_{..}$) captures the intended meaning of the loose sequence notation $\sigma_1 \,..\, \sigma_2$. Two sub-traces of a well-formed sequence σ which have corresponding starting and ending events are equivalent ($= \mathit{seq}$) since all events are unique in a well-formed sequence. Any sequence $\sigma_1;\sigma_2$ can be seen as a special case of a loose sequence $\sigma_1 \,..\, \sigma_2 = \sigma_1\,;\sigma\,;\sigma_2$ for an empty σ ($\mathit{weakening}$). Rules ($\mathit{sub}_l, \mathit{sub}_r, \mathit{sub}_m$) are used to focus on sub-parts of a given loose sequence. On the contrary loose sequences can be combined using Rules ($\mathit{merge}, \mathit{join}$). All of these rules are necessary and sufficient to conduct the case study on possible completion with the events $\mathbb{s}_w^k, \mathbb{f}_w^k, \mathbb{s}_r^p, \mathbb{f}_r^p$ of the loose sequence $\mathbb{t}_w^k \,..\, \mathbb{t}_r^p \,..\, \mathbb{t}_w^{k+1}$ used in the correctness property.

Fig. 8. Operations and relations between sequences (;) sub-sequences ($\subseteq$) and loose sequences (..)

4.1 The sketch of the proof and its construction

The proof of the correctness property of Figure 6 is driven by an induction on k of a property Φ that implies the correctness property. It is common in proofs of programs that the expected property is not inductive and therefore requires the proof of a stronger property Φ that is inductive and entails the desired one [4]. The Φ property extends the conclusion of the correctness property in a conjunction of the initial goal with additional propositions which state that 1) tasks other than r and w do not affect the arrays $A\texttt{[}r\texttt{]}\ldots\texttt{[...]}$ and $A\texttt{[}w\texttt{]}\ldots\texttt{[...]}$ for any array A used in the protocol and 2) in the target state of a triggering event $\mathbb{t}_r^p$ (resp. $\mathbb{t}_w^k$) the assignment counter of $\texttt{inp[}r\texttt{][}w\texttt{]}$ (resp. $\texttt{out[}w\texttt{]}$) is equal to p (resp. k). The interesting part in the proof of Φ consists in a case study of all the possible ways to complete the sub-sequence $\mathbb{t}_w^k \,..\, \mathbb{t}_r^p \,..\, \mathbb{t}_w^{k+1}$ with the events $\mathbb{s}_w^k$, $\mathbb{f}_w^k$, $\mathbb{s}_r^p$, $\mathbb{f}_r^p$. Then,

$$\frac{\langle \mathcal{V}, v\texttt{=}e, \mathcal{V}'\rangle}{\text{EVAL}(\mathcal{V}',v) = \text{EVAL}(\mathcal{V},e)}\ asg_1 \qquad \frac{\langle \mathcal{V}, v\texttt{=}e, \mathcal{V}'\rangle}{\forall x,\ \neg\text{ALIAS}(\mathcal{V},v,x) \Rightarrow \text{EVAL}(\mathcal{V}',x) = \text{EVAL}(\mathcal{V},x) \wedge \text{AC}(\mathcal{V}',x) = \text{AC}(\mathcal{V},x)}\ asg_2$$

$$\frac{\langle \mathcal{V}, v\texttt{=}e, \mathcal{V}'\rangle}{\text{AC}(\mathcal{V}',v) = \text{AC}(\mathcal{V},v) + 1}\ asg_3 \qquad \text{INT}(x),\text{INT}(y)\ \frac{x \overset{syntax}{\neq} y}{\forall \mathcal{V}\ \neg\text{ALIAS}(\mathcal{V},x,y)}\ no{-}alias_1$$

$$\text{DATA}(t[\ldots]),\text{DATA}(t'[\ldots])\ \frac{t \overset{syntax}{\neq} t' \ \vee\ \text{EVAL}(\mathcal{V},e_1) \neq \text{EVAL}(\mathcal{V},e_1') \vee \ldots \vee \text{EVAL}(\mathcal{V},e_n) \neq \text{EVAL}(\mathcal{V},e_n')}{\forall \mathcal{V}\ \neg\text{ALIAS}(\mathcal{V}, t\texttt{[}e_1\texttt{]}\texttt{[}\ldots\texttt{]}\texttt{[}e_n\texttt{]}, t'\texttt{[}e_1'\texttt{]}\ldots\texttt{[}e_n'\texttt{]})}\ no{-}alias_2$$

$$\frac{\langle \mathcal{V}_1, P_1\texttt{;}P_2, \mathcal{V}_2\rangle}{\exists \mathcal{V}\ \langle \mathcal{V}_1, P_1, \mathcal{V}\rangle \wedge \langle \mathcal{V}, P_2, \mathcal{V}_2\rangle}\ seq(\uparrow\downarrow) \qquad \frac{\langle \mathcal{V}, P, \mathcal{V}'\rangle \quad \text{EVAL}(\mathcal{V},exp)}{\langle \mathcal{V}, \texttt{if(}exp\texttt{)\{}P\texttt{\} else \{\_\}}, \mathcal{V}'\rangle}\ if_1 \qquad \frac{\langle \mathcal{V}, P, \mathcal{V}'\rangle \quad \neg(\text{EVAL}(\mathcal{V},exp))}{\langle \mathcal{V}, \texttt{if(}exp\texttt{)\{\_\} else \{}P\texttt{\}}, \mathcal{V}'\rangle}\ if_2$$

$$\frac{\langle \mathcal{V}, \texttt{for(i=0;i<0;i=i+1)\{\_\}}, \mathcal{V}'\rangle}{\langle \mathcal{V},\ \texttt{i=0}\ , \mathcal{V}'\rangle}\ for_1(\uparrow\downarrow) \qquad \texttt{safe}(P,\{\texttt{i},n\})\ \frac{\langle \mathcal{V}, \texttt{for(i=0;i<}n\texttt{;i=i+1)\{}P(\texttt{i})\texttt{\}}, \mathcal{V}'\rangle \wedge 0 < n}{\langle \mathcal{V}, \texttt{for(i=0;i<}n-1\texttt{;i=i+1)\{}P(\texttt{i})\texttt{\}}\ \texttt{;}\ P(n), \mathcal{V}'\rangle}\ for_2(\uparrow\downarrow)$$

$$\text{DECL}(\texttt{proc(}p_1,\ldots,p_n\texttt{)\{}Body(p_1,\ldots,p_n)\texttt{\}})\ \frac{\langle \mathcal{V},\ Body(\text{EVAL}(\mathcal{V},v_1),\ldots,\text{EVAL}(\mathcal{V},v_n))\ , \mathcal{V}'\rangle}{\langle \mathcal{V},\ \texttt{proc(}v_1,\ldots,v_n\texttt{)}\ , \mathcal{V}'\rangle}\ proc{-}call$$

Rule (asg_1) states that after the C instruction v=e the value of v in the target state is that of e in the source state. In the meantime, all variables which are not aliased to v are not affected by the assignment of v: neither their value nor their assignment counter (asg_2). The guarantee of non-aliasing is obtained by rules $(no{-}alias_1)$ and $(no{-}alias_2)$ which exploit the type of variables, the syntactic comparison of variable name, and in some cases a simple reasoning on the indices of arrays. Each assignment of a variable v increases its counter of assignment (asg_3). The simple `for` loops of the protocol are tackled by standard proofs of loop invariants that use the induction principle ($ind_{\mathbb{N}}$, Fig. 1) and the Rules (for_1, for_2) which define the meaning of a `for` loop in terms of simpler instructions. The condition $\texttt{safe}(P,V)$ of Rule (for_2) syntactically checks that the program P doesn't write variables of V. Rules (if_1, if_2) are borrowed from standard operational semantics of conditional instructions. Rule $(proc{-}call)$ relates procedure calls to the execution of their body. No other rule is needed to reason on the procedures `os` and `task` of the protocol.

Fig. 9. Semantic rules for the C instructions used in the protocol

$$\frac{}{\exists \mathcal{V},\mathcal{V}'\ \mathcal{V} \xrightarrow{\sigma} \mathcal{V}'}\ seq{-}state_1 \qquad \frac{\mathcal{V}_1 \xrightarrow{\sigma_1\,;\,\sigma_2} \mathcal{V}_2}{\exists \mathcal{V}\ \mathcal{V}_1 \xrightarrow{\sigma_1} \mathcal{V}\ \wedge\ \mathcal{V} \xrightarrow{\sigma_2} \mathcal{V}_2}\ seq{-}state_2(\uparrow\downarrow)$$

$$\frac{\exists \sigma \quad \mathcal{V} \xrightarrow{\sigma} \mathcal{V}'}{\exists P\ \langle \mathcal{V}, P, \mathcal{V}'\rangle}\ seq{-}prg_1(\uparrow\downarrow) \qquad \text{PCSE}(P_1,\sigma_1),\text{PCSE}(P_2,\sigma_2)\ \frac{\mathcal{V} \xrightarrow{\sigma_1|\sigma_2} \mathcal{V}'}{\langle \mathcal{V}, P_1|P_2, \mathcal{V}'\rangle}\ seq{-}prg_2(\uparrow\downarrow)$$

$$\frac{\mathcal{V} \xrightarrow{\sigma} \mathcal{V}' \quad \text{SIZE}(\sigma,0)}{\mathcal{V} = \mathcal{V}'}\ size_0 \qquad \text{EVT}(e)\ \frac{}{\text{SIZE}(e,1)}\ size_1 \qquad \frac{\text{SIZE}(\sigma,n) \quad \text{SIZE}(\sigma',n')}{\text{SIZE}(\sigma|\sigma', n+n')}\ size_2$$

$$\frac{\forall \mathcal{V}_1,\ \langle \mathcal{V}, P_1, \mathcal{V}_1\rangle \Rightarrow \text{AC}(\mathcal{V},v) = \text{AC}(\mathcal{V}_1,v) \quad \langle \mathcal{V}, P_2, \mathcal{V}_2\rangle}{\langle \mathcal{V}, P_1|P_2, \mathcal{V}'\rangle \Rightarrow \text{EVAL}(\mathcal{V}',v) = \text{EVAL}(\mathcal{V}_2,v)}\ independency$$

Rules $(seq{-}state_1)$ and $(seq{-}state_2)$ link memory states and sequences of events by means of the predicate $\mathcal{V} \xrightarrow{\sigma} \mathcal{V}'$. Then, Rules $(seq{-}prg_1)$ and $(seq{-}prg_2)$ associate sequences of events to executions of programs and *vice versa*: Rule $(seq{-}prg_1)$ assumes the existence of a correspondence between executions of programs and sequences of events. Rule $(seq{-}prg_2)$ exploits the sequences of events associated to programs by the static predicate PCSE in order to bind the interleaved execution of two programs to the interleaving of their sequences of events. Two states that are joined by a sequence of size 0 are equal $(size_0)$. An event is a sequence of size 1 $(size_1)$. No event is lost in the interleaving of two sequences $(size_2)$. The last rule (*independency*) states a crucial property of the interleaved execution of two programs P_1 and P_2. If P_1 has no effect on a variable v – *i.e.* its assignment counter $\text{AC}(\ldots, v)$ has not been modified from $\mathcal{V}$ to $\mathcal{V}_1$ – then, the effect of $P_1|P_2$ on v is that of P_2. This rule is a reformulation of the Owicki-Gries' rule for concurrent programs.

Fig. 10. Relating memory states, sequences of events and sequences of instructions

for each sub-sequence, the execution of the procedures associated to events are examined to prove $(\texttt{inp[}r\texttt{][}w\texttt{]})^p = (\texttt{out[}w\texttt{]})^k$ in every case. All of these proofs rely on the same two lemmata which show that: 1) tasks other than r and w that can be interleaved with r and w do not affect the pair of buffers used by r and w ; 2) the effect of $\texttt{os(}r\texttt{)}$, $\texttt{os(}w\texttt{)}$, $\texttt{task(}r\texttt{)}$, $\texttt{task(}w\texttt{)}$ realize the expected data-flow.

Large parts of the proof were conducted automatically with the help of a symbolic interpreter of C programs developed for this occasion. The interpreter interacts with the user for conditional instructions and records the derivation steps in a proof-term that is built during the guided execution. The resulting proof-term contains proof-obligations which must be completed afterward. The inductions and the proof of independency were done by hand with the help of lemmata.[5]

By using a prover such as PVS or B we would have benefited from simplifications and decision procedures for arithmetic but it is not possible to produce an independent proof-term using these provers. Therefore, we would have had to include the prover in the TCB. Actually we started with the COQ proof-assistant which produces proof-terms but the overhead, the complexity of formalization and the numerous proof-obligations revealed that COQ was not inappropriate for our purpose, even if it provides tactics to reduce the proof activity. Moreover, the COQ proof-checker is not yet certified.

5 Conclusion

The evaluators need to be completely convinced before delivering a certificate of correctness for an application. At the highest level of certification, evaluators must be suspicious (the proof must address the actual correctness property and the actual program instead of an abstract model of the problem), conservative (they dislike new theories and new tools that require a deep understanding to validate their verdict) and rigorous (they will investigate the provided evidence until they reach a deep understanding of the proof and will reject any opaque deduction). Finally, they are skeptical (they only accept combinations of low level evidence), and, justly, paranoiac (they trust almost no tools and ask for a minimal TCB). Generally speaking, the less theoretical background is needed to understand a proof, the easier they are to convince.

PCC, FPCC and verification tools that produce an independent checkable witness of their verdict give a great level of confidence in a software. However, these techniques fail to fulfill all of the evaluators' requirements. To date, the greatest level of confidence has been reached by developments in LF or COQ. These frameworks produce formal proofs using the smallest known set of general purpose derivation rules and, consequently, they offer small proof-checkers [1]. Even then, the evaluators can reject such proofs if they are not familiar with the underlying theory of LF and COQ which are based on dependant types and the Curry-Howard isomorphism [18]. More likely, such proofs can be rejected due to the overly detailed formalization of the semantic model that is required in these provers [9].

[5] Lemmata cannot introduce potential flaws as they are just systematic combinations of derivation rules.

In this paper we present the notion of *convincing proof* designed to meet evaluators' requirements. It can be summarized as follows: the logic of the property and the semantics of the system are explicit; they are given as derivation rules which must be validated by the evaluators; the proof uses only these validated rules; the proof-checker is also validated as it consists in a straightforward translation of the derivation rules in a recursive function. Therefore the TCB is close to minimum: {a compiler,[6] a machine}. Our methodology reduces the evaluators' task to the validation of the derivation rules and only relies on the theoretical background of any competent bachelor in computer science.

We illustrate our framework on the correctness proof of a data exchange protocol used to implement multi-tasking real-time data-flow applications on an event-based operating system featuring priority and preemption [19]. The proof of the protocol takes into account all of the system characteristics and provides evidence at a reasonable level of detail, avoiding the full description of an operational semantics of concurrency (to render the management of tasks by the OS).

Our goal is now to produce convincing proofs by instrumentation of existing automatic verification tools. This is achievable once the set of derivation rules that correspond to the verification steps has been identified. We also seek for a way to benefit from the power of the COQ proof-assistant [6] and its numerous libraries of mathematical theories while maintaining the global proof at a convincing level of reasoning – unnecessary details must be kept out of the proof. Our plan is to isolate purely mathematical theorems and to reserve COQ for their demonstration.

References

[1] Appel, A., N. Michael, A. Stump and R. Virga, *A trustworthy proof checker*, Journal of Automated Reasoning **31** (2003), pp. 231–260.

[2] Arons, T., A. Pnueli, S. Ruah, J. Xu and L. Zuck, *Parameterized verification with automatically computed inductive assertions*, in: *Computer Aided Verification*, LNCS **2102** (2001), pp. 221–234, (CAV'01).

[3] Barras, B., *Verification of the interface of a small proof system in coq*, in: *Types for Proofs and Programs*, LNCS **1512** (1996), pp. 28–45, (TYPES'96).

[4] Bensalem, S., Y. Lakhnech and S. Owre, *InVeSt: A tool for the verification of invariants*, in: *Computer Aided Verification*, LNCS **1427** (1998), pp. 505–510, (CAV'98).

[5] Benveniste, A., P. Caspi, S. Edwards, n. Halbwachs, P. Le Guernic and R. de Simone, *The synchronous languages 12 years later*, IEEE Computer Society **91** (2003), pp. 64–83.

[6] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)," Texts in Theoretical Computer Science, Springer, 2004.

[7] Filliâtre, J.-C., *Verification of non-functional programs using interpretations in type theory*, Journal of Functional Programming **13** (2003), pp. 709–745.

[8] Filliâtre, J.-C. and C. Marché, *The Why/Krakatoa/Caduceus platform for deductive program verification*, in: *Computer Aided Verification*, LNCS **4590** (2007), pp. 173–177, (CAV'07).

[9] Hamid, N. and Z. Shao, *Interfacing Hoare logic and type systems for foundational proof-carrying code*, in: *Theorem Proving in Higher Order Logics*, LNCS **3223** (2004), pp. 118–135, (TPHOL'04).

[6] Attempts to remove compilers from the TCB are not convincing, unless we can certify one compiler. The research on this topic has recently obtained encouraging results [12].

[10] Hennell, M., J. Woodcock and M. Woodward, *The safety integrity levels of IEC 61508 and a revised proposal*, in: *Embedded Systems Show* (2006), (ESS'06).

[11] Henzinger, T., R. Jhala, R. Majumdar, G. Necula, G. Sutre and W. Weimer, *Temporal-safety proofs for systems code*, in: *Computer Aided Verification*, LNCS **2404** (2002), pp. 526–538, (CAV'02).

[12] Leroy, X., *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, in: *ACM Conference on Principles of Programming Languages* (2006), pp. 42–54, (POPL'06).

[13] Namjoshi, K. S., *Certifying model checkers*, in: *Computer Aided Verification*, LNCS **2102** (2001), pp. 2–13, (CAV'01).

[14] Necula, G. C., *Proof-carrying code*, in: *ACM Conference on Principles of Programming Languages* (1997), pp. 106–119, (POPL'97).

[15] Peled, D. and L. Zuck, *From model checking to a temporal proof*, in: *SPIN Workshop on Model Checking Software*, LNCS **2057** (2001), pp. 1–14, (SPIN'01).

[16] Randimbivololona, F., J. Souyris, P. Baudin, A. Pacalet, J. Raguideau and D. Schoen, *Applying formal proof techniques to avionics software: A pragmatic approach*, in: *Formal Methods in the Development of Computing Systems*, LNCS **1709** (1999), pp. 1798–1815, (FM'99).

[17] Tan, T. and R. Cleaveland, *Evidence-based model checking*, in: *Computer Aided Verification*, LNCS **2404** (2002), pp. 455–470, (CAV'02).

[18] Thompson, S., "Type theory and functional programming," Addison-Wesley, 1991.

[19] Tripakis, S., C. Sofronis, N. Scaife and P. Caspi, *Semantics-preserving and memory-efficient implementation of inter-task communication under static-priority or EDF schedulers*, in: *ACM Conference on Embedded Software* (2005), pp. 353–360, (EMSOFT'05).

[20] van Dalen, D., "Logic and Structure," Springer-Verlag, 1997.

[21] Wilding, M., D. A. Greve and D. Hardin, *Efficient simulation of formal processor models*, Formal Methods in System Design **18** (2001), pp. 233–248.

[22] Wu, D., A. W. Appel and A. Stump, *Foundational proof checkers with small witnesses*, in: *ACM Conference on Principles and Practice of Declarative Programming* (2003), pp. 264–274, (PPDP'03).